Efficient code

The R Bootcamp
Twitter: @therbootcamp

September 2017

What is efficient code?

> "Programmers waste enormous amounts of time thinking about, or worrying about, the speed of noncritical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered."
>
> -- Donald Knuth

Donald E. Knuth

Author of The Art of Programming

source http://www-cs-faculty.stanford.edu/

Why is R slow? And, is it?

> R is not a fast language. This is not an accident. R was purposely designed to make data analysis and statistics easier for you to do. It was not designed to make life easier for your computer. While R is slow compared to other programming languages, for most purposes, it's fast enough.
>
> -- Hadley Wickham

Reasons for R being slow

Extreme dynamism - allows you to code flexibly.

Name lookup (in environments) - allows you to import packages and name your objects flexibly.

Profiling

The first step to making your code efficient is to **identify critical parts** of your code. Do this using one of the following:

Function	Package	Description
`proc.time()`	`base`	Returns the time.
`system.time()`	`base`	Runs one expression once and returns elapsed CPU time
`microbenchmark()`	`microbenchmark`	Runs one or many expressions multiple times and returns statistics on elapsed time.
`lineprof()`, `shine()`	`lineprof`	Evaluates entire scripts. (From Hadley's Github)

Profiling: Example

Often some small part of your code takes orders of magnitudes longer than everything else. Profiling is about figuring out which parts of your code take so long.

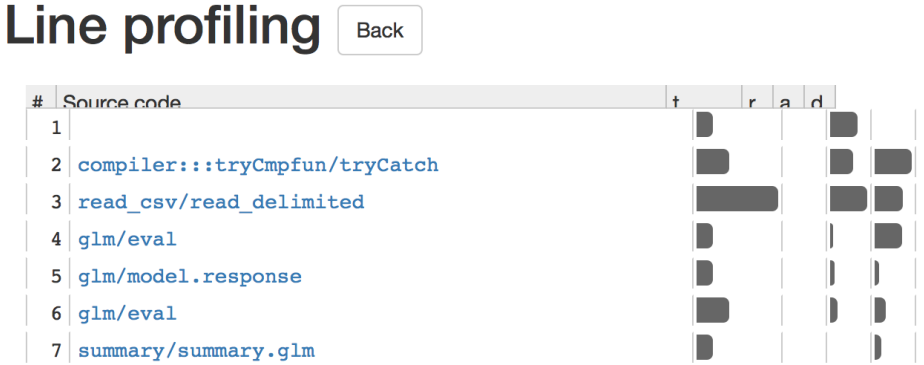
```r
# load data
data <- read_csv('titanic.csv')

# remove first column
data <- data[,-1]

# mutate
data <- data %>%
  mutate(months = Age * 12)

# multiple regression
model <- glm(Survived ~ Sex * Age,
             data = data,
             family = 'binomial')

# evaluate model
summary(model)
```



Improving performance

beginner

1. Look for existing solutions
2. Do less work
3. Vectorise
4. Parallelise
5. Avoid copies
6. Byte-code compile

advanced

7. Rcpp
8. Using a different R

Look for an existing solution

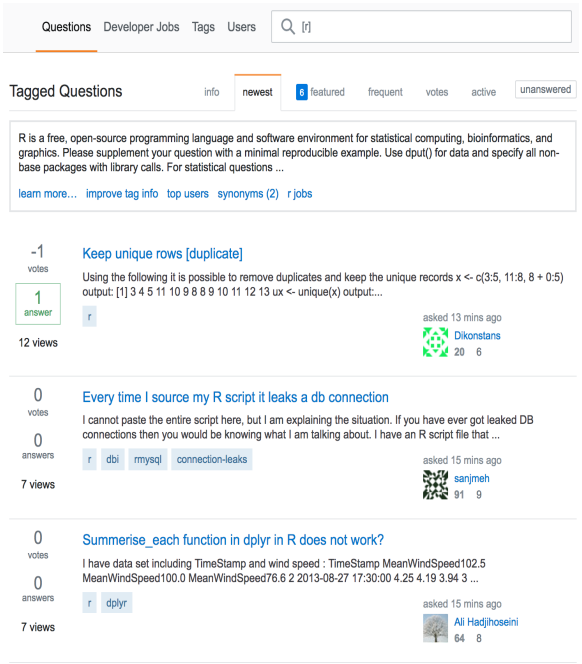
Almost always your problem has been solved by someone else.

Look for solutions in:

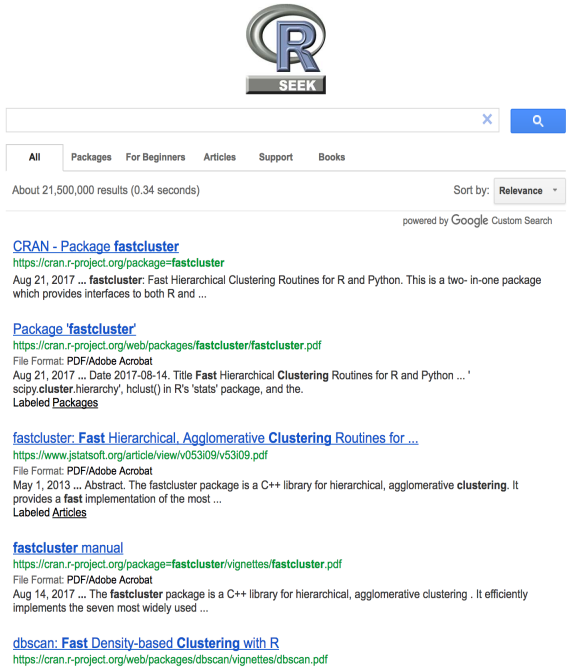
Base R which can be amazingly fast.

