ps5

Kehsin Su Esther 3033114294

October 17, 2017

```
knitr::opts_chunk$set(tidy = TRUE, cache = TRUE)
library(pryr)

## Warning:  package 'pryr' was built under R version 3.3.3
```

1 Q1

(skip)

2 Q2

There are 64 bits used to store double. Since 1 bit is used for sign and 11 bits are allocated to the exponent, there are only 52 bits for significand As show in the following, the approach after 2^{52} will become 2 each calculation. Hence, when calculating exceed 2^{52}, it will become overflow and lose precision.

```
options(digits = 22)
bits(2^52 - 1)

## [1] "01000011 00101111 11111111 11111111 11111111 11111111 11111111 11111110"

bits(2^52)

## [1] "01000011 00110000 00000000 00000000 00000000 00000000 00000000 00000000"

bits(2^52 + 1)

## [1] "01000011 00110000 00000000 00000000 00000000 00000000 00000000 00000001"

bits(2^53 - 1)

## [1] "01000011 00111111 11111111 11111111 11111111 11111111 11111111 11111111"

bits(2^53)

## [1] "01000011 01000000 00000000 00000000 00000000 00000000 00000000 00000000"

# lose precision when over 2^52, and will use the closet number 2^53
2^53 + 1  #only can approximate to 2^(53-52) precision

## [1] 9007199254740992

# can approach precisely since it can approach with 2^1
2^53 + 2  #can approach to 2^53+2^1 precision

## [1] 9007199254740994
```

```r
2^54

## [1] 18014398509481984

bits(2^54)

## [1] "01000011 01010000 00000000 00000000 00000000 00000000 00000000 00000000"

2^54 + 2  #only can approximate to 2^(54-52) precision

## [1] 18014398509481984

bits(2^54 + 2)  #unable to approach precision over 2^54 within 2^2

## [1] "01000011 01010000 00000000 00000000 00000000 00000000 00000000 00000000"

2^54 + 3  #unable to approach precision over 2^54 within 2^2

## [1] 18014398509481988

bits(2^54 + 3)  #unable to approach precision over 2^54 within 2^2

## [1] "01000011 01010000 00000000 00000000 00000000 00000000 00000000 00000001"

2^54 + 4  #can approach to 2^54+2^2

## [1] 18014398509481988

2^54 + 7  #unable to approach precisely

## [1] 18014398509481992

2^54 + 8

## [1] 18014398509481992
```

3 Q3

3.1 (a)

As show in the followng example, the time of double copy is nearly two times of integer copy. Hence, we can conclude that the running speed will increase if we use less precise numbers.

```r
library(data.table)
a <- as.integer(rnorm(1e+06, 2, 10))
typeof(a)

## [1] "integer"

b <- rnorm(1e+06, 2, 10)
typeof(b)

## [1] "double"

(at <- system.time(a1 <- copy(a)))

##    user  system elapsed
##       0       0       0
```

```
(bt <- system.time(b1 <- copy(b)))

##    user  system elapsed
##    0.03    0.00    0.03

bt[3]/at[3]

## elapsed
##     Inf
```

3.2 (b)

No, as showing below, sometimes the subset time of double faster than integate, but sometimes integers are faster.

```
(ats <- system.time(a2 <- a[1:1e+06/2]))

##    user  system elapsed
##    0.00    0.02    0.02

(bts <- system.time(b2 <- b[1:1e+06/2]))

##    user  system elapsed
##    0.02    0.00    0.02

bts[3]/ats[3]

## elapsed
##       1

(ats2 <- system.time(a2 <- subset(a, a < 2)))

##    user  system elapsed
##    0.02    0.01    0.03

(bts2 <- system.time(b2 <- subset(b, b < 2)))

##    user  system elapsed
##    0.01    0.00    0.01

bts2[3]/ats2[3]

##   elapsed
## 0.3333333
```

4 Q4

4.1 (a)

The time spend will increase when divide the column into more parts. Since when there are more parts, there will be more communication and more tasks a worker need to handle.

