Introduction to doMPI

Steve Weston
stephen.b.weston@gmail.com

June 13, 2013

1 Introduction

The doMPI package is what I call a "parallel backend" for the foreach package. Since the foreach package is not a parallel programming system, but a parallel programming framework, it needs a parallel programming system to do the actual work in parallel. The doMPI package acts as an adaptor to the Rmpi package, which in turn is an R interface to an implementation of MPI. MPI, or *Message Passing Interface*, is a specification for an API for passing messages between different computers. There are a number of MPI implementations available that allow data to be moved between computers quite efficiently.

Programming with MPI is rather difficult, however, since it is a rather large and complex API. For example, the Rmpi package defines about 110 R functions, only a few of which are only for internal use. And the MPI standard includes many more functions that aren't supported by Rmpi, such as the file functions. Of course, you only need to learn a small percentage of those functions in order to start using MPI effectively, but it can take awhile just to figure out which functions are really important, and which ones you can safely ignore.

The foreach package is an attempt to make parallel computing much simpler by providing a parallel for-loop construct for R. As an adaptor to MPI, the doMPI package is an attempt to give you the best of both worlds: the ease of use of parallel for-loops, with the efficiency of MPI.

Unfortunately, there are still a wide variety of problems that you may run into. First, you need to have MPI installed on your computers, and then you have to build and install Rmpi to use that MPI installation. That is where many people run into problems, particularly on Windows. However, on Debian / Ubuntu, it is very easy to install Open MPI, and Rmpi works quite well with it.

Another thing to be aware of is that you have to run your R programs differently in order to execute on multiple computers, and the exact method varies depending on your MPI implementation. If you just start a normal R session, your workers will only run on your local machine. That's great for testing, but the point of using doMPI is to execute on multiple computers. To do that, you'll

need to start your R session using a command such as mpirun or mpiexec, depending on your MPI installation.

Hopefully, this document will help you to get started, but it primarily deals with programming issues, not configuring or administering computers. And for running doMPI scripts, I only discuss the use of Open MPI. Very likely you will have some problem that I couldn't help you with anyway, and so I highly recommend that you join the R-sig-hpc mailing list. It is a very friendly and helpful community, and there isn't even a lot of traffic on it.

2 Installing the software

Assuming that you already have R and MPI installed on your computers, you will need to install doMPI and the packages that it depends on. That can be done with a single "install.packages" command, but you might want to explicitly install Rmpi first, so you can clearly see if it installs correctly or not. On Debian / Ubuntu, you can actually install Rmpi by using apt-get to install the Debian r-cran-rmpi package. That's the most fool-proof way to install Rmpi on Ubuntu, for example, since apt-get will automatically install a compatible version of MPI, if necessary. But on other systems you can install it with:

```
> install.packages("Rmpi")
```

As I mentioned, if you have problems, I strongly recommend the R-sig-hpc mailing list if you don't have a local expert.

Once Rmpi is installed, the rest should be easy:

```
> install.packages("doMPI", dependencies=TRUE)
```

That will also install packages such as foreach and iterators.

3 Getting Started

So if I haven't scared you off with my introduction, and you got all of the software installed, then it's time to see if it actually works! For testing purposes, we'll just run on a single computer. In that case, you just start a normal R session. I'll talk about running on multiple computers later, in section 8.

Once you've started an R session, load the doMPI package:

```
> library(doMPI)
```

You'll no doubt get an error at this point if `Rmpi` isn't properly installed. If that happens, you should restart the `R` session, load `Rmpi`, make a copy of the error message, google around for the error, scratch your head for awhile, and then post a polite, well-thought-out request for help to R-sig-hpc that includes the entire error message. Very possibly, someone on the list has encountered that error before, and will offer advice on how to fix it. But don't be too surprised if someone suggests that you switch to Ubuntu.

