Getting Started with doMC and foreach

Steve Weston

doc@revolutionanalytics.com

December 9, 2017

1   Introduction

The `doMC` package is a "parallel backend" for the `foreach` package. It provides a mechanism needed to execute `foreach` loops in parallel. The `foreach` package must be used in conjunction with a package such as `doMC` in order to execute code in parallel. The user must register a parallel backend to use, otherwise `foreach` will execute tasks sequentially, even when the %dopar% operator is used.[1]

The `doMC` package acts as an interface between `foreach` and the `multicore` functionality of the `parallel` package, originally written by Simon Urbanek and incorporated into `parallel` for R 2.14.0. The `multicore` functionality currently only works with operating systems that support the `fork` system call (which means that Windows isn't supported). Also, `multicore` only runs tasks on a single computer, not a cluster of computers. That means that it is pointless to use `doMC` and `multicore` on a machine with only one processor with a single core. To get a speed improvement, it must run on a machine with multiple processors, multiple cores, or both.

2   A word of caution

Because the `multicore` functionality starts its workers using `fork` without doing a subsequent `exec`, it has some limitations. Some operations cannot be performed properly by forked processes. For example, connection objects very likely won't work. In some cases, this could cause an object to become corrupted, and the R session to crash.

In addition, it usually isn't safe to run `doMC` and `multicore` from a GUI environment.

---

[1] `foreach` will issue a warning that it is running sequentially if no parallel backend has been registered. It will only issue this warning once, however.

3 Registering the doMC parallel backend

To register doMC to be used with foreach, you must call the registerDoMC function. This function takes only one argument, named "cores". This specifies the number of worker processes that it will use to execute tasks, which will normally be equal to the total number of cores on the machine. You don't need to specify a value for it, however. By default, the multicore package will use the value of the "cores" option, as specified with the standard "options" function. If that isn't set, then multicore will try to detect the number of cores, and use approximately half that many workers.

Remember: unless registerDoMC is called, foreach will *not* run in parallel. Simply loading the doMC package is not enough.

4 An example doMC session

Before we go any further, let's load doMC, register it, and use it with foreach:

```
> library(doMC)
> registerDoMC(2)
> foreach(i=1:3) %dopar% sqrt(i)

[[1]]
[1] 1

[[2]]
[1] 1.414214

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

> Note well that this is *not* a practical use of doMC. This is my "Hello, world" program for parallel computing. It tests that everything is installed and set up properly, but don't expect it to run faster than a sequential for loop, because it won't! sqrt executes far too quickly to be worth executing in parallel, even with a large number of iterations. With small tasks, the overhead of scheduling the task and returning the result can be greater than the time to execute the task itself, resulting in poor performance. In addition, this example doesn't make use of the vector capabilities of sqrt, which it must to get decent performance. This is just a test and a pedagogical example, *not* a benchmark.

But returning to the point of this example, you can see that it is very simple to load doMC with all of its dependencies (foreach, iterators, multicore, etc), and to register it. For the rest of

the R session, whenever you execute `foreach` with `%dopar%`, the tasks will be executed using `doMC` and `multicore`. Note that you can register a different parallel backend later, or deregister `doMC` by registering the sequential backend by calling the `registerDoSEQ` function.

5 A more serious example

Now that we've gotten our feet wet, let's do something a bit less trivial. One good example is bootstrapping. Let's see how long it takes to run 10,000 bootstrap iterations in parallel on 2 cores:

```
> x <- iris[which(iris[,5] != "setosa"), c(1,5)]
> trials <- 10000
> ptime <- system.time({
+   r <- foreach(icount(trials), .combine=cbind) %dopar% {
+     ind <- sample(100, 100, replace=TRUE)
+     result1 <- glm(x[ind,2]~x[ind,1], family=binomial(logit))
+     coefficients(result1)
+   }
+ })[3]
> ptime

elapsed
 17.586
```

Using `doMC` and `multicore` we were able to perform 10,000 bootstrap iterations in 17.586 seconds on 2 cores. By changing the `%dopar%` to `%do%`, we can run the same code sequentially to determine the performance improvement:

```
> stime <- system.time({
+   r <- foreach(icount(trials), .combine=cbind) %do% {
+     ind <- sample(100, 100, replace=TRUE)
+     result1 <- glm(x[ind,2]~x[ind,1], family=binomial(logit))
+     coefficients(result1)
+   }
+ })[3]
> stime

elapsed
 33.244
```

The sequential version ran in 33.244 seconds, which means the speed up is about 1.9 on 2 workers.[2] Ideally, the speed up would be 2, but no multicore CPUs are ideal, and neither are the operating systems and software that run on them.

At any rate, this is a more realistic example that is worth executing in parallel. I'm not going to explain what it's doing or how it works here. I just want to give you something more substantial than the `sqrt` example in case you want to run some benchmarks yourself. You can also run this example on a cluster by simply registering a different parallel backend that supports clusters in order to take advantage of more processors.

6   Getting information about the parallel backend

To find out how many workers `foreach` is going to use, you can use the `getDoParWorkers` function:

```
> getDoParWorkers()
```

```
[1] 2
```

This is a useful sanity check that you're actually running in parallel. If you haven't registered a parallel backend, or if your machine only has one core, `getDoParWorkers` will return one. In either case, don't expect a speed improvement. `foreach` is clever, but it isn't magic.

The `getDoParWorkers` function is also useful when you want the number of tasks to be equal to the number of workers. You may want to pass this value to an iterator constructor, for example.

You can also get the name and version of the currently registered backend:

```
> getDoParName()
```

```
[1] "doMC"
```

```
> getDoParVersion()
```

```
[1] "1.3.5"
```

This is mostly useful for documentation purposes, or for checking that you have the most recent version of `doMC`.

---

[2]If you build this vignette yourself, you can see how well this problem runs on your hardware. None of the times are hardcoded in this document. You can also run the same example which is in the examples directory of the `doMC` distribution.

7 Specifying multicore options

The `doMC` package allows you to specify various options when running `foreach` that are supported by the underlying `mclapply` function: "preschedule", "set.seed", "silent", and "cores". You can learn about these options from the `mclapply` man page. They are set using the `foreach .options.multicore` argument. Here's an example of how to do that:

```
> mcoptions <- list(preschedule=FALSE, set.seed=FALSE)
> foreach(i=1:3, .options.multicore=mcoptions) %dopar% sqrt(i)


[[1]]
[1] 1

[[2]]
[1] 1.414214

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

The "cores" options allows you to temporarily override the number of workers to use for a single `foreach` operation. This is more convenient than having to re-register `doMC`. Although if no value of "cores" was specified when `doMC` was registered, you can also change this value dynamically using the `options` function:

```
> registerDoMC()
> getDoParWorkers()
[1] 3
> options(cores=2)
> getDoParWorkers()
[1] 2
> options(cores=3)
> getDoParWorkers()
[1] 3
```

If you did specify the number of cores when registering `doMC`, the "cores" option is ignored:

```
> registerDoMC(2)
> options(cores=4)
> getDoParWorkers()
```

```
[1] 2
```

As you can see, there are a number of options for controlling the number of workers to use with `multicore`, but the default behaviour usually does what you want.

8   Conclusion

The `doMC` and `multicore` packages provide a nice, efficient parallel programming platform for multiprocessor/multicore computers running operating systems such as Linux and Mac OS X. It is very easy to install, and very easy to use. In short order, an average R programmer can start executing parallel programs, without any previous experience in parallel computing.