Using The `foreach` Package

Steve Weston

November 26, 2018

1 Introduction

One of R's most useful features is its interactive interpreter. This makes it very easy to learn and experiment with R. It allows you to use R like a calculator to perform arithmetic operations, display data sets, generate plots, and create models.

Before too long, new R users will find a need to perform some operation repeatedly. Perhaps they want to run a simulation repeatedly in order to find the distribution of the results. Perhaps they need to execute a function with a variety a different arguments passed to it. Or maybe they need to create a model for many different data sets.

Repeated executions can be done manually, but it becomes quite tedious to execute repeated operations, even with the use of command line editing. Fortunately, R is much more than an interactive calculator. It has its own built-in language that is intended to automate tedious tasks, such as repeatedly executing R calculations.

R comes with various looping constructs that solve this problem. The `for` loop is one of the more common looping constructs, but the `repeat` and `while` statements are also quite useful. In addition, there is the family of "apply" functions, which includes `apply`, `lapply`, `sapply`, `eapply`, `mapply`, `rapply`, and others.

The `foreach` package provides a new looping construct for executing R code repeatedly. With the bewildering variety of existing looping constructs, you may doubt that there is a need for yet another construct. The main reason for using the `foreach` package is that it supports *parallel execution*, that is, it can execute those repeated operations on multiple processors/cores on your computer, or on multiple nodes of a cluster. If each operation takes over a minute, and you want to execute it hundreds of times, the overall runtime can take hours. But using `foreach`, that operation can be executed in parallel on hundreds of processors on a cluster, reducing the execution time back down to minutes.

But parallel execution is not the only reason for using the `foreach` package. There are other reasons that you might choose to use it to execute quick executing operations, as we will see later in the document.

2 Getting Started

Let's take a look at a simple example use of the `foreach` package. Assuming that you have the `foreach` package installed, you first need to load it:

```
> library(foreach)
```

Note that all of the packages that `foreach` depends on will be loaded as well.

Now I can use `foreach` to execute the `sqrt` function repeatedly, passing it the values 1 through 3, and returning the results in a list, called `x`[1]:

```
> x <- foreach(i=1:3) %do% sqrt(i)
> x

[[1]]
[1] 1

[[2]]
[1] 1.414214

[[3]]
[1] 1.732051
```

This is a bit odd looking, because it looks vaguely like a `for` loop, but is implemented using a binary operator, called `%do%`. Also, unlike a `for` loop, it returns a value. This is quite important. The purpose of this statement is to compute the list of results. Generally, `foreach` with `%do%` is used to execute an R expression repeatedly, and return the results in some data structure or object, which is a list by default.

You will note in the previous example that we used a variable `i` as the argument to the `sqrt` function. We specified the values of the `i` variable using a named argument to the `foreach` function. We could have called that variable anything we wanted, for example, `a`, or `b`. We could also specify other variables to be used in the R expression, as in the following example:

```
> x <- foreach(a=1:3, b=rep(10, 3)) %do% (a + b)
> x
```

[1]Of course, `sqrt` is a vectorized function, so you would never really do this. But later, we'll see how to take advantage of vectorized functions with `foreach`.

```
[[1]]
[1] 11

[[2]]
[1] 12

[[3]]
[1] 13
```

Note that parentheses are needed here. We can also use braces:

```
> x <- foreach(a=1:3, b=rep(10, 3)) %do% {
+   a + b
+ }
> x

[[1]]
[1] 11

[[2]]
[1] 12

[[3]]
[1] 13
```

We call `a` and `b` the *iteration variables*, since those are the variables that are changing during the multiple executions. Note that we are iterating over them in parallel, that is, they are both changing at the same time. In this case, the same number of values are being specified for both iteration variables, but that need not be the case. If we only supplied two values for `b`, the result would be a list of length two, even if we specified a thousand values for `a`:

```
> x <- foreach(a=1:1000, b=rep(10, 2)) %do% {
+   a + b
+ }
> x

[[1]]
[1] 11

[[2]]
[1] 12
```

Note that you can put multiple statements between the braces, and you can use assignment statements to save intermediate values of computations. However, if you use an assignment as a way of communicating between the different executions of your loop, then your code won't work correctly in parallel, which we will discuss later.

