Introduction to parallel computing in R

Clint Leach

April 10, 2014

1 Motivation

When working with R, you will often encounter situations in which you need to repeat a computation, or a series of computations, many times. This can be accomplished through the use of a `for` loop. However, if there are a large number of computations that need to be carried out (i.e. many thousands), or if those individual computations are time-consuming (i.e. many minutes), a `for` loop can be very slow That said, almost all computers now have multicore processors, and as long as these computations do not need to communicate (i.e. they are "embarrassingly parallel"), they can be spread across multiple cores and executed in parallel, reducing computation time. Examples of these types of problems include:

- Run a simulation model using multiple different parameter sets,
- Run multiple MCMC chains simultaneously,
- Bootstrapping, cross-validation, etc.

2 Parallel backends

By default, R will not take advantage of all the cores available on a computer. In order to execute code in parallel, you have to first make the desired number of cores available to R by registering a 'parallel backend', which effectively creates a cluster to which computations can be sent. Fortunately there are a number of packages that will handle the nitty-gritty details of this process for you:

- `doMC` (built on `multicore`, works for unix-alikes)
- `doSNOW` (built on `snow`, works for Windows)
- `doParallel` (built on `parallel`, works for both)

The `parallel` package is essentially a merger of `multicore` and `snow`, and automatically uses the appropriate tool for your system, so I would recommend sticking with that.

Creating a parallel backend (i.e. cluster) is accomplished through just a few lines of code:

```r
library(doParallel)

# Find out how many cores are available (if you don't already know)
detectCores()

## [1] 4

# Create cluster with desired number of cores
cl <- makeCluster(3)

# Register cluster
registerDoParallel(cl)

# Find out how many cores are being used
getDoParWorkers()

## [1] 3
```

The syntax for the other packages is essentially the same, just with `register<Package>(cl)`.

3 Executing computations in parallel

Regardless of the application, parallel computing boils down to three basic steps: split the problem into pieces, execute in parallel, and collect the results.

3.1 Using `foreach`

These steps can all be handled through the use of the `foreach` package. This provides a parallel analogue to a standard `for` loop.

```r
library(foreach)

x <- foreach(i = 1:3) %dopar% sqrt(i)

x

## [[1]]
## [1] 1
##
## [[2]]
## [1] 1.414
##
## [[3]]
## [1] 1.732
```

As in a `for` loop, `i` defines an iterator (though note the use of = instead of `in`) splits the problem into pieces (step 1). For each value of the iterator then, the `%dopar%` operator passes the defined computations, here `sqrt(i)`, to the availables cores (step 2). Also note that, in contrast to a `for` loop, `foreach` collects the results (step 3) and returns an object, a list by default. This can be changed through the `.combine` option:

```
# Use the concatenate function to combine results
x <- foreach(i = 1:3, .combine = c) %dopar% sqrt(i)

# Now x is a vector
x

## [1] 1.000 1.414 1.732

# Can also use + or * to combine results
x <- foreach(i = 1:3, .combine = "+") %dopar% sqrt(i)

# Now x is a scalar, the sum of all the results
x

## [1] 4.146
```

Other options include `rbind` and `cbind`.

It is also important to note that `foreach` will automatically export any necessary variables (i.e. all variables defined in your current environment will be available to the cores), but any packages needed by the computations need to be passed using the `.packages` option.

3.2 Parallel `apply` functions

The `parallel` package also provides parallel analogues for the apply family of functions.

```
parLapply(cl, list(1, 2, 3), sqrt)

## [[1]]
## [1] 1
##
## [[2]]
## [1] 1.414
##
## [[3]]
## [1] 1.732
```

3.3 Random number generation

If the calculations that you are parallelizing involve random number generation (as they often will), you will need to explicilty set up your random number generators so that the random numbers used by the different cores are independent. This can be handled fairly easily through package doRNG, which generates independent, reproducible random number chains for each core:

```
library(doRNG)

# Set the random number seed manually before calling foreach
set.seed(123)

# Replace %dopar% with %dorng%
rand1 <- foreach(i = 1:5) %dorng% runif(3)

# Or set seed using .options.RNG option in foreach
rand2 <- foreach(i = 1:5, .options.RNG = 123) %dorng% runif(3)

# The two sets of random numbers are identical (i.e. reproducible)
identical(rand1, rand2)

## [1] TRUE
```

