URL: https://yinzhang@bitbucket.org/yinzhang/stat242.git
STAT242hw5
Yin Zhang

Working with Big Data

Abstract:

The report focuses on comparing shell, R and C performance in dealing with big data and reducing of real running time. There are two problems. For the first problem of calculation of less time between tolls amount and total amount, I use the three methods with parallel, block , then analyses the three types time(user, system, real time) and stability. For the second problem of regression, first I use shell to check fare and data match, then use C, R with parallel and welford method to calculate the regression parameter, then analysis the running time of three methods. The results show that C have a better performance in real running time and it is stable, regardless of programming time. For all three methods, reading data costs much more time than the time of calculation, and ways to reduce is to do calculation at the same time when read the data(such as parallel, block).

1 less value for tolls to total

1.1 Methods

The main idea is first calculate the frequency table, then find the decile of it. I mainly use three ways to calculate the value: shell, R and C.

For shell: At first, I plan to use shell to have a for loop to run all the file. I run it for one file, the time is: real 198.27s, user 160.38s, sys 3.75s. So if I do it for a “for loop”, it will take almost half an hour. At the same time if I just calculate the less value and put them into a .txt file, it will take lots of memory even just for one line. According to it, I change the method. For example, I can calculate the frequency table and then calculate the. The other way is to parallel and shell together. In the report, I used the second one.

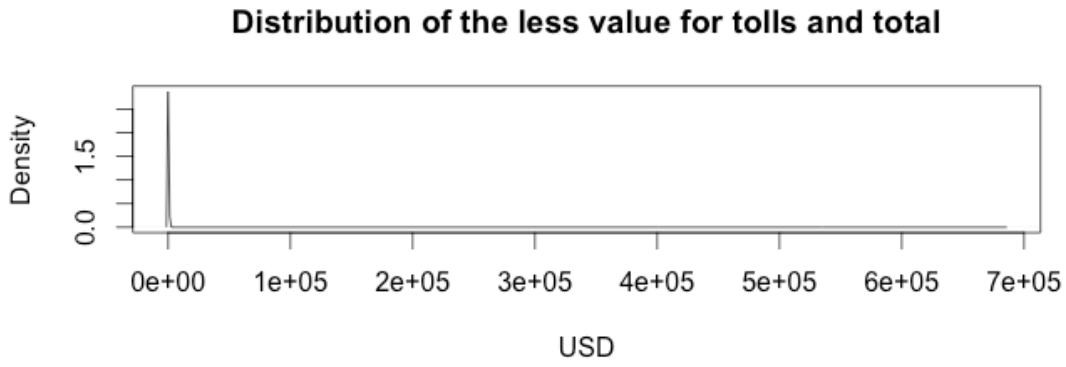
For R: I use block to read and calculate the less value, then parallel it to get the frequency table. Here I use R block because I find that the ways of using block and the way of not using block, they have similar running time (I think it is

because for the “not use block” method, writing to variables takes lots of time, while “uses block” method, the “for loop” takes time). Considering for the memory part, I used block in this report.

For C, according to the first two methods, R and shell takes lots of time to read the data. But when I get the frequency table, R can calculate the decile quickly, so in order to save time and save memory, I use C to read the data and calculate the less value, then parallel it to get the frequency table.

1.2 Results

The decile of the total amount less the tolls is 6 in USD. The density plot of less value is shown below:



The plot shows that the distribution of less value of toll and total are extremely heavy tail. The majority of them are around 10, and there are extreme large values. Moreover, the frequency table shows some negative value. In frequency table, the number that 9 USD, 6.5 USD, 8.0 USD and 7.0 USD have the largest frequency (6139139, 5855774, 5829132 and 5674008 respectively). There are 172 different negative values, 3615 of 173179759 (<0.00001) total number. The lowest one is -1430.0 USD which absolute value is really big. There are also extreme large values, such as 685908.00 USD, 541433.00 USD, 158996.00 USD … but the frequency of them are 1 of the extreme large value. They are almost in file8. Further discussion will be in Part 2.

1.3 Evaluation of each methods

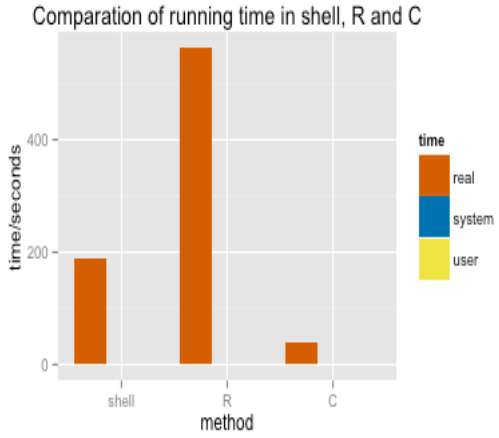
I evaluate each measure by the running time, the memory they takes and the stable of the methods three aspects.