3 The .combine Option

So far, all of our examples have returned a list of results. This is a good default, since a list can contain any R object. But sometimes we'd like the results to be returned in a numeric vector, for example. This can be done by using the `.combine` option to `foreach`:

```
> x <- foreach(i=1:3, .combine='c') %do% exp(i)
> x
```

```
[1]  2.718282  7.389056 20.085537
```

The result is returned as a numeric vector, because the standard R `c` function is being used to concatenate all the results. Since the `exp` function returns numeric values, concatenating them with the `c` function will result in a numeric vector of length three.

What if the R expression returns a vector, and we want to combine those vectors into a matrix? One way to do that is with the `cbind` function:

```
> x <- foreach(i=1:4, .combine='cbind') %do% rnorm(4)
> x
```

```
        result.1    result.2   result.3   result.4
[1,] -0.34897230  1.49356996 -0.5648695  1.4066929
[2,]  0.64779160  0.56157614 -1.2031425  2.3621397
[3,] -1.08359296 -0.06590355  0.7542198  0.5585404
[4,] -0.05305822 -0.37430178 -0.1567882 -0.1441656
```

This generates four vectors of four random numbers, and combines them by column to produce a 4 by 4 matrix.

We can also use the `"+"` or `"*"` functions to combine our results:

```
> x <- foreach(i=1:4, .combine='+') %do% rnorm(4)
> x
```

```
[1]  1.025751  1.450846 -2.931059 -2.781771
```

You can also specify a user-written function to combine the results. Here's an example that throws away the results:

```
> cfun <- function(a, b) NULL
> x <- foreach(i=1:4, .combine='cfun') %do% rnorm(4)
> x

NULL
```

Note that this `cfun` function takes two arguments. The `foreach` function knows that the functions `c`, `cbind`, and `rbind` take many arguments, and will call them with up to 100 arguments (by default) in order to improve performance. But if any other function is specified (such as `"+"`), it assumes that it only takes two arguments. If the function does allow many arguments, you can specify that using the `.multicombine` argument:

```
> cfun <- function(...) NULL
> x <- foreach(i=1:4, .combine='cfun', .multicombine=TRUE) %do% rnorm(4)
> x

NULL
```

If you want the combine function to be called with no more than 10 arguments, you can specify that using the `.maxcombine` option:

```
> cfun <- function(...) NULL
> x <- foreach(i=1:4, .combine='cfun', .multicombine=TRUE, .maxcombine=10) %do% rnorm(4)
> x

NULL
```

The `.inorder` option is used to specify whether the order in which the arguments are combined is important. The default value is `TRUE`, but if the combine function is `"+"`, you could specify `.inorder` to be `FALSE`. Actually, this option is important only when executing the R expression in parallel, since results are always computed in order when running sequentially. This is not necessarily true when executing in parallel, however. In fact, if the expressions take very different lengths of time to execute, the results could be returned in any order. Here's a contrived example, that executes the tasks in parallel to demonstrate the difference. The example uses the `Sys.sleep` function to cause the earlier tasks to take longer to execute:

```
> foreach(i=4:1, .combine='c') %dopar% {
+   Sys.sleep(3 * i)
+   i
+ }

[1] 4 3 2 1

> foreach(i=4:1, .combine='c', .inorder=FALSE) %dopar% {
+   Sys.sleep(3 * i)
+   i
+ }

[1] 4 3 2 1
```

The results of the first of these two examples is guaranteed to be the vector c(4, 3, 2, 1). The second example will return the same values, but they will probably be in a different order.

4 Iterators

The values for the iteration variables don't have to be specified with only vectors or lists. They can be specified with an *iterator*, many of which come with the `iterators` package. An iterator is an abstract source of data. A vector isn't itself an iterator, but the `foreach` function automatically creates an iterator from a vector, list, matrix, or data frame, for example. You can also create an iterator from a file or a data base query, which are natural sources of data. The `iterators` package supplies a function called `irnorm` which can return a specified number of random numbers for each time it is called. For example:

```
> library(iterators)
> x <- foreach(a=irnorm(4, count=4), .combine='cbind') %do% a
> x

       result.1   result.2   result.3    result.4
[1,]  2.3142980 -1.4846688 -0.5444280  0.36410604
[2,] -1.0065438 -1.3176158 -1.8797260 -1.18736054
[3,]  0.2743127  0.5039962  1.9917742  0.01261119
[4,]  0.5086706 -1.7966186 -0.3854015  0.87226388
```

This becomes useful when dealing with large amounts of data. Iterators allow the data to be generated on-the-fly, as it is needed by your operations, rather than requiring all of the data to be generated at the beginning.

