Stat243: Problem Set 5

Qingan Zhao
SID: 3033030808

17 Oct. 2017

Problem 1

This is the reading assignment.

Problem 2

First, let's convert 1, 2, 3, ..., $2^{53}-2$, $2^{53}-1$ to binary. Then demonstrate all these numbers into the format of $(-1)^S \times 1.d \times 2^{e-1023}$. Apparently S is always 0, so let's just determine d and e for each number. The results are shown in the table below:

Table 1: Number 1 to $2^{53}-1$

Decimal	Binary	$(-1)^S \times 1.d \times 2^{e-1023}$	d	e
1	1	$(-1)^0 \times 1.0 \times 2^0$	0	1023
2	10	$(-1)^0 \times 1.0 \times 2^1$	0	1024
3	11	$(-1)^0 \times 1.1 \times 2^1$	1	1024
4	100	$(-1)^0 \times 1.00 \times 2^2$	00	1025
5	101	$(-1)^0 \times 1.01 \times 2^2$	01	1025
...	...	...	...	...
$2^{53}-2$	$\underbrace{1...1}_{52}0$	$(-1)^0 \times 1.\underbrace{1...1}_{51}0 \times 2^{52}$	$\underbrace{1...1}_{51}0$	1075
$2^{53}-1$	$\underbrace{1...1}_{53}$	$(-1)^0 \times 1.\underbrace{1...1}_{52} \times 2^{52}$	$\underbrace{1...1}_{52}$	1075

Hence, from the table we know that number 1 to $2^{53}-1$ can be represented exactly. Now let's consider larger numbers. $2^{53}-1$ can be written as $(-1)^0 \times 1.\underbrace{0...0}_{53} \times 2^{53}$. However, we know that d is represented as 52 bits. Luckily $\underbrace{0...0}_{53}$ is the same as 0, so 0 can be restored as d for 2^{53}. Similarly, $2^{53}+2$ can be witten as $(-1)^0 \times 1.\underbrace{0...0}_{52}10 \times 2^{53}$ and d can be $\underbrace{0...0}_{52}1$ instead of $\underbrace{0...0}_{52}10$. For $2^{53}+1$, however, it can only be wriiten as $(-1)^0 \times 1.\underbrace{0...0}_{52}1 \times 2^{53}$, which means d can only be $\underbrace{0...0}_{52}1$, so d is overflowed. Hence, $2^{53}+1$ cannot be represented exactly, and the spacing of numbers of this magnitude is 2.

Similarly, for numbers starting with 2^{54}, we know that 2^{54} can be written as $(-1)^0 \times 1.\underbrace{0...0}_{54} \times 2^{54}$ and $2^{54}+4$ can be written as $(-1)^0 \times 1.\underbrace{0...0}_{51}100 \times 2^{54}$. So for these two numbers, d is no greater than 52 bits (i.e., 0 and $\underbrace{0...0}_{51}1$). However, for $2^{54}+1$, $2^{54}+2$, and $2^{54}+3$, d can only be greater than 52 bits (not enough zeros

at the end of d). Hence, the spacing of numbers of this magnitude is 4.

Finally, let's confirm it in R.

```
#execute 2^53-1, 2^53, and 2^53+1
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"
```

We can see that the result of $2^{53} + 1$ is the same as 2^{53}, which means $2^{53} + 1$ is not represented exactly.

Problem 3

(a)

Let's create a large vector of integers and convert the vector to numeric type. Calculate and compare the time of copying the integer vector and the numeric vector.

```
#set the vector large enough (10^8 numbers)
n <- 1e8
int <- 1:n
num <- as.numeric(int)
#calculate the time of copying the integer vector and the numeric vector
system.time(copy(int))

##    user  system elapsed
##   0.155   0.161   0.335

system.time(copy(num))

##    user  system elapsed
##   0.366   0.336   0.848
```

As seen in the results, copying a integer vector is faster than a numeric vector of the same length in R. It is quite straightforward because a numric vector basically uses twice as much memory as a integer vector.

(b)

Let's calculate and compare the time of taking the subset of half size from each vector.

```
#calcuate the time of taking the subset of size n/2 from each vector.
system.time(int[1:(n/2)])

##    user  system elapsed
##   0.444   0.237   0.721

system.time(num[1:(n/2)])

##    user  system elapsed
##   0.555   0.324   0.924
```

As seen in the results, taking a subset of half size from the integer vector is also faster than from the numeric vector, but not so much.

Problem 4

(a)

Breaking up Y into n individual column means that we let p be n. Given that n is very large, there will be too many tasks for the parallelization, and the communication overhead of starting and stopping those tasks will reduce the efficiency. Hence, it is better to break up Y into resonable p blocks of $m = n/p$ columns, which means p should not be too large or too small. The key is to keep the parallelization load-balanced without too much communication.