1.3.1 Running time

The system time, user time and real running time of each methods and their compare are shown below:

Table1: running time(seconds) in each methods

Time(seconds)	system	user	real
Shell	0.003	0.039	187.532
R mainly	0.004	0.041	563.929
C mainly	0.003	0.057	40.008



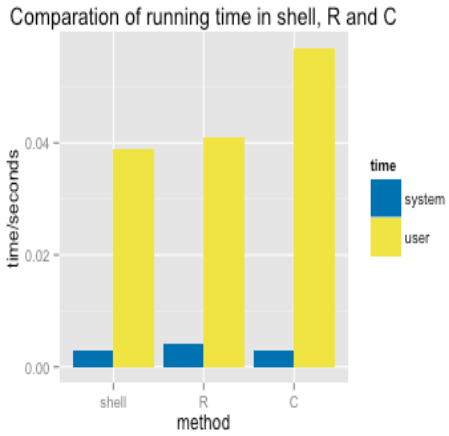
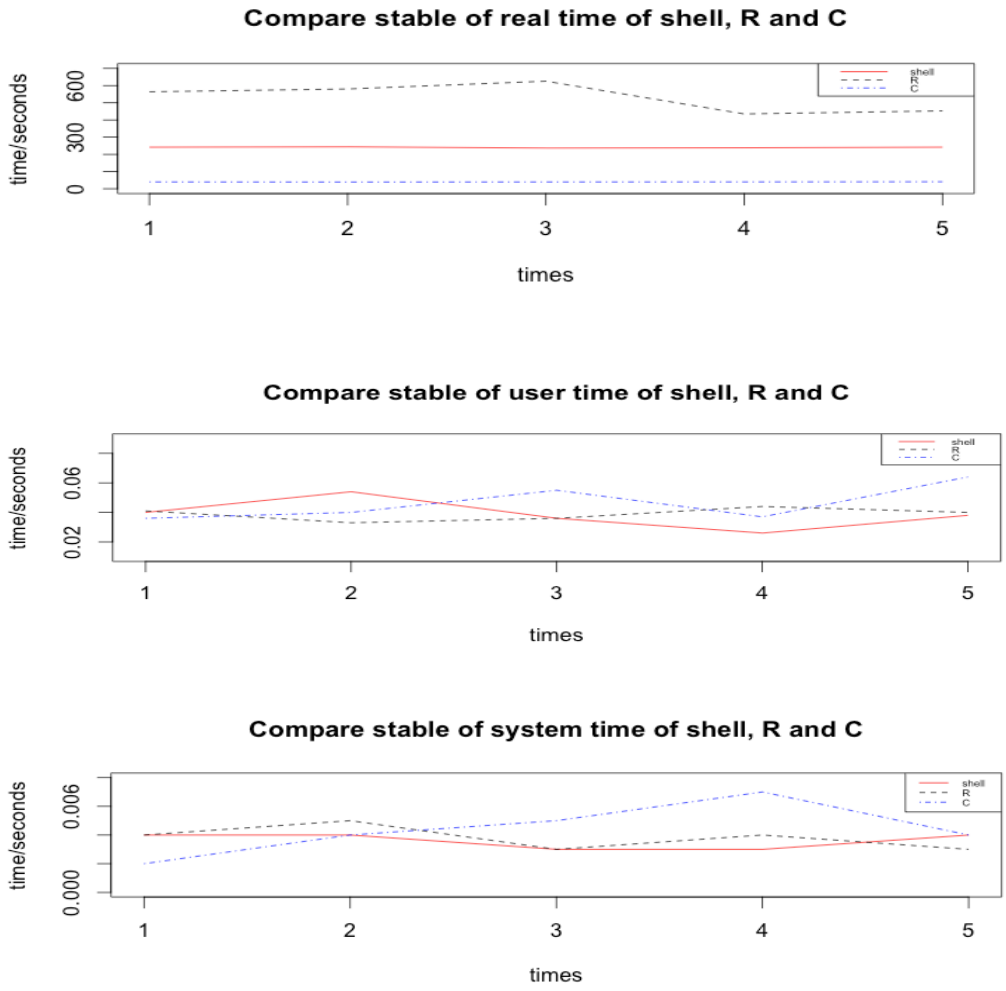


Table1 shows that for the real time, the time of running C is almost 5 times faster than shell and 16 times faster then R. The plot on the left shows that system and user time are so small that when they plot with real time, the time of them are almost 0. In order to compare user time and system time, I also make a plot of them(the plot on the right hand). It shows that on the contrary, C takes the longest use time and shell takes shortest time. For system time, they are almost the same. According to the help page of system time[1] and Wikipedia[2], "The 'user time' is the CPU time charged for the execution of user instructions of the calling process", and "The 'system time' is the CPU time charged for execution by the system on behalf of the calling process. The real time which is much larger than the other two is because the three methods spend much time waiting and not executing at all (whether in user mode or system mode). So for reading data, R takes the most time, C takes least time. For use time system time of the three methods, it shows that the three methods have almost the same CPU system call, which means they requests the service little form an operation system's

kernel[3]. For user time, C takes more time for CPU spent perform some actions. The total time of CUP of three methods are all less than 0.01, which means the big problem for running code too long is reading the data.

