PS4

Mengying Yang

October 18, 2017

Problem 2

part a)

1 in base 2 is 0 01111111111 0000000000000000000000000000000000000000000000000000

double: $(-1)^0 \times 1 \times 2^{(1023-1023)}$

2 in base 2 is 0 10000000000 0000000000000000000000000000000000000000000000000000

double: $(-1)^0 \times 1 \times 2^{(1024-1023)}$

3 in base 2 is 0 10000000000 1000000000000000000000000000000000000000000000000000

double: $(-1)^0 \times 1.1 \times 2^{(1024-1023)}$

4 $(-1)^0 \times 1 \times 2^{(1025-1023)}$

5 $(-1)^0 \times 1.01 \times 2^{(1025-1023)}$

...

$2^{53} - 2$ 0 10000110011 1111111111111111111111111111111111111111111111111110

$(-1)^0 \times 1.1111111111111111111111111111111111111111111111111110 \times 2^{(1075-1023)}$

$2^{53} - 1$ 0 10000110011 1111111111111111111111111111111111111111111111111111

$(-1)^0 \times 1.1111111111111111111111111111111111111111111111111111 \times 2^{(1075-1023)}$

```
library(pryr)
bits(1)

## [1] "00111111 11110000 00000000 00000000 00000000 00000000 00000000 00000000"

bits(2)

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

bits(3)

## [1] "01000000 00001000 00000000 00000000 00000000 00000000 00000000 00000000"

bits(4)

## [1] "01000000 00010000 00000000 00000000 00000000 00000000 00000000 00000000"

bits(2^53-2)

## [1] "01000011 00111111 11111111 11111111 11111111 11111111 11111111 11111110"

bits(2^53-1)

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

part b)

2^{53} in base 2 is 0 10000110100 0000000000000000000000000000000000000000000000000000

double: $(-1)^{\mathbf{0}} \times \mathbf{1} \times \mathbf{2}^{(\mathbf{1076-1023})}$
$\mathbf{2^{53}+1}$ in base 2 is 0 10000110100 0000000000000000000000000000000000000000000000000000
double: $(-1)^{\mathbf{0}} \times \mathbf{1} \times \mathbf{2}^{(\mathbf{1076-1023})}$

because there are 52-precision-number stored, but when a integer more than 2^{53} will need 53-precision-number to store. The last digit will be rounded. Therefore, the abusolute spacing for this magnitude will be $2^{53} \times (2^{-52}) = 2$. That's why 2^{53} cannot be represent exactly.

$\mathbf{2^{53}+2}$ in base 2 is 0 10000110100 0000000000000000000000000000000000000000000000000001
double: $(-1)^{\mathbf{0}} \times \mathbf{1.0000000000000000000000000000000000000000000000000001} \times \mathbf{2}^{(\mathbf{1076-1023})}$

```
options(digits = 22)
2^53+2

## [1] 9007199254740994

2^53

## [1] 9007199254740992

2^53+1

## [1] 9007199254740992

2^53+2-(2^53+1) # The spacing of this magnitude is 2

## [1] 2

.Machine$double.eps*(2^53+1)

## [1] 2

bits(2^53)

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

bits(2^53+1)

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

bits(2^53+2)

## [1] "01000011 01000000 00000000 00000000 00000000 00000000 00000000 00000001"
```

part c)
because there are 52-precision-number stored, but when a integer more than $2 * 54$ will need 54-precision-number to store. The last two digits don't have a space to store therefore they will be rounded. The abusolute spacing for integer more than 2^{54} will be $2^{54} \times (2^{(-52)}) = 4$.

$\mathbf{2^{54}}$ in base 2 is 0 10000110101 0000000000000000000000000000000000000000000000000000
double: $(-1)^{\mathbf{0}} \times \mathbf{1} \times \mathbf{2}^{(\mathbf{1077-1023})}$
$\mathbf{2^{54}+1}$ in base 2 is 0 10000110101 0000000000000000000000000000000000000000000000000000
(only have 52 precision-number, so the $\mathbf{+1}$ at here cannot be represent)
double: $(-1)^{\mathbf{0}} \times \mathbf{1} \times \mathbf{2}^{(\mathbf{1077-1023})}$

