Parallel Programming in R

Isaac J. Michaud

Overview

Parallel programming is a vital computational tool that every statistics graduate student needs to utilize. Parallel programming can be complicated, but with just a few extra lines of code, I will show you how you can complete a simulation study in a fraction of the time. Simulation studies typically involve generating many synthetic data sets and applying a statistical method to them. Because the data sets are independent, we can easily divide up the entire set of replication and run each chunk on a different computer. A simulation study is an example of an *embarrassingly parallel* problem. In theory, if we have 100 computers available to use then our code will run in 1/100th of the time compared to a single computer.

Note: Because parallel computing is system and hardware specific some of the code in this Rmarkdown file will not execute correctly on all machines. The document was produced on OSX 10.11.6, but should execute on any Linux/Unix based system. Most code will run on Windows and I have indicated a few places where this will not be the case. In the future, I hope to write a more detailed version specifically for Windows.

Just Don't: BOM Cluster

Let's admit it, we have all done this: your final project is due at 8 am tomorrow morning, and you need to get 20,000 simulations done in the next hour while you grab dinner. You manually divide the task, run it on four computers, and paste the results together. This approach to parallelism is quick, dirty, and a bad habit. Here are some reasons why you should avoid making a BOM cluster:

- Inefficient use of resources (one simulation running on an eight-core machine)
- Vulnerable to disruption (power and people)
- Heterogeneous hardware and software versions
- Random number seeds become difficult to manage
- Manually intensive (leads to mistakes)
- **Not reproducible**

Parallel programming is annoying because it takes upfront effort, but it pays dividends.

Parallel Programming on a Laptop

We will start with how to utilize the multiple cores within your laptop's processor to run parallel code. Once we know how to do that, scaling up to the Department's cluster is easy. Let's consider the following simulation code:

```r
my_simulation <- function(x) { #STAND BACK! We are simulating!
  return(rnorm(1,x,1))
}
```

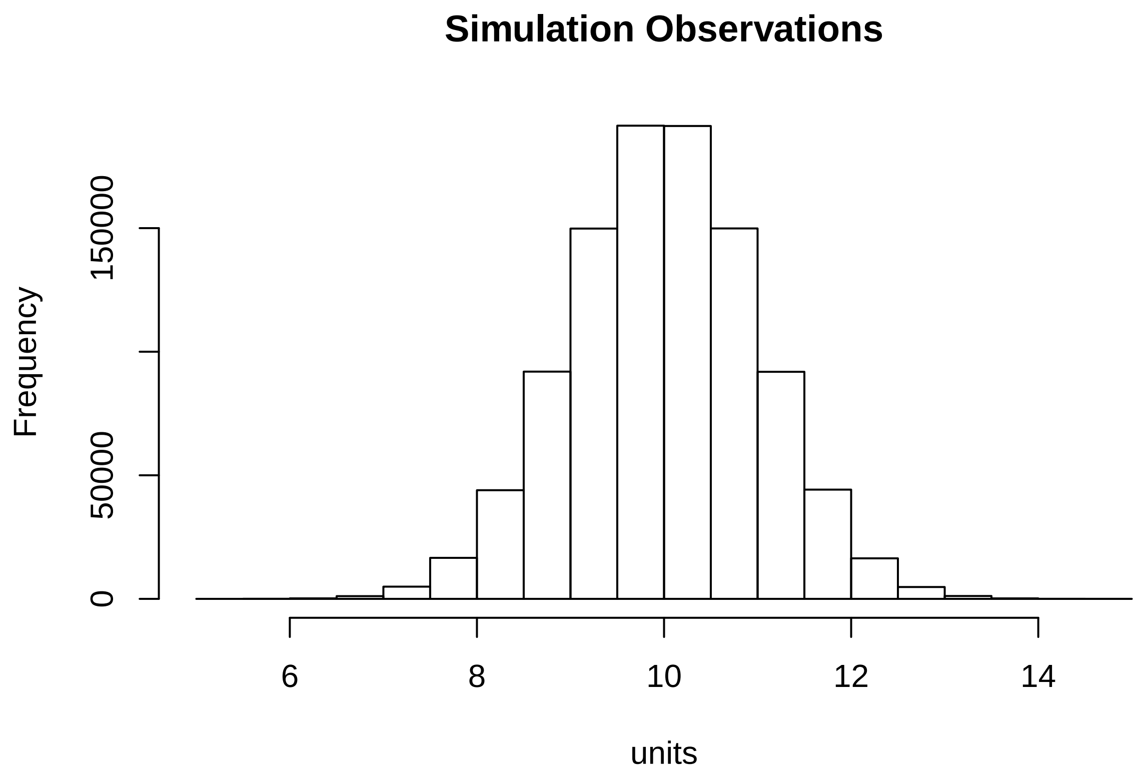
The simulation study we would like to perform is the following simple `for` loop that repeatedly calls our simulation code. This code could be made more efficient by vectorizing the operations, but this format make it a little easier to understand what is happening.

```r
set.seed(123)
start_time  <- proc.time()
N           <- 1e6
p           <- 10
result      <- rep(0,N)

for (i in 1:N) { #simple simulation study
  result[i] <- my_simulation(p)
}

print(proc.time() - start_time)
```

```
##    user  system elapsed
##   4.014   0.165   4.404
```

