Introduction to Parallel Computation in R*

Brenton Kenkel

May 6, 2015

1 Introduction

Political science research becomes more computationally intensive every year. Not only do we have Big Data™ to deal with—we've also gotten more sophisticated in how we draw inferences from small data. We shoot ourselves in the foot whenever we don't use all the computational resources at our disposal. At best, we spend a day waiting for an analysis to run when it should have taken only an hour. At worst, we walk away from good research projects by mistakenly convincing ourselves that certain analyses are computationally infeasible.

This workshop is a baby step toward ensuring that we're using the available resources as best as possible. Its goals are threefold:

1. To show how easy it is to rewrite R code to take advantage of parallel processing.
2. To show that parallel processing can lead to substantial speed improvements.
3. To give a rough outline of the conditions under which those speed improvements are most substantial.

2 Running Example

As a running example throughout the workshop, we'll be replicating part of the analysis in Josh Clinton's article "Representation in Congress: Constituents and Roll Calls in the 106th House" (*Journal of Politics*, 2006). The paper is concerned with whether (and how) the preferences of the constituents in a Congress member's district affect her vote choices. The part of the analysis we'll be replicating is summarized in Table 4 of the paper. The analysis consists of running more than 800 logistic regression models, each with a different roll call vote as the dependent variable, on three covariates: an indicator for the member's party ID, the average ideology of the member's same-party constituents, and the average ideology of the member's different-party constituents. The goal of the analysis is to track the number of models in which each covariate is statistically significant.

You can download a cleaned-up version of Josh's data from my website in R data format:

TABLE 4 Frequency of Significant Covariation on Individual Roll Calls

	Pooled	Rep.	Dem.
Same-Party Only	75	145	70
Nonsame-Party Only	89	82	217
Party Indicator Only	103		
Same-Party & Party Indicator	47		
Nonsame-Party & Party Indicator	136		
Same-Party & Nonsame-Party	62	74	146
Same-Party, Nonsame-Party & Party Indicator	142		
None	219	281	233
Total	873	582	666

The numbers denote the times each covariate combination is statistically non-zero in a logistic regression predicting the probability of voting yea on a roll call for which agreement was less than 97.5%. Party cohesion results in fewer votes in the party subsamples. Survey-based measurement error is ignored.

Figure 1. Table 4 from Clinton 2006, summarizing the analysis we'll be partially replicating.

```
load(url("http://bkenkel.com/data/clinton-jop.rda"))
```

The data file contains two objects, `house106` and `votes106`. `house106` is a 432×3 data frame containing legislator characteristics—the independent variables in each of our logistic regression models.

```
str(house106)
```

```
## 'data.frame':    432 obs. of  3 variables:
##  $ republican   : num  1 1 1 1 0 1 0 1 1 0 ...
##  $ prefs_same   : num  0.25628 0.35948 0.20438 0.18065 -0.00476 ...
##  $ prefs_nonsame: num  0.0352 0.1111 -0.0219 -0.0129 0.3429 ...
```

`votes106` is a 432×852 roll call matrix, where the ij'th entry gives legislator i's vote on bill j.[1]

```
str(votes106)
```

```
##  num [1:432, 1:852] 1 1 1 1 1 1 0 1 1 1 ...
```

[1]To avoid being derailed by spurious warnings during the workshop, I removed votes for which there was separation in the corresponding logistic regression, so the number of votes analyzed is slightly smaller than in the original paper.

So we're going to run 852 logits: one for each column of `votes106`, each with the three variables in `house106` as regressors. As an example, let's do that with the first vote recorded in `votes106`.

```
## Substitute given vote into member data
house106$vote <- votes106[, 1]

## Run logit of vote choice on member attributes
logit_fit <- glm(vote ~ republican + prefs_same + prefs_nonsame,
                 data = house106,
                 family = binomial(link = "logit"))
```

