Getting Started with doSMP and foreach

Steve Weston
doc@revolutionanalytics.com

February 27, 2011

1 Introduction

The `doSMP` 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 `doSMP` 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]

It is important to note that the `doSMP` package only runs tasks on a single computer, not a cluster of computers.[2] That means that it is pointless to use `doSMP` 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 Starting and stopping worker processes

To register `doSMP` to be used with `foreach`, you must first create a "workers" object by calling the `startWorkers` function. `startWorkers` takes various arguments to specify the number of workers to start, verbose mode, etc. If the number of workers isn't specified, it will start the number returned by `getOptions("cores")`. Thus, you can use the standard `options` function to control the number of workers to start. If the "cores" option isn't set either, than three workers are started.[3]

It is important to explicitly stop the workers using the `stopWorkers` function before exiting from R. If you forget, the workers will probably continued to execute, and you may also leak some IPC

[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.

[2]You can use the `doNWS` package to execute on a cluster using the `nws` package. `doNWS` is available from Revolution Analytics: `http://www.revolutionanalytics.com/products/parallel-r.php`.

[3]The default value of three may change in the future to be the number of cores on the machine. Currently, `doSMP` does not try to determine the number of cores.

resources.

When calling `startWorkers`, you may see a warning message that there are existing doSMP sessions. That is important to know, because your performance will probably suffer if there is another parallel job running on your machine at the same time. But it's also possible that a previous doSMP session wasn't properly shut down using `stopWorkers`. If you know that the other job is defunct, you can shut it down using the `rmSessions` function. For example, if you get a warning message that there is an existing doSMP session using "doSMP1", you can shut down and clean up that previous session using the command `rmSessions('doSMP1')`. You can clean up all sessions using `rmSessions(all=TRUE)`, but don't do that if you have a current session, because that will also shut down your current session.

3 Registering the `doSMP` parallel backend

Once you're started some workers, you must register `doSMP` to be used for parallel execution, specifying the "workers" object using the `registerDoSMP` function.

Remember: unless `registerDoSMP` is called, `foreach` will *not* run in parallel. Simply loading the `doSMP` package is not enough.

4 An example `doSMP` session

Before we go any further, let's load `doSMP`, start some workers, register `doSMP`, and use it with `foreach`:

```
> library(doSMP)
> w <- startWorkers(workerCount = 4)
> registerDoSMP(w)
> 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 `doSMP`. 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 `doSMP` with all of its dependencies (`foreach`, `iterators`, etc), and to register it. For the rest of the R session, whenever you execute `foreach` with `%dopar%`, the tasks will be executed using `doSMP`. Note that you can register a different parallel backend later, or deregister `doSMP` 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 4 cores:

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

elapsed
 57.614
```

Using `doSMP` we were able to perform 10,000 bootstrap iterations in 57.614 seconds on 4 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
104.426
```

The sequential version ran in 104.426 seconds, which means the speed up is about 1.8 on 4 workers.[4] Ideally, the speed up would be 4, 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] 4
```

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:

[4] 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 `doSMP` distribution.

```
> getDoParName()
```

```
[1] "doSMP"
```

```
> getDoParVersion()
```

```
[1] "1.0-1"
```

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

7 Specifying `doSMP` options

The `doSMP` package allows you to specify various options when running `foreach`: “chunkSize”, “info”, “initEnvir”, “initArgs”, “finalEnvir”, and “finalArgs”. You can learn about these options from the `doSMP` man page. They are set using the `foreach .options.smp` argument. Here’s an example of how to do that:

```
> smpopts <- list(chunkSize = 2)
> foreach(i = 1:6, .options.smp = smpopts) %dopar% sqrt(i)
```

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

[[2]]
[1] 1.414214

[[3]]
[1] 1.732051

[[4]]
[1] 2

[[5]]
[1] 2.236068

[[6]]
[1] 2.449490
```

8 Conclusion

The `doSMP` package provides a nice, efficient parallel programming platform for multiprocessor/multicore computers running operating systems such as Windows, 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.