Once you can successfully load `doMPI`, the next step is to create an `MPI` cluster object. There are lots of options, but here's one way to do that[1]:

```
> cl <- startMPIcluster(count=2)
```

This starts two cluster workers, or slaves, as `MPI` calls them. Note that if you're using `Open MPI`, those two processes will immediately start using as much of your CPU as they can. This is a known issue with `Open MPI`. They are waiting for the master process to send them a message, but they are "busy waiting" for that message. That doesn't make much sense in this context, but it can give better performance in some contexts. At any rate, it is an issue that the `Open MPI` group is planning to address in a future release.

The next step is to register the `MPI` cluster object with the `foreach` package. This is done with the "registerDoMPI" function:

```
> registerDoMPI(cl)
```

This tells `foreach` that you want to use `doMPI` as the parallel backend, and that you want to use the specified `MPI` cluster object to execute the tasks. If you don't register a cluster object, your tasks won't be executed in parallel, even though you use `foreach` with the `%dopar%` operator.

When you are finished using the `MPI` cluster object, it is important to shut it down, otherwise you may leak processes on your cluster.[2] You do that with the `closeCluster` function:

```
> closeCluster(cl)
```

And finally, you should probably call `mpi.quit` to exit your `R` session when using `doMPI`. You could call `mpi.finalize` instead if you want to use `doMPI` interactively during testing. Both of these functions will "finalize" `MPI`, which is a requirement for `MPI` programs. I'm not sure what the consequences are of not finalizing `MPI`, but I have seen error messages from `mpirun` when I forgot to finalize.

[1] I'm using the `count` argument because we're in an interactive R session. However, in section 8, I'll explain why I don't recommend using the `count` argument for most other circumstances.

[2] Shutting down the workers is particularly important with `Open MPI`, since they use CPU time even when they are "idle".

4 A simple example

Now we're ready to run a little test that actually runs in parallel. I always do that with a very simple program that executes the `sqrt` function in parallel. It's not a practical use of `foreach`, because the `sqrt` function runs so fast that it makes no sense to execute it in parallel. But it makes a great test.

```
> foreach(i=1:3) %dopar% sqrt(i)

[[1]]
[1] 1

[[2]]
[1] 1.414214

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

There are a lot of things to comment on in this little example. First of all, notice that a list containing three numbers is returned by this command. That is quite different from a for-loop. If we did this in a for-loop, we would execute `sqrt` three times, but the result would be thrown away. A for-loop normally uses assignments to save its results, but `foreach` works more like `lapply` in this respect, and that is what makes it work well for parallel execution.

Secondly, notice that we're using a strange binary operator, named `%dopar%`. `R` doesn't really make it easy to define new control structures, but since arguments to functions are lazily evaluated, it is possible. You don't have to worry about that, however. Just keep in mind that you need to put `%dopar%` in between the `foreach` and the loop body.

In a real program, you'll obviously want to save the results of the computations. You may also have several operations to perform, so the way you'll usually write the previous example is as:

```
> x <- foreach(i=1:3) %dopar% {
+     sqrt(i)
+ }
> x

[[1]]
[1] 1

[[2]]
```

```
[1] 1.414214

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

The braces are a good practice since they force the correct operator precedence. And if you squint hard, it looks just like a for-loop.

5 The .combine option

But what if we want our results to be returned in a vector, rather than a list? We could convert it to a list afterwards, but there's another way. The `foreach` function takes a number of additional arguments, all of which start with a "." to distinguish them from the "variable" arguments. The one we need in this case is named `.combine`. You can use it to specify a function that will be used to combine the results. This function must take at least two input arguments, and return a value which is a "combined" or possibly "reduced" version of its input arguments. To combine many numeric values into a single vector of numeric values, we can use the standard `c` function, which concatenates its arguments:

```
> x <- foreach(i=1:3, .combine="c") %dopar% {
+     sqrt(i)
+ }
> x

[1] 1.000000 1.414214 1.732051
```

This works well for numeric, character, and logical values.