```
.Machine$double.eps*(2^54)

## [1] 4

.Machine$double.eps*(2^54+1)

## [1] 4

.Machine$double.eps*(2^54+2)

## [1] 4

bits(2^54)

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

bits(2^54+1)

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

Problem 3

```
options(digits = 10)
library(data.table)

##
## Attaching package:  'data.table'
## The following object is masked from 'package:pryr':
##
##    address

library(microbenchmark)
Integervector <- 1:10000
numericvector <- Integervector /5.55
microbenchmark(copy(Integervector))

## Unit: microseconds
##                 expr   min     lq     mean median     uq     max neval
##  copy(Integervector) 4.927 13.959 18.00304 15.601 17.243 135.481   100

microbenchmark(copy(numericvector))

## Unit: microseconds
##                 expr    min     lq     mean median      uq     max neval
##  copy(numericvector) 32.845 44.545 79.96716 47.419 54.3985 2741.64   100
```

Yes, copying a large integer vector is faster copying a same size numeric vector.

```
microbenchmark(sample(Integervector,5000))

## Unit: microseconds
##                         expr    min     lq      mean median       uq
##  sample(Integervector, 5000) 81.289 82.726 127.21296 86.831 113.1065
##       max neval
##  1247.245   100
```

```
microbenchmark(sample(numericvector,5000))

## Unit: microseconds
##                         expr    min   lq      mean  median      uq
##  sample(numericvector, 5000) 80.057 81.7 117.57742 89.7055 106.948
##       max neval
##  1266.131   100
```

They use about the same time.

Problem 4

a)

It might be better to break up Y into p blocks rather than n individual conlumn-wise computation because the following reasons:
1) Usually the avaiable cores to help us calculate is less than n, the more effecient way is to use the fixed number of cores avaiable on our computers as to split up the tasks.
2) This is a tradeoff problem. Although seperate the matrix to n part to calculate will save the time of waiting to calculate when we seperate to p parts, communications cost between each thread are also time-comsuming. Sometimes, maybe the waiting time is less than the communication time if we break up tasks to too many part.
Therefore, breaking up to p parts usually is better than using single core and is also better than breaking into n parts individual column in this situation.

b)

1
The memory for approach A uses is store $\boldsymbol{(n \times n + n \times m) \times p + n \times m \times p = n^2p + 2n^2}$ numbers ($\boldsymbol{(n \times n + m \times p) \times p}$ is number when doing calculation, $\boldsymbol{n \times m \times p = n \times n}$ is memory to save result)

The memory for approach B uses is store $\boldsymbol{(m \times n + n \times m + m \times m) \times p = 2n^2 + n \times m}$ numbers

2
The total number of pass to the workers for Approach A and B are $\boldsymbol{(n \times n + n \times m + n \times m) \times p = n^2p + 2n^2}$ and $\boldsymbol{(n \times m + n \times m + m \times m) \times p \times p = 2n^2p + n^2}$ respectively
Approach B is better for minimzing memory usage.
Approach A is better for minimzing communication

Extra Credit

```
bits(0.25)

## [1] "00111111 11010000 00000000 00000000 00000000 00000000 00000000 00000000"

bits(0.04)

## [1] "00111111 10100100 01111010 11100001 01000111 10101110 00010100 01111011"

bits(0.39)
```

```
## [1] "00111111 11011000 11110101 11000010 10001111 01011100 00101000 11110110"

0.25+0.04 == 0.39

## [1] FALSE
```

This related to the how these numbers represented by binary. If the two numbers' binary sum more than 52 digits and the 53th digits is one, then the number will round up, if not then the number will be truncated. Therefore, it will depend on the summation of the binary of the two numbers. So some pattern is hard to find. Beyond that, When the number is 2 to the power of an integer(0.5 is; 0.1, 0,2,0.3 are not), then the number can be represent in R exactly. For 0,1, 0.2, 0.3, they are rounded to the nearest number and when they add to a power of 2, their magnitude error space is same. For example, When 0.2 + 0.3, 0.2 round up one error space and 0.3 round down one error space, when they add together the positive spacing error plus negative spacing error who cancel each other and get the exact 0.5.