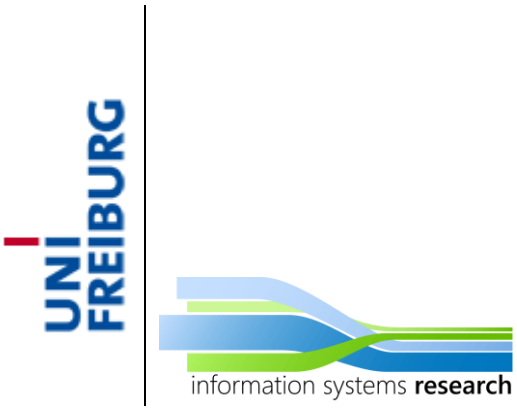

Applications of Business Intelligence

– SEMINAR WINTER SEMESTER 2014/2015 –

Parallel Execution in R

– SEMINAR PAPER –

Submitted by:
Arin Öcal

Advisor:
Prof. Dr. Dirk Neumann

Table of Contents

1. Introduction

Since its launch in 1993, R has proven itself to be an immensely useful tool for individuals who are dealing with statistical computing and data analysis. Emergence of Big Data contributed to the need of having such an environment like R intensely. Several methods of measuring present a significant increase in the popularity of R over the past decade, compared to other environments in the same area (Muenchen, 2012). On the other hand, Big Data also helped to reveal where R can be inefficient due to the requirements of today's data analysis scene. Given the amount of data to process nowadays, parallelism comes to aid of many who require heavy computational tasks to be performed. These computations can be done in much smaller intervals of time, which is the main motivation behind parallelism. With the current widespread availability of multi core systems, it would be inefficient not to take advantage of these hardware resources.

However, parallelism or high performance computing was not on the agenda when R was initially being developed. It is now difficult to overlook the need with ever-growing data sets and more complex simulations arising from advanced methodologies. A good example for data sets would be practices in bioinformatics, where DNA sequencing data sets take more than a DVD to fit. Performance increases in hardware do not match the growth rate of data sets. From a methodological point of view, advancements in simulation and resampling techniques brought the need for advanced computations. Markov Chain Monte Carlo and Gibbs sampling, bootstrapping and Monte Carlo simulations which are used in geographical, econometric and biological data analysis demand higher performance for efficiency. Parallelism can be an effective solution for both cases, when the subjects are data sets or simulations. Big data sets can be divided into individual "chunks" before they go through parallel processing. Simulations that can be run independently from each other can also be parallelized in a similar manner (Schmidberger et. al, 2009).

The rest of this paper is organized as follows: Section 2 makes a comparison between sequential versus parallel computing, as well as analyzing the conditions under which parallelization can be advantageous. In Section 3, infrastructural aspects of parallel computing, such as underlying hardware and software technologies are being mentioned. Finally, specific packages for parallel computing in R have been brought into attention on Section 4, as well as relevant examples.

2. Sequential versus Parallel Computing

Parallelism refers to taking advantage of more than one tool of calculation (such as multiple CPUs) to handle tasks which require many computations to be performed. By utilizing hardware and software to carry out many calculations simultaneously, it mainly aims at improving calculation capacity.

In order to have a better understanding of how the concept of parallelization actually works, it would be necessary to compare it to the traditional way of handling the computations, also known as

sequential processing. In sequential (also known as serial) processing, tasks are performed one after each other on a single processor, as visualized in Figure 1 - Sequential Processing. It is only possible to process one task at the same time.