Suppose we want to track the statistical significance of the Republican indicator. We can extract the p-value from the `summary()` of our fitted model:

```
summary(logit_fit)
```

```
##
## Call:
## glm(formula = vote ~ republican + prefs_same + prefs_nonsame,
##     family = binomial(link = "logit"), data = house106)
##
## Deviance Residuals:
##    Min      1Q  Median      3Q     Max
## -2.713   0.237   0.283   0.334   0.581
##
## Coefficients:
##               Estimate Std. Error z value Pr(>|z|)
## (Intercept)      1.855      0.705    2.63   0.0085
## republican       2.232      1.576    1.42   0.1567
## prefs_same      -2.038      2.953   -0.69   0.4902
## prefs_nonsame    5.452      2.842    1.92   0.0551
##
## (Dispersion parameter for binomial family taken to be 1)
##
##     Null deviance: 146.54  on 396  degrees of freedom
## Residual deviance: 142.37  on 393  degrees of freedom
##   (35 observations deleted due to missingness)
## AIC: 150.4
##
## Number of Fisher Scoring iterations: 6
```

```
p_vals <- summary(logit_fit)$coef[, 4]
p_vals[2]
```

```
## republican
##    0.15669
```

What we'll be doing in this workshop is to repeat this process 852 times—and to use parallel processing to do so as efficiently as possible.

3 Preliminaries

You'll need to have the `microbenchmark`, `foreach`, and `doParallel` packages installed to follow along with the code I'm running.

```
install.packages(c("microbenchmark", "foreach", "doParallel"))
```

You should probably also check that you're not running an outdated version of R (3.0 or later ought to be fine). Before we get into parallel processing, let's briefly review *writing functions*, *looping*, and *benchmarking* in R.

3.1 Functions

One of the basic principles of programming is to repeat yourself as little as possible. If you're going to perform the same operation repeatedly, you should write up the routine as a function. In that vein, let's take the code we used to find the p-value on Republican in the first vote, and generalize it to find the p-value on any coefficient in any vote.

```
pval_by_vote <- function(vote, coefficient) {
    ## Substitute given vote into member data
    house106$vote <- votes106[, vote]

    ## Run logit of vote choice on member attributes
    logit_fit <- glm(vote ~ republican + prefs_same + prefs_nonsame,
                     data = house106,
                     family = binomial(link = "logit"))

    ## Retrieve p-value
    p_vals <- summary(logit_fit)$coef[, 4]
    return(p_vals[coefficient])
}
```

Now let's apply this to the example we already did, and make sure it gives us the same result.

```
pval_by_vote(vote = 1, coefficient = 2)
```

```
## republican
##    0.15669
```

3.2 Loops

The easiest way to repeat the same operation on different data in R is through a *for loop*. For example, suppose we wanted to gather the p-values on Republican for the first five votes in the data. First, we'd set up a vector to store our results.

```
p <- rep(NA, 5)
```

Then, we'd use `for` to loop through the first five votes, calculating the p-value for each one using our `pval_by_vote()` function.[2]

```
for (i in 1:5) {
    p[i] <- pval_by_vote(vote = i, coefficient = 2)
}
```

We can then calculate in how many cases the Republican indicator was statistically significant at the conventional level.

```
p
```

```
## [1] 1.5669e-01 8.7264e-04 2.8046e-04 2.5317e-06 5.2281e-03
```

```
sum(p < 0.05)
```

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

For the purposes of comparing `for` loops to other methods, it will be convenient to make a function to calculate p-values for a range of votes via a `for` loop.

```
pvals_for <- function(votes, coefficient) {
    p <- rep(NA, length(votes))
    for (i in 1:length(votes)) {
        p[i] <- pval_by_vote(vote = votes[i], coefficient = coefficient)
    }
    return(p)
}
```

Once again, let's verify that this gives us the same result as before.

```
pvals_for(1:5, 2)
```

```
## [1] 1.5669e-01 8.7264e-04 2.8046e-04 2.5317e-06 5.2281e-03
```

[2] A cleaner, but no faster, way to accomplish the same end result would be to run `sapply(1:5, pval_by_vote, coefficient = 2)`. The `*apply` family of functions is beyond the scope of this workshop.

3.3 Benchmarking

The goal of this workshop is to help you run code faster. To know whether you're making improvements, you need to be able to track your execution speed. The easiest way to do that, though somewhat *ad hoc*, is the `system.time()` function. For example, let's see how long it takes to get the *p*-values on Republican for the first five votes.

```
system.time(pvals_for(1:5, 2))
```

```
##    user  system elapsed
##   0.016   0.002   0.018
```

The number you should be interested in is the third value, the `elapsed` time. This represents the "wall time," or how long the operation took from the perspective of the clock on the wall. (This is an important distinction when we start working in parallel, when an operation can take a lot of processor time but very little wall time.)

