Parallelization in R: Making your Lazy Computer Work Up to its Fullest Potential

Lecture 7

Kevin McAlister

University of Michigan

March 17, 2017

Efficiency

What is my computer actually doing?

- R is a programming language that leverage components of your computer to execute code.
- Code is nothing more than a series of commands that you CPU can interpret.
- R code is many levels above CPU code. This is why it's so easy to learn.
- How does R actually work?
  1. R passes instructions to CPU.
  2. CPU finds relevant data in memory (hard or working).
  3. CPU performs operations.
  4. CPU writes output from relevant operations back to memory.

Lazy R

- The process is pretty straightforward.
- Last week we discussed how to *implicitly* increase the efficiency of code using data.table.
- data.table makes the instructions given to the CPU more efficient.
- We can also achieve this by using apply functions, plyr, etc.
- There's another way that we can increase efficiency of operations.
- Parallelization

What is parallelization?

- CPUs have gotten much better over time.
- Most (if not all) CPUs that exist today have multiple physical *cores*.
- What is a core? A core is a physical CPU that can perform operations.
- If I tell R to take the sum of a set of values using `sum()`, R is using one physical CPU.
- Your computer generally has multiple physical CPUs. Why not tell R to use all of them?
- When utilizing multiple cores, we can do multiple operations simultaneously!
- Newer computers also have *logical* CPUs. Each physical core is now capable of doing multiple tasks (generally 2ish) simultaneously. R can operate on all logical CPUs your system has.

A Simple Example

- Lets think about taking the sum of N values.
- The time it takes to take a sum increases linearly, $O(N)$.
- A sum is an operation that we can *split*.
  - Divide the N values into 4 chunks.
  - Take the sum of each chunk.
  - Take the sum of the chunks.
- The results are identical. A sum is an *embarrassingly parallel* operation.
- Send the each sum operation to a different CPU. The sums can be taken simultaneously!
- Thus, the time needed to complete the sum is $O(\frac{N}{4})$.

A Simple Example

- The simple example actually has many important parts.
- The most important aspect of why parallelization works is that the operation can be split.
- Data can be mapped to each CPU. Each CPU can perform a portion of the operations. Each CPU can write the output from each sub-operation. The *master* node can *combine* the sub-output into the final output.
- Split/Recombine, Map/Reduce, Divide/Conquer
- These are all phrases related to operations that can be parallelized.

How do I know if a problem can be split?

- Understanding when parallelization is possible is almost more art than science.
- Look for parts of a problem which can be solved and then recombined,
- This will be more apparent in examples.
- Today I'm going to try to teach by example. We are going to create a parallel linear regression in R.

Parallelized Linear Regression

- Linear regression is an embarrassingly parallel problem.
- There are many parts of the problem that can be sped up by using a parallel architecture.
- Let's think about the steps needed to calculate the necessary components for a linear regression:
    1. Covariance Matrix: $V = (X'X)^{-1}$
    2. OLS coefficients: $B = VX'Y$
    3. Residuals: $R = Y - XB$
- With these three matrices, we can calculate everything we need for linear regression.
- Notice that B is needed for R.

Parallelized Linear Regression

- Let's think about the process for linear regression:
  1. Read in data
  2. Calculate coefficients
  3. Write coefficients
  4. Calculate Residuals
  5. Calculate Standard Errors
  6. Write standard errors
- What parts of this process can be split?