The simulation is run one million times, takes almost 4 seconds, and produces the following observations:

```r
hist(result,main="Simulation Observations",xlab ="units")
```



Realistically, your simulation study will be faced with the following three slow downs:

- Many replications
- Many parameters
- `my_simulation` takes hours, for example MCMC

Parallelizing the entire simulation study will allow us to bring all of our computing resources to bear on the most challenging problems.

Using lapply

The key feature of our simulation study was that the `for` loop doesn't need to be performed sequentially, its purpose is to run the same code block repeatedly. We can perform the same function with R's many `apply` functions. The `lapply` function allows you to repeatedly evaluate some code and get a list back.

```
set.seed(123)
lapply(c(10,10,10),my_simulation)
```

```
## [[1]]
## [1] 9.439524
##
## [[2]]
## [1] 9.769823
##
## [[3]]
## [1] 11.55871
```

`lapply` can be thought of as a `for` loop where the guts of the loop are written as a function and the index set is passed to `lapply` as an argument. A similar function is `sapply` which simplifies the returning list to an array if your function has scalar output. Performing our simulation study with a `sapply` statement shortens the code, but doesn't fix our efficiency problems.

```
set.seed(123)
start_time  <- proc.time()
result      <- rep(0,N)
result      <- sapply(rep(10,N),my_simulation)
print(proc.time() - start_time)
```

```
##    user  system elapsed
##   4.587   0.217   5.196
```

`lapply` and `sapply` are useful because any `for` loop that can be converted into one of these `ply` statements is parallelizable. If you cannot write your loops with `ply` statements then you are probably updating variables on each iteration which requires sequential execution (for example MCMC). Such cases either cannot be parallized or you have to get creative.

Multicore Programming

With your code rewritten using `ply` functions allows you to parallelize your code with the `parallel` package by replacing `lapply` with `parLapply`. We first set up a cluster. The R session creating the cluster is called the master and the cluster is a pool of workers. You can make a cluster with any number of workers you want, but making more workers than available processor cores will just cause the workers to fight for processor time and slow each other down. It is a good idea to use at least one fewer cores than you have available so that you can still use the computer while the simulation runs to write your report or watch cat videos.

```
library(parallel)
num_cores  <- detectCores() - 1
my_cluster <- makeCluster(num_cores)
```

We can then call `parLapply` with the sample simulation code and tell it the cluster we would like it to run on.

```
parLapply(my_cluster,1:3,my_simulation)
```

```
## [[1]]
## [1] 2.079011
##
```

```
## [[2]]
## [1] 2.401475
##
## [[3]]
## [1] 1.178485
```