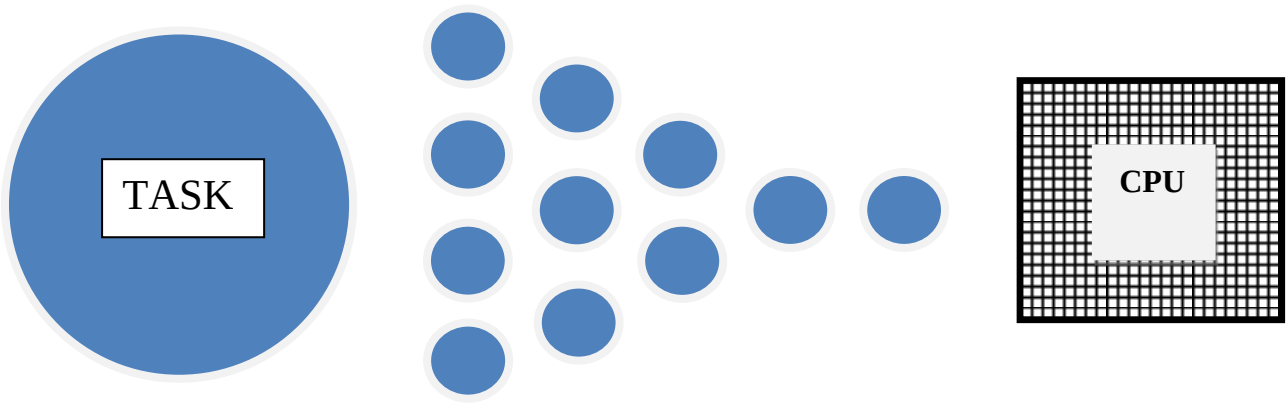

Figure 1 - Sequential Processing (cf. Zhang, 2014).

Things go substantially different when we utilize parallel processing. Firstly, the task is separated into different parts of work. Afterwards, every single part is being handled by different processors at the same time. An illustration in Figure 2 - Parallel Processing is given below.

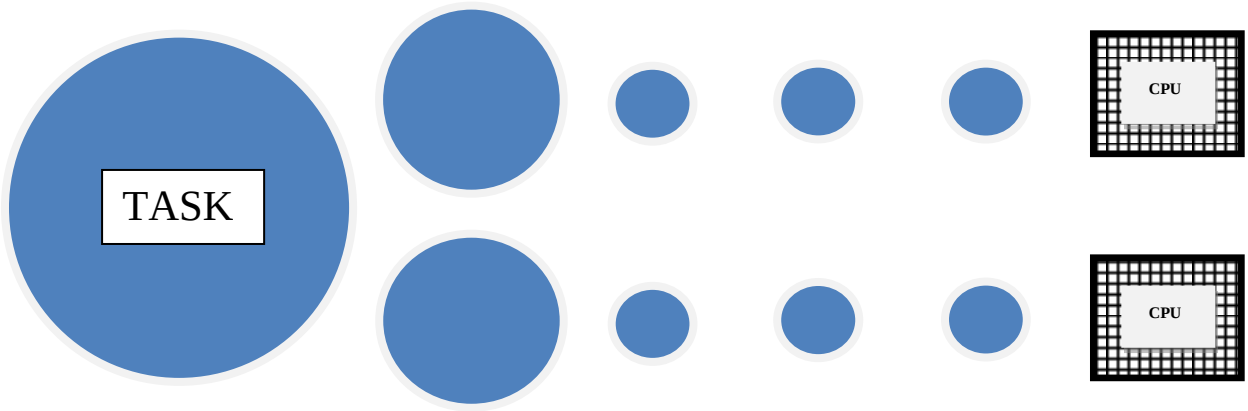

Figure 2 - Parallel Processing (cf. Zhang, 2014).