Matrix Algebra

- Consider the matrices X and Y.
- How do we calculate $X'X$?

$$X'X = \begin{bmatrix} \sum_{i=1}^{N} x_{i,1}^2 & \sum_{i=1}^{N} x_{i,1}x_{i,2} & \cdots & \sum_{i=1}^{N} x_{i,1}x_{i,P} \\ \sum_{i=1}^{N} x_{i,1}x_{i,2} & \sum_{i=1}^{N} x_{i,2}^2 & \cdots & \sum_{i=1}^{N} x_{i,2}x_{i,P} \\ \vdots & \vdots & \ddots & \vdots \\ \sum_{i=1}^{N} x_{i,1}x_{i,P} & \sum_{i=1}^{N} x_{i,2}x_{i,P} & \cdots & \sum_{i=1}^{N} x_{i,P}^2 \end{bmatrix}$$

Matrix Algebra

- What about $X'Y$?

$$X'Y = \begin{bmatrix} \sum_{i=1}^{N} x_{i,1}y_i \\ \sum_{i=1}^{N} x_{i,2}y_i \\ \vdots \\ \sum_{i=1}^{N} x_{i,P}y_i \end{bmatrix}$$

Parallelized Linear Regression

- The first steps can be split!
- Matrix multiplication is just a series of sums and multiplication.
- We can partition X and Y, calculate the corresponding sums, sum the corresponding sub-matrices, taken the inverse of $X'X$, and then multiply the two pieces together.
- Recall that calculation of the OLS coefficients scales linearly, $O(N)$.
- If we have C available CPUs and create C partitions of the data, then P-OLS Coefficients requires $O(\frac{N}{C})$ to complete.
- Let's get a function going to calculates OLS coefficients in R.

Parallelization in R

- Parallelization in R can be done in a variety of ways.
- Typically to get the computer to work a harder you must work a bit harder to arrange calculations in a parallel friendly manner. In the best circumstances somebody has already done this for you.
- (Not For Class Today) One tool that has already been created for performing simple operations in parallel in Revolution R. RevR (now a subsidiary of M$) is a distribution of R that has built in parallel BLAS/LAPACK libraries. Using this distro, simple operations and many linear algebra operations will be performed using available cores. I recommend you install this on your own time and use it.
- Many libraries exist for common functions with parallel capabilities.
- Check out the CRAN page on High Performance Computing in R.

Parallelization in R

- Many times, however, we're going to be writing our own parallel functions.
- Writing parallel functions requires intimate knowledge of the problem at hand.
- Hard to generalize.
- For this class, we are going to utilize three packages:
  - `library(doParallel)`
  - `library(foreach)`
  - `library(data.table)`
- Go ahead and install both of these packages and load them into your R session.

Parallelization in R

- By default, R does not utilize all available CPUs.
- We have to tell R to do this. Generally, this is done by registering a parallel backend.
- In other words, we're creating an internal *cluster*.
- First, let's check how many logical CPUs our computer has: `detectCores()`
- My general rule of thumb is to always leave one core available for dicking around on Facebook and whatnot. Memorize this number, C.
- We first need to make the cluster: `cl <- makeCluster(C)`
- Then we register the cluster: `registerDoParallel(cl)`
- We can check that it is actually working: `getDoParWorkers()`
- Once we're done using the cluster, we want to decouple it from R: `stopCluster(cl)`
- Now that the cluster is recognized by R, we still need to instruct R to utilize multicore algorithms.

Parallelization in R

- Once the cluster is running, we need to use some set of functions that tell R to use all of the threads.
- For now, we're going to be using `library(foreach)`.
- foreach creates parallel for loops. Each iterations of the loop is passed to an available *worker* node. Once the worker node is done, it writes the worker's output to the *master* node (i.e. your main R session).
- foreach is nice for a number of reasons. Without getting into too much detail, foreach automatically passes the current global environment to the new worker environment.
- `makeCluster()` spins up a *socket cluster*. This is the most basic of parallel architectures. Nodes in a socket cluster are designed to have a *vanilla* instance of R (i.e. no frills R).

foreach Syntax

- foreach has a special syntax:

  `for(i = 1:N, .combine = c(), .packages = c()) %dopar% fun(i,...)`
- `i = .` is the iterator. Think for loop.
- `.combine = .` tells foreach how to combine the output. We'll look at this more in examples.
- `.packages = .` is a character vector of packages to send to each worker node's R environment. Variables pass through automatically, packages do not.

