Parallel Computing in R

Kevin Johnson

November 20, 2014

1 Introduction

Data is becoming bigger and bigger, and advances in processing power have not been able to keep up (partially due to the laws of physics but that's a discussion for another time). The last few years have seen a trend of stacking multiple processors on top of each other rather than making one super fast processor. This combined with the exponential growth in the size of available data has resulted in huge growth in the area of parallel computing.

The world of parallel computing is large and far too extensive to cover in our limited time. This document will get you up and running if you want to do some basic parallel computing from within R. Python has similar functionality (and some would argue that Python should be used over R if speed is a factor), but for now I'll stick with R.

2 Available Packages

The CRAN website has a great overview of parallel computing in R located here: `http://cran.r-project.org/web/views/HighPerformanceComputing.html`. if you need something that goes beyond the scope of this document, that's the first place you should look.

For this I'm going to focus on the `foreach` package in combination with the texttt doMC package. These packages focus on parallel computing on a single workstation (e.g. your home desktop running a quad-core processor). If you have a large for loop that could benefit from running in parallel, these are the packages for you.

```r
library(foreach)
library(doMC)
```

3 The Problem

Let's say you are writing a custom simulation with several nested for loops. For this example, I'll randomly generate some data and then evaluate some function inside of a few nested for loops. Here's the base algorithm:

```
start <- proc.time()
people <- numeric(100000)
for (k in 1:100000) {
    x1 <- rnorm(0,1)
    x2 <- rnorm(10,10)
    x3 <- rnorm(5,1)
    x4 <- rnorm(5,5)
    people <- c(people, 1/(1+exp(-(x1 + 2*x2 + 0.5*x3 + 1.5*x4))))
}
proc.time() - start

##    user  system elapsed
##  70.865   4.265  75.209
```

4 Parallel Implementation

Each instance of the for loops can be computed independently, which means we can compute several instances at the same time. Processor speed increases have slowed down significantly in recent years (from what I hear it's due to the laws of physics, but I'm not an expert in that subject). Instead of making faster processors we've been stacking processors on top of each other and connecting them to work in unison. This sounds great, but it poses significant challenges for us because things like R and Python do not take advantage of this by default. A typical desktop these days will have 4 cores, which means you are only using 25% of the computer's power at any given time.

Parallel processing is a very deep subject that you could make an entire career out of, but I'll start with the `foreach` package in R. The `foreach` package provides an alternate framework for for loops that allows for parallel processing. The `doMC` package sets up a miniature computing cluster on your machine that allows R to run on multiple cores at once.

You can find a manual for using the `foreach()` function here: http://cran.r-project.org/web/packages/fore
The most important thing to consider is the `.combine` parameter that specifies how the result of each loop is combined. This is a fundamental concept in all of parallel processing, you must specify how to combine the results of your loop instances. By default, this is a list, but lists are annoying so I usually try to specify something better than that.

There are three main values that `.combine` takes built in:

- `"c"`: This will concatenate the results of each instance of the for loop into a 1d vector.
- `"cbind"`: If your for loop returns a vector then you can use this to bind those vectors together into a matrix.
- `"+"`: This simply adds the results of each instance of the for loop.

Let's use the `"c"` parameter to implement the previous algorithm in parallel. Here's what it looks like:

```
registerDoMC(cores = 4)
start <- proc.time()
people <- foreach(k = 1:100000, .combine = "c") %dopar% {
    x1 <- rnorm(0,1)
    x2 <- rnorm(10,10)
    x3 <- rnorm(5,1)
    x4 <- rnorm(5,5)
    return(1/(1+exp(-(x1 + 2*x2 + 0.5*x3 + 1.5*x4))))
}
proc.time() - start

##    user  system elapsed
##  32.325   0.120  29.539
```