In conclusion: 1 the reason of real running time is so long is because the speed of reading data. 2 the different of time among the three methods it mainly because C can read data quicker and R is slowest, not the calculation time. So even I take parallel, the reading time still too large. In order to develop the method, one way is to find some way when read the data, do not wait it to finish but calculate at the same time.

1.3.2 Stability:



From the three plots above, they show that for the real time, R is not stable, comparing with shell and C; for user time, the stability of three methods are

almost the same(C is a little bit larger); for system time, C is not stable comparing with shell and R methods. Combining with the results in 1.3.1 of running time, it shows that if the running time is larger, the corresponding stability will be less.

Moreover, I find that I can calculate the time in each file to see the stability(just put the system.time in the sapply function). However, the two methods show the similar things. So I just put one of them here.

2 linear regression of less value in part1 with trip time

Because some of the time are too long and the stable have been discussed in part 1, in this part, I just discuss the running time and memory that they take of each method.

2 .1check matched and outlier data

I check the medallion and pickup_datetime to see whether they are matched. Because if a person have the same medallion and the trip time(pickup_datatime) are the same, it is the same trip and can calculate same trip process. The results shows that they are matched.(details in Appendix 1)

2.2 Calculate beta0 and beta1

2.2.1 Methods

According to the results and the programming time of mine, I choose to use R, C and to calculate the regression parameter (because I am not familiar with shell, it takes me a long time to write the code). Because welford methods and the calculation of covariance are mainly for iteration[4,5], in order to save time, I use C to write the welford method.

Compare the results of welford method and other two methods: Because in R and C, I paid more attention to the accuracy of the calculation. I use a list to store the results and sum in block which can void big data adding small data. The results of the two methods are almost the same with the welfold method(they just have a same value in this calculation!) However, for welford methods, because I have not find a way to use parallel to calculate the variance, so I stored all needed data then calculate the variance and covariance, it takes me a long time and lots of memory.

In R and C, I calculate the sufficient statistics

$\sum_{i=1}^{n} x_i$, $\sum_{j=1}^{n} y_j$, $\sum_{i=1}^{n} x_i y_i$, $\sum_{i=1}^{n} {x_i}^2$ and use the formula:

$$\hat{\beta}_1 = \frac{\sum_{i=1}^{n} x_i y_i - \sum_{i=1}^{n} x_i \sum_{j=1}^{n} y_j}{\sum_{i=1}^{n} {x_i}^2 - \frac{1}{n}(\sum_{i=1}^{n} x_i)^2}$$

In Welford method, I calculate the variance of x and covariance of y[4,5], using the formula:

$$\hat{\beta}_1 = \frac{Cov[x,y]}{Var[x]}$$

2.2.2 Results

According to the results in part1, there are negative values and extreme data in file 8(trip_fare_8.csv and trip_data_8.csv). So I calculate two results which are with file 8 and without file 8 to see how much of the influence of data in file 8. The results shows that if there with or without the 8th file, the regression parameter is:

Table3: results of regression calculation in each methods

	Include file8	Exclude file8
Beta0	14.52334	2.430292
Beta1	2.06023×10^{-5}	0.0160375

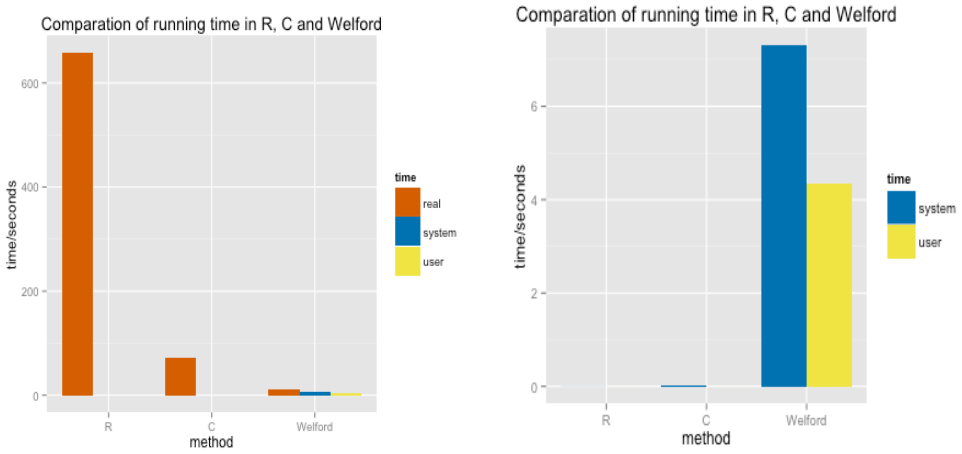
The table shows that within file 8 and without file 8, the regression line change a lot. It also indicates that the data in file 8 are outliers. Analysis which results are better: According to the results in frequency table of total – tolls, the majority less value are around 4 to 20. So the corresponding trip_time(seconds) is around (-510784.7, 265827.6) when use regression methods that include file 8(trip_data_8.csv and trip_fare_8.csv). On the other hand, for the regression which does not contain file 8, the corresponding trip_time(seconds) is (97.87735, 1095.539) which is nearer the real life. So the second one beta0 = 2.43, beta1 = 0.016 are more reliable.