A Basic Example

- Let's do a basic parallel example.

```
rm(list=ls())
library(doParallel)
library(foreach)
#Function finds the sum of the sqrt of each whole number from 1 to x
rec.sqrt <- function(x){
  sum.sqrt <- sum(sqrt(seq(1,x)))
  return(sum.sqrt)
}
#Make our cluster
cl <- makeCluster(3)
registerDoParallel(cl)
#Run function in parallel
out <- foreach(i = 1:20) %dopar% rec.sqrt(i)
#Returns list
#Run function so it returns a vector
out <- foreach(i = 1:20, .combine = 'c') %dopar% rec.sqrt(i)
#Return sum of all the output vals
out <- foreach(i = 1:20, .combine = '+') %dopar% rec.sqrt(i)
stopCluster(cl)
```

Parallelized Linear Regression

- Let's think about how we might parallelize the code to find OLS coefficients.
- Recall that the sum of inverses is not equal to the inverse of the sum.
- A good habit to get into is to only read in data when needed and delete it from RAM when not needed.
- For larger files, it's also good to never read the full file in all at once.
- Keeping with tis good practice, our linear regression function will have the following form:
  1. Read in a chunk of data from the file.
  2. Process to fit our needs.
  3. Calculate $X'X$ and $X'Y$ for the chunk
  4. Return the two matrices

Parallelized Linear Regression

- We're going to run a regression on bigdats.csv.
- Let's start by thinking about how we can chunk the data.
- One important piece of information about large data files is the number of observations or rows.
- We don't want to (or can't) read in all the data at once. So, we need another method for counting rows.
- We'll use system commands.

Parallelized Linear Regression

- Let's start by creating the system command using `paste()`.
- Paste concatenates character strings into a single string.
- For Windows, the command that we're after is:

```
fl <- "file.path"
sys.com <- paste('find /v /c "" ',fl,sep="")
system(sys.com)
```

- For UNIX:

```
fl <- "file.path"
sys.com <- paste('wc -l ',fl,sep="")
system(sys.com)
```

- These functions return the line count. Write this number in your code as a comment and create an object `C <- ..`

Parallelized Linear Regression

- Now that we know how many observations are in our file, we can start to build the parallelized function.
- We want to run the following regression:

$$SPENT = TREAT1 + TREAT2 + TREAT1 * TREAT2$$

- Recall what we need to do in our function. Let's start by figuring out how we're going to read in chunks of data.

```
#X is the iterator, chunk.size is equal to the chunk size,
#and C is the total number of rows in the data
parallel.lr <- function(x, chunk.size, C){
  #Figure out the last row that we'll read in on the Xth chunk
  end <- min(C,(x*chunk.size))
  #Figure out the starting point from the end point
  start <- end - chunk.size + 1
  #Read in the data skipping to start and reading to end
  dats <- fread("file.path", nrows = chunk.size, skip = start)
```

Parallelized Linear Regression

- The chunk of data in now in R. We can start to process it. Use data.table as it's much faster.

```
#Set the names of the new data
setnames(dats, c("SID","TREAT1","TREAT2","SPENT","N.ITEMS"))
#Get rid of unneeded columns
dats <- dats[,SID := NULL]
dats <- dats[,N.ITEMS := NULL]
#Create the column for the interaction term
dats <- dats[,TREAT.INT := TREAT1*TREAT2]
```

Parallelized Linear Regression

- We have almost everything we need. Now separate the data.table into X and Y.

```
#Take SPENT and make it the Y matrix
yy <- data.table(dats[,SPENT])
#Get rid of the SPENT to make the X matrix
xx <- data.table(dats[,SPENT := NULL])
#Add a column for the intercept
xx <- dats[,INT := rep(1,chunk.size)]
#Remove the full data set
rm(dats)
#Garbage Collection!
gc()
```

