Homework 7

Problem 2

In this problem, I compute the sums of squares(SST) using two traditional methods: for loop, verctorization form and two parallel methods: dopar and parSapply. I would like to see whether the parallel computing indeed improve the computing speed. The following code implement these four methods:

```r
########### Problem 2 ##########
set.seed(12345)
y <- rnorm(n = 1e+07, mean = 1, sd = 1)
ymean <- mean(y)
l <- length(y)
#-----   for loop   --------
sstotal1 <- 0
t1 <- system.time({
    for (i in 1:length(y)) {
        sstotal1 <- sstotal1 + (y[i] - ymean)^2
    }
})
#-----  vectorization -------
t2 <- system.time({
    sstotal2 <- t(y - ymean) %*% (y - ymean)
})
#------  dopar --------
cl <- makeCluster(2)
registerDoParallel(cl)
clusterExport(cl, varlist = c("y", "ymean", "l"))
t3 <- system.time({
    sstotal3 <- foreach(a = y[1:1e+05], b = rep(ymean, 1e+05),
        .combine = "+") %dopar% {
        (a - b)^2
    }
})
stopCluster(cl)

#------parSapply------
core_fun <- function(x) {
    sum(x - ymean)^2
}
cl <- makeCluster(2)
clusterExport(cl, varlist = c("core_fun", "ymean"))
registerDoParallel(cl)
t4 <- system.time({
    sstotal4 <- parSapply(cl, y, function(x) core_fun(x))
    sstotal4 <- sum(sstotal4)
})
stopCluster(cl)
# to show the computing times in a table
time <- data.frame(loop = as.vector(t1)[1:3], vectorize = as.vector(t2)[1:3],
    dopar = as.vector(t3)[1:3], parSapply = as.vector(t4)[1:3])
rownames(time) <- c("user", "system", "elapsed")
kable(time, caption = "Computing Times with Different Method")
```

Table 1: Computing Times with Different Method

	loop	vectorize	dopar	parSapply
user	6.981	0.192	46.331	13.400
system	0.087	0.078	2.763	1.917
elapsed	7.144	0.276	52.192	31.338

When using dopar to do parallel computing, my computer never ends running. So I just use 1% of y to compute the SST and the time is already the longest one. From the above table we can see that in this particular problem, vectorization performs the best. But what surprised me is that neither of the parallel computing is faster than the for loop. So I think the parallel computing doesn't always more efficient than the traditional computings.

Problem 3

In the gradient descent problem, since the core part is iteration and the next iteration will use the value from the last one, so I didn't come up a good way to parallel on the iterations. And I'm thinking parallelize around the step size to see the influence of it.

```
########## Problem 3 ############
#----generate the data---
set.seed(1256)
theta <- as.matrix(c(1, 2), nrow = 2)
X <- cbind(1, rep(1:10, 10))
h <- X %*% theta + rnorm(100, 0, 0.2)
h_0 <- function(X, theta) {
    hval <- X %*% theta
    return(hval)
}
#----- set initial values----
tolerance <- tolerance <- 1e-05

#----parallel computing---
cl <- makeCluster(2)
registerDoParallel(cl)
clusterExport(cl, varlist = c("h_0", "theta_vec", "tolerance"))

theta.est <- foreach(alpha = 1:10/5000, .combine = "cbind",
    .export = "h_0") %dopar% {
    theta0 <- 1.1
    theta1 <- 1.9
    theta_vec <- c(theta0, theta1)
    while (TRUE) {
        h_diff <- h_0(X, theta = theta_vec) - h
        theta0_new <- theta_vec[1] - alpha * mean(h_diff)
        theta1_new <- theta_vec[2] - alpha * mean(h_diff *
            X[, 2])
        theta_new <- c(theta0_new, theta1_new)
        if ((abs(theta0 - theta0_new) < tolerance) | (abs(theta1 -
            theta1_new) < tolerance)) {
            break
        } else {
            theta_vec <- theta_new
        }
    }
    theta_vec
}
stopCluster(cl)
step.size <- paste("alpha", 1:10/5000, sep = "=")
colnames(theta.est) <- step.size
rownames(theta.est) <- c("beta1", "beta2")
kable(theta.est, caption = "Estimation with Different Step Size",
    digits = 4, header = F)
```

(Above code kept crashed when I trying to knit, so when I knit it I didn't really run the code.)

Problem 4

In this problem I used bootstrap to estimate the betas in the linear regression model: $Y = X\beta + \epsilon$.
The procedure is following:
For $b \in \{1, ..., B\}$

- Sample from the original dataset with replacement and the sample size should be the same as the original dataset.

- Calculate $\hat{\beta}$ using lm with the sampled data.

After we got B $\hat{\beta}$, we can compute the mean of those $\hat{\beta}$s to get the esimation of β from bootstrap.