For a more systematic look at the time an operation takes, and to be able to compare operations easily, I use the `microbenchmark` package. It runs a given operation 100 times (by default) and displays summary statistics about how long it took. As a kind of toy example, let's use `microbenchmark` to compare grabbing the first five *p*-values to grabbing the first ten.

```
library("microbenchmark")
microbenchmark(five = pvals_for(1:5, 2),
               ten = pvals_for(1:10, 2))
```

```
## Unit: milliseconds
##  expr    min     lq   mean median     uq    max neval cld
##  five 15.060 16.227 18.150 17.041 17.453 49.031   100  a
##   ten 36.564 37.849 41.241 39.203 40.142 72.810   100   b
```

It should come as no shock that the latter operation takes about twice as long.

4 From `for` to `foreach`

When you run a typical `for` loop, R executes it strictly in sequence: the second calculation in the loop doesn't start until the first one finishes, the third doesn't start until the second finishes, and so on. When you have more than one processor available, this is kind of a waste. What we'd like to do is automatically distribute tasks across processors, running them in parallel while using all of the computing power available to us. The first step to doing this in R is to translate `for` loops into `foreach` loops.

```
library("foreach")
```

There are two main differences between `for` and `foreach` loops. The first, simpler difference is syntax. You can translate almost any `for` loop into a `foreach` loop by replacing `for (i in values)` with `foreach (i = values) %do%`. The second difference is in the output. Before using a `for` loop, you have to set up a vector (or matrix, or other appropriate object) to store the output of each computation. Not so with `foreach`. When `foreach` finishes running, it returns a list containing the output of the expression evaluated in each iteration of the loop.

To see `foreach` in action, let's translate our earlier loop over `pval_by_vote()`:

```
p1 <- foreach (i = 1:5) %do% {
    pval_by_vote(vote = i, coefficient = 2)
}
p1
```

```
## [[1]]
## republican
##    0.15669
##
## [[2]]
## republican
## 0.00087264
##
## [[3]]
## republican
## 0.00028046
##
## [[4]]
## republican
## 2.5317e-06
##
## [[5]]
## republican
##  0.0052281
```

Notice that instead of assigning `p1[i] <- ...` within the loop, as we would have in a `for` loop, we just assign the whole output of the loop to `p1`.[3] You might also notice that the output is a list instead of a vector. That's a bit inconvenient, but we can fix it via the `.combine` argument to `foreach`:

```
p2 <- foreach (i = 1:5, .combine = "c") %do% {
    pval_by_vote(vote = i, coefficient = 2)
}
p2
```

[3] The downside is that you can't access previous computations from within the loop. In other words, the i'th calculation can't depend on the results of the $i-1$'th. Of course, if your goal is to run the calculations in parallel, that would have to be the case anyway.

```
## republican republican republican republican republican
## 1.5669e-01 8.7264e-04 2.8046e-04 2.5317e-06 5.2281e-03
```

If you're building a matrix rather than a vector, you can use `.combine = "rbind"` or `.combine = "cbind"` as appropriate. `foreach` has many more useful arguments that I won't get into here, but which you can read about in the package documentation.

As we did with our `for` loop method, let's turn this into a function that computes *p*-values for a range of votes using `foreach`/`%do%`:

```
pvals_do <- function(votes, coefficient) {
    foreach (i = votes, .combine = "c") %do% {
        pval_by_vote(vote = i, coefficient = coefficient)
    }
}
pvals_do(1:5, 2)
```

```
## republican republican republican republican republican
## 1.5669e-01 8.7264e-04 2.8046e-04 2.5317e-06 5.2281e-03
```

We can now use `microbenchmark()` for a side-by-side comparison of the `for` and `foreach` methods:

```
microbenchmark(for_loop = pvals_for(1:5, 2),
               do = pvals_do(1:5, 2))
```

```
## Unit: milliseconds
##      expr    min     lq   mean median     uq    max neval cld
##  for_loop 15.459 16.731 18.111 17.496 18.477 60.093   100  a
##        do 18.366 19.729 21.065 20.837 21.476 58.674   100   b
```

I told you I'd help make your code faster ... but this seems to be no different, or even a smidge slower! That's because we still have yet to harness the connection between `foreach` and parallel processing.

