Problem Set 5

Hsieh Cheng Han

October 17, 2017

2.

(Ans)
First we determine how integers from 1 to 2^{53} are stored exactly. Apparently $S = 0$, we only concern about d and e (here we replace e - 1023 with e for convinence).
$1 = 1_{(2)} = 1.0 \times 2^0$, so 1 is stored as d = 0 and e = 0.
$2 = 10_{(2)} = 1.0 \times 2^1$, so 2 is stored as d = 0 and e = 1.
$3 = 11_{(2)} = 1.1 \times 2^1$, so 3 is stored as d = 1 and e = 1.
...
2^{53} - 1 = $\underbrace{1...1}_{53}{}_{(2)} = 1.\underbrace{1...1}_{52} \times 2^{52}$, so 2^{53} - 1 is stored as d = $\underbrace{1...1}_{52}$ and e = 52.
$2^{53} = 1\underbrace{0..0}_{53}{}_{(2)} = 1.0 \times 2^{53}$, so 2^{53} is stored as d = 0 and e = 53.
In general, x = $1d_1d_2...d_{n(2)}$, n $\leq$ 52 and $d_k = 0 (k = \text{n+1} \sim 52)$, e = n.
Now consider $2^{53} + 1$:
$2^{53} + 1 = 1\underbrace{0..0}_{52}1_{(2)} = 1.\underbrace{0...0}_{52}1 \times 2^{53}$, so the corresponding d should be $\underbrace{0...0}_{52}1$, which exceeds the constraint that d has only 52 digits.
Finally, consider $2^{53} + 2$:
$2^{53} + 2 = 1\underbrace{0..0}_{51}1_{(2)} = 1.\underbrace{0...0}_{51}1 \times 2^{53}$, so the corresponding d = $\underbrace{0...0}_{51}1$ and e = 53.
That is, both 2^{53} and $2^{53} + 2$ can be stored exactly while 2^{53} - 1 cannot.
Similarily, all d's corresponding to $2^{54} + 1$, $2^{54} + 2$, $2^{54} + 3$ have digits number greater than 52 while 2^{54} and $2^{54} + 4$ don't.
The reason of these "spacing of numbers" phenomena is due to the constraint that d has only 52 digits. In fact, as e increases, the spacing of numbers increases as well. More precisely speaking, we have the formula that spacing of numbers = 2^{e-52}.

```
# Test the storage of 2^53 - 1, 2^53, 2^53 + 1, 2^53 + 2 :
# We only notice last 52 bits.
library(pryr)
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"
```

```
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"
```

```
# 2^53 - 1 has overflow of d, causing its storage being equivalent with that of 2^53.
```

3.

(a)

Is it faster to copy a large vector of integers than a numeric vector of the same length in R? Do a bit of experimentation and see what you find.

```
# We want to use copy() within data.table package.
library(data.table)

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

# Set the length of the vectors.
n <- 1e7
x1 <- 1 : n
# Now that x1 is a large vector of integers, we copy it to vector y.
system.time(y <- copy(x1))

##    user  system elapsed
##   0.041   0.012   0.053

# Then, create a large numeric vector with the same length.
x2 <- as.numeric(x1)
system.time(z <- copy(x2))

##    user  system elapsed
##   0.052   0.023   0.078
```

From the elapsed times we can find it faster to copy a large vector of integers than a numeric vector of the same length in R. I think the reason is the twice size of double floating numbers compared to single floating numbers.

(b)

Is it faster to take a subset of size k n/2 from an integer vector of size n than from a numeric vector of size n?

```
# Subset the first n/2 elements in a large vector of integers.
system.time(y <- x1[1:(n/2)])

##    user  system elapsed
##   0.044   0.022   0.070

system.time(z <- x2[1:(n/2)])

##    user  system elapsed
##   0.033   0.007   0.040
```

From the elapsed times we can find it even slower when coping a large vector of integers compared to a numeric vector of the same length in R. It may be the reason that subsetting the vector doesn't have relation with their elements' size.

4.