I also use the `cbind` function as a `.combine` function quite often. I use it to turn vectors into matrices:

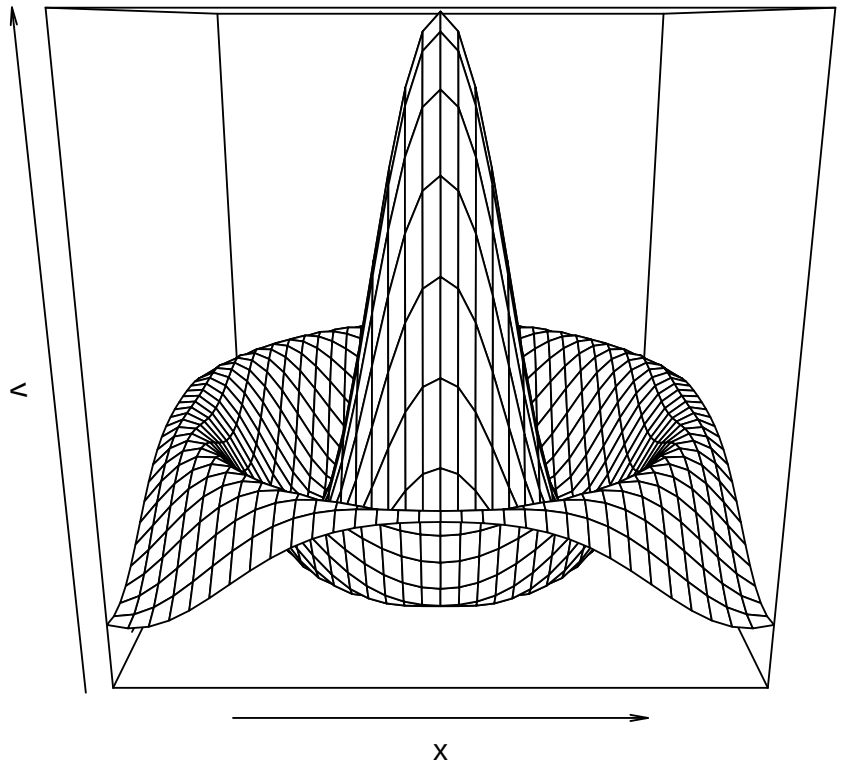
```
> x <- foreach(seed=c(7, 11, 13), .combine="cbind") %dopar% {
+     set.seed(seed)
+     rnorm(3)
+ }
> x

       result.1    result.2   result.3
[1,]  2.2872472 -0.59103110  0.5543269
[2,] -1.1967717  0.02659437 -0.2802719
[3,] -0.6942925 -1.51655310  1.7751634
```

6 Computing the sinc function

Now let's try a slightly more realistic example. Let's evaluate the `sinc` function over a grid of values. We'd like to return the results in a matrix, as in the previous example, so we'll use `cbind` to combine the results:

```
> x <- seq(-8, 8, by=0.5)
> v <- foreach(y=x, .combine="cbind") %dopar% {
+     r <- sqrt(x^2 + y^2) + .Machine$double.eps
+     sin(r) / r
+ }
> persp(x, x, v)
```



One thing to note in this example is that I'm doing an assignment to the variable `r` in the loop. You can do that with `foreach`, but if you want your code to work correctly in parallel, you need to be careful not to use variable assignments as a way of communicating between different iterations of the loop. That's what parallel programmers call a "loop dependence". In this case I'm only using it to communicate within a single iteration, so it's safe.

Also note that I'm using the variable `x` inside the loop, which was defined before the loop. In many other parallel computing systems, you'd have to explicitly export that variable somehow or other. With `foreach`, it's done automatically.

As a comparison, let's write this program using the standard `lapply` function:

```
> x <- seq(-8, 8, by=0.5)
> sinc <- function(y) {
+     r <- sqrt(x^2 + y^2) + .Machine$double.eps
+     sin(r) / r
+ }
> r <- lapply(x, sinc)
> v <- do.call("cbind", r)
> persp(x, x, v)
```

As you can see, there is a similarity between these two versions. But it makes me very happy that the parallel version is actually shorter and easier to explain than the sequential version.