Running the same batch of simulations in parallel shows that we get a speed increase over a single core:

```
start_time  <- proc.time()
result = parSapply(my_cluster,rep(10,N),my_simulation)
print(proc.time() - start_time)
```

```
##    user  system elapsed
##   1.315   0.132   4.089
```

We can see the cluster in the activity monitor. It shows a number of R processes that are running in the background. They operate like little consoles that will accept R commands and return the results.

```
open /Applications/Utilities/Activity\ Monitor.app
```

We can shift the workload around to make it run faster:

```
my_better_simulation <- function(x) {
  rnorm(floor((1e6)/3),x,1)
}

start_time  <- proc.time()
result = parSapply(my_cluster,rep(10,3),my_better_simulation)
print(proc.time() - start_time)
```

```
##    user  system elapsed
##   0.117   0.017   0.191
```

It's a good idea to shut down the cluster once you have finished. It is not absolutely necessary because it will timeout eventually, but closing it will release the system resources immediately.

```
stopCluster(my_cluster)
```

Exporting Data and Functions

```
set.seed(123)
my_fake_data <- matrix(rnorm(1000),10,10)

compute_row_mean <- function(i) {
  mean(my_fake_data[i,])
}

my_cluster <- makeCluster(num_cores)
```

The following code will throw an error because the data each worker needs is defined in the master's memory, which they don't have access to. By default, `makeCluster` creates a "PSOCK" cluster where each worker is separate invocation of R with clean environment.

```
parLapply(my_cluster,1:10,compute_row_mean)
```

To work around this problem we must manually export the data to the workers to operate on.

```
clusterExport(my_cluster,'my_fake_data')
parSapply(my_cluster,1:10,compute_row_mean)
```

```
##  [1]  0.046873456 -0.249768906  0.106093701  0.215668594  0.013254547
##  [6]  0.395790948  0.384644252 -0.222217996  0.001616181  0.212104309
```

```
stopCluster(my_cluster)
```

An alternative, when using Linux or MacOS, is to make a FORK cluster which spawn copies of the master process to create the workers. Any variables present in the master are then available to the workers.

```
my_cluster   <- makeCluster(num_cores,type = 'FORK')
parSapply(my_cluster,1:10,compute_row_mean)
```

```
##  [1]  0.046873456 -0.249768906  0.106093701  0.215668594  0.013254547
##  [6]  0.395790948  0.384644252 -0.222217996  0.001616181  0.212104309
```

```
stopCluster(my_cluster)
```

Reproduciblity

If we are performing many stochastic simulations we need to seed the random number generator to allow us to reproduce our results. Using `set.seed` is not enough when using a cluster because it will only set the seed in the master. Repeated evaluations of the following code will not produce the same results:

