G-FORGE

A BLOG ABOUT ORTHOPAEDIC SURGERY, R, RESEARCH AND MORE

How-to go parallel in R – basics + tips

DON'T WAIST ANOTHER SECOND, START PARALLELIZING YOUR COMPUTATIONS TODAY! THE IMAGE IS CC BY SMUDGE 9000

Today is a good day to start parallelizing your code. I've been using the parallel package since its integration with R (v. 2.14.0) and its much easier than it at first seems. In this post I'll go through the basics for implementing parallel computations in R, cover a few common pitfalls, and give tips on how to avoid them.

Contents [hide]

1 Learning lapply is key
2 The parallel package
  2.1 Variable scope
  2.2 Using parSapply
3 The foreach package
  3.1 Variable scope
4 Fork or sock?
  4.1 Memory handling
5 Debugging
  5.1 The tryCatch - list approach
  5.2 Creating a common output file
  5.3 Creating a node-specific file
  5.4 The partools package
6 Caching
7 Load balancing
  7.1 Work load
  7.2 Memory load
  7.3 Memory tips
8 Other tips
    8.0.1 Share this:

The common motivation behind parallel computing is that something is taking too long time. For me that means any computation that takes more than 3 minutes – this because parallelization is incredibly simple and most tasks that take time are embarrassingly parallel. Here are a few common tasks that fit the description:

- Bootstrapping
- Cross-validation
- Multivariate Imputation by Chained Equations (MICE)
- Fitting multiple regression models

Learning `lapply` is key

One thing I regret is not learning earlier `lapply` . The function is beautiful in its simplicity: It takes one parameter (a vector/list), feeds that variable into the function, and returns a list:

```
lapply(1:3, function(x) c(x, x^2, x^3))
```

```
[[1]]
 [1] 1 1 1

[[2]]
 [1] 2 4 8

[[3]]
 [1] 3 9 27
```

You can feed it additional values by adding named parameters:

```
lapply(1:3/3, round, digits=3)
```

```
[[1]]
[1] 0.333

[[2]]
[1] 0.667

[[3]]
[1] 1
```

The tasks are embarrassingly parallel as the elements are calculated independently, i.e. second element is independent of the result from the first element. After learning to code using `lapply` you will find that parallelizing your code is a breeze.

The `parallel` package

The `parallel` package is basically about doing the above in parallel. The main difference is that we need to start with setting up a cluster, a collection of "workers" that will be doing the job. A good number of clusters is the *numbers of available cores – 1*. I've found that using all 8 cores on my machine will prevent me from doing anything else (the computers comes to a standstill until the R task has finished). I therefore always set up the cluster as follows:

```
library(parallel)

# Calculate the number of cores
no_cores <- detectCores() - 1

# Initiate cluster
cl <- makeCluster(no_cores)
```

Now we just call the parallel version of `lapply`, `parLapply`:

```
parLapply(cl, 2:4,
          function(exponent)
            2^exponent)
```

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

[[2]]
[1] 8

[[3]]
[1] 16
```

Once we are done we need to close the cluster so that resources such as memory are returned to the operating system.

```
stopCluster(cl)
```

VARIABLE SCOPE

On Mac/Linux you have the option of using `makeCluster(no_core, type="FORK")` that automatically contains all environment variables (more details on this below). On Windows you have to use the *Parallel Socket Cluster* (PSOCK) that starts out with only the base packages loaded (note that PSOCK is default on *all* systems). You should therefore always specify exactly what variables and libraries that you need for the parallel function to work, e.g. the following fails:

```
cl<-makeCluster(no_cores)
base <- 2

parLapply(cl,
          2:4,
          function(exponent)
            base^exponent)

stopCluster(cl)
```

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

While this passes:

```
cl<-makeCluster(no_cores)

base <- 2
clusterExport(cl, "base")
parLapply(cl,
          2:4,
          function(exponent)
            base^exponent)

stopCluster(cl)
```

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

[[2]]
[1] 8

[[3]]
[1] 16
```

Note that you need the `clusterExport(cl, "base")` in order for the function to see the **base** variable. If you are using some special packages you will similarly need to load those through `clusterEvalQ`, e.g. I often use the `rms` package and I therefore use `clusterEvalQ(cl, library(rms))`. Note that any changes to the variable after `clusterExport` are ignored:

```
cl<-makeCluster(no_cores)
clusterExport(cl, "base")
base <- 4
# Run
parLapply(cl,
          2:4,
          function(exponent)
            base^exponent)

# Finish
stopCluster(cl)
```