I should also point out that in this `sinc` example, I've been very careful to make use of vector operations in the loop. The primary rule for getting good performance in `R` is to use vector operations whenever possible. I flagrantly broke that rule in the `sqrt` example. In the `sinc` example, I used vector operations to compute the columns, and I used `foreach` to execute those vector operations in parallel. The `sinc` example is still not really compute intensive enough to really merit executing it in parallel, but at least it uses much better `R` programming style.

7 Processing a directory of data files

Now let's try a different kind of example that you probably haven't seen in any parallel programming book, but simply reeks of practicality. Let's say that you have a directory full of CSV data files, and you want to read each of them, analyze the data in some way, and produce a plot of the results that you write to output files. Obviously, it's embarrassingly parallel, it can be time consuming, and it's a common task to perform.

Let's perform that kind of operation using Andy Liaw's `randomForest` package:

```
> ifiles <- list.files(pattern="\\.csv$")
> ofiles <- sub("\\.csv$", ".png", ifiles)
```

```
> foreach(i=ifiles, o=ofiles, .packages="randomForest") %dopar% {
+     d <- read.csv(i)
+     rf <- randomForest(Species~., data=d, proximity=TRUE)
+     png(filename=o)
+     MDSplot(rf, d$Species)
+     dev.off()
+     NULL
+ }
```

Note that we're iterating over two different vectors in this example: `ifiles` and `ofiles`, which are both the same length. The `foreach` function let's you specify any number of variables to iterate over, and it stops when one of them runs out of values.

Also note the `NULL` at the end of the loop body. There is no value to return in this case, so I used a `NULL` to make sure that I wasn't needlessly sending anything large back to the master, which would hurt the performance somewhat. The real results of executing the loop are written to disk by the workers directly.

8 Running doMPI on a cluster

Now that we've run some interesting examples on a single computer, let's try running on multiple computers, which is the real purpose of the `doMPI` package.

To execute `doMPI` scripts on multiple computers, you need to execute the `R` interpreter using a command such as `mpirun` or `mpiexec`. In this vignette, I'll only discuss the Open MPI version of `mpirun`, although it should be pretty simple to translate my instructions for other implementations.

When you execute `mpirun`, you can specify the hosts to use as a comma-separated list using the `-H` or `-host` option, or by specifying a "host file" using the `-hostfile` option. You can also use the `-n` or `-np` option to specify how many copies of your script to run on those nodes. If you specify that only one copy should be executed, then `doMPI` will start workers for you using the `Rmpi` `mpi.comm.spawn` function when you execute `startMPIcluster`. If you specify that more than one copy of your script should be executed on those hosts, then `doMPI` will not spawn workers, assuming that you have started all of the workers that you wanted using `mpirun`[3]. That means that multiple instances of your script are going to start running, or in other words, the workers are going to execute your script. In that case, your script should call the `startMPIcluster` function near the beginning. The `startMPIcluster` function will notice when a "worker" calls it, and will execute the `workerLoop` function, in order to execute tasks submitted by the master, or "rank 0" process.

Now let's try some examples. Here's an example that will run the script "sincMPI.R" using two

[3] Actually, `startMPIcluster` will always spawn workers unless the `comm` argument is `0`, but `comm` defaults to `0` if `mpi.comm.size(0)` is greater than one. See the documentation on `startMPIcluster` for more information.

workers:

```
$ mpirun -H localhost,n2,n3 R --slave -f sincMPI.R
```

The `mpirun` command will run the `R` interpreter on "localhost", "n2", and "n3", and `R` will execute the script named "sincMPI.R" in the current directory. `sincMPI.R` is an example that comes in the `doMPI` package, so you can try this command out by going to the "examples" directory of the `doMPI` installation.