5 Local Parallelization

Once you've translated a `for` loop into a `foreach` loop, it's pretty easy to get it running in parallel. The first thing you have to do is replace `%do%` with `%dopar%` (i.e., "do in parallel"):

```
foreach (i = 1:5, .combine = "c") %dopar% {
    pval_by_vote(vote = i, coefficient = 2)
}
```

```
## Warning: executing %dopar% sequentially: no parallel backend registered
```

```
## republican republican republican republican republican
## 1.5669e-01 8.7264e-04 2.8046e-04 2.5317e-06 5.2281e-03
```

If that's all you do, `foreach` will issue the warning above, telling you that you've asked for parallel but it's running sequentially. To actually run in parallel, you need to register a *parallel backend*. There are a number of different backends available, but we're going to use one that has the advantage of working on Windows machines as well as Mac and Linux.[4] We'll be using the `doParallel` package for this.

```
library("doParallel")
```

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

Before deciding how many cores to use, you should know how many cores you have. Given that even most smartphones now have dual-core processors, your laptop probably has at least two processing cores available. To find out, use the `detectCores()` function:

```
## Results will vary across machines
detectCores()
```

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

Now we'll set up a local cluster to use for parallel processing, and then register it as the backend to `%dopar%`. I'm going to just use two cores—go ahead and use more if you have more cores available.

```
cl <- makePSOCKcluster(2)
registerDoParallel(cl)
```

When you're done using the cluster, make sure to run `stopCluster(cl)`. Otherwise you'll have lingering R processes using up resources on your machine.

If you were to try to use `%dopar%` at this point, it would probably throw an error. That's because the R processes we've spawned on our local cluster don't have our data sets or the functions we've defined to operate on them. To populate the cluster instances of R with the objects in your R session, use the `clusterExport()` function:

```
clusterExport(cl, varlist = ls())
```

Now we're ready to run our loop in parallel.

[4] Caveat: I don't have a Windows machine available to test with, so I apologize if unanticipated issues crop up.

```
foreach (i = 1:5, .combine = "c") %dopar% {
    pval_by_vote(vote = i, coefficient = 2)
}
```

```
## republican republican republican republican republican
## 1.5669e-01 8.7264e-04 2.8046e-04 2.5317e-06 5.2281e-03
```

Notice that there was no warning this time, since we had a parallel backend registered. For the purposes of speed comparisons, let's define a function for grabbing p-values in parallel for a range of votes. This is exactly the same as `pvals_do()`, just replacing `%do%` with `%dopar%`:

```
pvals_dopar <- function(votes, coefficient) {
    foreach (i = votes, .combine = "c") %dopar% {
        pval_by_vote(vote = i, coefficient = coefficient)
    }
}
```

First let's see how our three methods stack up against each other for a small task, like getting the first five p-values.

```
microbenchmark(for_loop = pvals_for(1:5, 2),
               do = pvals_do(1:5, 2),
               dopar = pvals_dopar(1:5, 2))
```

```
## Unit: milliseconds
##      expr    min     lq   mean median     uq    max neval cld
##  for_loop 15.638 17.238 19.238 18.066 19.137 62.443   100  b
##        do 18.809 20.466 22.150 21.495 22.800 62.803   100   c
##     dopar 14.370 15.339 16.307 15.982 16.608 29.484   100 a
```

There's no real advantage to parallel execution for a tiny task like this. The efficiency gains from executing in parallel are swamped by the additional overhead required to collect results from across different R processes. We really start seeing benefits when we run a more significant task—like collecting the p-values on the Republican indicator for the full set of 852 votes in the sample, as in the original analysis.

```
cols <- 1:ncol(votes106)
system.time(p_do <- pvals_do(cols, 2))
```

```
##    user  system elapsed
##   3.667   0.029   3.698
```

```
system.time(p_dopar <- pvals_dopar(cols, 2))
```

```
##    user  system elapsed
##   0.254   0.022   2.035
```

The parallel version finishes in about 60% the time it takes the sequential version. Not a big deal when you're talking about 3 seconds versus 5 seconds, but pretty good for 3 hours versus 5 hours, or 3 days versus 5 days. And that's just with using one additional core! We can confirm that both methods come up with the same results:

```
all.equal(p_do, p_dopar)
```

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

And, finally, we can get our answer: for how many votes is the coefficient on the Republican indicator statistically significant?

```
sum(p_dopar < 0.05)
```

```
## [1] 428
```

