Norwegian School of Economics
Bergen, November 2018

BAN421 - Data Structures in R
Parallel computing with R

Group 3
Candidates 38 and 21

Abstract

In this project we look at different approaches to implementing parallel computing in R. There are many helpful packages to use, both in base R and other packages available on the CRAN repository. The performance gains of using packages utilizing multiple cores were significant, and we were able to speed up computational tasks almost linearly with the amount of cores used. This shows there are great advantages to using parallel processing in R.

Contents

1 Introduction

In this paper, we are going to explore different methods for parallel processing in R. We are going to benchmark base R's single-threaded looping function `lapply` against its multicore functionality, `mcapply`, and other functions for parallel computing provided by the `parallel`, `DoParallel` and `foreach` packages.

1.1 Motivation

The motivation for looking into parallel processing is because most modern computers have multiple cores at their disposal. Since R is single threaded (Zhao, 2016), R does utilize all the processing power available on the computer. With large data sets to process, parallel processing can give large increases in performance.

We wish to investigate the performance gains actually obtained by implementing parallel processing in regularly performed tasks, and see how difficult it is to implement this correctly. Are the performance gains large enough to justify the change in how the code is written and are there any major challenges to overcome with this implementation?

1.2 Expected results

Considering one processor or core will perform a task or calculation in t time units, we would expect n processors to be able perform in the same task in $\frac{1}{n}t + c$, where c is some extra time consumed for dividing the task into smaller tasks and combining the result afterwards. If the time consumption for this c is low enough, we could expect close to linear performance gains for each processor added to the computer system. This also means that if c is sufficiently large, there is no gain to performing the task in parallel at all, since we would be using too much time just dividing up the task at hand rather than performing said task.

2 Parallel computing

Generally, parallel computing in R can be achieved either via *forking* or via a *socket cluster*. Both work by creating new processes, splitting up the problem in multiple pieces, and computing them in parallel. Forking clones the original process, so each new process uses the exact same data for its computations. That also means that all data in the environment, i.e. variables and loaded packages, will be the same for all processes. In contrast, the socket approach creates an entirely new R session, analogous to opening a new window in RStudio. This means that e.g. packages needs to be loaded explicitly within the call. (Errickson, 2017) (Dursi, 2017)

Each method has its use cases. Forking is the easiest one to implement, and can be faster than using a cluster. However, the cluster approach has its advantage with big data. To combat memory issues, one could divide the dataset in more manageable chunks, and assign each chunk to a node in the cluster for parallel processing. In addition, the cluster can even be on a remote computer, or on a cloud computing platform.

2.1 Packages

2.1.1 Parallel

The package `Parallel` is base R's take on parallel processing. It builds on earlier packages `multicore` and `snow`. The `multicore` package was created with a focus on spreading work load onto different cores or processors (forking), and the `snow` package was created for the purpose of distributing work load onto different computer systems for distributed computing (using socket clusters) (R Core Team, 2018).

2.1.2 Foreach and doParallel

`Foreach` is one of R's most popular packages for parallel processing. It introduces a new looping construct for parallel execution. The difficulty of parallel processing occurs when the parallel computations needs to be merged. `Foreach` provides a framework for doing this, making it much less time-consuming. Each computation is stored in a list, which

is what the `foreach()` function returns. Another feature is the flexibility in how the computations are combined to the final output. (Microsoft and Weston, 2017)

`Foreach` needs a back-end package loaded for performing parallel computing. This could be either the package `doMC` for forking, or `parallel` for using a cluster, among others. Another package, such as `doParallel`, is needed as an interface between `foreach` and the back-end.

3 Performance analysis

For assessing the performance gains of parallel processing we are going to perform several synthetic tests on simulated data. This is in order to ensure the replicability of our results and because this is the easiest way for us to measure the performance difference between sequential and parallel performance without providing access to a larger real world data set. This also means we can check the performance gains on small sets of data against larger sets of data quite easily. We also assume this is within the scope of this final project.

3.1 Birthday paradox benchmark

This example is from a talk at the European R Users Meeting by David Smith (2018). A function calculates the probability that, in a room with n people, two of them will have birthday on the same day. The function is used in a loop to calculate the probability for all n.

Various looping methods are compared. Base R's single threaded `lapply` is compared to `foreach` and `parallel`'s `mclapply` and `parLapply`. In addition, we wanted to explore the difference between using hyper-threading (HT) to register more workers than physical cores available, and using only the amount of workers equal to the number of physical cores on the computer. The computer used for the benchmark is a Macbook Pro with a single processor. The Intel i7 processor has four cores, but also supports hyper-threading, where each core consists of two pseudo-cores. Consequently, when calling `detectCores()`, it will return 8, the number of cores avvailable when using hyper-threading. By setting the parameter `logical` to `FALSE`, it will the detect the number of physical cores on the

processor.