Table4: running time for regression calculation in each methods

Times(seconds)	User	system	real

R	0.004	0.001	658.619
C	0.005	0.001	72.778
Welford in C	7.320	4.335	12.293

*** the third one only calculate the time of calculation regression parameter, not include the time of read data.**



The table shows the similar things as question 1 for R and C methods. However, when I run the welford method in C, the user time and system time are not that much different with real time as R and C methods. In the plot, it is obvious to see that differences. In the plot on the left side, the system and user time are so little to 0 for R method, but for the welford method in C, the three methods are not different too much. The plot on the left hand shows that, comparing with welford method in C, running time of user and system in R code and C code are almost equal to 0. I think it is because for the third welford in C method, I just calculate the calculation time for regression, not include the time of reading data. The welford method for variance and stable one-pass algorithm for covariance methods are complex comparing with just calculating the sufficient statistics. So it will takes more time for CPU to run.

3 Use multiple regression and add the surchage as a regressor, estimating the resulting coefficients.

I use "biglm" package to do it. The codes are shown in the Appendix. Can also use matrix to calculate the coefficient parameter.

Conclusion:

Compare of three methods: the lower the level languages, the running time is shorter and the corresponding methods are more stable. Shell is good at read file and write file, but does not good at calculate. On the contrary, R is good at calculate, but reading a file will takes a long time(not using other package). C seems good at both but the C script is longest then the others(which might takes lots of programming time). But running time for C is more stable than the other two methods. So when estimating a parameter, it is better to use C.

The way to decrease running time: For the situation when the calculation is simple but the data is large, the main method to develop the running speed is to decrease the reading time, that is, when read the data, do not wait to finish reading but calculate at the same time. Some ways can find from this paper are: using parallel, using block, using pipe to reduce running time.

Further discussion

1 There are lots of ways to do this job, such as hadoop, and Spark…..The codes are not very hard which is similar with Duncan's code of the airlines.

2 The time calculated in this paper are all based on the server of poisson and gcox. They are influenced by the other users. So the time might be not much accurate.

[1] system.time help page: http://www.inside-r.org/r-doc/base/system.time
[2] time (Unix): http://en.wikipedia.org/wiki/Time_%28Unix%29
[3] System call: http://en.wikipedia.org/wiki/System_call
[4] Comparing three methods of computing standard deviation, http://www.johndcook.com/blog/2008/09/26/comparing-three-methods-of-computing-standard-deviation/
[5] stable one-pass algorithm for covariance https://en.wikipedia.org/wiki/Algorithms_for_calculating_variance#Covariance

Appendix 1

Match:

I use three ways to check whether they match. .They all matched but in file 8 there are outliers. The first method that I use is blocking with 1000000 and then I sample the index and read the corresponding lines for each file to check whether they match. However, there is a trade-off of confident and running time. In order to achieve at least 0.9 confident, it should sample $0.9\times total\_lines$. However, it takes a long time to know the number of lines in each file and sample $0.9\times total\_lines$ is actually not save much time. Then I use shell command and parallel to check whether they are match. It is much quicker then the first, because it runs in shell which can give more memory and does not need for loop to block. So at last, I choose the second way. The results show that the 12 files are all matched (trip_fare data and trip_data data). The running time shows below:

Table2: running time(seconds) in match

Time(seconds)	User	System	real
Shell + parallel	0.004	0.002	1080.086

The results show that the real running time is much larger than the other two methods, even comparing to the part 2. I think it is because I use “system” command, not “pipe” command, and I do not use block in this methods.

