Nesting `Foreach` Loops

Steve Weston

November 26, 2018

1 Introduction

The `foreach` package provides a looping construct for executing R code repeatedly. It is similar to the standard `for` loop, which makes it easy to convert a `for` loop to a `foreach` loop. Unlike many parallel programming packages for R, `foreach` doesn't require the body of the `for` loop to be turned into a function. `foreach` differs from a `for` loop in that its return is a list of values, whereas a `for` loop has no value and uses side effects to convey its result. Because of this, `foreach` loops have a few advantages over `for` loops when the purpose of the loop is to create a data structure such as a vector, list, or matrix: First, there is less code duplication, and hence, less chance for an error because the initialization of the vector or matrix is unnecessary. Second, a `foreach` loop may be easily parallelized by changing only a single keyword.

2 The nesting operator: %:%

An important feature of `foreach` is the `%:%` operator. I call this the *nesting* operator because it is used to create nested `foreach` loops. Like the `%do%` and `%dopar%` operators, it is a binary operator, but it operates on two `foreach` objects. It also returns a `foreach` object, which is essentially a special merger of its operands.

Let's say that we want to perform a Monte Carlo simulation using a function called `sim`.[1] The `sim` function takes two arguments, and we want to call it with all combinations of the values that are stored in the vectors `avec` and `bvec`. The following doubly-nested `for` loop does that. For testing purposes, the `sim` function is defined to return $10a + b$:[2]

```
> x <- matrix(0, length(avec), length(bvec))
> for (j in 1:length(bvec)) {
```

[1] Remember that `sim` needs to be rather compute intensive to be worth executing in parallel.

[2] Of course, an operation this trivial is not worth executing in parallel.

```
+   for (i in 1:length(avec)) {
+     x[i,j] <- sim(avec[i], bvec[j])
+   }
+ }
> x
```

```
     [,1] [,2] [,3] [,4]
[1,]   11   12   13   14
[2,]   21   22   23   24
```

In this case, it makes sense to store the results in a matrix, so we create one of the proper size called `x`, and assign the return value of `sim` to the appropriate element of `x` each time through the inner loop.

When using `foreach`, we don't create a matrix and assign values into it. Instead, the inner loop returns the columns of the result matrix as vectors, which are combined in the outer loop into a matrix. Here's how to do that using the `%:%` operator:[3]

```
> x <-
+   foreach(b=bvec, .combine='cbind') %:%
+     foreach(a=avec, .combine='c') %do% {
+       sim(a, b)
+     }
> x
```

```
     result.1 result.2 result.3 result.4
[1,]       11       12       13       14
[2,]       21       22       23       24
```

This is structured very much like the nested `for` loop. The outer `foreach` is iterating over the values in "bvec", passing them to the inner `foreach`, which iterates over the values in "avec" for each value of "bvec". Thus, the "sim" function is called in the same way in both cases. The code is slightly cleaner in this version, and has the advantage of being easily parallelized.

3 Using %:% with %dopar%

When parallelizing nested `for` loops, there is always a question of which loop to parallelize. The standard advice is to parallelize the outer loop. This results in larger individual tasks, and larger

[3] Due to operator precedence, you cannot put braces around the inner `foreach` loop. Unfortunately, that causes Sweave to format this example rather badly, in my opinion.

tasks can often be performed more efficiently than smaller tasks. However, if the outer loop doesn't have many iterations and the tasks are already large, parallelizing the outer loop results in a small number of huge tasks, which may not allow you to use all of your processors, and can also result in load balancing problems. You could parallelize an inner loop instead, but that could be inefficient because you're repeatedly waiting for all the results to be returned every time through the outer loop. And if the tasks and number of iterations vary in size, then it's really hard to know which loop to parallelize.

But in our Monte Carlo example, all of the tasks are completely independent of each other, and so they can all be executed in parallel. You really want to think of the loops as specifying a single stream of tasks. You just need to be careful to process all of the results correctly, depending on which iteration of the inner loop they came from.

That is exactly what the `%:%` operator does: it turns multiple `foreach` loops into a single loop. That is why there is only one `%do%` operator in the example above. And when we parallelize that nested `foreach` loop by changing the `%do%` into a `%dopar%`, we are creating a single stream of tasks that can all be executed in parallel:

```
> x <-
+   foreach(b=bvec, .combine='cbind') %:%
+     foreach(a=avec, .combine='c') %dopar% {
+       sim(a, b)
+     }
> x
```

```
     result.1 result.2 result.3 result.4
[1,]       11       12       13       14
[2,]       21       22       23       24
```

Of course, we'll actually only run as many tasks in parallel as we have processors, but the parallel backend takes care of all that. The point is that the `%:%` operator makes it easy to specify the stream of tasks to be executed, and the `.combine` argument to `foreach` allows us to specify how the results should be processed. The backend handles executing the tasks in parallel.

4 Chunking tasks

Of course, there has to be a snag to this somewhere. What if the tasks are quite small, so that you really might want to execute the entire inner loop as a single task? Well, small tasks are a problem even for a singly-nested loop. The solution to this problem, whether you have a single loop or nested loops, is to use *task chunking*.

