Parallel Computing in R

Matthew Olson

October 27, 2015

Overview

- Parallel backends in R: `Rmpi`, `doParallel`, `snow`, `multicore`, ...
- A number of packages (including `glmnet` and `dplyr`) can take advantage of a parallel backend, but you need to set everything up (as shown) below to take advantage.
- We'll be concerned with "embarassing parallel" computing, where tasks can be divided up in a communication free way

Your Best Friend: `forEach`

```
library(rbenchmark) # just to do timing

# Commands to set up a local cluster
library(doParallel)
cl <- makeCluster(4)
registerDoParallel(cl)

detectCores() # number of available workers on machine
getDoParWorkers() # number of registered workers

# Series Loop (NOTE: %do% !!!!!)
n <- 1e5
nSim <- 20
time1 <- benchmark(
                    res <- foreach(i = 1:nSim) %do% {
                    x <- mean(rnorm(n))
                    x
                    },
                    replications = 10)
time1$elapsed

# Parallel Loop (NOTE: %dopar% !!!!!)
time2 <- benchmark(
                    res <- foreach(i = 1:nSim) %dopar% {
                    x <- mean(rnorm(n))
                    x
                    },
                    replications = 10)
time2$elapsed

stopCluster(cl) # shutdown cluster when done
```

```
## [1] 4
## [1] 4
```

```
## [1] 2.168
## [1] 0.851
```

The method of setting up the parallel workers above is the most robust way, and works both on Windows and Unix. You can also do it more succinctly as below. One issue to keep in mind: the `mulitcore` package relies on multithreading, which Windows doesn't support (says the docs). The `snow` package is robust across Windows and Unix.

```
library(doParallel)
registerDoParallel(cores=4)
```

Function anatomy: `foreach` takes other args

(1) `.combine`: by default the result of each iteration of the loop is appended together in a list; by specifying `.combine` you can append output in other ways:

```
# return as list
sqrt1 <- foreach(i = 1:5) %dopar% sqrt(i)
# return as vector
sqrt2 <- foreach(i = 1:5, .combine="c") %dopar% sqrt(i)
# take the sum (i.e. any legit "map-reduce" type operation)
sqrt3 <- foreach(i = 1:5, .combine="+") %dopar% sqrt(i)
```

(2) `.inorder`: should the `.combine` operator be applied to output in the same order as the iterator, or in the order returned?
(3) `.packages`: when using packages in the loop, need to specify them in this arguement

```
n <- 100 #n.b. foreach is smart enough to export variables to each of the
p <- 200 # workers that are defined outside the loop
out <- foreach(i = 1:5, .packages = c("glmnet")) %dopar% {
  X <- matrix(rnorm(n*p), n)
  y <- rowSums(X) + rnorm(n)
  mod <- glmnet(X, y)
}
```

```
## Loading required package: Matrix
## Loaded glmnet 2.0-2
```

Gotchas

(1) Replicating your results: `doRNG` package

```
set.seed(123)
rand1 <- foreach(i = 1:5) %dopar% runif(3)
set.seed(123)
rand2 <- foreach(i = 1:5) %dopar% runif(3)
identical(rand1, rand2)
```

```
## [1] FALSE
```

```r
library(doRNG)
set.seed(123)
# NOTE %dorng% !!!!!!!!!
rand1 <- foreach(i = 1:5) %dorng% runif(3)
set.seed(123)
# NOTE %dorng% !!!!!!!!!
rand2 <- foreach(i = 1:5) %dorng% runif(3)
identical(rand1, rand2)
```

```
## [1] TRUE
```

(2) Load Balancing

You need to be smart when parallel computing: communication / scheduling costs can kill any performance benefits. Ideally, you should try to break your task up into chunks which are at least moderately computationally expensive. It only happens in the best case that we get a speed-up by a factor of nCores; often it is much less.

```r
# Serial
time1 <- benchmark(foreach(i = 1:5, .combine="c") %do% sqrt(i))

# Parallel
time2 <- benchmark(foreach(i = 1:5, .combine="c") %dopar% sqrt(i))

# Serial is FASTER! (we pay a fixed cost for scheduling the job)
time1$elapsed
```

```
## [1] 0.346
```

```r
time2$elapsed
```

```
## [1] 1.729
```

On the Grid

(Thanks to Sameer!)

(See `Code/gridExample` for a the complete example that runs on the grid)

If you are used to submitting task arrays, dealing with the `iterSGE` variable, saving results from each node, and then doing post-processing to aggregate your results, `foreach` will be a huge time saver!

Other Handy Functions from `parallel` Package

Tip: functional programming (map, reduce, filter, ...) makes parallel computing very easy since a lot of the parallelism can be handled "under the hood"

- `mclapply`: parallel version of `lapply`; relies on forking, so I don't believe that it will run in parallel on Windows...
- `mcmapply`: parallel version of `Map`; same caveat above applies

- `parLapply`: parallel version of `lapply` that takes a "cluster" arg
- `parSapply`: "" `sapply` ""

```
# will work on either windows or *nix
library(doParallel)
cl <- makeCluster(4)
registerDoParallel(cl)

nSim <- 1e3
n <- 200
#parLapply(cl, X = 1:nSim, fun = function(x) n)

# gotcha, need to export variables to the nodes (not so with foreach)
clusterExport(cl, varlist = c("n"))
out <- parLapply(cl, X = 1:nSim, fun = function(x) n)

# A more "realistic" example
time1 <- system.time(out1 <- parLapply(cl, 1:nSim,
                              function(x) max(svd(matrix(runif(n*n),n))$d)))
time2 <- system.time(out2 <- lapply(1:nSim,
                               function(x) max(svd(matrix(runif(n*n),n))$d)))

stopCluster(cl)
```

What Wasn't Covered

The types of situations we covered we are "embarassing parallel", i.e. you can work on subproblems independently, and then combine the result. Other types of computation might require more sophisticated communication at intermediate steps: i.e. node 1 relies on the result of node 2 relies on the result from node 3, etc. You can still do this kind of thing in R: see `Rmpi`. A nice reference on more advanced topics can be found at (https://github.com/berkeley-scf/parallelR-biostat-2015).