lab_09

Problem 1: Think

Image processing

Batch-processing tasks

Physics processing

Problem 2: Before you

generates a n x k dataset with all its entries distributed poission with mean lambda

```
set.seed(1235)
fun1 <- function(n = 100, k = 4, lambda = 4) {
  x <- NULL

  for (i in 1:n)
    x <- rbind(x, rpois(k, lambda))

  return(x)
}
f1 <- fun1(1000,4)
mean(f1)
```

```
## [1] 4.03725
```

```
fun1alt <- function(n = 100, k = 4, lambda = 4) {
  # YOUR CODE HERE
  x = matrix(rpois(n*k, lambda), ncol = 4)

  return(x)
}

# Benchmarking
microbenchmark::microbenchmark(
  fun1(),
  fun1alt()
)
```

```
## Unit: microseconds
##       expr     min       lq      mean  median      uq      max neval
##     fun1() 296.438 325.7955 436.12339 350.066 519.600 1482.009   100
##  fun1alt()  18.545  19.9890  63.95856  20.970  27.054 4005.512   100
```

```r
d <- matrix(1:16, ncol=4)
d
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    5    9   13
## [2,]    2    6   10   14
## [3,]    3    7   11   15
## [4,]    4    8   12   16
```

```r
diag(d)
```

```
## [1]  1  6 11 16
```

```r
d[c(1,6,11,16)]
```

```
## [1]  1  6 11 16
```

```r
d[cbind(1:4, 1:4)]
```

```
## [1]  1  6 11 16
```

Find the column max (hint: Checkout the function max.col())

```r
# Data Generating Process (10 x 10,000 matrix)
set.seed(1234)
x <- matrix(rnorm(1e4), nrow=10)
M <- matrix(runif(12), ncol=4)

# Find each column's max value
fun2 <- function(x) {
  apply(x, 2, max)
}

fun2(x=M)
```

```
## [1] 0.8765384 0.6978030 0.9274793 0.4548257
```

```r
fun2alt <- function(x) {
  # YOUR CODE HERE
  max.col(t(x))
}

fun2alt(x=M)
```

```
## [1] 2 2 3 3
```

```r
# Benchmarking
microbenchmark::microbenchmark(
  fun2(x),
  fun2alt(x)
)
```

```
## Unit: microseconds
##         expr      min        lq      mean   median       uq       max neval
##     fun2(x) 1027.258 1078.7125 1280.8483 1111.548 1175.533 10948.847   100
##  fun2alt(x)   93.822  123.4185  144.0695  129.733  134.748  1450.178   100
```

Problem 3: Parallelize everyhing

```
library(parallel)

my_boot <- function(dat, stat, R, ncpus = 1L) {

  # Getting the random indices
  n <- nrow(dat)
  idx <- matrix(sample.int(n, n*R, TRUE), nrow=n, ncol=R)

  # Making the cluster using `ncpus`
  # STEP 1: GOES HERE
  cl <- makePSOCKcluster(4)
  # STEP 2: GOES HERE
  clusterSetRNGStream(cl, 123)
  clusterExport(cl, c("stat", "dat","idx"), envir = environment())


    # STEP 3: THIS FUNCTION NEEDS TO BE REPLACES WITH parLapply
  ans <- lapply(seq_len(R), function(i) {
    stat(dat[idx[,i], , drop=FALSE])
  })

  # Coercing the list into a matrix
  ans <- do.call(rbind, ans)

  # STEP 4: GOES HERE
  ans

}
```

Use the previous pseudocode, and make it work with parallel. Here is just an example for you to try:

```
# Bootstrap of an OLS
my_stat <- function(d) coef(lm(y ~ x, data=d))

# DATA SIM
set.seed(1)
n <- 500; R <- 1e4

x <- cbind(rnorm(n)); y <- x*5 + rnorm(n)

# Checking if we get something similar as lm
ans0 <- confint(lm(y~x))
ans1 <- my_boot(dat = data.frame(x, y), my_stat, R = R, ncpus = 2L)
```

```
# You should get something like this
t(apply(ans1, 2, quantile, c(.025,.975)))
```

```
##                   2.5%      97.5%
## (Intercept) -0.1386903 0.04856752
## x            4.8685162 5.04351239
```

```
##                   2.5%      97.5%
## (Intercept) -0.1372435 0.05074397
## x            4.8680977 5.04539763
ans0
```

```
##                   2.5 %     97.5 %
## (Intercept) -0.1379033 0.04797344
## x            4.8650100 5.04883353
```

```
##                   2.5 %     97.5 %
## (Intercept) -0.1379033 0.04797344
## x            4.8650100 5.04883353
```

Check whether your version actually goes faster than the non-parallel version

```
system.time(my_boot(dat = data.frame(x, y), my_stat, R = 4000, ncpus = 1L))
```

```
##    user  system elapsed
##   2.994   0.064   3.819
```

```
system.time(my_boot(dat = data.frame(x, y), my_stat, R = 4000, ncpus = 2L))
```

```
##    user  system elapsed
##   2.909   0.035   3.287
```