Code optimization best practices

Mikhail Dozmorov

Summer 2018

Timing

Use `system.time()` functions to measure the time of execution.

```
> # make a function
> myFun <- function(x) {
+     y = vector(length=x)
+     for (i in 1:x) y[i]=i/(i+1)
+     y
+ }
```

```
> # execute the function, measuring the time of the execution
> system.time( myFun(100000) )
   user  system elapsed
  0.107   0.002   0.109
```

Memory

Use `pryr::object_size()` function to measure memory footprint of R objects

```
> library(pryr)
> object_size(USArrests)
5.23 kB
object_size(1:10^6)
4 MB
```

Code speedup: Use vectors

```
> # using loops
> g1 <- function(x) {
+     y = vector(length=x)
+     for (i in 1:x) y[i]=i/(i+1)
+     y
+ }

> # execute the function
> system.time( g1(100000) )
   user  system elapsed
  0.107   0.002   0.109
```

Code speedup: Use vectors

```
> # using vectors
> x <- (1:100000)
> g2 <- function(x) {
+     x/(x+1)
+ }

> # execute the function
> system.time( g2(x) )
 user  system elapsed
  0.002   0.000   0.003
```

Pre-allocate arrays

```
> vec1<-NULL
```

```
> # execute the command
> system.time(
+ for(i in 1:100000)
+ vec1 <- c(vec1,mean(1:100)))
   user  system elapsed
 58.181   0.193  58.417
```

Pre-allocate arrays

```
> vec2 <- vector(
+ mode="numeric",length=100000)

> # execute the command
> system.time(
+ for(i in 1:100000)
+ vec2[i] <- mean(1:100))
   user  system elapsed
  2.324   0.063   2.388
```

Use optimized R-functions

- `rowSums()`, `rowMeans()`, `table()`, etc.

```
> matx <- matrix(rnorm(1000000),100000,10)

> # execute the command
> system.time(apply(matx,1,mean))
   user  system elapsed
  2.686   0.057   2.748

> matx <- matrix(rnorm(1000000),100000,10)

> # execute the command
> system.time(rowMeans(matx))
   user  system elapsed
  0.013   0.000   0.014
```

Rcpp = R and C++

- R is a high-level *interpreted* language
- C/C++ are low-level *compiled* languages
- C is approximately more than 50X times faster than R
- R is much better for prototyping - one line of code in R is typically many lines of code in C/C++
- `Rcpp` was created by Dirk Eddelbuettel and Romain Francois in 2011. Permits direct interchange of rich R objects between R and C++

https://darrenjw.wordpress.com/2011/07/16/gibbs-sampler-in-various-languages-revisited/

http://adv-r.had.co.nz/Rcpp.html

http://dirk.eddelbuettel.com/code/rcpp.html

Code profiling

Profiling is a tool, which can be used to find out how much time is spent in each function. Code profiling can give a way to locate those parts of a program which will benefit most from optimization.

- `Rprof()` – turn profiling on
- `Rprof(NULL)` – turn profiling off
- `summaryRprof("Rprof.out")` – Summarize the output of the `Rprof()` function to show the amount of time used by different R functions.

```
> summaryRprof("bmslow.out")
$by.self
         self.time self.pct total.time total.pct
"cbind"     400.52    99.39     400.52     99.39
"rnorm"       1.70     0.42       1.70      0.42
"bmslow"      0.74     0.18     402.96    100.00
```

Code profiling

- `microbenchmark` - Accurate Timing Functions. Provides infrastructure to accurately measure and compare the execution time of R expression
- `profvis` - Interactive Visualizations for Profiling R Code Overview,
- `bench` - High Precision Timing of R Expressions

https://cran.r-project.org/web/packages/microbenchmark/index.html

https://rstudio.github.io/profvis/

http://r-lib.github.io/bench, https://github.com/r-lib/bench

R goodies

- `skimr` - A frictionless, pipeable approach to dealing with summary statistics, https://github.com/ropenscilabs/skimr
- `data.table` - fast data reading, subsetting, aggregating, summarizing, https://github.com/Rdatatable/data.table/wiki/Getting-started
- Whenever you get a strange execution error it is sometimes helpful to show the history of all the function calls leading to that error. This is done by typing `traceback()` at the command prompt