Since the thing we did inside the for loop is repeated each time. So I used dopar to parallel the computing in order to improve the computing efficiency.

```
######## Problem 4 ########
#-----generate the data-----------
set.seed(1267)
n <- 200
B <- 1000
X <- 1/cbind(1, rt(n, df = 1), rt(n, df = 1), rt(n, df = 1))
beta <- c(1, 2, 3, 0)
Y <- X %*% beta + rnorm(100, sd = 3)

# ------- construct core functions --------- sample.est:
# sampling from the data once and use lm to get beta
# estimation output: estimation of beta
sample.est <- function() {
    id <- sample(1:n, size = 200, replace = T)
    x.new <- X[id, ]
    y.new <- Y[id, ]
    beta.hat <- coefficients(lm(y.new ~ x.new + 0))
    return(beta.hat)
}
#--------do the parallel computing using dopar----------
cl <- makeCluster(3)
clusterExport(cl, varlist = c("core_fun", "ymean"))
registerDoParallel(cl)
beta.est <- foreach(b = 1:B, .combine = data.frame) %dopar%
```

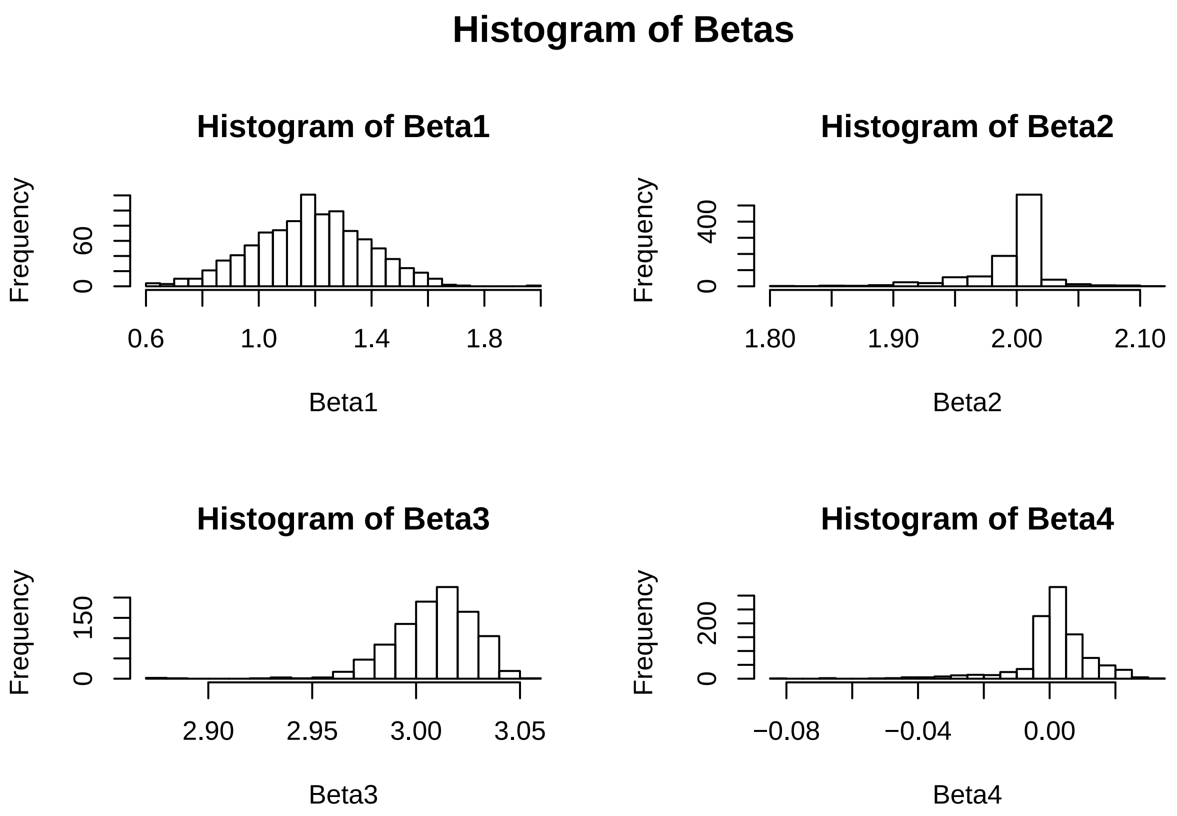
```r
    {
        data.frame(sample.est())
    }
stopCluster(cl)

#--------clean and summary the boostrap betas-------------------
beta.est <- t(beta.est)
colnames(beta.est) <- c("beta0", "beta1", "beta2", "beta3")
mean_beta <- apply(beta.est, MARGIN = 2, FUN = mean)
mean_beta <- round(mean_beta, 4)
conint.beta <- apply(beta.est, 2, quantile, c(0.025, 0.975))
conint.beta <- round(conint.beta, 4)
conint.vec <- c(paste("[", conint.beta[1, 1], ",", conint.beta[2,
    1], "]", sep = ""), paste("[", conint.beta[1, 2], ",",
    conint.beta[2, 2], "]", sep = ""), paste("[", conint.beta[1,
    3], ",", conint.beta[2, 3], "]", sep = ""), paste("[",
    conint.beta[1, 4], ",", conint.beta[2, 4], "]", sep = ""))
sum.beta <- data.frame(mean = mean_beta, Confidence_Interval = conint.vec)
kable(t(sum.beta), caption = "Summary of Beta Estimation")
```

Table 2: Summary of Beta Estimation

	beta0	beta1	beta2	beta3
mean	1.1850	1.9956	3.0083	0.0016
Confidence_Interval	[0.7945,1.5592]	[1.9094,2.0398]	[2.969,3.0386]	[-0.0292,0.021]

```r
#------create histogram for betas----------
par(oma = c(0, 0, 2.5, 0))
layout(matrix(1:4, nrow = 2, byrow = T))
for (i in 1:4) {
    hist(beta.est[, i], main = paste("Histogram of Beta",
        i, sep = ""), xlab = paste("Beta", i, sep = ""),
        breaks = 20)
}
layout(1)
title(main = "Histogram of Betas", outer = T)
```



From the above summary table, we could see that the mean of boostrapped βs gives us a pretty good estimation of the true parameters. And the 95% confidence intervals of betas also include the true value. The histogram shows that the β_2 and β_4 are highly centered around the true values while $\beta 1$ and $\beta 3$ is kind of dispersing around the true value.