---

title: “foreach_slides” author: “Henry Scharf” date: “November 7, 2014” output: html_document: highlight: tango keep_md: yes theme: readable toc: yes

THOUGHTS:

- this example is a lot of reps of teeny calculations. Might see better speed up if the reps were larger things. MCMC?
- need something more compelling for `iterators`
- more complicated example for `.combine`? mean?
- do their own code? fix some code?
- images for heavy analysis?
- model search? random forest? svm? other ML topics?
- references
- dorng in my bootstrap?

Why?

Embarassingly parallel tasks

- computational tasks which involve many, separate, independently executable calculations
- common statistical examples of embarassingly parallel processes:
  - bootstrapping
  - cross-validation
  - simulating independent random variables (`dorng`)
- non-parallel processes:
  - MCMC algorithms
  - stepwise model selection (e.g.: `step()`)
- For loops that do not explicitly involve dependent calculations are wasteful, especially if we have multiple processors available
- Perhaps even worse, the time cost of using a wasteful approach can put some useful statistical tools beyond our reach!

Options in R

- changing from a for loop to one of the `apply()` functions can help, but still doesn’t use multiple processors
- `parallel` package (thanks, Miranda!)
- don’t use R

Why foreach?

We would like to find a way to make use of our whole computer, and make useful tasks like bootstrapping available to use, but without having to invest large amounts of time in learning new programming languages. Enter `foreach`, which keeps the structure of a for loop, but allows us to drop two key assumptions:

- sequentiality
- single processor architecture

Goal: transform a traditional for loop into a foreach loop

We will begin with a simple chunk of R code involving a for loop, and transform it. Along the way, we'll take a look at the equivalent computation done with an `apply()` function, and see that using foreach and multiple processors outperforms this too.

Bootstrapping with a for loop

We now suppose that we are conducting an analysis wherein we'd like to fit a linear regression, then obtain estimates of standard errors for each slope parameter through bootstrapping.

Here's some familiar code:

```
data(airquality)
dim(airquality)
```

```
## [1] 153   6
```

```
names(airquality)
```

```
## [1] "Ozone"   "Solar.R" "Wind"    "Temp"    "Month"   "Day"
```

```
N <- 10^2
## make a function to be used in all three instances
boot.func <- function(i) {
    train.set <- sample(1:153, size = 153, replace = T)
    fit <- lm(Ozone ~ .^2, data = airquality[train.set, ])
    coef.mse <- summary(fit)$coef[, 2]
    return(coef.mse)
}
## 'for' version
coef.mse.for <- NULL
system.time(
    for (i in 1:N) {
        coef.mse.for <- cbind(coef.mse.for, boot.func(i))
    }
    )[3]
```

```
## elapsed
##   0.376
```

Bootstrapping with an apply function

If you're good with `apply()` functions you might upgrade to

```
## apply version
system.time(
    coef.mse.sapply <- sapply(X = 1:N, FUN = boot.func)
    )[3]
```

```
## elapsed
##   0.358
```

Why do we think we can do better?

A fun anecdote. There's a common experience among kids when they first learn to use a handsaw. Perhaps a young girl is attempting to cut a 2 x 4 in half, under the watchful eye of her grandfather. Likley, the pair are constructing a treehouse or some other charming project. As the girl grapples with her grandpa's tool, one made for a full grown adult, she begins to move the teeth of the saw back and forth, and in an effort to maintain control of the cumbersome object moves the saw maybe an inch or two in each direction, making steady but meager progress. After a minute or so of this, the grandpa inevitably interjects: "I bought the *whole* saw!"

Neither of the first two methods take advantage of multiple processors. While `apply()` functions avoid the inherently sluggish nature of for loops in `R`, they are still ignorant of the processor structure. We want to chop the job into halves, fourths, etc. and use the *whole* computer!

Bootstrapping with a foreach loop

Here is the same computation written with a `foreach()` loop

```
## foreach version
library(foreach)
```

```
## foreach: simple, scalable parallel programming from Revolution Analytics
## Use Revolution R for scalability, fault tolerance and more.
## http://www.revolutionanalytics.com
```

```
library(doRNG)
```

```
## Loading required package: rngtools
## Loading required package: pkgmaker
## Loading required package: registry
```

```
library(doParallel)
```

```
## Loading required package: iterators
## Loading required package: parallel
```

```
registerDoParallel(cores = 4) ## multi-threading issues with cores = 2
system.time(
    coef.mse.foreach <- foreach(i = 1:N,
                                .inorder = FALSE,
                                .combine = 'cbind',
                                .options.RNG = 1985) %dorng% boot.func(i)
    )[3]
```

```
## elapsed
##   0.434
```

Components of a foreach loop

- `%do%` and `%dopar%`
- arguments to consider
  - `.combine` can take on the intuitive values `c`, `cbind`,
  - `.inorder`

iterators

Sometimes the list or vector that we are iterating over (in the above case, the vector 1:N) can be a very large object. In our case, the vector is quite reasonable in size, but perhaps the object we were iterating over was a multi-dimensional object, with many values, and a high level of precision. In this case, we'd be storing a massive object, which could potentiall fill up our useable memory and slow things down. We could save memory by only keeping the piece of our list we need for a given calculation, and dumping the rest. This is the idea behind the `iterators` package in R.

Our same example with an iterator instead of a list