Appendix 2

#1

```
#method 1: shell get the less value of total-tolls that we need for one file
#filename is the path of the file and its name that we need
sdif = function (filename)
{
   shellCMD = paste("cut -f 10,11 -d,", filename,    "|awk -F,
'NR>=2{printf(\"%s\\n\", $2-$1)}'")
   less = as.numeric(system(shellCMD, intern=TRUE))
   #change it to be data frame to further merge table of 12 files
   as.data.frame(table(less), stringsAsFactors = FALSE)
}
```

```
#get the 12 files
paths = list.files(path ="/home/data/NYCTaxis", pattern = "trip_fare.*.csv", recursive = TRUE)
filename = lapply(paths, function(x) paste("/home/data/NYCTaxis/", x, sep = ""))

#parallel for all 12 files
library("parallel", lib.loc = "~/Rpackages")
cl = makeCluster(12, "FORK")
#Stime = replicate(5, system.time(clusterApply(cl, filename, sdif)))
clusterApply(cl, filename, function(x) system.time(sdif(x)))
stopCluster(cl)

#method 2 : R block + parallel
#use block to read the file and calculate less value that we need
dif = function(filepath)
{
   f = pipe(filepath, "r")    # open standard input
   #make the blockSize to be 1000000 because I find that the length of each file is at 10^7 level
   blockSize = 1000000
   lines = readLines(f, 1) #remove the first row which is the name of variables
   i = 1
   diff = list() #I use a list not vector because list can give me more accurate velue
   #diff is a list which is

   #run for each block
   while(TRUE) {
      lines = readLines(f, blockSize) #This command use lots of time

      #means if read at the end of the file, break the loop
      if(length(lines) == 0)
         break
```

```
      #diff is a list,    the ithe is the less value of tolls and total
      diff[[i]] = sapply( strsplit(lines, ","), function(x) as.numeric(x[2]) - as.numeric(x[1]))
      i = i+1
   }
   close(f)
   #make in the results as a data frame in order to merge the 12 tables
   as.data.frame(table(unlist(diff)), stringsAsFactors = FALSE)
}

#get the path of the files
paths = list.files(path ="/home/data/NYCTaxis", pattern = "trip_fare.*.csv", recursive = TRUE)
file_shell = sapply(paths, function(x) paste("cut -f 10,11 -d, /home/data/NYCTaxis/", x, sep = ""))

#parallel
cl = makeCluster(12, "FORK")
#Rtime = replicate(5, system.time(clusterApply(cl, file_shell, dif))) # run the dif function for all files
results = clusterApply(cl, file_shell, dif)#run the dif function for each file
stopCluster(cl)

###the merge function is to merge the tables that we get in each file together, x, y is the table that we wnat to merge
#and the result will give one table which merge table x and table y together
merge = function(x, y) {
   combine = rbind(x,y)
   #sum the frequncy which have the same name(same less value)
   freq = tapply(combine$Freq, combine$Var1, sum, na.rm=TRUE)
   #make the result as a table in order to interation for merge
   one_table = data.frame(Var1 = names(freq), Freq = freq, row.names = NULL)
}
```

```
f = Reduce(merge, results)#apply merge function to the 12 files table
f$Var1 = as.numeric(as.character(f$Var1))
#because use merge, the value will because charactor, in order to sort them and get the decile, they should be changed to be numeric

#C version for calculate the less value of tolls and total
#the filename have the same means as the shell version
cdif = function(filename)
{
   dyn.load("cdif.so")
   #calculate the length of the file
   length =as.numeric(strsplit(x = system (paste("wc -l", filename),intern=TRUE), split = " +")[[1]][1])-1
   #connect with C give the filename that we want to calculate the less time and details in C script
   cresult = .C("cdif", filename = as.character(filename), less = as.double(rep(0,length)))
   as.data.frame(table(cresult$less),stringsAsFactors=FALSE ) #give the same results as dif function in R and also can save memory
}

#get the path, which is the same as before
#parallel for the 12 files
library("parallel", lib.loc = "~/Rpackages")
cl = makeCluster(12, "FORK")    #make a cluster
results = clusterApply(cl, filename, cdif)
#Ctime = replicate(5, system.time(clusterApply(cl, filename, cdif))) #calculate the running time
stopCluster(cl)

##########################analysis of the results
f = read.table("freq.txt") #freq.txt is the frequency table of total - tolls, the first col V1 is frequency, the second col V2 is the corresponding value
```

```
n=subset(f,V2<0) #the number of less values that little than 0

#order them according to the less value f$Var1 is the less value calculated before
order_diff = f[order(f$Var1),]
#get the decile(0.1 posibility of whole sum)
order_diff$Var1[which(cumsum(order_diff$Freq)>sum(order_diff$Freq)*0.1)[1]
]

#plot the density
f_freq= as.vector(rep(f$V2, f$V1))
dens = density(f_freq)
plot(dens, xlab = "USD", main = "Distribution of the less value for tolls and total")
########################plot
library(reshape)
library(ggplot2)
library(Hmisc)