The `sincMPI.R` script first loads `doMPI`, and then calls `startMPIcluster`. The copy of `sincMPI.R` running on the master, or "rank 0" process, will return from `startMPIcluster`, but the other two processes (which are running on hosts "n2" and "n3") will not return, since they will execute the `workerLoop` function. That is why it's a good idea to execute `startMPIcluster` near the beginning of the script, otherwise the workers may run into problems executing code that was intended to be executed by the master.

The last example used what I call "non-spawn" or "static" mode. Now let's run in "spawn" or "dynamic" mode. We'll start the master on node "n1", but we'll also list hosts "n2" and "n3". That will include those nodes in the `MPI` "universe", so that `startMPIcluster` will be able to spawn processes to them:

```
$ mpirun -H n1,n2,n3 -n 1 R --slave -f sincMPI.R
```

Because we specified `-n 1`, the `mpirun` command only starts one copy of the `R` interpreter running. `startMPIcluster` will then spawn two workers, since it notices that `mpi.universe.size()` is three. They will execute on hosts "n2" and "n3", since `MPI` knows that they are part of the "universe".

So which of these two methods should you use? The `Open MPI` documentation advices using "non-spawn" mode, since that will give better performance. You probably also need to use "non-spawn" mode to run with a batch queueing system, such as "slurm". But if your script needs to perform other operations before `startMPIcluster` that would fail on the worker processes, or if you need to create multiple clusters, then you should use the "spawn" mode, by specifying `-n 1`.

Here's a final example that starts multiple processes on the hosts using "non-spawn" mode:

```
$ mpirun -H localhost,n2,n3 -n 6 R --slave -f sincMPI.R
```

This will start two processes on each of the hosts for a total of six processes. The master and one worker will run on "localhost", and "n2" and "n3" will both run two workers. In order to spawn five workers, you would use `-n 1`, but you'd also have to modify the script to call `startMPIcluster` with `count` set to `5`.

In general, I'd suggest not using the `startMPIcluster count` argument unless you're doing something fancy, like creating multiple clusters. I think that's the best method, since it let's you

control the number of workers strictly from the `mpirun` command, which is what we've been doing in all of these examples. But if you do specify a value for `count` while in "non-spawn" mode, it's got to be equal to `mpi.comm.size(0) - 1`, or `startMPIcluster` will issue an error. That's one reason that none of the examples or benchmarks that come with `doMPI` use the `count` argument.

9 Host setup

Of course, in order for any of these examples to work, you've got to set up a lot of things correctly on all of the hosts specified via `-H`. Generally speaking, I would strongly suggest that you setup your hosts so they have an environment that is almost identical to your local machine. That is, they should all have the same user account, with the same home directory, with passwordless ssh enabled, and all of the same software installed using the same file paths. For example, `R` should be installed in the same place on all machines, and the `doMPI` package should be installed in the same directory on all machines, as well. That's a pretty big requirement, but that's the way most people do things with `MPI`, and so that's the way that many computer clusters are configured.

There are various tricks that you can use to make differently configured machines work together. One trick is to create symbolic links on the worker machines to make them look more like the master machine. And if your user account has different names on different machines, you can edit the ssh configuration file on the master machine to use the appropriate user name for each worker machine. But if you're going to do much parallel execution, it's worth the time and effort to set up your computers so that you don't have to depend on tricks.

10 Conclusion

I'd like to end by saying how easy parallel computing on a computer cluster can be if you use `foreach` and `doMPI`. And it is a lot simpler than many of the alternatives. But running on networks of computers always seems to present challenges. Hopefully, by using `foreach` and `doMPI`, most of those will be system administration challenges, and not programming challenges. Since `foreach` allows you to separate your parallel program from your parallel environment, you can develop your program on a single computer, without using any parallel backend at all, let alone a tricky one. Once your program is debugged and working, you can run it on a multicore workstation using a "low maintenance" parallel backend such as `doMC`[4]. And once you've got a computer cluster set up that has all the software installed by some friendly sysadmin, with an NFS-mounted home directory, you can use the `doMPI` package to run that same parallel program very efficiently using `MPI` without having to learn anything about message passing.

[4]The `doMC` package is also available on CRAN.