We've gone from 75 seconds down to 30 seconds, a 60% decrease. It's worth noting that running your algorithm in parallel is not always going to result in a faster runtime. There's a lot going on in the background to make this work and it's not always worth it. For example, here's what happens when you have a small number of iterations.

```r
start <- proc.time()
people <- numeric(1000)
for (k in 1:1000) {
    x1 <- rnorm(0,1)
    x2 <- rnorm(10,10)
    x3 <- rnorm(5,1)
    x4 <- rnorm(5,5)
    people <- c(people, 1/(1+exp(-(x1 + 2*x2 + 0.5*x3 + 1.5*x4))))
}
proc.time() - start

##    user  system elapsed
##   0.026   0.005   0.029

registerDoMC(cores = 4)
start <- proc.time()
people <- foreach(k = 1:1000, .combine = "c") %dopar% {
    x1 <- rnorm(0,1)
    x2 <- rnorm(10,10)
    x3 <- rnorm(5,1)
    x4 <- rnorm(5,5)
    return(1/(1+exp(-(x1 + 2*x2 + 0.5*x3 + 1.5*x4))))
}
proc.time() - start

##    user  system elapsed
##   0.273   0.022   0.294
```

The normal version takes 0.029 seconds and the parallel version takes 0.294 seconds. Generally, implementing algorithms in parallel is worth it if there is a very large number of iterations (like in our first example) or if the inner function takes a significant amount of time to run (think minutes instead of seconds).

5 Diamonds Example

What if you want to do something more complicated than combining some numbers together? Let's say you want to do cross validation and run a bunch of different regression models on different sections of your dataset? Let's do that with the `diamonds` dataset we used previously. If you don't remember, I'll copy a brief description of the data here.

This dataset contanis prices and other attributes of more than 50,000 diamonds. The variables included are:

- price: price in US dollars ($326–$18,823)
- carat: weight of the diamond (0.2–5.01)
- cut: quality of the cut (Fair, Good, Very Good, Premium, Ideal)
- colour: diamond colour, from J (worst) to D (best)
- clarity: a measurement of how clear the diamond is (I1 (worst), SI1, SI2, VS1, VS2, VVS1, VVS2, IF (best))
- x: length in mm (0–10.74)
- y: width in mm (0–58.9)
- z: depth in mm (0–31.8)
- depth: total depth percentage
- table: width of top of diamond relative to widest point (43–95)

```
library(ggplot2)
head(diamonds)

##   carat       cut color clarity depth table price    x    y    z
## 1  0.23     Ideal     E     SI2  61.5    55   326 3.95 3.98 2.43
## 2  0.21   Premium     E     SI1  59.8    61   326 3.89 3.84 2.31
## 3  0.23      Good     E     VS1  56.9    65   327 4.05 4.07 2.31
## 4  0.29   Premium     I     VS2  62.4    58   334 4.20 4.23 2.63
## 5  0.31      Good     J     SI2  63.3    58   335 4.34 4.35 2.75
## 6  0.24 Very Good     J    VVS2  62.8    57   336 3.94 3.96 2.48
```

Let's try to predict the price of a diamond based on the carat, cut quality, color, and clarity. Here's a simple linear regression to do just that. Note that a linear regression may or may not be the most appropriate method here, this is just for demonstration.

```
fit <- lm(price ~ carat + factor(cut) + factor(color) + factor(clarity),
          data = diamonds)
summary(fit)

##
## Call:
## lm(formula = price ~ carat + factor(cut) + factor(color) + factor(clarity),
##     data = diamonds)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -16813.5   -680.4   -197.6    466.4  10394.9
##
## Coefficients:
##                     Estimate Std. Error  t value             Pr(>|t|)
## (Intercept)        -3710.603     13.980 -265.414 < 0.0000000000000002 ***
## carat               8886.129     12.034  738.437 < 0.0000000000000002 ***
## factor(cut).L        698.907     20.335   34.369 < 0.0000000000000002 ***
## factor(cut).Q       -327.686     17.911  -18.295 < 0.0000000000000002 ***
## factor(cut).C        180.565     15.557   11.607 < 0.0000000000000002 ***
## factor(cut)^4         -1.207     12.458   -0.097                0.923
## factor(color).L    -1910.288     17.712 -107.853 < 0.0000000000000002 ***
## factor(color).Q     -627.954     16.121  -38.952 < 0.0000000000000002 ***
## factor(color).C     -171.960     15.070  -11.410 < 0.0000000000000002 ***
## factor(color)^4       21.678     13.840    1.566                0.117
## factor(color)^5      -85.943     13.076   -6.572        0.00000000005 ***
## factor(color)^6      -49.986     11.889   -4.205        0.00002620629 ***
## factor(clarity).L   4217.535     30.831  136.794 < 0.0000000000000002 ***
## factor(clarity).Q  -1832.406     28.827  -63.565 < 0.0000000000000002 ***
## factor(clarity).C    923.273     24.679   37.411 < 0.0000000000000002 ***
## factor(clarity)^4   -361.995     19.739  -18.339 < 0.0000000000000002 ***
## factor(clarity)^5   216.616     16.109   13.447 < 0.0000000000000002 ***
## factor(clarity)^6     2.105     14.037    0.150                0.881
## factor(clarity)^7   110.340     12.383    8.910 < 0.0000000000000002 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 1157 on 53921 degrees of freedom
## Multiple R-squared:  0.9159,Adjusted R-squared:  0.9159
## F-statistic: 3.264e+04 on 18 and 53921 DF,  p-value: < 0.00000000000000022
```

