Problem Set 5

Cameron Adams

October 17, 2017

1 In class we discussed the idea of a closure as a function ...

2 In class I mentioned that integers as large as 2^{53} can be stored exactly in the double precision floating point representation.

2.1 Demonstrate how the integers $1, 2, 3, ..., 2^{53} - 2, 2^{53} - 1$ can be stored exactly in the $(1)^S \times 1.d \times 2^{e1023}$ format where d is represented as 52 bits

For a double, there are 8 bytes=64 bits to represent the floating point number.

- S (1 bits) reprsents the sign of teh number (neg/pos)
- e (11 bits) is the base-2 exponent number, $2^1 1 = 2048$
- d (52 bits) the computer accuracy of the real number

We can solve for d by rearrange the following equation: $decimal/integer = S \times e \times d$. Therefore, $1, 2, ..., 2^{53}, 2^{53} - 1$ can be represented by:

$$\begin{aligned}
1 &= (-1)^0 \times 1.0000000000000000 \times 2^{1023-1023} \\
2 &= (-1)^0 \times 1.0000000000000000 \times 2^{1024-1023} \\
2^{53} - 2 &= (-1)^0 \times \frac{2^{53} - 2}{2^{52}} \times 2^{1075-1023} \\
2^{53} - 1 &= (-1)^0 \times \frac{2^{53} - 1}{2^{52}} \times 2^{1075-1023} \\
2^{53} &= (-1)^0 \times 1.0000000000000000 \times 2^{1076-1023}
\end{aligned}$$

See code below for equalities.

```
#set digits to higher precision than default
options(digits = 22)

1 == 1*2^(1023-1023)

## [1] TRUE

2 == 1*2^(1024-1023)

## [1] TRUE

2^53 - 2 == ((2^53 - 2) / 2^52) * 2^(1075 - 1023)

## [1] TRUE

2^53 - 1 == ((2^53 - 1) / 2^52) * 2^(1075 - 1023)
```

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

```
2^53 == 1*2^(1076-1023)
```

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

2.2 Then show that 2^{53} and $2^{53}+2$ can be represented exactly but $2^{53}+1$ cannot, so the spacing of numbers of this magnitude is 2.

These numbers in floating point format are:

$$\begin{aligned} 2^{53} &= (-1)^0 \times 1.0000000000000000 \times 2^{1076-1023} \\ 2^{53}+1 &= (-1)^0 \times 1.0000000000000001 \times 2^{1076-1023} \\ 2^{53}+2 &= (-1)^0 \times 1.0000000000000002 \times 2^{1076-1023} \end{aligned}$$

However, $2^{53}+1$ requires more digits than there is precision. Precision is determined by machine epsilon, ϵ, and absolute spacing is $x\epsilon = 2^{53} \times 2^{-52} = 2$. Therefore, at 2^{53} power, we can represenet numbers with spacing $== 2$. So 2^{53} and $2^{53}+2$ work, but $2^{53}+1$ does not.

See code below for equalities.

```
2^53
```

```
## [1] 9007199254740992
```

```
2^53 + 1
```

```
## [1] 9007199254740992
```

```
2^53 + 2
```

```
## [1] 9007199254740994
```

```
2^53 == 1.000000000000000 * 2^{1076-1023}
```

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

```
2^53 == 1.0000000000000001 * 2^{1076-1023}
```

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

```
2^53 + 1 == 1.0000000000000001 * 2^{1076-1023}
```

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

```
2^53 + 2 == 1.0000000000000002 * 2^{1076-1023}
```

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

2.3 Finally show that for numbers starting with 2^{54} that the spacing between integers that can be represented exactly is 4.

This answer is similar to above. We can determine absolute spacing with $x\epsilon = 2^{54} \times 2^{-52} == 4$.

See code below for equalities.

```
2^54
```

```
## [1] 18014398509481984
```

```
2^54 + 2
```

```
## [1] 18014398509481984
```

```
2^54 + 4
```

```
## [1] 18014398509481988
```

```
2^54 == 1.000000000000000 * 2^{1077-1023}
```

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

```
2^54 == 1.0000000000000001 * 2^{1077-1023}
```

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

```
2^54 + 2 == 1.0000000000000001 * 2^{1077-1023}
```

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

```
2^54 + 4 == 1.0000000000000002 * 2^{1077-1023}
```

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

3 GPUs tend to use single precision floating point calculations (4 bytes per real number). One advantage of this is that one is moving around half as much data during the process of computations so that should speed up computation, at the expense of precision. Lets explore a similar question related to integers.

