performance enhancement in R

The is a constant trade-off between time and space in computer programming. Concretely, the notion of a fast-running program often requires more space in memory. Conversely, a conservative program using less memory will likely sacrifice in performance speed. The trade-off between time and space is relevant in the context of R for the following reasons:

… R is an interpreted language. Many of the commands are written in C and thus do run in fast machine code. But other commands, and your own R code, are pure R and thus interpreted. So, there is a risk that your R application may run more slowly than you would like.
… All objects in an R session are stored in memory. More precisely, all objects are stored in R's memory address space. R places a limit of 231 − 1 bytes on the size of any object, even on 64-bit machines and even if you have a lot of RAM. Yet some applications do encounter larger objects.

writing fast code in R

The main tools available to make code faster in R are as follows:

… Optimize your R code through vectorization, use of byte-code compilation, and other approaches
… Write the key, CPU-intensive parts of code in a compiled language such as C/C++
… Write code in some form of parallel R

To optimize code effectively, an understanding is necessary of R's functional programming nature and the way R allocated memory.

the for loop in R

An alternate to using **for loops** in R is to leverage various ***vectorization*** techniques.

For the use of ***vectorization*** to speed up code, the following vector **x** and **y** are of equal lengths:

```
> z <- x + y        #vectorized summation of all elements in x and y, respectively
```

The above is considerably more computationally cheap than the following **for loop** option:

```
> for (i in 1:length(x)) z[i] <- x[i] + y[i]
```

The proof is seen in runtime comparisons of the above methods of element-wise vector summation:

```
> x <- runif(1000000)
> y <- runif(1000000)
> z <- vector(length=1000000)
> system.time(z <- x + y)
   user system elapsed
   0.052 0.016 0.068

> system.time(for (i in 1:length(x)) z[i] <- x[i] + y[i])
   user system elapsed
   8.088 0.044 8.175
```

One type of vectorization is ***vector filtering***. Using the **oddcount( )** function from prior examples:

```
oddcount <- function(x) return(sum(x%%2==1))
```

Though there is no explicit **loop** in the above example, R will internally loop through the array in native machine code; therefore, the anticipated speedup does occur.

```
> x <- sample(1:1000000,100000,replace=T)
> system.time(oddcount(x))
   user system elapsed
   0.012 0.000 0.015
```

The following performs the same **oddcount( )** function in the form of a **loop**:

```
> system.time(
+ {
+ c <- 0
+ for (i in 1:length(x))
+ if (x[i] %% 2 == 1) c <- c+1
+ return(c)
+ }
+ )
   user system elapsed
   0.308 0.000 0.310
```

Although both cases ran in less than a second, the timing extrapolated with function complexity and data.

Other forms of vectorized function in R that will potentially increase code performance:

The following functions are vectorized in R:

`ifelse(), which(), where(), any(), all(), cumsum(), and cumprod()`

In the case of matrices: `rowSums(), colSums(), etc.`

In “all possible combination” types: `combin(), outer(), lower.tri(), upper.tri(), expand.grid()`

Although the **apply( )** function eliminates an explicit loop, it is implemented in R rather than C and will not typically cause an increase in code performance. In contrast, other apply functions such as **lapply( )**, can be useful to help code performance.

functional programming and memory issues in R

Most object in R are ***immutable*** (unchangeable); thus, R operations are implemented as functions that reassign to the given object, a trait that can have performance implications.

Once source of performance issues in R is with certain uses of ***vector assignments***:

```
> x[3] <- 8
> z <- "[<-"(z,3,value = 8)        #the above actually calls replacemet function "[<-"
```

Concretely, the above example reassigns the entire vector **z** even though the semantics instruct changing a single element. The effect leads to mass computational cost in long vectors, or small vectors within loops.

Another source for performance issues in R is with ***copy-change-policy***:

```
> y <- z
```

Assuming the above assignment, variable **y** shares the same memory as variable **z**. If either changes, a copy is made in a different area of memory. However, only the ***first*** change is affected; the relocating of the moved variable removes any sharing issues. The **tracemem( )** function reports such memory locations.

R adheres to ***copy-on-change*** semantics; a situation where R does not exhibit location-change behavior:

```
> z <- runif(10)
> tracemem(z)
[1] "<0x88c3258>"

> z[3] <- 8
> tracemem(z)
[1] "<0x88c3258>"
```

```
> z <- 1:10000000
> system.time(z[3] <- 8)
   user system elapsed
   0.180 0.084   0.265
> system.time(z[33] <- 88)
user system elapsed
   0      0       0
```

The above illustrates the location of **z** never changing between assignments. In an event where the copying is performed, it is copied through the **duplicate( )** function within R’s internal code.

finding slow code spots in R

The **Rprof( )** function provides z report of (approximately) how much time code is spending on each function being called. The usefulness arises due to not every section of code always needing optimizing. Sacrifices often related to optimization can often be code writing time and code complexity.

How Rprof( ) works:

R inspects the call stack to determine which function calls are in effect evert 2 seconds (default value). Each inspection result is written to a file (Rprof.out by default).

The **summaryRprof( )** function will summarize all the latter **Rprof( )** function results written to the file. The **Rprof( )** results can often by cryptic in scenarios where the profiling code produces many functions (indirect calls) and might often not be the best route for understanding the inner workings of an R program.

byte code compilations in R

R includes a ***byte code compiler*** that can be used to potential improve code performance. Returning to the example from before:

```
> z <- x + y                                #faster than the for loop below
> for (i in 1:length(x)) z[i] <- x[i] + y[i]
```

Applying the ***byte code compiler*** to the above illustration as follows:

```
> library(compiler)
> f <- function() for (i in 1:length(x)) z[i] <<- x[i] + y[i]
> cf <- cmpfun(f)
> system.time(cf())
   user system elapsed
  0.845  0.003   0.848
```

Noted how significantly faster the compiled **for loop** function above runs compared to the original below:

```
> system.time(for (i in 1:length(x)) z[i] <- x[i] + y[i])
   user system elapsed
   8.088 0.044 8.175
```

However, overall performance remains best with the vectorized implementation as **z <- x + y**.

conquering max memory in R

Remembering that all R objects are stored in-memory, R has a limit of 2^{31}—1 bytes on the size of any given object, regardless of word size (32 vs 64-bit) and RAM.

One way to overcome these limitations is through the use of what is known as **chunking**. The concept is to read data from a disk file in portions. This is achieved through using the **skip** argument within **read.table( )**.

```
For example, a dataset with 1,000,000 records can be divided into 10 chunks. The skip = 0
argument will be set at the initial call. The second call will set the skip = 100000
argument; the process repeats until all of the data is read and computed on as needed.
```

An additional method to deal with R memory limits is through the use of specialized **R packages**. The **RMySQL** package provides a more convenient way to handle large databases through connecting to external SQL databases (database knowledge required). The package essentially performs the large memory-required transactions back at the database and reads the results to R. Another option is the **biglm** package that performs regression and general linear-model analysis on large datasets. To more commonly used packages are **ff** and **bigmemory**. The **ff** package sidesteps memory limitations by storing data on-disk, opposed to in-memory. The **bigmemory** packages does the same, while also storing on a machine's main memory (ideal for multicore machines).