#plot for the user time, system time and real time for the three methods
times = data.frame(time =c("system", "user", "real"), shell = c(0.003, 0.039,
187.532), R = c(0.004, 0.041, 563.929), C = c(0.003, 0.057, 40.008))

#reshape time in order to use ggplot
times= melt(times, id ='time', variable_name="methods")
ggplot(times,aes(methods,value,fill=time))+

geom_bar(stat="identity",position='dodge')+scale_fill_manual(values=c( "#D55E
00", "#0072B2","#F0E442"))+ #find the color properly in
http://www.cookbook-r.com/Graphs/Colors_%28ggplot2%29/#a-colorblind-fri
endly-palette
  xlab("method")+ylab("time/seconds")+ggtitle("Comparation of running time
in shell, R and C")

#the plot without real time
```

```
ggplot(times[-c(3,6,9),],aes(methods,value,fill=time))+

geom_bar(stat="identity",position='dodge')+scale_fill_manual(values=c("#0072
B2","#F0E442"))+
   xlab("method")+ylab("time/seconds")+ggtitle("Comparation of running time
in shell, R and C")
# times[-c(3,6,9),] means exclude the real time

#i =1,2,3 for user time, sys time and elapsed time repectely, can use lapply to plot
each one
plot(1:5, Stime[i,], col = "red",type = "l", ylim = range(0.01, 0.09), ylab =
"time/seconds", xlab = "times", main = "Compare stable of user time of shell, R
and C")
lines(Rtime[i,], col = "black", lty = 2)
lines(Ctime[i,], col = "blue", lty = 4)
legend("topright", legend=c("shell","R","C"),col=c('red', 'black', 'blue'),
lty=c(1,2,4), cex=0.5)

##########################################################
#########
##2
###chaeck wether the trip_fare and trip_data match
##get the shell command for fare data
fname = list.files(path ="/home/data/NYCTaxis", pattern = "trip_fare.*.csv",
recursive = TRUE)
f_all = sapply(fname, function(x) paste("cut -f 1,4 -d, /home/data/NYCTaxis/", x,
"|sed 1d", sep = ""))
#get the shell command for trip_data
dname = list.files(path ="/home/data/NYCTaxis", pattern = "trip_data.*.csv",
recursive = TRUE)
d_all = sapply(dname, function(x) paste("cut -f 1,6 -d, /home/data/NYCTaxis/", x,
"|sed 1d", sep = ""))
#mix them together
```

```
filename = as.data.frame(rbind(f_all, d_all),stringsAsFactors = FALSE)

#match function is to check whether the trip_fare match the reip_data (whether the same line describe the same thing)
#it returns a string, which tells you wether they match
match = function(filename) #filename[1] is the data from trip_fare, filename[2] is the data from trip_data
{
   fdata = system(filename[1], intern = TRUE) #use system to connect
   ddata = system(filename[2], intern = TRUE)

   #check whether they decribe the same thing in the same line, fdata is for fare, ddata is for trip_data
   if(!all(fdata == ddata))
   {ans = "They are not matched"}else
      ans = "They are matched"

   ans
}

library("parallel", lib.loc = "~/Rpackages")
cl = makeCluster(12, "FORK")
whetherMatch = clusterApply(cl, filename, match)
#system.time(clusterApply(cl, filename, match)) # run the dif function for all files
stopCluster(cl)
whetherMatch

############################calculate the regression coefficient
#method 1 R block+parallel
#calculate the sufficient statistics of bata1 and bata0
suffic = function(file) #calcuate the total amount less the tolls in fare, and the trip time in data
```

```
{
   farec = pipe(file[1], "r")    # open standard input
   datac = pipe(file[2], "r")

   #connection the file and intialized the variable
   blockSize = 1000000
   linesF = readLines(farec, 1)
   linesD = readLines(datac, 1)#remove the first row which is the name of variables
   sumx = 0; sumy = 0; sumx2 = 0; sumxy = 0

   #run the file by block
   while(TRUE) {
      linesF = readLines(farec, blockSize)
      linesD = readLines(datac, blockSize)#This command use lots of time

      if(length(linesF) == 0)
         break

      #if it is not end of the file, then calculate the sufficient statistics for beta0 and beta1, use vectorlized to develop the running speed
      y = sapply( strsplit(linesF, ","), function(x) as.numeric(x[2]) - as.numeric(x[1])) #the totoal -toll value
      x = as.numeric(linesD)
      sumx = sumx + sum(x)
      sumy = sumy + sum(y)
      sumx2 = sumx2 + sum(x^2)
      sumxy = sumxy +sum(x*y)
   }
   #    close(farec); close(datac)
   length =as.numeric(system (paste(file[1], "|wc -l", sep = ""),intern=TRUE))-1
   list(sumx, sumy, sumx2, sumxy, length) #I use list not vector becuase list gives more memory to make the results more accurate
```

```
}

#get the path of 12 shells which is similar below
cl = makeCluster(12, "FORK")
#results = clusterApply(cl, file_shell, suffic)
system.time(clusterApply(cl, file_shell, suffic)) # run the dif function for all files
stopCluster(cl)

#Method 2 C
suff = function(file)
{
   dyn.load("suf.so")
   length =as.numeric(strsplit(x = system (paste("wc -l", file[1]),intern=TRUE), split = " +")[[1]][1])-1
   #length =as.numeric(system (paste("wc -l", file[1]),intern=TRUE))-1