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

[[2]]
[1] 8

[[3]]
[1] 16
```

USING `parSapply`

Sometimes we only want to return a simple value and directly get it processed as a vector/matrix. The `lapply` version that does this is called `sapply` , thus it is hardly surprising that its parallel version is `parSapply` :

```
parSapply(cl, 2:4,
          function(exponent)
            base^exponent)
```

```
[1]  4  8 16
```

Matrix output with names (this is why we need the `as.character`):

```
parSapply(cl, as.character(2:4),
          function(exponent){
            x <- as.numeric(exponent)
            c(base = base^x, self = x^x)
          })
```

```
     2  3   4
base 4  8  16
self 4 27 256
```

The `foreach` package

The idea behind the `foreach` package is to create *'a hybrid of the standard for loop and lapply function'* and its ease of use has made it rather popular. The set-up is slightly different, you need "register" the cluster as below:

```
library(foreach)
library(doParallel)

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

Note that you can change the last two lines to:

```
registerDoParallel(no_cores)
```

But then you need to remember to instead of `stopCluster()` at the end do:

```
stopImplicitCluster()
```

The `foreach` function can be viewed as being a more controlled version of the `parSapply` that allows combining the results into a suitable format. By specifying the `.combine` argument we can choose how to combine our results, below is a vector, matrix, and a list example:

```
foreach(exponent = 2:4,
        .combine = c)  %dopar%
  base^exponent
```

```
[1]  4  8 16
```

```
foreach(exponent = 2:4,
        .combine = rbind)  %dopar%
  base^exponent
```

```
         [,1]
result.1    4
result.2    8
result.3   16
```

```
foreach(exponent = 2:4,
        .combine = list,
        .multicombine = TRUE)  %dopar%
  base^exponent
```

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

[[2]]
[1] 8

[[3]]
[1] 16
```

Note that the last is the default and can be achieved without any tweaking, just `foreach(exponent = 2:4) %dopar%` . In the example it is worth noting the `.multicombine` argument that is needed to avoid a nested list. The nesting occurs due to the sequential `.combine` function calls, i.e. `list(list(result.1, result.2), result.3)` :

```
foreach(exponent = 2:4,
        .combine = list)  %dopar%
  base^exponent
```

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

[[1]][[2]]
[1] 8



[[2]]
[1] 16
```

VARIABLE SCOPE

The variable scope constraints are slightly different for the `foreach` package. Variable within the same *local* environment are by default available:

```
base <- 2
cl<-makeCluster(2)
registerDoParallel(cl)
foreach(exponent = 2:4,
        .combine = c)  %dopar%
  base^exponent
stopCluster(cl)
```

```
 [1]  4  8 16
```

While variables from a parent environment will not be available, i.e. the following will throw an error:

```
test <- function (exponent) {
  foreach(exponent = 2:4,
          .combine = c)  %dopar%
    base^exponent
}
test()
```

```
 Error in base^exponent : task 1 failed - "object 'base' not found"
```

A nice feature is that you can use the `.export` option instead of the `clusterExport`. Note that as it is part of the parallel call it will have the latest version of the variable, i.e. the following change in "base" will work:

```
base <- 2
cl<-makeCluster(2)
registerDoParallel(cl)

base <- 4
test <- function (exponent) {
  foreach(exponent = 2:4,
          .combine = c,
          .export = "base")  %dopar%
    base^exponent
}
test()

stopCluster(cl)
```

```
[1]  4  8 16
```

Similarly you can load packages with the `.packages` option, e.g. `.packages = c("rms", "mice")`. I strongly recommend always exporting the variables you need as it limits issues that arise when encapsulating the code within functions.

Fork or sock?

I do most of my analyses on Windows and have therefore gotten used to the PSOCK system. For those of you on other systems you should be aware of some important differences between the two main alternatives:

FORK: "to divide in branches and go separate ways"
Systems: Unix/Mac (not Windows)
Environment: Link all

PSOCK: Parallel Socket Cluster
Systems: All (including Windows)
Environment: Empty

MEMORY HANDLING

Unless you are using multiple computers or Windows or planning on sharing your code with someone using a Windows machine, you should try to use FORK (I use capitalized due to the `makeCluster` *type* argument). It is leaner on the memory usage by linking to the same address space. Below you can see that the memory address space for variables exported to PSOCK are not the same as the original:

```
library(pryr) # Used for memory analyses
cl<-makeCluster(no_cores)
clusterExport(cl, "a")
clusterEvalQ(cl, library(pryr))

parSapply(cl, X = 1:10, function(x) {address(a)}) == address(a)
```

```
[1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
```