For example, let's say that we want to sum together a thousand random vectors:

```
> set.seed(123)
> x <- foreach(a=irnorm(4, count=1000), .combine='+') %do% a
> x

[1]   9.097676 -13.106472  14.076261  19.252750
```

This uses very little memory, since it is equivalent to the following `while` loop:

```
> set.seed(123)
> x <- numeric(4)
> i <- 0
> while (i < 1000) {
+   x <- x + rnorm(4)
+   i <- i + 1
+ }
> x

[1]   9.097676 -13.106472  14.076261  19.252750
```

This could have been done using the `icount` function, which generates the values from one to 1000:

```
> set.seed(123)
> x <- foreach(icount(1000), .combine='+') %do% rnorm(4)
> x

[1]   9.097676 -13.106472  14.076261  19.252750
```

but sometimes it's preferable to generate the actual data with the iterator (as we'll see later when we execute in parallel).

In addition to introducing the `icount` function from the `iterators` package, the last example also used an unnamed argument to the `foreach` function. This can be useful when we're not intending to generate variable values, but only controlling the number of times that the R expression is executed.

There's a lot more that I could say about iterators, but for now, let's move on to parallel execution.

5 Parallel Execution

Although `foreach` can be a useful construct in its own right, the real point of the `foreach` package is to do parallel computing. To make any of the previous examples run in parallel, all you have to do is to replace `%do%` with `%dopar%`. But for the kinds of quick running operations that we've been doing, there wouldn't be much point to executing them in parallel. Running many tiny tasks in parallel will usually take more time to execute than running them sequentially, and if it already runs fast, there's no motivation to make it run faster anyway. But if the operation that we're executing in parallel takes a minute or longer, there starts to be some motivation.

5.1 Parallel Random Forest

Let's take random forest as an example of an operation that can take a while to execute. Let's say our inputs are the matrix `x`, and the factor `y`:

```
> x <- matrix(runif(500), 100)
> y <- gl(2, 50)
```

We've already loaded the `foreach` package, but we'll also need to load the `randomForest` package:

```
> library(randomForest)
```

If we want want to create a random forest model with a 1000 trees, and our computer has four cores in it, we can split up the problem into four pieces by executing the `randomForest` function four times, with the `ntree` argument set to 250. Of course, we have to combine the resulting `randomForest` objects, but the `randomForest` package comes with a function called `combine` that does just that.

Let's do that, but first, we'll do the work sequentially:

```
> rf <- foreach(ntree=rep(250, 4), .combine=combine) %do%
+   randomForest(x, y, ntree=ntree)
> rf

Call:
 randomForest(x = x, y = y, ntree = ntree)
               Type of random forest: classification
                     Number of trees: 1000
No. of variables tried at each split: 2
```

To run this in parallel, we need to change %do%, but we also need to use another foreach option called .packages to tell the foreach package that the R expression needs to have the randomForest package loaded in order to execute successfully. Here's the parallel version:

```
> rf <- foreach(ntree=rep(250, 4), .combine=combine, .packages='randomForest') %dopar%
+   randomForest(x, y, ntree=ntree)
> rf

Call:
 randomForest(x = x, y = y, ntree = ntree)
               Type of random forest: classification
                     Number of trees: 1000
No. of variables tried at each split: 2
```

If you've done any parallel computing, particularly on a cluster, you may wonder why I didn't have to do anything special to handle x and y. The reason is that the %dopar% function noticed that those variables were referenced, and that they were defined in the current environment. In that case %dopar% will automatically export them to the parallel execution workers once, and use them for all of the expression evaluations for that foreach execution. That is true for functions that are defined in the current environment as well, but in this case, the function is defined in a package, so we had to specify the package to load with the .packages option instead.

Beginning with foreach 0.5.0, foreach will use the future package, if available, to automatically detect needed packages. (This also requires appropriately updated foreach backends.) You can retain the old behavior by setting options(foreachGlobals="foreach").

5.2 Parallel Apply

Now let's take a look at how to make a parallel version of the standard R apply function. The apply function is written in R, and although it's only about 100 lines of code, it's a bit difficult to understand on a first reading. However, it all really comes down two for loops, the slightly more complicated of which looks like:

```
> applyKernel <- function(newX, FUN, d2, d.call, dn.call=NULL, ...) {
+   ans <- vector("list", d2)
+   for(i in 1:d2) {
+     tmp <- FUN(array(newX[,i], d.call, dn.call), ...)
+     if(!is.null(tmp)) ans[[i]] <- tmp
+   }
+   ans
+ }
> applyKernel(matrix(1:16, 4), mean, 4, 4)
```

```
[[1]]
[1] 2.5

[[2]]
[1] 6.5

[[3]]
[1] 10.5

[[4]]
[1] 14.5
```

I've turned this into a function, because otherwise, R will complain that I'm using "..." in an invalid context.