Parallelized Linear Regression

- With X and Y, calculate $X'X$ and $X'Y$ for the chunk.
- We want foreach to write both matrices to the list output, so we're going to make a sublist and send that as the function's output.

```
#Generate X'X
xpx <- t(as.matrix(xx))%*%as.matrix(xx)
#Generate X'Y
xpy <- t(as.matrix(xx))%*%as.matrix(yy)
#Remove X and Y
rm(xx,yy)
gc()
```

Parallelized Linear Regression

- We want foreach to write both matrices to the list output, so we're going to make a sublist and send that as the function's output.

```
  #Create a sublist
  out <- list()
  #put X'X in the first page
  out[[1]] <- xpx
  #X'Y in the second
  out[[2]] <- xpy
  return(out)
}
```

Parallelized Linear Regression

- Using the function we just wrote, we can now spin up the cluster and apply foreach over the data.

```
#How many iterations do I need to cover the data?
n.its <- ceiling(C/25000)
#Now we can spin up the cluster and run our function.
cl <- makeCluster(3)
registerDoParallel(cl)
#Run function in parallel
out <- foreach(m = 1:n.its, .packages = c("data.table")) %dopar%
  parallel.lr(m, 25000, C)
```

Parallelized Linear Regression

- Check your output. We should have 80 list elements with 2 nested list elements.
- The function performed the split step. Now we need to recombine the data.
- Because of the setup of the problem, we know that we're going to sum over the matrices.
- My preference is to put all elements of the list into corresponding array and use apply.

Parallelized Linear Regression

```
#Create array holders
xpx.arr <- array(dim=c(dim(out[[1]][[1]])[1],
           dim(out[[1]][[1]])[2],length(out)),c(0))
xpy.arr <- array(dim=c(dim(out[[1]][[2]])[1],
           dim(out[[1]][[2]])[2],length(out)),c(0))
#Loop through list and get everything in array
for(i in 1:length(out)){
  xpx.arr[,,i] <- out[[i]][[1]]
  xpy.arr[,,i] <- out[[i]][[2]]
}
#Aggreate!
xpx <- apply(xpx.arr,c(1,2),sum)
xpy <- apply(xpy.arr,c(1,2),sum)
#Coefficients
betas <- solve(xpx)%*%xpy
```

Parallel Linear Regression

- Voila! We have the coefficients.
- That took a decent amount of work, but the benefits are great when we're trying to calculate regression coefficients on very large data sets.
- Unfortunately, we are not done quite yet.
- We want to calculate standard errors associated with each of the coefficients.
- We already have $X'X$, so all we need now is σ^2.
- Estimate using:
$$\sigma^2 = \frac{\|Y - X\beta\|^2}{n - p}$$
- We know n because we checked before hand. p is the number of coefficients that we just calculated (4).

Parallel Linear Regression

- This creates a parallel workflow:
  1. Read in chunks of data and calculate matrices
  2. Aggregate matrices and calculate coefficients
  3. Pass coefficients to cluster, read in data, and calculate sum of residuals for chunk.
  4. Aggregate residuals and sum.
  5. Calculate σ^2 and $\sqrt{diag(\sigma^2(X'X)^{-1})}$.

Parallel Linear Regression

- To calculate standard errors, we need to know the sum of squared residuals.
- Thus, we need Y, X, and β.
- Fortunately, we've already done most of the legwork for this function:

```
sum.resids <- function(x,chunk.size,C,coefs){
  #Figure out the last row that we'll read in on the Xth chunk
  end <- min(C,(x*chunk.size))
  #Figure out the starting point from the end point
  start <- end - chunk.size + 1
  #Read in the data skipping to start and reading to end
  dats <- fread("file.path", nrows = chunk.size, skip = start)
```