Task chunking allows you to send multiple tasks to the workers at once. This can be much more efficient, especially for short tasks. Currently, only the doNWS backend supports task chunking. Here's how it's done with doNWS:

```
> opts <- list(chunkSize=2)
> x <-
+   foreach(b=bvec, .combine='cbind', .options.nws=opts) %:%
+     foreach(a=avec, .combine='c') %dopar% {
+       sim(a, b)
+     }
> x
```

```
     result.1 result.2 result.3 result.4
[1,]       11       12       13       14
[2,]       21       22       23       24
```

If you're not using doNWS, then this argument is ignored, which allows you to write code that is backend-independent. You can also specify options for multiple backends, and only the option list that matches the registered backend will be used.

It would be nice if the chunk size could be picked automatically, but I haven't figured out a good, safe way to do that. So for now, you need to specify the chunk size manually.[4]

The point is that by using the %:% operator, you can convert a nested for loop to a nested foreach loop, use %dopar% to run in parallel, and then tune the size of the tasks using the "chunkSize" option so that they are big enough to be executed efficiently, but not so big that they cause load balancing problems. You don't have to worry about which loop to parallelize, because you're turning the nested loops into a single stream of tasks that can all be executed in parallel by the parallel backend.

5 Another example

Now let's imagine that the "sim" function returns a object that includes an error estimate. We want to return the result with the lowest error for each value of b, along with the arguments that generated that result. Here's how that might be done with nested for loops:

```
> n <- length(bvec)
> d <- data.frame(x=numeric(n), a=numeric(n), b=numeric(n), err=numeric(n))
```

[4] In the future, the backend might decide that it will execute the tasks in parallel. That could be very useful when running on a cluster with multiprocessor nodes. Multiple tasks are sent across the network to each node, which then executes them in parallel on its cores. Maybe in the next release...

```
> for (j in 1:n) {
+   err <- Inf
+   best <- NULL
+   for (i in 1:length(avec)) {
+     obj <- sim(avec[i], bvec[j])
+     if (obj$err < err) {
+       err <- obj$err
+       best <- data.frame(x=obj$x, a=avec[i], b=bvec[j], err=obj$err)
+     }
+   }
+   d[j,] <- best
+ }
> d
```

```
   x a b err
1 11 1 1   0
2 22 2 2   0
3 23 2 3   1
4 24 2 4   2
```

This is also quite simple to convert to `foreach`. We just need to supply the appropriate ".combine" functions. For the outer `foreach`, we can use the standard "rbind" function which can be used with data frames. For the inner `foreach`, we write a function that compares two data frames, each with a single row, returning the one with a smaller error estimate:

```
> comb <- function(d1, d2) if (d1$err < d2$err) d1 else d2
```

Now we specify it with the ".combine" argument to the inner `foreach`:

```
> opts <- list(chunkSize=2)
> d <-
+   foreach(b=bvec, .combine='rbind', .options.nws=opts) %:%
+     foreach(a=avec, .combine='comb', .inorder=FALSE) %dopar% {
+       obj <- sim(a, b)
+       data.frame(x=obj$x, a=a, b=b, err=obj$err)
+     }
> d
```

```
   x a b err
1 11 1 1   0
```

```
2 22 2 2   0
3 23 2 3   1
4 24 2 4   2
```

Note that since the order of the arguments to the "comb" function is unimportant, I have set the ".inorder" argument to FALSE. This reduces the number of results that need to be saved on the master before they can be combined in case they are returned out of order. But even with niceties such as parallelization, backend-specific options, and the ".inorder" argument, the nested foreach version is quite readable.

But what if we would like to return the indices into "avec" and "bvec", rather than the data itself? A simple way to do that is to create a couple of counting iterators that we pass to the foreach functions:[5]

```
> library(iterators)
> opts <- list(chunkSize=2)
> d <-
+   foreach(b=bvec, j=icount(), .combine='rbind', .options.nws=opts) %:%
+     foreach(a=avec, i=icount(), .combine='comb', .inorder=FALSE) %dopar% {
+       obj <- sim(a, b)
+       data.frame(x=obj$x, i=i, j=j, err=obj$err)
+     }
> d
```

```
   x i j err
1 11 1 1   0
2 22 2 2   0
3 23 2 3   1
4 24 2 4   2
```

These new iterators are infinite iterators, but that's no problem since we have "bvec" and "avec" to control the number of iterations of the loops. Making them infinite means we don't have to keep them in sync with "bvec" and "avec".

6 Conclusion

Nested for loops are a common construct, and are often the most time consuming part of R scripts, so they are prime candidates for parallelization. The usual approach is to parallelize the outer

[5] It is very important that the call to icount is passed as the argument to foreach. If the iterators were created and passed to foreach using a variable, for example, we would not get the desired effect. This is not a bug or a limitation, but an important aspect of the design of the foreach function.

loop, but as we've seen, that can lead to suboptimal performance due to an imbalance between the size and the number of tasks. By using the `%:%` operator with `foreach`, and by using chunking techniques, many of these problems can be overcome. The resulting code is often clearer and more readable than the original R code, since `foreach` was designed to deal with exactly this kind of problem.