2.1 Embarrassing or Non-Embarrassing Parallelism

Although it seems very appealing, there are a few points to consider before attempting parallelization. One of the conditions for parallel processing can be named as having "embarrassingly parallel" computations for processing. This basically refers to computations having no dependency on each other in order to be successfully executed. In other words, they should not be "communicating" with each other (Leach, 2012). A very good example of an embarrassingly parallel problem would be the multiplication of matrix A with the vector X. AX can be computed in parallel by assigning individual cores or processors, each handling one row of matrix A for the multiplication process. No matter what the word "embarrassing" suggests, it is actually a preferable situation to have embarrassingly parallel applications. They are much easier to program, and they are essential for good performance since their communication requirements are very low, if not non-existent. It is true that non-embarrassingly parallel problems can also be expected. However, parallelizing them can be a difficult task when it is apparent that computations will need more communication between each other. Since communication necessities between processes will be time consuming already, these problems should be run on multi-core systems rather than clusters. A classical example of a non-embarrassingly parallel problem would be linear regression analysis. The necessity of doing matrix inversions or QR factorizations in this case would demand more effort from the programmer and hardware alike (Matloff, 2013). R package library offers solutions also for less-embarrassing problems on multi-core systems, although they are a bit more difficult to deal with compared to their embarrassing counterparts (Leach, 2014).

3. Technologies of Parallelization

Before going into practical details of parallel execution in R, it is useful to know that parallelization has two infrastructural aspects. While the hardware side is dealing with the physical environment of parallel computation, software side is about how to organize the communication among the components.

3.1 Hardware

There are basically three types of parallel computing environments. First and the most easily accessible one is a computer with multi-core or multiple processors. Although there is more than one CPU is present, memory is shared between the processing elements. On distributed computer systems, machines are connected to each other with all their components ready to work on the same task. If the computers are connected to the same local area network, this setting is called a cluster. In terms of hardware and software, machines on a cluster are usually homogenous. Furthermore, when machines are connected to each other through a wide area network, e.g. the internet, it is called grid computing. It is expected that the machines on a grid are heterogeneous, thus requiring extra effort while working in parallel (Schmidberger et al., 2009).

3.2 Software

A parallelization framework is based on a master/slave design. The master process creates slave processes throughout the cluster, and distributes the task to them. As the slaves are done with their tasks, they send them back to the master. An illustration can be seen below (Figure 3 - Master/Slave .

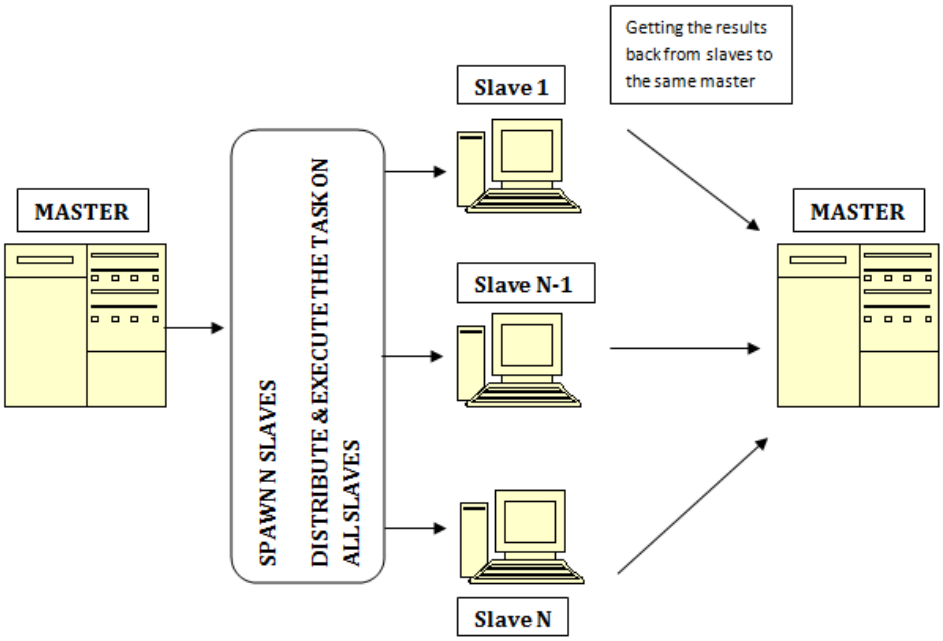

Figure 3 - Master/Slave Design (cf. Mahdi, 2014).