Not bad, that's actually a much better regression than I was expecting. Now, let's say we want to use cross validation to obtain a mean squared error for our predictive model. I explained this in the regression notes, but here's the basic idea:

1. Leave out one out of n observations.
2. Fit the regression model on the remaining $n-1$ observations.
3. Use that model to predict the observation you left out.
4. Compute the error in your prediction.
5. Repeat for all n observations to get the root mean squared error (or some other measure of accuracy).

We're going to implement this in parallel because it can be computationally expensive to fit that many models. I subset the data to the first 2000 rows to make it run in a short amount of time.

```r
registerDoMC(cores = 4)
temp <- diamonds[1:2000,]
start <- proc.time()
errors <- foreach(i = 1:nrow(temp), .combine = "+") %dopar% {
    fit <- lm(price ~ carat + factor(cut) + factor(color) + factor(clarity),
              data = temp[-i,])
    pred <- predict(fit, temp[i,])
    return((temp$price[i] - pred)^2)
}
sqrt(errors/2000)
```

```
##        1
## 228.5501

proc.time() - start

##    user  system elapsed
##  48.027   0.262  13.751
```

6 Alternative Technologies

6.1 R

Of course the `foreach` package in R is not the only way to do parallel processing. Revolution Analytics is a company that specializes in products for doing big data analytics in R. They recently released Revolution R Open (`http://mran.revolutionanalytics.com/open/`) which includes some extra features revolving around multi-threaded computations and reproducible R code/documents. It's free so you can download it and play around with it if you want. They also offer cloud based services for a fee.

There's also projects like SparkR (`https://amplab.cs.berkeley.edu/2014/01/26/large-scale-data-`
and H2O (`http://0xdata.com/h2o/`) that offer a way to easily connect R to software and machines that are capable of large scale analytics. Parallel processing in R is still in its infancy at this point, but there are a lot of exciting projects out there trying to bring parallelization to the masses (so to speak).

Generally speaking, R is one of the slowest languages out there. If speed is a concern then parallelization may not be enough. Other languages like Python and C++ are much faster (especially C++). For loops in R are notoriously slow. You should avoid using for loops for anything if at all possible. If you must, then the `RCpp` package allows you to embed C++ code within R. I recently did this for a simulation from my research and decreased my runtime from 65 hours to 50 minutes. Yeah, that's how bad for loops in R can be.

6.2 Hadoop

Hadoop is the big daddy of the parallel processing world. It's an open source framework created by a couple guys at Yahoo (now offered by Apache) that allows for distributed storage and processing of extremely large data sets across clusters of machines (`http://hadoop.apache.`

`org/`). It's based on the MapReduce paradigm first developed by Google to handle large amounts of data (`http://research.google.com/archive/mapreduce-osdi04.pdf`). Teaching you the ins and outs of MapReduce is beyond the scope of this document, but I will say that if you intend on doing big data analytics then you need to know and understand how MapReduce works. It's related to the `foreach` package we discussed earlier in that its primary function is to specify how to combine the results of parallel computations.