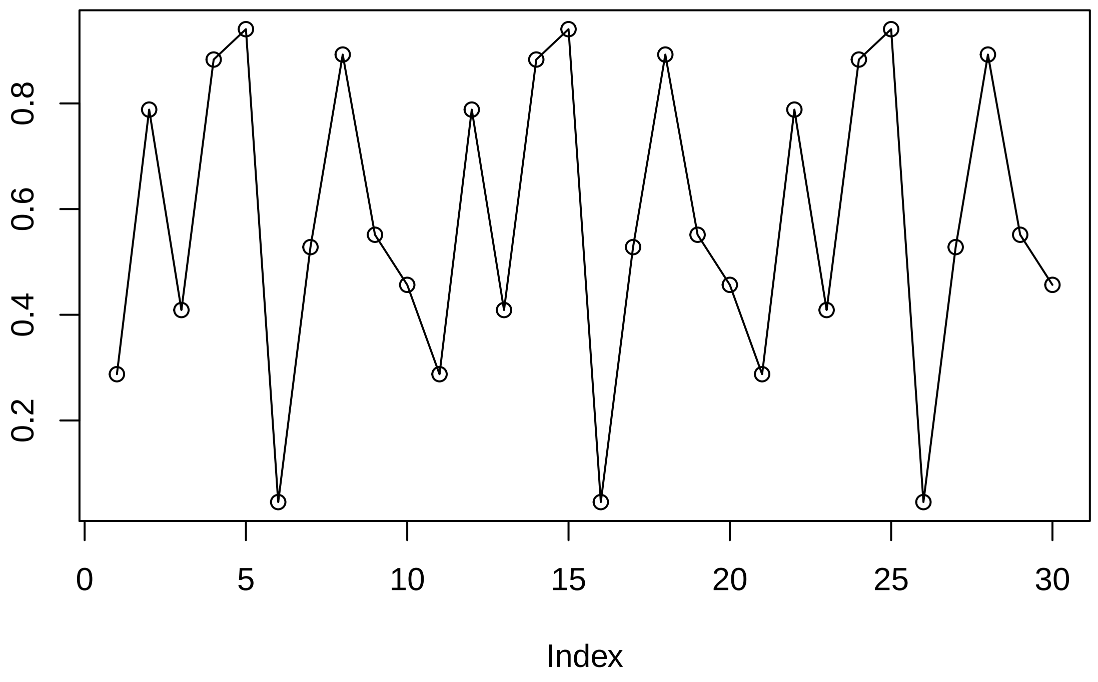
```
set.seed(123)
my_cluster <- makeCluster(num_cores)
seeds = parSapply(my_cluster,rep(1,30),function(x){runif(1)})
print(seeds)
```

```
##  [1] 0.522288018 0.494634112 0.127867582 0.348102263 0.199167423
##  [6] 0.593533084 0.776288707 0.857266723 0.575796320 0.008602140
## [11] 0.593251924 0.348070206 0.851232240 0.296646927 0.954238757
## [16] 0.009601016 0.925126767 0.874902505 0.503134885 0.527046543
## [21] 0.537757863 0.870684555 0.036094248 0.900018071 0.744760940
## [26] 0.793529966 0.602458999 0.320860794 0.622800823 0.986772227
```

```
stopCluster(my_cluster)
```

The situation is even worse if we use a FORK cluster because all of the workers will share the same seed as the master. Every worker will draw the same exact stream of pseudo-random numbers!

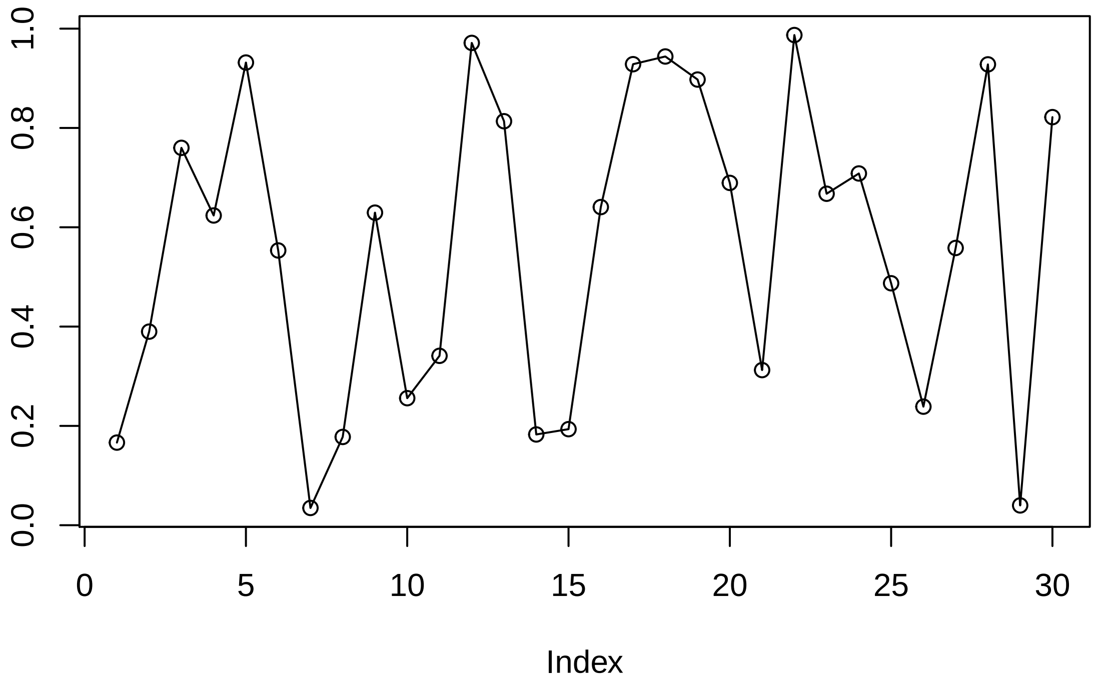
```
set.seed(123)
my_cluster <- makeCluster(num_cores,type = 'FORK')
plot(parSapply(my_cluster,rep(1,30),function(x){runif(1)}),ylab="",type='o')
```



```
stopCluster(my_cluster)
```