The need for communication between master and slaves can be done through different mechanisms. They can use a socket interface via the operating system, or utilize the Private Virtual Machine (PVM) message passing framework, or the Message Passing Interface (MPI). On R, the last two methods are realized through packages rpvm and Rmpi. These packages are already implemented in the popular parallelization packages such as snow. While it is possible to specify which one of these mechanisms to use in snow, details of how these low-level communication mechanisms are used are not visible to the user of the high-level interface.(Rossini, Tierney, & Li, 2007).

4. Parallelization Packages for R

Fortunately, R can be equipped with a wide selection of packages to help analyzers maintain various tasks. It is no exception when it comes to parallel computing. In this section, the most significant and/or popular R packages on parallelization will be introduced. Before going further into the details of the each package being mentioned on this paper, a quick overview of them is included in the table below (Table 1).

Package	Communication Mechanism - Operating System	Remarks
snow	MPI, PVM, NetWorkSpaces, SOCK - Unix, Windows	Especially convenient for traditional clusters
multicore	Fork system call - Unix	Only useful on a single multi-core machine
parallel	Same as snow and multicore combined - Unix, Windows	Merger of snow and multicore
foreach	Looping construct - Unix, Windows	Can iterate loops in parallel by using a backend
doParallel	Same as parallel - Unix, Windows	Parallel backend for foreach

Table 1 - Comparison of Parallelization Packages

4.1 snow[1]

"Simple Network of Workstations" might be the most well-known of all parallel computing solutions for R environment. It is most frequently used on Linux clusters, but can be utilized on multi-core machines running Windows. This flexibility is due to snow's capability of working with various low-level communication mechanisms such as socket connections, MPI, PVM and NetWorkSpaces (through packages Rmpi, Rpvm and nws). Since there are other specific solutions available for multi-core systems, snow is mostly convenient for clusters (McCallum & Weston, 2012).

It can easily be loaded by using the code

```
library("snow")
```

However, it must be installed on all machines running on the cluster, all at the same path location. Afterwards, the first step for parallelization should be creating a cluster object which will be used to communicate with the workers on the cluster

```
clusterobject <- makeCluster(4, type="SOCK")
```

This expression creates 4 workers using the socket connection layer. Instead of "SOCK", another communication layer argument such as "MPI", "PVM" or "NWS" can be used depending on the clustering decision.

Spawned workers will run until they are told to stop with the command

```
stopCluster(cl)
```

[1] ***cran**.r-project.org/package=**snow***

4.2 multicore[2]

multicore is another famous package for parallel computing on R. As the name suggests, it lets the user take advantage of their multi-core or multi-processor system. It is relatively easy to use on Linux and Mac OS due to its utilization of fork() system call, which is an operation that allows a process to create copies of itself on Unix-like systems. Therefore, it cannot be used on Windows. Nevertheless, it offers a better experience compared to snow in terms of memory efficiency.

multicore is mostly based on the usage of the function `mclapply()`. It is essentially the same as `lapply()`, but makes use of multi-cores or processors available. Loading the multicore package and replacing the `lapply()` function in the code with `mclapply()` would mostly be sufficient in parallelizing the code. (McCallum & Weston, 2012).

4.3 parallel[3]

The package "parallel" comes built in starting from R version 2.14.0. It can be referred as a combination of packages "snow" and "multicore", making it very useful in different operating system environments. Functionality of both packages is present in "parallel". On Unix-like systems, the function `mclapply()` can be used the same way as in the multicore package. On multi-core Windows systems, snow-like functionality of the parallel package should be used.

4.4 foreach[4]

foreach is a looping construct for R, written in order to provide multiple execution of a code without the need of a separate loop counter. It is similar to the `lapply()`, in terms of applying the function over a set of items. R supports a variety of other looping statements such as `for`, `repeat` and `while`. However, the most important feature of foreach is the ability to support parallel processing. Parallelism basically consists of breaking a problem into smaller pieces, and putting the outcomes back together after processing them in parallel. Foreach helps the programmer to do exactly these things. Iterators of foreach help splitting the task, `%dopar%` runs the code in parallel and `.combine` argument brings the results together (Weston, 2014). It is excessively used together with the package "doParallel", which will be introduced in the next section.