(b)

First let's compare the two approaches in terms of memory usage. Assume that 1 unit of memory is used when storing a single number.

Approach A:

For a single task, the units of memory used for storing the input matrix X and submatrix of Y should be:

$$n \times n + n \times m = n^2 + mn \tag{1}$$

During the multiplication process, the units of memory used for storing the results of multiplications of every two numbers should be:

$$n \times n \times m = mn^2 \tag{2}$$

The units of memory used for storing the output matrix should be:

$$n \times m = mn \tag{3}$$

Hence, for a single task, all units of memory used should be:

$$n^2 + mn + mn^2 + mn = n^2 + 2mn + mn^2 \tag{4}$$

The total number of tasks is n/m, so the total units of memory used in Approach A should be:

$$(n^2 + 2mn + mn^2) \times \frac{n}{m} = n^3 + 2n^2 + \frac{n^3}{m} \tag{5}$$

Approach B:

For a single task, the units of memory used for storing the input submatrix of X and submatrix of Y should be:

$$2 \times m \times n = 2mn \tag{6}$$

During the multiplication process, the units of memory used for storing the results of multiplications of every two numbers should be:

$$n \times m \times m = m^2 n \tag{7}$$

The units of memory used for storing the output matrix should be:

$$m \times m = m^2 \tag{8}$$

Hence, for a single task, all units of memory used should be $2mn + m^2n + m^2$.

The total number of tasks is $(n/m)^2$, so the total units of memory used in Approach B should be:

$$(2mn + m^2n + m^2) \times \left(\frac{n}{m}\right)^2 = n^3 + n^2 + \frac{2n^3}{m} \tag{9}$$

Now let's compare the two approachs (Equation(5)-Equation(9)):

$$(n^3 + 2n^2 + \frac{n^3}{m}) - (n^3 + n^2 + \frac{2n^3}{m}) = n^2\left(1 - \frac{n}{m}\right) < 0 \quad (n > m) \tag{10}$$

Hence, Approach A is better for minimizing memory use.

Second, let's compare the two approaches in terms of communication.

Approach A:

For a single task, the total number of numbers that passed to the workers should be:

$$n \times n + n \times m = n^2 + mn \tag{11}$$

The number of numbers that passed back is mn, so the communication cost per task should be:

$$n^2 + mn + mn = n^2 + 2mn \tag{12}$$

Then the total communication cost in Approach A should be:

$$(n^2 + 2mn) \times \left(\frac{n}{m}\right) = 2n^2 + \frac{n^3}{m} \tag{13}$$

Approach B:

For a single task, the total number of numbers that passed to the workers should be:

$$n \times m \times 2 = 2mn \tag{14}$$

The number of numbers that passed back is m^2, so the communication cost per task should be $2mn + m^2$.

Then the total communication cost in Approach B should be:

$$(2mn + m^2) \times \left(\frac{n}{m}\right)^2 = n^2 + \frac{2n^3}{m} \tag{15}$$

Now let's compare the two approachs (Equation(13)-Equation(15)):

$$(2n^2 + \frac{n^3}{m}) - (n^2 + \frac{2n^3}{m}) = n^2\left(1 - \frac{n}{m}\right) < 0 \quad (n > m) \tag{16}$$

Hence, Approach A is better for minimizing communication.

Problem 5

To solve this problem, we can look back at Problem 2. We know that numbers are stored as S, d, and e in R. S and e are easy to describe exactly for all numbers, but the real problem is d. For 0.2, the actual digit of d is infinite so R has to round it to 52 digits. Hence, the actual value of 0.2 in R is not exactly 0.2. Similarly, the actual value of 0.3 in R is not exactly 0.3 either. Nor are 0.1, 0.01 and 0.49. However, for 0.5, d equals 0 instead of infinite numbers, so R does not have to round it. Hence, the actual value of 0.5 in R is exactly 0.5.

Hence, it makes sense that $0.3 == 0.2 + 0.1$ is FALSE in R because they are not the exact value in R. But why $0.2 + 0.3 == 0.5$ and $0.01 + 0.49 == 0.5$ are TRUE in R? Now let's find out the exact values of these numbers in R.

```
options(digits=22)
0.2

## [1] 0.20000000000000000111022

0.3

## [1] 0.29999999999999988978

0.01

## [1] 0.010000000000000020817

0.49

## [1] 0.48999999999999911182

0.1

## [1] 0.10000000000000055511
```

As seen from the result, we know that $0.2 + 0.3 == 0.5$ and $0.01 + 0.49 == 0.5$ are TRUE in R because the "error" are canceled when they are added together. In other words, the actual value of 0.2 in R plus the actual value of 0.3 in R equals the actual value of 0.5 in R. However, the actual value of 0.2 in R plus the actual value of 0.1 in R is not equal to the actual value of 0.3 in R, so $0.3 == 0.2 + 0.1$ is FALSE in R.