Chapter 38: Parallel processing

Section 38.1: Parallel processing with parallel package

The base package `parallel` allows parallel computation through forking, sockets, and random-number generation.

Detect the number of cores present on the localhost:

```
parallel::detectCores(all.tests = FALSE, logical = TRUE)
```

Create a cluster of the cores on the localhost:

```
parallelCluster <- parallel::makeCluster(parallel::detectCores())
```

First, a function appropriate for parallelization must be created. Consider the **`mtcars`** dataset. A regression on `mpg` could be improved by creating a separate regression model for each level of `cyl`.

```
data <- mtcars
yfactor <- 'cyl'
zlevels <- sort(unique(data[[yfactor]]))
datay <- data[,1]
dataz <- data[,2]
datax <- data[,3:11]


fitmodel <- function(zlevel, datax, datay, dataz) {
  glm.fit(x = datax[dataz == zlevel,], y = datay[dataz == zlevel])
}
```

Create a function that can loop through all the possible iterations of `zlevels`. This is still in serial, but is an important step as it determines the exact process that will be parallelized.

```
fitmodel <- function(zlevel, datax, datay, dataz) {
  glm.fit(x = datax[dataz == zlevel,], y = datay[dataz == zlevel])
}


for (zlevel in zlevels) {
  print("*****")
  print(zlevel)
  print(fitmodel(zlevel, datax, datay, dataz))
}
```

Curry this function:

```
worker <- function(zlevel) {
    fitmodel(zlevel,datax, datay, dataz)
  }
```

Parallel computing using `parallel` cannot access the global environment. Luckily, each function creates a local environment `parallel` can access. Creation of a wrapper function allows for parallelization. The function to be applied also needs to be placed within the environment.

```
wrapper <- function(datax, datay, dataz) {
  # force evaluation of all parameters not supplied by parallelization apply
  force(datax)
```

```
  force(datay)
  force(dataz)
  # these variables are now in an enviroment accessible by parallel function

  # function to be applied also in the environment
  fitmodel <- function(zlevel, datax, datay, dataz) {
    glm.fit(x = datax[dataz == zlevel,], y = datay[dataz == zlevel])
  }

  # calling in this environment iterating over single parameter zlevel
  worker <- function(zlevel) {
    fitmodel(zlevel,datax, datay, dataz)
  }
  return(worker)
}
```

Now create a cluster and run the wrapper function.

```
parallelcluster <- parallel::makeCluster(parallel::detectCores())
models <- parallel::parLapply(parallelcluster,zlevels,
                              wrapper(datax, datay, dataz))
```

Always stop the cluster when finished.

```
parallel::stopCluster(parallelcluster)
```

The `parallel` package includes the entire **`apply()`** family, prefixed with **`par`**.

Section 38.2: Parallel processing with foreach package

The `foreach` package brings the power of parallel processing to R. But before you want to use multi core CPUs you have to assign a multi core cluster. The `doSNOW` package is one possibility.

A simple use of the foreach loop is to calculate the sum of the square root and the square of all numbers from 1 to 100000.

```
library(foreach)
library(doSNOW)

cl <- makeCluster(5, type = "SOCK")
registerDoSNOW(cl)

f <- foreach(i = 1:100000, .combine = c, .inorder = F) %dopar% {
    k <- i ** 2 + sqrt(i)
    k
}
```

The structure of the output of `foreach` is controlled by the `.combine` argument. The default output structure is a list. In the code above, `c` is used to return a vector instead. Note that a calculation function (or operator) such as `"+"` may also be used to perform a calculation and return a further processed object.

It is important to mention that the result of each foreach-loop is the last call. Thus, in this example `k` will be added to the result.

Parameter	Details
.combine	combine Function. Determines how the results of the loop are combined. Possible values are `c`, **`cbind`**, **`rbind`**, `"+"`, `"*"`...

.inorder	if `TRUE` the result is ordered according to the order of the iteration vairable (here `i`). If `FALSE` the result is not ordered. This can have postive effects on computation time.
.packages	for functions which are provided by any package except `base`, like e.g. `mass`, `randomForest` or else, you have to provide these packages with `c("mass", "randomForest")`

Section 38.3: Random Number Generation

A major problem with parallelization is the used of RNG as seeds. Random numbers by the number are iterated by the number of operations from either the start of the session or the most recent `set.seed()`. Since parallel processes arise from the same function, it can use the same seed, possibly causing identical results! Calls will run in serial on the different cores, provide no advantage.

A set of seeds must be generated and sent to each parallel process. This is automatically done in some packages (`parallel`, `snow`, etc.), but must be explicitly addressed in others.

```
s <- seed
for (i in 1:numofcores) {
    s <- nextRNGStream(s)
    # send s to worker i as .Random.seed
}
```

Seeds can be also be set for reproducibility.

```
clusterSetRNGStream(cl = parallelcluster, iseed)
```

Section 38.4: mcparallelDo

The `mcparallelDo` package allows for the evaluation of R code asynchronously on Unix-alike (e.g. Linux and MacOSX) operating systems. The underlying philosophy of the package is aligned with the needs of exploratory data analysis rather than coding. For coding asynchrony, consider the `future` package.

Example

Create data

```
data(ToothGrowth)
```

Trigger mcparallelDo to perform analysis on a fork

```
mcparallelDo({glm(len ~ supp * dose, data=ToothGrowth)},"interactionPredictorModel")
```

Do other things, e.g.

```
binaryPredictorModel <- glm(len ~ supp, data=ToothGrowth)
gaussianPredictorModel <- glm(len ~ dose, data=ToothGrowth)
```

The result from mcparallelDo returns in your targetEnvironment, e.g. .GlobalEnv, when it is complete with a message (by default)

```
summary(interactionPredictorModel)
```

Other Examples

```
# Example of not returning a value until we return to the top level
for (i in 1:10) {
  if (i == 1) {
```

```
    mcparallelDo({2+2}, targetValue = "output")
  }
  if (exists("output")) print(i)
}

# Example of getting a value without returning to the top level
for (i in 1:10) {
  if (i == 1) {
    mcparallelDo({2+2}, targetValue = "output")
  }
  mcparallelDoCheck()
  if (exists("output")) print(i)
}
```