   cresult = .C("suff", fare = as.character(file[1]), data = as.character(file[2]), x = as.double(0), y = as.double(0), xy = as.double(0), x2 = as.double(0))
   list(cresult$x, cresult$y, cresult$x2, cresult$xy, length)
}


path_fare = list.files(path ="/home/data/NYCTaxis", pattern = "trip_fare.*.csv", recursive = TRUE)
#path_fare = path_fare[!path_fare%in%"trip_fare_8.csv"] #do not include file 8
file_fare = sapply(path_fare, function(x) paste("/home/data/NYCTaxis/", x, sep = ""))
path_data = list.files(path ="/home/data/NYCTaxis", pattern = "trip_data.*.csv", recursive = TRUE)
#path_data = path_data[!path_data%in%"trip_data_8.csv"] #do not include file 8
file_data = sapply(path_data, function(x) paste("/home/data/NYCTaxis/", x, sep = ""))
file_shell = as.data.frame(rbind(file_fare, file_data, deparse.level = 0),stringsAsFactors = FALSE)
```

```
#the other way way is to put those command in the function, but I choose here becaue in this way I do not need to repeat it in two functions
#and there are only 12 files, so sapply did not takes much time. So there is not need to put one file in the function and paralell them. Morever,
#in this way, the function also can be used directly if the path of the file change.

cl = makeCluster(12, "FORK")
results = clusterApply(cl, file_shell, suff) # run the dif function for all files
#system.time(clusterApply(cl, file_shell, suff))
stopCluster(cl)

#############calculate the beta1 and beta2
ef = as.data.frame(matrix(unlist(results), nrow = 5))
#ef means each file result, it is a data frame with 12 col and 5 rows, each col represent a file result, each row represents sumx, sumy, sumx2, sumxy, length respectly for each file
#use the formular which mentioned in the report
beta1 = (sum(ef[4,]) - 1/sum(ef[5,])*sum(ef[1,])*sum(ef[2,]))/(sum(ef[3,]) - 1/sum(ef[5,])*sum(ef[1,])^2)
beta0 = 1/sum(ef[5,])*sum(ef[2,]) - beta1*1/sum(ef[5,])*sum(ef[1,])

########method 2: shell + C
getData = function(filename)
{
   ddata = system(paste(filename), intern = TRUE)
   as.numeric(ddata)
}
fname = list.files(path ="/home/data/NYCTaxis", pattern = "trip_fare.*.csv", recursive = TRUE)
f_all = sapply(fname, function(x) paste("cut -f 7 -d, /home/data/NYCTaxis/", x, "|sed 1d", sep = ""))

library("parallel", lib.loc = "~/Rpackages")
```

```
cl = makeCluster(12, "FORK")
x = clusterApply(cl, d_all, getData) # run the dif function for all files
#system.time(clusterApply(cl, f_all, getData))
stopCluster(cl)
#y for less value do the similar thing as bove, the y value can be download in my git

regression = function (x, y)
{
   dyn.load("wff.so")
   dyn.load("mycov.so")

   #calculate the variance of x
   varx = .C("welf_var", x = as.double(x), length = as.integer(length(x)), var = as.double(0))$var
   #calculate the covariance of y and x, and the mean od x, mean of y
   xyResult =  .C("mycov", x = as.double(x), y = as.double(y), length = as.integer(length(x)), cov = as.double(0), meanx = as.double(0), meany = as.double(0))

   #the value that needed to calculate beta1 beta0 mentioned in the report part 2.2.1
   covxy = xyResult$cov
   meanx = xyResult$meanx
   meany = xyResult$meany
   beta1 = covxy/varx
   list(beta0 = meany - meanx*beta1, beta1 = beta1)
}

#put the value of x(trip_time), y(less time) in the function
regres = regression(x,y)
wtime = system(regression(x,y))#welford time
```

```
########################analysis of the results plot
#plot the user, system and real time for the question 2, same as question1
times = data.frame(time =c("system", "user", "real"), R = c(0.004, 0.001, 658.619),
C = c(0.005, 0.001, 72.778), Welford= c(7.320, 4.335, 12.293))
times= melt(times, id ='time', variable_name="methods")
ggplot(times,aes(methods,value,fill=time))+

geom_bar(stat="identity",position='dodge')+scale_fill_manual(values=c( "#D55E
00", "#0072B2","#F0E442"))+ #find the color properly in
#http://www.cookbook-r.com/Graphs/Colors_%28ggplot2%29/#a-colorblind-f
#riendly-palette
   xlab("method")+ylab("time/seconds")+ggtitle("Comparation of running time
in R, C and Welford")
ggplot(times[-c(3,6,9),],aes(methods,value,fill=time))+

geom_bar(stat="identity",position='dodge')+scale_fill_manual(values=c("#0072
B2","#F0E442"))+
   xlab("method")+ylab("time/seconds")+ggtitle("Comparation of running time
in R, C and Welford")
# times[-c(3,6,9),] means exclude the real time

#############3
library(biglm)
#I calculate the less time(y), trip_time(x) and sucharge(x2) which can see in my
git, just dowload it and use the command below

data = as.data.frame(cbind(y=y, x=x, x2=x2))
f = y~x+x2
chunk = list()
# similar with the example of
http://cran.r-project.org/web/packages/biglm/biglm.pdf
chunk[[1]] = data[1:10000000,]
fit = biglm(f, chunk[[1]], na.action=na.omit)
```

```
for (i in 1 :17)#because the number of row in data is 173179759
{chunk[[i+1]] = data[10000000*i+1:10000000*(i+1), ];
  fit = updata(fit, chunk[[i+1]])
}
```