While they are for FORK clusters:

```
cl<-makeCluster(no_cores, type="FORK")
parSapply(cl, X = 1:10, function(x) address(a)) == address(a)
```

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

This can save a lot of time during setup and also memory. Interestingly, you do not need to worry about variable corruption:

```
b <- 0
parSapply(cl, X = 1:10, function(x) {b <- b + 1; b})
# [1] 1 1 1 1 1 1 1 1 1 1
parSapply(cl, X = 1:10, function(x) {b <<- b + 1; b})
# [1] 1 2 3 4 5 1 2 3 4 5
b
# [1] 0
```

Debugging

Debugging is especially hard when working in a parallelized environment. You cannot simply call `browser` / `cat` / `print` in order to find out what the issue is.

THE `tryCatch` - `list` APPROACH

Using `stop()` for debugging without modification is generally a bad idea; while you will receive the error message, there is a large chance that you have forgotten about that `stop()`, and it gets evoked once you have run your software for a day or two. It is annoying to throw away all the previous successful computations just because one failed (yup, this is default behavior of all the above functions). You should therefore try to catch errors and return a text explaining the setting that caused the error:

```
foreach(x=list(1, 2, "a"))  %dopar%
{
  tryCatch({
    c(1/x, x, 2^x)
  }, error = function(e) return(paste0("The variable '", x, "'",
                                      " caused the error: '", e, "'")))
}
```

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

[[2]]
[1] 0.5 2.0 4.0

[[3]]
[1] "The variable 'a' caused the error: 'Error in 1/x: non-numeric argument to binary operator\n'"
```

This is also why I like lists, the `.combine` may look appealing but it is easy to manually apply and if you have function that crashes when one of the element is not of the expected type you will loose all your data. Here is a simple example of how to call `rbind` on a `lapply` output:

```
out <- lapply(1:3, function(x) c(x, 2^x, x^x))
do.call(rbind, out)
```

```
     [,1] [,2] [,3]
[1,]    1    2    1
[2,]    2    4    4
[3,]    3    8   27
```

CREATING A COMMON OUTPUT FILE

Since we can't have a console per worker we can set a shared file. I would say that this is a "last resort" solution:

```
cl<-makeCluster(no_cores, outfile = "debug.txt")
registerDoParallel(cl)
foreach(x=list(1, 2, "a"))  %dopar%
{
  print(x)
}
stopCluster(cl)
```

```
starting worker pid=7392 on localhost:11411 at 00:11:21.077
starting worker pid=7276 on localhost:11411 at 00:11:21.319
starting worker pid=7576 on localhost:11411 at 00:11:21.762
[1] 2]

[1] "a"
```

As you can see due to a race between first and the second node the output is a little garbled and therefore in my opinion less useful than returning a custom statement.

CREATING A NODE-SPECIFIC FILE

A perhaps slightly more appealing alternative is to a have a node-specific file. This could potentially be interesting when you have a dataset that is causing some issues and you want to have a closer look at that data set:

```
cl<-makeCluster(no_cores, outfile = "debug.txt")
registerDoParallel(cl)
foreach(x=list(1, 2, "a"))  %dopar%
{
  cat(dput(x), file = paste0("debug_file_", x, ".txt"))
}
stopCluster(cl)
```

A tip is to combine this with your `tryCatch - list` approach. Thereby you can extract any data that is not suitable for a simple message (e.g. a large data.frame), load that, and debug it without parallel. If the `x` is too long for a file name I suggest that you use digest as described below for the cache function.

THE `partools` PACKAGE

There is an interesting package `partools` that has a dbs() function that may be worth looking into (unless your on a Windows machine). It allows coupling terminals per process and debugging through them.

Caching

I strongly recommend implementing some caching when doing large computations. There may be a multitude of reasons to why you need to exit a computation and it would be a pity to waist all that valuable time. There is a package for caching, R.cache, but I've found it easier to write the function myself. All you need is the built-in `digest` package. By feeding the data + the function that you are using to the `digest()` you get an unique key, if that key matches your previous calculation there is no need for re-running that particular section. Here is a function with caching:

```
cacheParallel <- function(){
  vars <- 1:2
  tmp <- clusterEvalQ(cl,
                      library(digest))

  parSapply(cl, vars, function(var){
    fn <- function(a) a^2
    dg <- digest(list(fn, var))
    cache_fn <-
      sprintf("Cache_%s.Rdata",
              dg)
    if (file.exists(cache_fn)){
      load(cache_fn)
    }else{
      var <- fn(var);
      Sys.sleep(5)
      save(var, file = cache_fn)
    }
    return(var)
  })
}
```

The when running the code it is pretty obvious that the `Sys.sleep` is not invoked the second time around:

```
system.time(out <- cacheParallel())
# user system elapsed
# 0.003 0.001 5.079
out
# [1] 1 4
system.time(out <- cacheParallel())
# user system elapsed
# 0.001 0.004 0.046
out
# [1] 1 4

# To clean up the files just do:
file.remove(list.files(pattern = "Cache.+\\.Rdata"))
```

Load balancing

Balancing so that the cores have similar weight load and don't fight for memory resources is core for a successful parallelization scheme.

WORK LOAD

Note that the `parLapply` and `foreach` are wrapper functions. This means that they are not directly doing the processing the parallel code, but rely on other functions for this. In the `parLapply` the function is defined as:

```
parLapply <- function (cl = NULL, X, fun, ...)
{
    cl <- defaultCluster(cl)
    do.call(c, clusterApply(cl, x = splitList(X, length(cl)),
        fun = lapply, fun, ...), quote = TRUE)
}
```

Note the `splitList(X, length(cl))` . This will split the tasks into even portions and send them onto the workers. If you have many of those cached or there is a big computational difference between the tasks you risk ending up with only one cluster actually working while the others are inactive. To avoid this you should when caching try to remove those that are cached from the X or try to mix everything into an even workload. E.g. if we want to find optimal number of neurons in a neural network we may want to change:

```
# From the nnet example
parLapply(cl, c(10, 20, 30, 40, 50), function(neurons)
  nnet(ir[samp,], targets[samp,],
       size = neurons))
```

to:

```
# From the nnet example
parLapply(cl, c(10, 50, 30, 40, 20), function(neurons)
  nnet(ir[samp,], targets[samp,],
       size = neurons))
```

MEMORY LOAD

Running large datasets in parallel can quickly get you into trouble. If you run out of memory the system will either crash or run incredibly slow. The former happens to me on Linux systems while the latter is quite common on Windows systems. You should therefore always monitor your parallelization to make sure that you aren't too close to the memory ceiling.

Using FORKs is an important tool for handling memory ceilings. As they link to the original variable address the fork will not require any time for exporting variables or take up any additional space when using these. The impact on performance can be significant (my system has 16Gb of memory and eight cores):

```
> rm(list=ls())
> library(pryr)
> library(magrittr)
> a <- matrix(1, ncol=10^4*2, nrow=10^4)
> object_size(a)
1.6 GB
> system.time(mean(a))
   user  system elapsed
  0.338   0.000   0.337
> system.time(mean(a + 1))
   user  system elapsed
  0.490   0.084   0.574
> library(parallel)
> cl <- makeCluster(4, type = "PSOCK")
> system.time(clusterExport(cl, "a"))
   user  system elapsed
  5.253   0.544   7.289
> system.time(parSapply(cl, 1:8,
                        function(x) mean(a + 1)))
   user  system elapsed
  0.008   0.008   3.365
> stopCluster(cl); gc();
> cl <- makeCluster(4, type = "FORK")
> system.time(parSapply(cl, 1:8,
                        function(x) mean(a + 1)))
   user  system elapsed
  0.009   0.008   3.123
> stopCluster(cl)
```

FORKs can also make your able to run code in parallel that otherwise crashes:

```
> cl <- makeCluster(8, type = "PSOCK")
> system.time(clusterExport(cl, "a"))
   user  system elapsed
 10.576   1.263  15.877
> system.time(parSapply(cl, 1:8, function(x) mean(a + 1)))
Error in checkForRemoteErrors(val) :
  8 nodes produced errors; first error: cannot allocate vector of size 1.5 Gb
Timing stopped at: 0.004 0 0.389
> stopCluster(cl)
> cl <- makeCluster(8, type = "FORK")
> system.time(parSapply(cl, 1:8, function(x) mean(a + 1)))
   user  system elapsed
  0.014   0.016   3.735
> stopCluster(cl)
```

Although, it won't save you from yourself 😬 as you can see below when we create an intermediate variable that takes up storage space:

```
> a <- matrix(1, ncol=10^4*2.1, nrow=10^4)
> cl <- makeCluster(8, type = "FORK")
> parSapply(cl, 1:8, function(x) {
+   b <- a + 1
+   mean(b)
+   })
Error in unserialize(node$con) : error reading from connection
```

MEMORY TIPS

- Frequently use `rm()` in order to avoid having unused variables around
- Frequently call the garbage collector `gc()` . Although this should be implemented automatically in R, I've found that while it may releases the memory locally it may not return it to the operating system (OS). This makes sense when running at a single instance as this is an time expensive procedure but if you have multiple processes this may not be a good strategy. Each process needs to get their memory from the OS and it is therefore vital that each process returns memory once they no longer need it.
- Although it is often better to parallelize at a large scale due to initialization costs it may in memory situations be better to parallelize at a small scale, i.e. in subroutines.

- I sometimes run code in parallel, cache the results, and once I reach the limit I change to sequential.
- You can also manually limit the number of cores, using all the cores is of no use if the memory isn't large enough. A simple way to think of it is: `memory.limit()/memory.size() = max cores`

Other tips

- A general core detector function that I often use is:

```
max(1, detectCores() - 1)
```

- **Never use** `set.seed()` , use `clusterSetRNGStream()` instead, to set the cluster seed if you want reproducible results
- If you have a Nvidia GPU-card, you can get huge gains from micro-parallelization through the `gputools` package (Warning though, the installation can be rather difficult…).
- When using `mice` in parallel remember to use `ibind()` for combining the imputations.

15 RESPONSES TO *HOW-TO GO PARALLEL IN R – BASICS + TIPS*

Markus Senn *says:*
FEBRUARY 26, 2015 AT 17:18

Hi Max,

Thanks again for your presentation at the Meetup.

Just to let you know: you convinced me. I started using parallel in R and I'm liking it! This blog post is really helpful as a reference.

Markus

REPLY

Shaun Ryan *says:*
MARCH 17, 2015 AT 13:55

Hi.

I have 4 cores. I set it to 3 but it still used all 4 (I can send the screen shot if no one believes me). I looked at the Windows Task Manager and can clearly see all 4 cores spiking due to the R call.
Am I misunderstanding something?

Thanks
Shaun

REPLY

Shaun Ryan *says:*
MARCH 17, 2015 AT 14:02

having looked into this further. Altering the core number has some effect but it doesn't look like a clean affinity partitioning. I think it's because they're logical cores maybe and not physical cores and therefore there might be some lower level affinity masking going on to decide where to run higher parallel stuff. If I switch it to 1 it definitely only runs on 1 core and it's comparatively very slow (I tuned up the work load a little than in your example)

REPLY

Henrik Bengtsson *says:*
JANUARY 21, 2016 AT 01:43

Note that the main/original R process uses one core/process. Then the ones you setup up with makeCluster() uses *additional* cores/processes. Same is true for mcapply() and mc.cores, cf. https://twitter.com/henrikbengtsson/status/686276062095904769

REPLY

jn *says:*
MAY 2, 2015 AT 19:20

Excellent post! Thank you for your great work.

REPLY

Paul Johnson *says:*
MAY 3, 2015 AT 23:59

This is very helpful. I've written notes like this for classes, but never as good as this set. Way to go.

REPLY

Jessica *says:*

MAY 4, 2015 AT 16:34

I am a relatively novice R user, and absolute novice at any sort of parallel computing. I am desperately trying to figure out how to use an extant r package (specifically I'm using LQMM) in parallel mode. I see lots of examples where you can write your own functions, but I have yet to find an example of combining parallel and other packages. Any advice?

REPLY

Max Gordon *says:*
JULY 24, 2015 AT 22:52

Unfortunately it is difficult to go parallel without changing the core code – you will most likely need to change the source and adapt those parts suitable for parallel execution. I would recommend avoiding this path if your new to R.

PS. Sorry for the late answer.

REPLY

Sérgio Casca *says:*
OCTOBER 6, 2015 AT 09:52

Great post! I've just started using parallel computing in R and your post has been really helpful.

Thanks,
Sérgio

REPLY

Juanmi *says:*
JANUARY 20, 2016 AT 09:09

Hello, very didactic and useful post. Thank you!!!

REPLY

Chris Kennedy *says:*

FEBRUARY 12, 2016 AT 22:04

Great stuff. One extra tip is that when you makeCluster(), if you specify outfile="" you will be able to see the printed output from parallel workers without needing a separate text file – it will go right to the console. There is an example here: http://stackoverflow.com/a/35329215/2463900

REPLY

PINGBACK: *PARALLEL PROGRAMMING IN R | LONEHAROON*

PINGBACK: *R WITH PARALLEL COMPUTING FROM USER PERSPECTIVES – PARALLELR*

PINGBACK: *BETTER CONFIDENCE INTERVALS FOR QUANTILES – MUBASHIR QASIM*

PINGBACK: *HOW TO MAKE R USE ALL PROCESSORS? - EXCEPTIONSHUB*