The `parallel` package includes the function `clusterSetRNGStream` which will handle the seeding for you. It takes a single seed and then passes seed to each of the workers that keeps them relatively independent.

```
my_cluster <- makeCluster(num_cores)
clusterSetRNGStream(cl = my_cluster, 123)
plot(parSapply(my_cluster,rep(1,30),function(x){runif(1)}),ylab="",type='o')
```



```
stopCluster(my_cluster)
```

NCSU Statistics Cluster

We are now ready to run parallel code on the Department's Beowulf cluster. You can submit jobs that use up to 20 cores, 8 gigs of memory per core, and 21 days of run time. It is a good idea to run long jobs on the cluster because you don't risk burning out or tieing up your own computer, and the cluster has a UPS so that you don't have to worry about the power going out losing your work.

So how do we use this resource? As always, you can always do the hard way. Some people will manual divide their simulation runs into seperate scripts and submit each one individually to the cluster. As with a BOM cluster, this approach leaves you responsible for pasting the results together after the runs finish. It is much better to use the methods we have already discussed and allow the code to handle the divvying.

An overview of the cluster can be found on the Department's website here NCSU Beowulf Cluster. To gain access to the cluster you will need to email Chris and request access. He will create an account using your unity login credentials. To run code on the cluster you will then need to copy your scripts to the cluster using command line tools, remotely log in using SSH, and submit the code to the Beowulf controller. The full details can be found here Cluster Instructions. After your code finishes, the cluster will send you and email and you can download the results.

Note: The cluster runs Linux, so it is easier to interact with it if you are using either Linux or MacOS because all the tools are installed by default. If you are using Windows on a personal computer you will need to install a few applications that will handle the communication.

Here is the script I am going to run on the cluster:

```r
library(parallel)

num_cores  <- 10 #must be 20 or less
my_cluster <- makeCluster(num_cores)

my_cluster_simulation <- function(x) {
  s      <- 1000
  result <- rep(0,s)
  for (i in 1:s) {
   result[i] <- mean(rexp(200))
  }
  return(result)
}
study_results <- parSapply(my_cluster,rep(0,1000),my_cluster_simulation)
save(study_results,file="cluster_simulation_demo_results.txt",ascii = TRUE)
```

Copy your code to the cluster using SCP

```
scp [program] unityid@beowulf.stat.ncsu.edu:
```

Pro Tip: Don't forget the colon at the end of hostname.

Log in to the cluster using SSH

```
ssh unityid@beowulf.stat.ncsu.edu
```

Submit code to the cluster controller

```
bwsubmit_multi 10 r sim.r
```

Copy the results back:

```
scp unityid@beowulf.stat.ncsu.edu:results.txt .
```

Note: If you don't want to use `scp`, you can also log in to the server using an FTP client using sFTP which allows you to copy file between your computer and the server with a graphical interface.

Resources

Max Gordan's Introduction to Parallel in R

Parallel Package Vignette

NCSU Statistics Beowulf Cluster

NCSU Statistics Cluster Instructions