STAT 243 PS5

Riv Jenkins

2)

A double precision float has 52 bits for precision, so the largest integer that can be represented exactly (without moving the decimal place) by this would be 2^52-1 because 111....11 (with 52 ones) is 2^52-1 in binary. However, each number in double precision is represented by 1.d with d being the precision dictated by the 52 bits. This means the largest number to be stored exactly is 2^53-1 = 111...111 (with 53 ones) because of the extra 1 implicit in the representation.

2^53 and 2^53+2 can be represented exactly because at this point we must move the decimal point to the left to be able to fit the number in 52 bit precision. This is simlar to what would happen if we tried to represent a decimal number without the last digit. 10 would become 1 and 20 would become 2, but 15 would have to be rounded to 1 or 2. The same concept applies here except that now we are in base 2 so the precision jumps from 1 to 2 instead of 1 to 10. So 2^53+1 cannot be represented exactly.

When we move to 2^54 we again have to move the decimal place one to the left, so our accuracy goes down proportionally (just as we saw in going to 2^53). We go from an accuracy of 2 to 4. If we were working in decimal notation this would be going from an accuracy of 10 to 100.

```
options(digits=16)
2^53 -1
```

```
## [1] 9007199254740991
```

```
2^53
```

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

```
2^53 +1
```

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

3 a)

```
microbenchmark(x = integer(1e7), y = numeric(1e7))
```

```
## Unit: milliseconds
##  expr       min         lq        mean     median         uq       max
##     x 10.674684 11.3589765 17.29186906 12.2802155 25.0579765 57.347982
##     y 22.028084 30.7574330 33.50288707 32.7411435 36.5575145 73.687095
##  neval
##    100
##    100
```

It is faster to copy a vector of integers as seen in the results above.

b)

```
x = integer(1e6)
y = numeric(1e6)
microbenchmark(x2<-x[1:length(x)/2], y2<-y[1:length(y)/2])
```

```
## Unit: milliseconds
##                    expr       min         lq        mean     median
##  x2 <- x[1:length(x)/2] 17.917423 19.2366990 23.90920067 22.4700725
##  y2 <- y[1:length(y)/2] 18.918314 21.1371875 26.60595745 24.1949705
##          uq               max neval
##  25.1208950 66.34529300000000   100
##  27.1043815 70.86157400000000   100
```

Based on the results above it looks like it takes slightly longer to take half the elements from the numeric vector, but the difference is much less drastic than we saw in part a.

4 a)

It may be better to break up the columns into p blocks because the amount of time needed to do one column multiplication may be very small especially compared to the amount of time necessary to start up a new process. This means that if we broke up the operations into n processes instead of m processes we could actually spend more computation time because the overhead to set up those processes is larger than the actual computation time of the process

b)

Approach A is good for minimizing the communication cost since there are only as many jobs as there are workers, so the jobs are just passed to the workers once. However, each job contains the entire matrix X, so there will be p copies of X across all the machines. If X is a large matrix, this could be infeasible. Approach B avoids this problem by splitting X up into smaller chunks, but there are now more overall jobs to be executed, so there will need to be more information passed to and from the individual workers.