Based on Zhao (2016), our hypothesis is that the hyper-threading method will only show marginal gains compared to using solely physical cores. We also expect the forking methods to be superior, because of the additional overhead by creating independent clusters will be particularly noticeable in our relatively simple loop.

3.1.1 Code

For the first `foreach` experiment, the name of the package for each function is shown explicitly. This is because we want to show how `parallel`, `registerDoParallel` and `foreach` works in conjunction with each other, as R's back-end for parallel computing, an interface between `parallel` and `foreach`, and execution mechanism, respectively.

```
1  library(rbenchmark)
2  library(parallel)
3  library(doParallel)
4  library(foreach)
5  library(dplyr)
6  library(ggplot2)
7
8  pbirthdaysim <- function(n) {
9    ntests <- 100000
10   pop <- 1:365
11   anydup <- function(i)
12     any(duplicated(
13       sample(pop, n, replace=TRUE)))
14   sum(sapply(seq(ntests), anydup)) / ntests
15 }
16
17 benchmark.loops <- function(fun, n) {
18   bmark <- benchmark(
19     "lapply" = lapply(1:n, fun),
20     "parLapply HT" = {
21       cores <- detectCores()
22       cluster <- makePSOCKcluster(cores)
23       parLapply(cluster, 1:n, fun)
24       stopCluster(cluster)
25     },
26     "parLapply PC" = {
27       cores <- detectCores(logical = FALSE)
28       cluster <- makePSOCKcluster(cores)
29       parLapply(cluster, 1:n, fun)
30       stopCluster(cluster)
31     },
32     "mclapply HT" = {
```

```
      cores <- detectCores()
      mclapply(1:n, fun, mc.cores = cores)
    },
    "mclapply PC" = {
      cores <- detectCores(logical = FALSE)
      mclapply(1:n, fun, mc.cores = cores)
    },
    "foreach HT" = {
      cores <- parallel::detectCores()
      cluster <- parallel::makeCluster(cores)
      doParallel::registerDoParallel(cluster)
      foreach::foreach(i = 1:n) %dopar% fun(i)
      parallel::stopCluster(cluster)
    },
    "foreach PC" = {
      cores <- detectCores(logical = FALSE)
      cluster <- makeCluster(cores)
      registerDoParallel(cluster)
      foreach(i=1:n) %dopar% fun(i)
      stopCluster(cluster)
    },
    replications = 1,
    columns = c("test", "elapsed", "relative")
  )
  bmark
}

for (i in c(1, 50, 100)) {
  assign(
    paste("bmark", i, sep = "."),
    benchmark.loops(fun = pbirthdaysim, n = i) %>%
      mutate(n = i)
  )
}
```

3.1.2 Results

Naturally, the results indicate that for a single computation ($n = 1$), `lapply()` seems to be the most efficient. For a smaller amount of computations, it seems like a forking method is the most fitting. Interestingly, when n gets larger, the cluster methods seems to be more efficient. This could be because the processes does not copy unneeded data, and hence experience a smaller memory footprint. Overall, it seems like the functions from the `parallel` packages are superior, and we are not sure why. Nonetheless, the difference is negligible for our problem.

The results differed a bit between each run. Because we are too impatient, we only did one repetition of the benchmark. For more robust results, we should have done more repetitions, and compared the box plots.

Using hyper-threading was significantly faster, but far from the extent one would have think. However Overall, the results show that when the number of independent computations reaches some level, parallel computing can offer multiple magnitudes of computational savings.

n = 1	elapsed	relative
foreach HT	3.997	2.861
foreach PC	2.817	2.016
lapply	1.397	1.000
mclapply HT	1.730	1.238
mclapply PC	1.685	1.206
parLapply HT	3.421	2.449
parLapply PC	2.789	1.996
n = 1:10	elapsed	relative
foreach HT	6.655	1.262
foreach PC	6.141	1.165
lapply	15.566	2.952
mclapply HT	5.273	1.000
mclapply PC	5.467	1.037
parLapply HT	6.479	1.229
parLapply PC	6.765	1.283
n = 1:100	elapsed	relative
foreach HT	57.387	1.065
foreach PC	61.940	1.150
lapply	199.996	3.712
mclapply HT	61.722	1.146
mclapply PC	66.256	1.230
parLapply HT	53.881	1.000
parLapply PC	65.607	1.218

