Parallel R

Andrew Vaughn

18 October, 2021

Many tasks that are computationally expensive are embarrassingly parallel. A few common tasks that fit the description:

- Simulations with independent replicates
- Bootstrapping
- Cross-validation
- Multivariate Imputation by Chained Equations (MICE)
- Fitting multiple regression models
- cross-validation

`lapply` Refresher

`lapply` 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
```

These 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` parallelizing your code is simple.

Table 1: Future Resolution Strategies

Name	OS	Description
synchronous		*non-parallel*
sequential	all	sequentially in current R process
transparent	all	as sequential w/ early signaling and w/out local
asynchronous		*parallel*
multiprocess	all	multicore if possible, multisession otherwise
multisession	all	background R sessions (current machine)
multicore	not Windows/Rstudio	forked process
cluster	all	external R session, current or local machines
remote	all	remote R sessions

`future` Package

The future package attempts to provide a simple and uniform framework for evaluating expressions on various resources.

```
library(future)

v %<-% {
  cat("Hello world!\n")
  3.14
}

v
```

```
## Hello world!
```

```
## [1] 3.14
```

Notice, the definition is evaluated when the variable is called, instead of when the variable is defined. There are several methods for controlling how futures are evaluated. The types of futures are listed in 1, and implemented as `plan()`s.

Futures can also be evaluated *asynchronously* in a different R process by setting the `plan()` to one of the *asynchronous* options.

```
plan(strategy = multiprocess)
```

```
## Warning: Strategy 'multiprocess' is deprecated in future (>= 1.20.0). Instead,
## explicitly specify either 'multisession' or 'multicore'. In the current R
## session, 'multiprocess' equals 'multisession'.
```

```
## Warning in supportsMulticoreAndRStudio(...): [ONE-TIME WARNING] Forked
## processing ('multicore') is not supported when running R from RStudio
## because it is considered unstable. For more details, how to control forked
## processing or not, and how to silence this warning in future R sessions, see ?
## parallelly::supportsMulticore
```

```r
w %<-% {
  cat("Hello world!\n")
  3.14
}

w
```

```
## Hello world!
```

```
## [1] 3.14
```

The `future` package is extended by the `future.apply` package, which implements the *apply* family of functions from base `R`.

```r
library(future.apply)

# set evaluation plan
# explicit about namespace so you know where these functions are coming from.
nCores = 2
future::plan(strategy = multiprocess, workers = nCores)

future.apply::future_sapply(X = 2:4, FUN = function(exponent){2^exponent})
```

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

Variable Scope

On Mac/Linux, you can set the `plan` to be explicitly `multicore` (this is what `multiprocess` defaults to on Mac/Linux). This creates forked processes, which use the current environmental variables. On Windows, you can set `multisession`, which is the Windows default of `multiprocess`, which creates background `R` sessions. This requires copying all necessary variables to the processes.

The `future` package attempts to handle most of this for you, using the globals package. See below, where the `plan` is explicitly `multisession`, meaning that it should fail because I didn't explicitly copy `base` to each process. However, `future` identifies that `base` is necessary and supplies it to each child process.
It provides a similar service for functions/objects in packages, except that packages are attached to the child process, so no copying is necessary. See the vignette on globals for the `future` package.

```r
nCores = 2
future::plan(strategy = multisession, workers = nCores)
base = 2

# should fail, but doesn't
future.apply::future_sapply(X = 2:4, FUN = function(exponent){base^exponent})
```

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

```r
# safer way
globals = "base"
future.apply::future_sapply(X = 2:4, FUN = function(exponent){base^exponent})
```

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

Using `future_sapply`

(I DO NOT RECOMMEND SAPPLY STATEMENTS)
Sometimes, we only want a simple return value, such as a vector/matrix. Here are a few examples using the `future_sapply` function.

```r
# setup plan
nCores = 2
future::plan(strategy = multisession, workers = nCores)

# setup base and name globals
#  remember, the globals thing is not necessary
base = 3
globals = "base"

# sapply
future.apply::future_sapply(X = 2:4, FUN = function(x){base^x})
```

```
## [1]  9 27 81
```

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

```r
future.apply::future_sapply(X = as.character(2:4), FUN = function(x){
  x <- as.numeric(x)
  c("base" = base^x, "self" = x^x)
})
```

```
##      2  3   4
## base 9 27  81
## self 4 27 256
```

The `foreach` Package via `doFuture`

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 the plan as below:

```r
#library(foreach)
library(doFuture)
```

```
## Loading required package: foreach
```

```r
nCores = 2
plan(strategy = multiprocess, workers = nCores)
registerDoFuture()
```

The `foreach` function can be viewed as being a more controlled version of the `future_sapply` 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]  9 27 81
```

Now using `rbind`.

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

```
##          [,1]
## result.1    9
## result.2   27
## result.3   81
```

Now a list.

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

```
## [[1]]
## [1] 9
##
## [[2]]
## [1] 27
##
## [[3]]
## [1] 81
```

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,
        .multicombine = FALSE)  %dopar% {
          base^exponent
        }
```

```
## [[1]]
## [[1]][[1]]
## [1] 9
##
```

```
## [[1]][[2]]
## [1] 27
##
##
## [[2]]
## [1] 81
```

Variable Scope

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

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

```
## [1]  9 27 81
```

While variables from a parent environment should not be available, i.e. the following will throw an error using a different parallel backend. However, `future` ensures that all globals necessary inside the function are available.

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

test()
```

```
## [1]  9 27 81
```

A nice feature is that you can use the `.export` option within `foreach` to ensure variables are exported to child processes. 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 <- 4

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

test()
```

```
## [1]  16  64 256
```

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.

Now move on to `scfOverview.pdf`