A Simple Guide to the Parallel Package in R

Joshua D. Ingram

1/6/2020

An Introduction to the Parallel Package

Performing large amounts of computations consumes lots of time and energy. Learning how to efficiently process data is valuable and the `Parallel` package in R allows us to do just that. For CPU-bound computations, this package allows us to use the multi-core processors in our computers. Rather than sequentially taking on tasks, the processor can take on multiple tasks at once by distributing them asmongst the cores. This allows for simultaneous computations, which can lead to shorter processing times.

System Specs

The first step to parallel processing and utilizing the `Parallel` package is looking at your system specs. Use the `detectCores` function as demonstrated below:

```r
detectCores()
```

```
## [1] 16
```

The machine being used for demonstration is an 8-core AMD processor, but the `detectCores` function outputs 16. This is because the function has a logical parameter `logical`. When set to `FALSE`, the function will return the number of physical cores. When set to `TRUE`, the default setting, the function returns the number of threads. In many cases, a CPU has multiple cores and each core has multiple threads.

Number of physical cores:

```r
detectCores(logical = FALSE)
```

```
## [1] 8
```

Number of threads:

```r
detectCores(logical = TRUE)
```

```
## [1] 16
```

For simplicity's sake, think of threads as "logical" cores. Meaning each thread will be able to act as its own core and take on a task. In our case, this system has a CPU with 8 physical cores and 16 threads. So 2 threads per core. You'll want to take note of this as you move forward with the package.

Using the Package for Parallel Processing

Now that we learned a little bit about the usefulness of parallel processing and our system, let's get started in R.

mclapply

`mclapply` is very similar to `lapply`, except that it allows us to distribute the tasks to all of the cores.

Setup:

```r
library(parallel)

rows <- seq(1,1250)
columns <- seq(1, 1250)
funct <- function(n.rows, n.cols){
  new.matrix <- matrix(nrow = n.rows, ncol = n.rows)
}

num.cores <- detectCores(logical = TRUE)
```

Using lapply:

```r
system.time(
  matrices <- lapply(c(rows, columns), funct)
)
```

```
##    user  system elapsed
##    1.12    0.43    1.56
```

Using mclapply:

```r
system.time(
  matrices <- mclapply(c(rows, columns), funct, mc.cores = 1)
)
```

```
##    user  system elapsed
##    0.51    0.57    1.08
```

Note: Using `mclapply` will have no use on Windows systems, as mc.cores $>$ 1 is not supported. The system used for demonstration is a Windows system, so mclapply is here for syntax, but is not being utilized properly. You should set `mc.cores = num.cores`.

parLapply

Considering Windows does not support mclapply, there is a more complicated approach to parallel processing. This is how to get started:

Start a Cluster

Use the function `makeCluster` to start a cluster with n nodes. n should be the number of cores to be used.

```r
# set to 4 so the output isn't too large
num.cores <- detectCores(logical = TRUE) - 12

clust <- makeCluster(num.cores)
```

Pre-processing Setup

With parLapply, the socket approach is generally used, as opposed to forking (used by mclapply). Due to this, some pre-processing setup is necessary becuase of the nature of each process. Rather than using existing variables and packages in a session, each of these must be moved into every process.

`clusterEvalQ` is a general way to do this. It has two inputs, the cluster and an expression:

```r
clusterEvalQ(clust, 3*3)
```

```
## [[1]]
## [1] 9
##
## [[2]]
## [1] 9
##
## [[3]]
## [1] 9
##
## [[4]]
## [1] 9
```

Since there is no inheritance, a variable assignment can be made within the function, but cannot be used in the main process:

```r
clusterEvalQ(clust, a <- 1)
```

```
## [[1]]
## [1] 1
##
## [[2]]
## [1] 1
##
## [[3]]
## [1] 1
##
## [[4]]
## [1] 1
```

`clusterEvalQ` can also be used to load packages:

```r
clusterEvalQ(clust, {library(RColorBrewer)})
```

```
## [[1]]
## [1] "RColorBrewer" "stats"        "graphics"     "grDevices"    "utils"
## [6] "datasets"     "methods"      "base"
```

```
##
## [[2]]
## [1] "RColorBrewer" "stats"        "graphics"     "grDevices"    "utils"
## [6] "datasets"     "methods"      "base"
##
## [[3]]
## [1] "RColorBrewer" "stats"        "graphics"     "grDevices"    "utils"
## [6] "datasets"     "methods"      "base"
##
## [[4]]
## [1] "RColorBrewer" "stats"        "graphics"     "grDevices"    "utils"
## [6] "datasets"     "methods"      "base"
```

`clusterExport` is a more robust way for pre-processing setup. Variables can be passed to the processes this way:

```
a <- 1
b <- 2
c <- 3
d <- c("a", "b", "c")
clusterExport(clust, d)
clusterEvalQ(clust, c(a,b))
```

```
## [[1]]
## [1] 1 2
##
## [[2]]
## [1] 1 2
##
## [[3]]
## [1] 1 2
##
## [[4]]
## [1] 1 2
```

Note: The second argument is a vector of strings that are the names of the variables to be passed, not the variables themselves.

parLapply and More

`Parallel` has its equivalents to `apply`, `lapply`, and `sapply`, being `parApply`, `ParLapply`, and `parSapply`, respectively. The only difference is that there is an argument that calls the cluster being used.

Example:

```
rows <- seq(1,1250)
columns <- seq(1, 1250)
vars <- c("rows", "columns")

num.cores <- detectCores(logical = TRUE)
clust <- makeCluster(num.cores)

funct <- function(n.rows, n.cols){
  new.matrix <- matrix(nrow = n.rows, ncol = n.rows)
}

clusterExport(clust, vars)

system.time(
  matrices <- parLapply(clust, c(rows, columns), funct)
)
```

```
##    user  system elapsed
##    3.66    7.00   10.92
```

Note: In the example with lapply, the timing was much shorter for the same function. This occurs because there is additional overhead and pre-processing setup with the socket approach. It's important to be able to determine when the socket approach is better than the forking (if you're not on a Windows system) and regular approaches.

Cluster Closing

Closing the cluster is a good habit to have, but not always necessary. Use `stopCluster` to do so:

```
stopCluster(clust)
```

Note: closing a cluster will result in anything saved there being lost, as well as packages needing to be re-loaded. The cluster object will still exist.

Overview of the Socket Approach

1. Create a cluster
2. Pre-processing setup
3. parApply, parLapply, or ParSapply
4. Close the cluster