What we've done so far is execute a lot of tiny tasks—852 logistic regressions—in parallel. What if we instead parallelized a few bigger tasks? For example, let's grab the p-values for each of the four coefficients in the analysis. We'll compare doing the whole thing sequentially, versus splitting it up in parallel by coefficient.[5]

```
system.time(for (j in 1:4) pvals_for(cols, j))
```

```
##    user  system elapsed
##  14.145   0.190  14.397
```

```
system.time(foreach (j = 1:4) %dopar% pvals_for(cols, j))
```

```
##    user  system elapsed
##   0.014   0.001   7.187
```

Notice that we're pretty close to the efficiency frontier here—the biggest improvement we could hope for is 50%, and that's just about what we've got.

```
stopCluster(cl)
```

5.1 For Mac and Linux Users: An Easier Way

If you're on a non-Windows machine, local parallelization is even easier than this, thanks to the `doMC` package. With `doMC`, you don't have to explicitly set up a cluster or export objects—just tell it how many cores you want, and go for it, as in the following example:

[5] This isn't the most efficient way to perform this particular job, since we really only need to run each logit once. It's just to illustrate how the efficiency gains grow with the size of the task being distributed.

```r
library("doMC")
registerDoMC(2)  # Number of cores to use

foreach (i = 1:5, .combine = "c") %dopar% {
    pval_by_vote(vote = i, coefficient = 2)
}
```

6 Applications

Parallel processing is useful for any task that meets the following conditions: (1) the task requires many similar calculations, and (2) each calculation does not depend on the results of previous calculations. Examples that come to mind include:

- Monte Carlo simulations
  - If each iteration is long: parallelize across iterations
  - If each iteration is short: parallelize across parameters
- Simulation-based statistics
  - Bootstrap
  - Permutation inference
  - Cross-validation (leave-one-out or k-fold)
- Ensemble estimators (e.g., random forests)
- Numerical solutions to formal models (parallelize across parameters)

Two computationally intense classes of problems that fail the second criterion (independence of calculations) are Markov Chain Monte Carlo and agent-based models. However, even for problems like these, you can parallelize across different sets of initial values.

7 Appendix: Remote Parallelization

You can use many more processors if you use a supercomputer, such as the cluster available through ACCRE at Vanderbilt. In terms of your R code, writing a script to parallelize on ACCRE doesn't look much different at all from what you would write to parallelize locally. Here's a self-enclosed script for running the Clinton 2006 analysis in parallel via MPI and the `doMPI` package on ACCRE:

```r
load(url("http://bkenkel.com/data/clinton-jop.rda"))

library("foreach")
library("doMPI")

cl <- startMPIcluster()
```

```
registerDoMPI(cl)

pval_by_vote <- function(vote, coefficient) {
    ## Substitute given vote into member data
    house106$vote <- votes106[, vote]

    ## Run logit of vote choice on member attributes
    logit_fit <- glm(vote ~ republican + prefs_same + prefs_nonsame,
                     data = house106,
                     family = binomial(link = "logit"))

    ## Retrieve p-value
    p_vals <- summary(logit_fit)$coef[, 4]
    return(p_vals[coefficient])
}

## Multiple-coefficient analysis (a few big jobs)
print(system.time(
    results0 <- foreach (j = 1:4) %do% {
        foreach (i = 1:ncol(votes106), .combine = "c") %do% {
            pval_by_vote(vote = i, coefficient = j)
        }
    }
))

print(system.time(
    results1 <- foreach (j = 1:4) %dopar% {
        foreach (i = 1:ncol(votes106), .combine = "c") %do% {
            pval_by_vote(vote = i, coefficient = j)
        }
    }
))

save(results0, results1, file = "clinton-2006-results.rda")

closeCluster(cl)
mpi.quit()
```

Suppose this script is named `run.r`. To schedule the job on the ACCRE cluster, you need a Bash script file that instructs the scheduler how many processors you need, how long the job will take, and so on. I use a script that looks like the following one:

```
#!/bin/bash
#SBATCH --mail-user=email@domain.com
#SBATCH --mail-type=ALL
#SBATCH --nodes=4
#SBATCH --ntasks=4
```

```
#SBATCH --time=1:00:00
#SBATCH --mem-per-cpu=250M
#SBATCH --output=clinton-2006.out

setpkgs -a gcc_compiler
setpkgs -a R_3.1.1
setpkgs -a openmpi_gcc
setpkgs -a mpiexec

mpirun Rscript /home/kenkelb/parallel-r-workshop/run.r
```

For detailed information on how to run scripts on the ACCRE cluster, see the ACCRE documentation on "Getting Started" and the SLURM documentation at `http://www.accre.vanderbilt.edu`.