Parallel Linear Regression

- After reading in the data, we process it in the same way we did before:

```
#Set the names of the new data
setnames(dats, c("SID","TREAT1","TREAT2","SPENT","N.ITEMS"))
#Get rid of unneeded columns
dats <- dats[,SID := NULL]
dats <- dats[,N.ITEMS := NULL]
#Create the column for the interaction term
dats <- dats[,TREAT.INT := TREAT1*TREAT2]
#Take SPENT and make it the Y matrix
yy <- data.table(dats[,SPENT])
#Get rid of the SPENT to make the X matrix
xx <- data.table(dats[,SPENT := NULL])
#Add a column for the intercept
xx <- dats[,INT := rep(1,chunk.size)]
#Remove the full data set
rm(dats)
#Garbage Collection!
gc()
```

Parallel Linear Regression

- Finally, we just calculate the sum of squared residuals for each chunk:

```
  #Calculate sum of squared residuals for chunk
  resids <- sum((yy - as.matrix(xx)%*%coefs)^2)
  return(resids)
}
```

Parallel Linear Regression

- Calculate the number of chunks and spin up the cluster:

```
#How many iterations do I need to cover the data?
n.its <- ceiling(C/25000)
#Now we can spin up the cluster and run our function.
cl <- makeCluster(3)
registerDoParallel(cl)
```

Parallel Linear Regression

- Call the function in parallel:

```
#Run function in parallel
out <- foreach(m = 1:n.its, .packages = c("data.table"),
    .combine = "+") %dopar% sum.resids(m, 25000, C, betas)
#STOP THE CLUSTER
stopCluster(cl)
```

Parallel Linear Regression

- Finally, calculate σ^2 and use this to calculate the standard errors:

```
#Store out sum as ssr
ssr <- out
#Calculate sig2
sig2 <- ssr/(C - dim(betas)[1])
#Get the corresponding standard errors
ses <- as.matrix(sqrt(sig2*diag(solve(xpx))))
#Regression Matrix
reg.mat <- cbind(betas,ses)
```

Parallel Linear Regression

	Estimate	Std. Error	t value	Pr(>|t|)
(Intercept)	15.2172	0.0511	297.91	0.0000
dats$TREAT1	0.9671	0.0715	13.52	0.0000
dats$TREAT2	-0.5918	0.0722	-8.20	0.0000
dats$TREAT1:dats$TREAT2	1.1650	0.1021	11.41	0.0000

Parallel Linear Regression

- It works!
- bigdats.csv is a relatively "small" data set.
- You can see the real power of this method on larger data sets.
- Using the provided script, calculate the OLS coefficients and standard errors for the realbigdats.csv.
- Start by finding C. Then set your chunk size to 500,000.
- realbigdats.csv is 2.07 GB!
- If you read a data set this large into R, you are bound to run into problems.
- One errant copy can cause you to hit your RAM limit.
- The parallel functionality we built does take a minute or two to run.
- Monitor your RAM and CPU usage while this function is running. See how the RAM never really increases?

Why don't we always parallelize code?

- This is great, right?
- Why don't I always do this?
- Consider the first situation where we reduce the time is takes to take a sum from $O(N)$ to $O\left(\frac{N}{4}\right)$. Recall that these values only work in the limit.
- In finite time, we can approximate the time for a serial sum as N and the parallel version as $\frac{N}{4} + S$.
- What is S?

Why don't we always parallelize code?

- This is great, right?
- Why don't I always do this?
- Consider the first situation where we reduce the time is takes to take a sum from $O(N)$ to $O\left(\frac{N}{4}\right)$. Recall that these values only work in the limit.
- In finite time, we can approximate the time for a serial sum as N and the parallel version as $\frac{N}{4} + S$.
- What is S?
- Node communication cost, unbatched processing time, read/write time, etc.
- Parallelization only benefits you when the gain in efficiency is greater than S.
- Computers are fast. Small samples are faster on one processor.

Optimal Chunk Sizes

- Picking the chunk size is an art.
- As chunk size decreases, S increases but the computational strain decreases. (Why?)
- As chunk size increases, S decreases but the computational strain increases.
- It's a balancing act. Ideally, you would want chunk size times the number of nodes in your cluster to be equal to the total RAM you machine has.
- Honestly, you really just have to feel out the problem and make a safe selection. Too small is always better than too big.