4 Conclusion

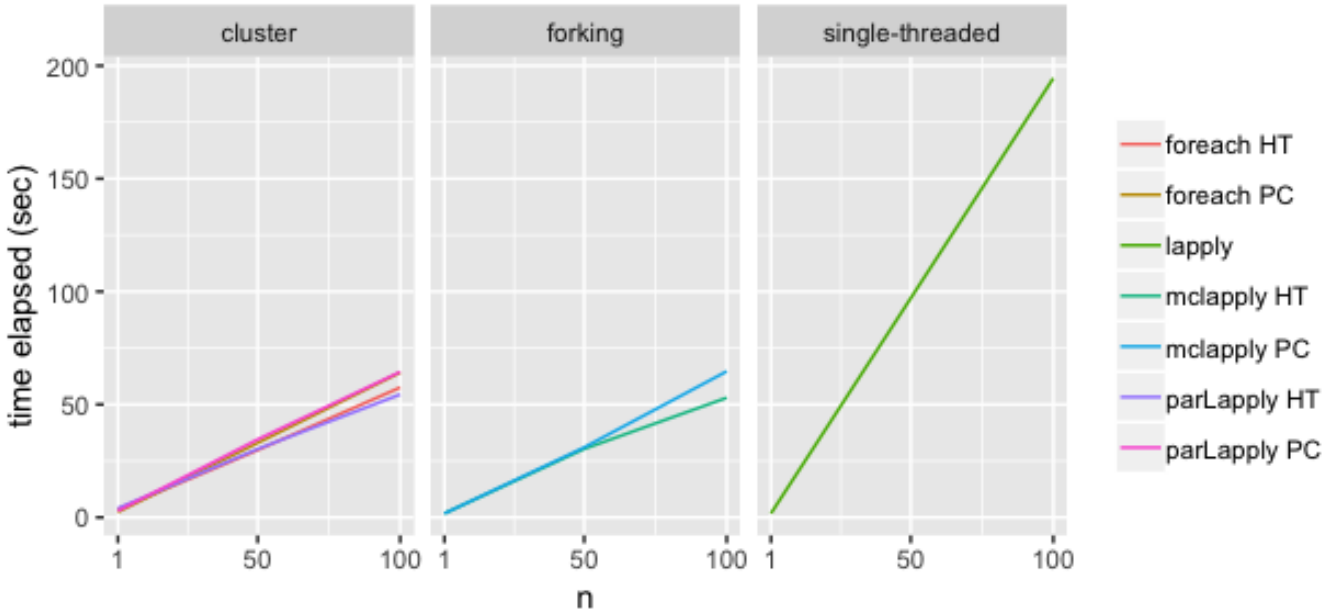

Figure 1: Comparison of parallel processing methods.

As we have seen, there are major increases in performance by using parallel processing. The implementations are also pretty straight forward, even using the base R functionality in the `parallel` package. It is easy to argue that this functionality should be handled internally in the R core when there are opportunities for parallel processing. In many basic operations and computations, one should be able to only specify the number of jobs to be dispatched, and R should then be able to process the request as fast as possible using the optimal resources available. This is however easier said than done, and is a major challenge for all programming languages.

When doing simple calculations on large sets of data we have seen large increases in performance, but problems will arise as soon as we would try to perform operations with side effects. From our perspective of being students studying economics and not software development, this is a problem best left for the developers, and we could rather exploit a final package once these become available.

References

Dursi, J. (2017). Beyond single-core r. https://ljdursi.github.io/beyond-single-core-R. (Accessed on 11/21/2018).

Errickson, J. (2017). Parallel processing in r. http://dept.stat.lsa.umich.edu/~jerrick/courses/stat701/notes/parallel.html. (Accessed on 11/21/2018).

Microsoft and Weston, S. (2017). Using the foreach package. https://cran.r-project.org/web/packages/foreach/vignettes/foreach.pdf. (Accessed on 11/21/2018).

R Core Team (2018). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria.

Smith, D. (2018). Speed up r with parallel programming in the cloud, european r users meeting, budapest 2018. https://www.slideshare.net/RevolutionAnalytics/speeding-up-r-with-parallel-programming-in-the-cloud. (Accessed on 11/21/2018).

Zhao, P. (2016). R with parallel computing from user perspectives. https://www.r-bloggers.com/r-with-parallel-computing-from-user-perspectives/. (Accessed on 11/21/2018).