[2] ***cran**.r-project.org/package=**multicore***
[3] *https://stat.ethz.ch/R-manual/R-devel/library/**parallel**/doc/**parallel**.pdf*
[4] ***cran**.r-project.org/package=**foreach***

4.5 doParallel[5]

In order to use the foreach package for parallel processing, there is the need of a "parallel backend". After registering the parallel backend, foreach can switch to execute codes in parallel. doParallel is no more than an interface between previously mentioned foreach and parallel packages. Since the "parallel" package itself is consisting of snow and multicore, doParallel automatically chooses to use snow functionality on Windows and multicore functions on single computers running Unix-like operating systems. Parallelization on clusters can also be made by means of snow functionality.

An example of a simple doParallel-foreach iteration process on a multi-core system running Windows is given below. The code creates a normally-distributed random variable of 100000 samples for 10 iterations.

```
#Importing required packages
library(foreach)
library(doParallel)

#Number of iterations
iters<-1e3

#Ordering R to create a "cluster" of 4. Since we are using a multi-core CPU, our cluster will be
#consisting of 4 cores.
cl<-makeCluster(4)

#Registering the parallel backend, so the incoming foreach loop can take advantage of the specified
#cluster.
registerDoParallel(cl)

#Foreach loop with %dopar% argument, specifically instructs foreach to run in parallel.
ls<-foreach(icount(iters)) %dopar% {
    to.ls<-rnorm(1e5)}

#Stopping the cluster to avoid leaking cluster workers in case the cluster type changes (McCallum &
#Weston, 2012).
stopCluster(cl)
```

[5] ***cran**.r-project.org/package=**doParallel***

On an Intel i5 quad-core laptop, this parallelized code executes in about 8 seconds. If we want to execute the same code sequentially, we do not have to specify how many cores R has to use, as well as not registering a parallel backend. The sequential version of the code is given below.

```
library(foreach)

iters<-1e3
ls<-foreach(icount(iters)) %do% {
    to.ls<-rnorm(1e5) }
```

On the same computer system that was mentioned before, this sequential code takes about 11 seconds to run.

As a remark, it should be mentioned that running a code with the `%dopar%` argument would cause the code be executed sequentially unless there is a registered parallel backend (Weston & Calaway, 2014).

4.5.1 Running a Random Forest in Parallel

Random forests are strong and useful tools in ensemble learning. They assist the prediction process by spawning many decision trees, which are bootstrapped according to random sampled variables. Then the final prediction is being made by combining the outcomes of all the trees. (Kabacoff, 2014) Spawning the individual trees in a random forest algorithm is an embarrassingly parallel process, therefore making it possible to take advantage of parallel processing.

In the example given below, R package "randomForest" has been used together with "doParallel" in order to parallelize the process. A random forest of 10000 trees has been created by running the randomForest function 4 times, which is the number of cores in the given computer (Weston, 2014).

```
#Inputs on which the random forest will be based have to be created first. Matrix X consists of 500
#randomly generated numbers on 100 rows. Factor Y has 50 replications of 2 levels.

x <- matrix(runif(500), 100)
y <- gl(2, 50)


#Required packages are being loaded. Since the package "foreach" is required to run "doParallel", it
#will not be loaded exclusively.
library(randomForest)
library(doParallel)


#Creating a "cluster" of 4 cores and registering the parallel backend.
cl<-makeCluster(4)
registerDoParallel(cl)
```

#While creating random forest of 2500 trees 4 times and combining the outcomes, processing time has #also been measured.

```
system.time ({ rf <- foreach (ntree=rep(2500,4), .combine=combine,
.packages="randomForest") %dopar%
+ randomForest (x,y,ntree=ntree)})
```

This parallel example runs in 1 second. When the %dopar% argument is changed into %do% for the aim of running the same code sequentially, calculation takes 1.3 seconds to finish.