4.2 (b)

Approach A:

for each worker, they will receive n*n elements from X and $n * m$ elements from Y, so they will totally receive n*(n+m) elements. The result will be a matrix with $n * m$ elements. Hence, at a single moement, when each worker do single task, they will need to $n * n + n + m + n * m$ elements memeory space to do calcuation and store the result. They will totally communicate $p*(n*n+n+m+n*m)$ elements, which is equal to n^2p+2n^2

Approach B:

for each worker, they will need to do p tasks. Under each task, they will receive m*n elements from X and $n * m$ elements from Y. The result for a single task will be a matrix with $m * m$ elements. When working on a single task, it will require to store $2nm + m^2$ elements, which conposed of the elements from X, Y and output. Since there are p workers and p tasks for each work, there will be $p^2 * 2nm + m^2$ elements sent to communicate, which equal to $2n^2p + n^2$.

Conclusion:

As appproach B seperate matrix into more parts, each worker need to do p tasks. However, the task size each worker deal is smaller than approach A and each worker can only deal with each task in a single moment, so the memeory used for approach B in the single moment is less than approach A. However, after sum up how many elements need to communicate totally, approach B needs to communicate more elements.

Approach	A	B
#workers	p	p
#tasks/worker	1	p
#elements in X/task	n*n	m*n
#elements in Y/task	n*m	n*m
#elements in output/task	n*m	m*m
elements memory used/task	n$^2 + 2nm$	m$^2 + 2nm$
#elements communicate totally	$n^2p + 2n^2$	2n$^2p + n^2$
memory used/time	greater	less
communicate	less	greater

```
# trial
require(parallel)  # one of the core R packages

## Loading required package:  parallel

require(doParallel)

## Loading required package:  doParallel
## Warning:  package 'doParallel' was built under R version 3.3.3
## Loading required package:  foreach
## Warning:  package 'foreach' was built under R version 3.3.3
## Loading required package:  iterators
## Warning:  package 'iterators' was built under R version 3.3.3

library(foreach)
cores <- detectCores(logical = F)
cl <- makeCluster(cores)
registerDoParallel(cl, cores = cores)
# split data by ourselves
test_mat1 <- matrix(rnorm(300 * 300, 3, 3), 300, 300)
test_mat2 <- matrix(rnorm(300 * 300, 3, 3), 300, 300)
```

```
chunk.size <- ncol(test_mat1)/cores

system.time(result1 <- foreach(i = 1:cores, .packages = "pryr") %dopar% {
    test_mat1 %*% test_mat2[, (1 + chunk.size * (cores - 1)):(chunk.size * cores)]
})

##    user  system elapsed
##    0.04    0.00    0.26

system.time(result2 <- foreach(i = 1:300, .packages = "pryr") %dopar% {
    test_mat1 %*% test_mat2[, i]
})

##    user  system elapsed
##    0.14    0.00    0.16

system.time(result3 <- foreach(test_mat2 = iter(test_mat2, by = "col"), .combine = cbind) %dopar%
    (test_mat1 %*% test_mat2))

##    user  system elapsed
##    0.09    0.02    0.14

system.time(result4 <- foreach(test_mat1 = iter(test_mat1, by = "row"), test_mat2 = iter(test_mat2,
    by = "col"), .combine = cbind) %dopar% (test_mat1 %*% test_mat2))

##    user  system elapsed
##    0.14    0.00    0.14

stopImplicitCluster()
stopCluster(cl)
```

5 Q5

Since the number system we use is 10, which composed of 2 and 5. Hence, only numbers contains of 2 and 5 can be finited represented. Also, the unit runoff equal to half of the machine epsilon, which has an upper bound on rounding that equal to $\frac{1}{2}$ b^{1-p} .Hence, the multiples of 5 can be precised reveal in the system.