This could be executed using `foreach` as follows:

```
> applyKernel <- function(newX, FUN, d2, d.call, dn.call=NULL, ...) {
+   foreach(i=1:d2) %dopar%
+     FUN(array(newX[,i], d.call, dn.call), ...)
+ }
> applyKernel(matrix(1:16, 4), mean, 4, 4)

[[1]]
[1] 2.5

[[2]]
[1] 6.5

[[3]]
[1] 10.5

[[4]]
[1] 14.5
```

But this approach will cause the entire `newX` array to be sent to each of the parallel execution workers. Since each task needs only one column of the array, we'd like to avoid this extra data communication.

One way to solve this problem is to use an iterator that iterates over the matrix by column:

```
> applyKernel <- function(newX, FUN, d2, d.call, dn.call=NULL, ...) {
+   foreach(x=iter(newX, by='col')) %dopar%
+     FUN(array(x, d.call, dn.call), ...)
+ }
> applyKernel(matrix(1:16, 4), mean, 4, 4)

[[1]]
[1] 2.5

[[2]]
[1] 6.5

[[3]]
[1] 10.5

[[4]]
[1] 14.5
```

Now we're only sending any given column of the matrix to one parallel execution worker. But it would be even more efficient if we sent the matrix in bigger chunks. To do that, we use a function called `iblkcol` that returns an iterator that will return multiple columns of the original matrix. That means that the R expression will need to execute the user's function once for every column in its submatrix.

```
> applyKernel <- function(newX, FUN, d2, d.call, dn.call=NULL, ...) {
+   foreach(x=iblkcol(newX, 3), .combine='c', .packages='foreach') %dopar% {
+     foreach(i=1:ncol(x)) %do% FUN(array(x[,i], d.call, dn.call), ...)
+   }
+ }
> applyKernel(matrix(1:16, 4), mean, 4, 4)

[[1]]
[1] 2.5

[[2]]
[1] 6.5

[[3]]
[1] 10.5

[[4]]
[1] 14.5
```

Note the use of the %do% inside the %dopar% to call the function on the columns of the submatrix x. Now that we're using %do% again, it makes sense for the iterator to be an index into the matrix x, since %do% doesn't need to copy x the way that %dopar% does.

6 List Comprehensions

If you're familar with the Python programming language, it may have occurred to you that the foreach package provides something that is not too different from Python's *list comprehensions*. In fact, the foreach package also includes a function called when which can prevent some of the evaluations from happening, very much like the "if" clause in Python's list comprehensions. For example, you could filter out negative values of an iterator using when as follows:

```
> x <- foreach(a=irnorm(1, count=10), .combine='c') %:% when(a >= 0) %do% sqrt(a)
> x
```

```
[1] 0.4055020 1.0835713 0.8704032 0.3653185 1.4166866 0.8115083
```

I won't say much on this topic, but I can't help showing how foreach with when can be used to write a simple quick sort function, in the classic Haskell fashion:

```
> qsort <- function(x) {
+   n <- length(x)
+   if (n == 0) {
+     x
+   } else {
+     p <- sample(n, 1)
+     smaller <- foreach(y=x[-p], .combine=c) %:% when(y <= x[p]) %do% y
+     larger  <- foreach(y=x[-p], .combine=c) %:% when(y >  x[p]) %do% y
+     c(qsort(smaller), x[p], qsort(larger))
+   }
+ }
> qsort(runif(12))
```

```
 [1] 0.05671936 0.05986948 0.19082846 0.22652967 0.54588779 0.62601549
 [7] 0.66316703 0.68171436 0.74671367 0.80146286 0.80993460 0.82453758
```

Not that I recommend this over the standard R sort function. But it's a pretty interesting example use of foreach.

7 Conclusion

Much of parallel computing comes to doing three things: splitting the problem into pieces, executing the pieces in parallel, and combining the results back together. Using the `foreach` package, the iterators help you to split the problem into pieces, the `%dopar%` function executes the pieces in parallel, and the specified `.combine` function puts the results back together. We've demonstrated how simple things can be done in parallel quite easily using the `foreach` package, and given some ideas about how more complex problems can be solved. But it's a fairly new package, and we will continue to work on ways of making it a more powerful system for doing parallel computing.