4.5.2 Parallelized Machine Learning with the caret Package

Resampling is the key method when it comes to optimizing predictive models. This process requires numerous different versions of a training set to be used, and it has to be done multiple times to come up with performance estimates that are applicable to other data sets. Resampled sets are independent of each other, so parallel processing can be used to reduce time and increase the efficiency ("Parallel Processing," 2014).

Caret has been developed to be "parallel-friendly", hence parallelizing a code is a simple procedure. As it has been done in the random forest example, a parallel backend has to be registered. Package functions stay as they are. A small example is given below:

#Loading packages caret and doParallel

```
library(doParallel)
library(caret)
```

#Creating a "cluster" of 4 cores and registering the parallel backend.

```
cl<-makeCluster(4)
registerDoParallel(cl)
```

#All the code in caret then runs in parallel, such as "`train`".

```
model <- train(y ~ ., data = training, method = "rf")
```

5. Conclusion

Parallel processing is definitely a time-saving concept for many researchers in various fields. Although it may take a few tweaking at times, the return can be enormously beneficial to those who were able to successfully parallelize their code. As Rosario (2012) beautifully states, parallelization should only be made "...if the cost of computation is higher than the cost of setting up the framework". Ongoing developments in R programming make parallelization easily available to people without backgrounds in software engineering. The package doParallel is a very good example of how simplified the process can be for an end-user. R is leveraging its position as a platform for parallel processing even further, with the capability of using the Hadoop framework for big data analysis. Since Hadoop utilizes local or cloud-based clusters for processing, memory limitations also cease to exist in addition to dealing with the single-threaded nature of R (McCallum & Weston, 2012).

6. References

Kabacoff, R. I. (2014). Quick-R: Tree-Based Models. Retrieved March 27, 2015, from http://www.statmethods.net/advstats/cart.html

Leach, C. (2014, April 10). Introduction to parallel computing in R. Retrieved from http://michaeljkoontz.weebly.com/uploads/1/9/9/4/19940979/parallel.pdf

Mahdi, E. (2014). A Survey of R Software for Parallel Computing. *American Journal of Applied Mathematics and Statistics*, *2.4*(2014), 224-230. Retrieved from http://pubs.sciepub.com/ajams/2/4/9/#

Matloff, N. (2013). Parallel Computing For Data Science. Retrieved February 15, 2015, from http://heather.cs.ucdavis.edu/paralleldatasci.pdf

McCallum, Q. E., & Weston, S. (2012). *Parallel R*. Sebastopol, CA: O'Reilly Media.

Muenchen, R. A. (2012). The Popularity of Data Analysis Software. Retrieved February 15, 2015, from http://r4stats.com/articles/popularity/

Parallel Processing. (2014, December 31). Retrieved March 27, 2015, from http://topepo.github.io/caret/parallel.html

Rosario, R. R. (2012, April 17). Parallelization in R, Revisited. Retrieved from http://www.bytemining.com/files/talks/larug/hpc2012/HPC_in_R_rev2012.pdf

Rossini, A. J., Tierney, L., & Li, N. (2007). Simple Parallel Statistical Computing in R. *Journal of Computational and Graphical Statistics*, *16*(2), 399-420. doi:10.1198/106186007X178979

Schmidberger, M., Morgan, M., Eddelbuettel, D., Yu, H., Tierney, L., & Mansmann, U. (2009). State of the Art in Parallel Computing with R. *Journal of Statistical Software*, *31*(1), 2-4.

Weston, S. (2014, April 10). Using the foreach package. Retrieved March 25, 2015, from http://cran.r-project.org/web/packages/foreach/vignettes/foreach.pdf

Weston, S., & Calaway, R. (2014, February 26). Getting Started with doParallel and foreach. Retrieved from http://cran.r-project.org/web/packages/doParallel/vignettes/gettingstartedParallel.pdf

Zhang, V. S. (2014). R workshop XX - Parallel Computing with R. Retrieved from http://www.slideshare.net/ShangxuanZhang/r-workshop-xx-parallel-computing-in-r?related=5

7. List of Figures

8. List of Tables