Appendix 3 C code

1 diff

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

/*
function for calculation the less time of between total and tolls
input:
filename: ths file that we want to calculate, pay attention the filename should include the whole path of the file
fee: it is a array, which is the less value of the the whole file in each line
output:
The same as input, and the fee is what we need of the less value
*/

void cdif(char **filename, double *fee)
{
    FILE *fp = fopen(filename[0], "r"); /* only read the filname, and the file name is such as ""/home/data/NYCTaxis/trip_fare_1.csv */

    char line[1024];
    char *toll = NULL, *total = NULL;
    int n = 0, i;

    fgets(line,1024,fp); /* except the first line which is for */
```

```
    while( fgets(line,1024,fp) ) /*if not end of the file*/
    {
        char *b = line;
        for(i = 0; i<10; i++){
            toll = strsep(&b, ","); /*tolls_amount*/
        }
            total = strsep(&b, "/n"); /* total_amount*/
        fee[n] = atof(total) - atof(toll);
        n = n + 1;
    }
}
```

2 suff:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <math.h>
/*
```

The suff function is a function which used to calculate the sufficient statistics (mentioned in report part2.2.1Methods)

Input:

fare: char, the file name of trip_fare, pay attention to put whole path of the file

data: char, the file name of trip_data

x: one of the sufficient statistics, sum x(trip time), should be double because there might be decimals

y: sum of y(less value of total and tolls), double for same reason as x

xy: sum of trip time *less value, double for same reason as x

x2: the square sum of trip time, double for same reason as x

output:

the same as the input, the value that we need is x, y, xy, x2

```
*/
```

```
void suff(char **fare, char **data, double *x, double *y, double *xy, double *x2)      /* argv[] is the file name */
{
        FILE *fp = fopen(fare[0], "r");
        FILE *dp = fopen(data[0], "r"); /* only read the filname, and the file name is such as ""/home/data/NYCTaxis/trip_fare_1.csv */

        char linef[1024], lined[1024];
        char *toll = NULL, *total = NULL, *t = NULL;
        int i, j;
        double less = 0, tt = 0, sumx = *x, sumy = *y, sumxy = *xy, sumx2 = *x2;

        fgets(linef,1024,fp);    /* except the first line which is for the names for fare */
        fgets(lined,1024,dp); /* except the first line which is for the name for data */

        while( fgets(linef,1024,fp) ) /*if not end of the file*/
        {     fgets(lined,1024,dp);

              char *f = linef;
              char *d = lined;

              /*calculate the sum of total amount less then tolls */
              for(i = 0; i<10; i++){
                    toll = strsep(&f, ","); /*tolls_amount*/
              }
                    total = strsep(&f, "/n"); /* total_amount*/
              less = atof(total) - atof(toll);
```

```
                sumy = sumy + less;

                /*calculate the sum of trip time*/
                for(j = 0; j<9; j++){
                        t = strsep(&d, ","); /*tolls_amount*/
                }
                tt = atof(t);
                sumx = sumx + tt;

                /*the other sufficient statistics, sum of xy and the sum of
square x */
                sumxy = sumxy + less*tt;
                sumx2 = sumx2 + pow(tt,2);

            }
        *x = sumx;
        *y = sumy;
        *xy = sumxy;
        *x2 = sumx2;

}
```

3 welford and one stable covariance method

```
/*Welford's method to calculate*/
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <math.h>

/*
x -- the variable which is to calculate it's variance
length -- the length of the x, it is int
var -- the variance of x that we need, it is double
```

```
*/
void welf_var( double *x, int *length, double *var)
{
    int N = *length, i;
    double M = 0, S = 0, Mprev = 0; /*Mprev means the previous value of
M*/

    /*use the iteration mentioned in the report part 2.2.1*/
    for ( i = 0; i < N; i++) {
        Mprev = M;
        M += (x[i] - Mprev)/(i+1);
                S += (x[i] - Mprev)*(x[i] - M);
            }
*var = S/(N-1);
}
```

4 Covariance

```
/*one-pass algorithm to calculate the covariance*/
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <math.h>

/*
x -- the variable which is to calculate it's variance
length -- the length of the x, it is int
var -- the variance of x that we need, it is double
mx -- mean of x
my -- mean of y
mxprev -- mx(mean of x) previous value
```

```
myprev --my(mean of y) previous value


*/
void mycov(double *x, double *y, int *length, double *cov, double *meanx, double *meany)
{
    int N = *length, i;
    double C = 0,mx = 0, my = 0, mxprev = 0, myprev = 0; /* Cprev means the previous value of C */

    for ( i = 0; i < N; i++) {
        mxprev = mx;
                myprev = my;
                mx += (x[i] - mxprev)/(i+1);
                my += (y[i] - myprev)/(i+1);
                C += (x[i] - mx)*(y[i] - myprev);
            }
*cov = C/(N-1);
*meanx = mx;
*meany = my;
}
```