3.4 Task-specific packages

These packages can take advantage of a registered parallel backend without needing `foreach`:

- `caret` – classification and regression training; cross-validation, etc; will automatically paralellize if a backend is registered.
- `bugsparallel` – provides tools for running parallel MCMC chains in WinBUGS using `R2WinBugs` (still active?).
- `dclone` – MCMC methods for maximum likelihood estimation, running BUGS chains in parallel.
- `pls` – partial least squares and principal component regression – built-in cross-validation tools can take advantage multicore by setting options.
- `plyr` – data manipulation and apply-like functions; can set options to run in parallel.

3.5 Caveats and Warnings

- There is communication overhead to setting up cluster – not worth it for simple problems.

- Error handling – default is to stop if one of the taks produces an error, but you lose the output from any tasks that completed successfully; use `.errorhandling` option in `foreach` to control how errors should be treated.
- Can use 'Performance' tab of the Windows Task Manager to double check that things are working correctly (should see CPU usage on the desired number of cores).
- Shutting down cluster – when you're done, be sure to close the parallel backend using `stopCluster(cl)`; otherwise you can run into problems later.

3.6 A (slightly) more substantial example

Returning to the tree data you've been working with, say we have girth and volume measurements for 100 species of trees, instead of just three, and we want to fit a linear regression for each species. This can still be done fairly quickly using a `for` loop on a single core, we can save a few seconds by running it in parallel.

```
# Generate fake tree data set with 100 observations for 100 species
tree.df <- data.frame(species = rep(c(1:100), each = 100), girth = runif(10000,
    7, 40))
tree.df$volume <- tree.df$species/10 + 5 * tree.df$girth + rnorm(10000, 0, 3)

# Extract species IDs to iterate over
species <- unique(tree.df$species)

# Run foreach loop and store results in fits object
fits <- foreach(i = species, .combine = rbind) %dopar% {
    sp <- subset(tree.df, subset = species == i)
    fit <- lm(volume ~ girth, data = sp)
    return(c(i, fit$coefficients))
}

head(fits)

##            (Intercept) girth
## result.1 1     0.09479 4.998
## result.2 2     0.79043 4.980
## result.3 3     0.06980 5.009
## result.4 4     0.25487 4.996
## result.5 5    -0.59531 5.055
## result.6 6     1.00723 4.984

# What if we want all of the info from the lm object? Change .combine
fullfits <- foreach(i = species) %dopar% {
    sp <- subset(tree.df, subset = species == i)
    fit <- lm(volume ~ girth, data = sp)
```

```
    return(fit)
}

attributes(fullfits[[1]])

## $names
##  [1] "coefficients"  "residuals"     "effects"       "rank"
##  [5] "fitted.values" "assign"        "qr"            "df.residual"
##  [9] "xlevels"       "call"          "terms"         "model"
##
## $class
## [1] "lm"
```

4 Less embarrassing parallel problems

Multicore computing is also useful for carrying out single, large computations (e.g. inverting very large matrices, fitting linear models with 'Big Data'). In these cases, the cores are working together to carry out a single computation and the thus need to communicate (i.e. not 'embarrassingly parallel' anymore). This type of parallel computation is considerably more difficult, but there are some packages that do most of the heavy lifting for you:

- HiPLAR – High Performance Linear Algebra in R – automaticaly replaces default matrix commands with multicore computations; Linux only, installation not necessarily straightforward.
- pbdR – programming with big data in R – multicore matrix algebra and statistics; available for all OS, with potentially tractable install. Also has extensive introduction manual, "Speaking R with a parallel accent."

5 Additional Resources

This overview has covered a very thin slice of the tools available, both within the above packages and in R more broadly. The help pages and vignettes for the above packages are very useful and provide additional details and examples. The CRAN task view on parallel computing is also a good resource for digging into the breadth of tools available:
`http://cran.r-project.org/web/views/HighPerformanceComputing.html`