HPC-R Exercises: Improving R Performance

Drew Schmidt

02/27/2015

1. Sort each element of this list:

```r
set.seed(1234)

l <- list(A=sample(0:1, size=10, replace=TRUE),
          B=runif(10),
          C=sample(letters)
)

l
```

```
## $A
##  [1] 0 1 1 1 1 1 0 0 1 1
##
## $B
##  [1] 0.6935913 0.5449748 0.2827336 0.9234335 0.2923158 0.8372956 0.2862233
##  [8] 0.2668208 0.1867228 0.2322259
##
## $C
##  [1] "i" "h" "d" "a" "e" "r" "k" "u" "o" "w" "y" "x" "v" "g" "c" "z" "l"
## [18] "j" "p" "f" "s" "m" "b" "n" "q" "t"
```

(Hint: use `lapply()` and `sort()`).

2. For the matrix:

```r
set.seed(1234)

x <- matrix(sample(0:1, size=1000*5, replace=TRUE), nrow=1000, ncol=5)
```

produce a vector containing the sum of the columns of x.

3. Suppose we wish to create a vector containing the square root of the numbers 1 to 10000. Do this in each of the following ways, and benchmark your implementations:

   - for loop without initialization
   - for loop with initialization
   - `sapply()`
   - vectorized

4. Revisit the solutions from exercise 1 above, now with the bytecode compiler.

5. How many numbers are there from 1 to 10,000,000 that are multiples of 5 or 17 (or both)? Solve this with:

   - `sapply()`

- vectorization

6. Remember that the bytecode compiler isn't nearly as clever as your typical C/C++ compiler. Consider these two functions:

```
f <- function (A, Q){
  n <- ncol (A)
  for (i in 1:n){
    tA <- t(A)
    Y <- tA %*% Q
    Q <- qr.Q (qr(Y))
    Y <- A %*% Q
    Q <- qr.Q(qr(Y))
  }

  Q
}


g <- function (A, Q){
  n <- ncol (A)
  tA <- t(A)
  for (i in 1:n){
    Y <- tA %*% Q
    Q <- qr.Q (qr(Y))
    Y <- A %*% Q
    Q <- qr.Q(qr(Y))
  }

  Q
}
```

Conceptually, these two functions do the same thing, but `g()` is more efficient than `f()`. We can easily show this on some randomly generated data:

```
nrows <- 100
ncols <- 100
A <- matrix(rnorm(nrows*ncols), nrow=nrows, ncol=ncols)
Q <- matrix(0, nrows, ncols)

library(rbenchmark)
benchmark(f=f(A, Q), g=g(A, Q), replications=10, order="relative",
                    columns=c("test", "elapsed", "relative"))
```

```
##   test elapsed relative
## 2    g   2.493    1.000
## 1    f   2.503    1.004
```

Compiling with the bytecode compiler may improve the overall performance, but won't fix the underlying problem. Use the `disassemble()` function on these functions to convince yourself that `f()` is still doing unnecessary operations.

Answers

1. The easy way to do this is to follow the hint:

```
lapply(l, sort)
```

```
## $A
##  [1] 0 0 0 1 1 1 1 1 1 1
##
## $B
##  [1] 0.1867228 0.2322259 0.2668208 0.2827336 0.2862233 0.2923158 0.5449748
##  [8] 0.6935913 0.8372956 0.9234335
##
## $C
##  [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q"
## [18] "r" "s" "t" "u" "v" "w" "x" "y" "z"
```

2. Some example solutions are:

```
### loop
sums <- numeric(5)
for (j in 1:5){
  for (i in 1:1000){
    sums[j] <- sums[j] + x[i, j]
  }
}
sums
```

```
## [1] 518 485 515 481 508
```

```
### apply
apply(x, 2, sum)
```

```
## [1] 518 485 515 481 508
```

```
### vectorized
colSums(x)
```

```
## [1] 518 485 515 481 508
```

3. Possible solutions are:

```
sqrt_loop_noinit <- function(n)
{
  ret <- c()
  for (i in 1:n)
    ret[i] <- sqrt(i)

  return(ret)
}
```

```r
sqrt_loop_withinit <- function(n)
{
  ret <- numeric(n)
  for (i in 1:n)
    ret[i] <- sqrt(i)

  return(ret)
}

sqrt_lapply <- function(n) lapply(1:n, sqrt)

sqrt_vec <- function(n) sqrt(1:n)
```

Benchmarking them for `n=10000`, the clear victor is using vectorization:

```r
library(rbenchmark)

n <- 10000
benchmark(sqrt_loop_noinit(n), sqrt_loop_withinit(n), sqrt_lapply(n), sqrt_vec(n),
                    order="relative", columns=c("test", "elapsed", "relative"))
```

```
##                    test elapsed relative
## 4           sqrt_vec(n)   0.009    1.000
## 3        sqrt_lapply(n)   0.340   37.778
## 2 sqrt_loop_withinit(n)   0.812   90.222
## 1   sqrt_loop_noinit(n)   8.122  902.444
```

To get a sense for possible performance improvements of `lapply()` over for loops, we can just compare these two:

```r
benchmark(sqrt_loop_withinit(n), sqrt_lapply(n), order="relative",
                    columns=c("test", "elapsed", "relative"))
```

```
##                    test elapsed relative
## 2        sqrt_lapply(n)   0.307    1.000
## 1 sqrt_loop_withinit(n)   0.812    2.645
```

4. Using the bytecode compiler helps, but not tremendously:

```r
library(compiler)

sqrt_loop_noinit <- cmpfun(sqrt_loop_noinit)
sqrt_loop_withinig <- cmpfun(sqrt_loop_withinit)
sqrt_lapply <- cmpfun(sqrt_lapply)
sqrt_vec <- cmpfun(sqrt_vec)

benchmark(sqrt_loop_noinit(n), sqrt_loop_withinit(n), sqrt_lapply(n), sqrt_vec(n),
                    order="relative", columns=c("test", "elapsed", "relative"))
```

```
##                    test elapsed relative
```

```
## 4           sqrt_vec(n)   0.009    1.000
## 3         sqrt_lapply(n)   0.303   33.667
## 2 sqrt_loop_withinit(n)   0.807   89.667
## 1   sqrt_loop_noinit(n)   7.837  870.778
```

And again just comparnig the loop with `lapply()`:

```r
benchmark(sqrt_loop_withinit(n), sqrt_lapply(n), order="relative",
                    columns=c("test", "elapsed", "relative"))
```

```
##                    test elapsed relative
## 2        sqrt_lapply(n)   0.302    1.000
## 1 sqrt_loop_withinit(n)   0.812    2.689
```

5. Possible solutions are:

```r
### sapply
div_by_5_or_17 <- function(n)
{
  if (n %%5 == 0 || n %% 17 == 0)
    return(TRUE)
  else
    return(FALSE)
}

div_sapply <- function(n) sum(sapply(1:n, div_by_5_or_17))



### Vectorization
div_vec <- function(n)
{
  numbers <- 1:n
  sum((numbers %% 5 == 0) | (numbers %% 17 == 0))
}
```

```r
library(rbenchmark)
n <- 100000

benchmark(sapply=div_sapply(n), vec=div_vec(n))
```

```
##     test replications elapsed relative user.self sys.self user.child
## 1 sapply          100  17.287   20.929    17.270    0.012          0
## 2    vec          100   0.826    1.000     0.826    0.000          0
##   sys.child
## 1         0
## 2         0
```