(a)

(Ans)
The communication overhead of starting and stopping the tasks will reduce efficiency. Break up the matrix into individual column may cause such result and may not be better than breaking up the matrix into column blocks.(It's a kind of trade-off between dimension of parallelization and its communication cost)

(b)

(Ans)
We first consider (1) The amount of memory used :

Approach A : Each task is to calculate the multiplication of n $\times$ n matrix X and n $\times$ m submatrix of Y. The input of the task uses n*n + n*m = n(n+m) units of memory(assuming each number occupies 1 unit memory). During the calculating process, extra n^2m units of memory is used(An element in X multipling an element in submatrix of Y occupies one extra unit memory). Finally, the output is a n $\times$ m matrix, which uses nm units of memory. As a result, the total amount of used memory is n(n+m) + n^2m + nm = n^2m + n^2 + 2nm units. Since there are simultaneously n/m tasks are executed, the total amount of memory used for all tasks at any single time is n^3(1+1/m) + $2n^2$ units.

Approach B : Analogous to approach A, we only need to replace matrix X with its m $\times$ n submatrix. Each task spends 2nm(input) + m^2n(extra) + m^2(output) units of memory. Since there are simultaneously $(n/m)^2$ tasks are executed, the total amount of memory used for all tasks at any single time is n^3(1+2/m) + n^2 units.

The difference of amount of memory used between approach A and B (A-B)is n^2(1-n/m) < 0. That is, **approach A** performs better for minimizing memory.

Then consider (2) The communication cost :

Approach A : Each task we need to pass n(n+m) numbers to the worker. Afterwards the worker passes back nm numbers to the master. As a result, the total communication cost is (n/m)(n(n+m) + nm) = n^2(2+n/m) units.

Approach B : Analogous to approach A, the total communication cost is $(n/m)^2(2nm + m^2) = n^2(1+2n/m)$ units.

The difference of communication cost between approach A and B (A-B)is n^2(1-n/m) < 0. That is, **approach A** also performs better for minimizing memory.

In terms of memory used, increasing dimension of parallelization always becomes a burden as the cost for its benefit like decreasing calculation time.

5.

First we focus on how 0.2 being stored in R : As problem(2) indicated, we convert 0.2 in binary form to see its storage form : $0.2 = 1.\overline{1001}_2 \times 2^{-3}$
We know the corresponding e = 3, but what happens to d ? Since 0.2 is a circulation number in binary from, d will overflow and be rounded off to its first 52 digits. As a result, corresponding d = 1001100110011001100110011001100110011001100110011010. The last four digits are 1010 since R rounds off the following 1(1001 $\rightarrow$ 1010). Due to the rounding off, the actual value stored in R will be slightly

greater than 0.2 The corresponding decimal value to 52-digits d is 0.200000000000000011102230..., which is the way how R stores 0.2 internally.

```
# Notice the last four digits.
bits(0.2)

## [1] "00111111 11001001 10011001 10011001 10011001 10011001 10011001 10011010"

options(digits = 22)
# The last part is 111022, which proves our assumption.
0.2

## [1] 0.20000000000000000111022
```

Now look at 0.3:
$0.3 = 1.\overline{0011}_2 \times 2^{-2}$.
The corresponding d = 00110011001100110011001100110011001100110011001100110011. The last four digits are 0011 since R rounds off the following 0. Due to the rounding off, the actual value stored in R will be slightly smaller than 0.3

```
bits(0.3)

## [1] "00111111 11010011 00110011 00110011 00110011 00110011 00110011 00110011"

0.3

## [1] 0.29999999999999999888978
```

More Interestingly, the "added" part of 0.2 and the "truncated" part 0.3 complement each other and will be canceled out if we add 0.2 to 0.3. That's why 0.2 + 0.3 == 0.5 becomes TRUE in R.

```
0.2 + 0.3 == 0.5

## [1] TRUE
```

It can be proved mathematically that a + b == c will be true if c can be presented in the form of $2^{-m} \times 5^{-n}$(both m, n are positive integers).
However, it will be hard to predict whether the equality holds when c cannot be presented in this form. Let's make some tests:

```
# Therotically they should be TRUE.
0.1 + 0.4 == 0.5

## [1] TRUE

0.12 + 0.13 == 0.25

## [1] TRUE

0.013 + 0.027 == 0.04

## [1] TRUE

# Difficult to predict.
0.1 + 0.2 == 0.3

## [1] FALSE
```

```r
0.6 + 0.2 == 0.8
```

```
## [1] TRUE
```

```r
0.112228 + 0.288338 == 0.400566
```

```
## [1] TRUE
```