3.1 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.

```
rm(list=ls())

require(dplyr)
require(data.table)

#return to default
options(digits = 7)

#specify integer and numeric vector of same length
x_num <- rnorm(1e7)
x_int <- as.integer(x_num)

x_int_copy <- copy(x_int)
x_num_copy <- copy(x_num)
```

```r
address(x_int); address(x_int_copy)

## [1] "0x123800000"
## [1] "0x11c898000"

address(x_num); address(x_num_copy)

## [1] "0x139630000"
## [1] "0x113000000"

#benchmark copy timing of vectors
require(microbenchmark)
microbenchmark(x_int_copy <- copy(x_int),
               x_num_copy <- copy(x_num),
               times = 20L, unit= "ms")

## Unit: milliseconds
##                       expr      min       lq     mean    median       uq
##  x_int_copy <- copy(x_int) 18.06868 46.95512 111.8652  66.75479 118.4839
##  x_num_copy <- copy(x_num) 82.63354 99.28604 205.6414 136.73618 218.8237
##       max neval
##  731.3262    20
##  676.3228    20
```

It is much faster (3̃x) to make a copy of the integer vector than numeric vector. This make sense as the integer vector is essentially half the size as the numeric vector, due to the number of bits required to store numeric (64) vs integer (32) numbers.

3.2 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?

```r
#specify size of subset
k <- length(x_num)/2

#get index of positions to subset, length k
subset <- sample(x = 1:length(x_num), size = k, replace = F)

#benchmark
microbenchmark(x_int[subset],
               x_num[subset],
               times = 20L,
               unit = "ms")

## Unit: milliseconds
##           expr      min       lq     mean   median       uq      max neval
##  x_int[subset] 345.7466 427.0215 592.1082 511.2336 657.3876 1300.393    20
##  x_num[subset] 366.7981 500.1221 703.2039 650.8605 891.7003 1152.722    20
```

It is faster to subset the integer vector than the numeric vector (1̃5-20% faster).

4 Lets consider parallelization of a simple linear algebra computation. In your answer, you can assume m = n/p is an integer.

4.1 Consider trying to parallelize matrix multiplication, in particular the computation XY where both X and Y are $n \times n$. There are lots of ways we can break up the computations, but lets keep it simple and consider parallelizing over the columns of Y . Given the considerations we discussed in Unit 7, when you parallelize the matrix multiplication, why might it be better to break up Y into p blocks of m = n/p columns rather than into n individual column-wise computations? Note: Im not expecting a detailed answer here a sentence or two is fine.

In this scenario, over parallelizing (n-idinvidiauls columns) could be slower than parallelizing over p processors. These computations are likley to be relatively fast, and sending each column as a process could introduce lag-time into each parallel process. It would be better to parallelize computations over to the nodes/cores For example, if Y is 40x40 matrix, it would make sense to divide our number of processes (40) by the number of cores (4), 40/4=10, and send subsets of 10 columns of Y to be processed by each core.

4.2 Lets consider two ways of parallelizing the computation and count (1) the amount of memory used at any single moment in time, when all p workers are doing their calculations, including memory use in storing the result and (2) the communication cost count the total number of numbers that need to be passed to the workers as well as the numbers passed from the workers back to the master when returning the result. Which approach is better for minimizing memory use and which for minimizing communication? Approach A: divide Y into p blocks of equal numbers of columns, where the first block has columns $1, ..., m$ where $m = \frac{n}{p}$ and so forth. Pass X and the j^{th} submatrix of Y to the j^{th} task. Approach B: divide X into p blocks of rows, where the first block has rows 1, ..., m and Y into p blocks of columns as above. Pass pairs of a submatrix of X and submatrix of Y to the different tasks.

Answer depends on the size of the matrix. The second approach has p^2 more processes than the first - dividing up both X and Y by m, require us to make sure each row is multiplied by all elements of each column. Additionally, reassembly of complete XY matrix is more difficult in the second approach, because we have p^2 more submatrices than in the first appraoch. There are also memory allocations to consider. There may be limited memory available, and having further atomizing operations (second approach vs first approach), may over burden available memeory. I would think the first approach would be best.

5 Extra Credit

Base-2 can store numbers in increments of 5, e.g. (5, 0.5, 0.005, 0.001, etc). Therefore, numbers divisible by 5 can be represented by their component parts. So 0.3 + 0.2 == 0.5, 0.3 + 0.7 == 1.0, 0.9 - 0.4 == 0.5, etc. Other numbers, such as 0.3, cannot be exactly reprented by base-2 numbers. Therefore, $0.2 + 0.1! = 0.3$

```
options(digits = 22)

#0.3 ex.
0.1 + 0.2 == 0.3
```

```
## [1] FALSE
```

```
0.1 + 0.2
```

```
## [1] 0.30000000000000004444089
```

```
0.3
```

```
## [1] 0.299999999999999888978
```

```
#0.5, 1.0 ex.
0.3 + 0.2 == 0.5
```

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

```
0.3 + 0.7 == 1
```

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

```
0.9 + 0.1 == 1
```

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

```
0.9 - 0.4 == 0.5
```

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

```
0.003 + 0.002 == 0.005
```

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