Other packages which often provide faster versions of one and the same function.

google, **stackoverflow**, **rseek**



https://stackoverflow.com



http://rseek.org

Do as little as possible

Or be as specific as possible

Use **tailor-made** functions, e.g., `.subset2()`.

Provide focus, e.g., `unlist(x, use.names = F)` vs. `unlist()`.

Don't repeat code.

```
# load package
library(microbenchmark, quietly = T)

# define link to data
link <- 'http://tinyurl.com/y99aj5ed'

# microbenchmark
microbenchmark(
  web   = read_csv(link),
  local = read_csv('data/titanic.csv'),
  fread = fread('data/titanic.csv'),
  times = 10)
```

```
## Unit: microseconds
##   expr    min     lq   mean median      uq     max neval cld
##    web 753036 784171 931712 950495 1046002 1077292    10   b
##  local   4190   4535   4925   4895    5136    6052    10  a
##  fread    929   1069   1374   1302    1499    2500    10  a
```

Vectorise

Whenever possible **use vector operations** or functions do vectorized operations.

In other words, **don't use loops** and stay away from all **apply idoms**, such as `apply()`, `sapply()`, `tapply()`, etc.

```
# create data
my_data <- matrix(rnorm(1000000), ncol = 10)

# microbenchmark
microbenchmark(
  colMeans = colMeans(my_data),
  apply = apply(my_data, 2, mean),
  times = 10)
```

```
## Unit: microseconds
##      expr   min    lq  mean median    uq    max neval cld
##  colMeans   774   778  1063    846   893   2508    10  a
##     apply 11877 12918 25289  13926 15315 126855    10   b
```

Avoid copies

R always copies.

whenever using c(), cbind(), rbind(), paste() R creates a copy large enough to contain its inputs.

```
# define character vectors
letters10  <- LETTERS[sample(1:26, 10, replace = T)]
letters100 <- LETTERS[sample(1:26, 100, replace = T)]

# define collapse function
collapse <- function(x) {
  my_string <- ''
  for(i in 1:length(x)) paste(my_string, x, sep = "")
}

# microbenchmark
microbenchmark(
  loop10  = collapse(letters10),
  loop100 = collapse(letters100),
  vec10   = paste(letters10, collapse = ""),
  vec100  = paste(letters100, collapse = "")
)
```

```
## Unit: microseconds
##     expr     min       lq     mean  median       uq     max neval cld
##   loop10   22.06   24.58   51.16   25.81   30.11 2365.99   100  a
##  loop100 1225.22 1304.80 1479.32 1420.30 1602.66 2337.05   100   b
##    vec10    1.79    2.23    2.79    2.46    2.92    7.88   100  a
##   vec100    9.64   10.08   12.01   11.01   12.46   30.45   100  a
```

Byte-code compilation

Using the base R package `compiler` code can be compiled to byte-code, a faster, **lower-level version of the code**.

As R compiles functions anyway after the first execution, this is **often not useful**.

```
# define unneccessarily complex function and compile
my_fun <- function(x, f, ...) {
  for (i in seq_along(x)) f(x[[i]], ...)
  }
my_fun_c <- compiler::cmpfun(my_fun)

# define some awfully complex data
x <- list(1:10, letters, c(F, T), NULL)

# microbenchmark
microbenchmark(my_fun(x, is.null), my_fun_c(x, is.null), unit='us')
```

```
## Unit: microseconds
##                   expr   min   lq mean median   uq    max neval cld
##    my_fun(x, is.null) 0.957 1.07 19.6   1.18 1.37 1840.0   100   a
##  my_fun_c(x, is.null) 0.954 1.10  1.4   1.20 1.38   14.6   100   a
```

```
# microbenchmark again
microbenchmark(my_fun(x, is.null), my_fun_c(x, is.null), unit='us')
```

```
## Unit: microseconds
##                   expr  min   lq mean median   uq   max neval cld
##    my_fun(x, is.null) 1.11 1.25 1.44   1.39 1.55  3.86   100   a
##  my_fun_c(x, is.null) 1.11 1.19 1.56   1.37 1.54 19.02   100   a
```

Parallel computing

When working with large data one of the **best way to speed up** execution is parallel execution.

Parallel execution splits the data in many **jobs** and then has many **workers** (separate R instances) working on them in parallel.

```
# define data and splitted data (jobs)
data       <- matrix(rnorm(10000000), ncol = 10)
split_data <- lapply(1:10, function(i) data[(1:1000)+(i-1)*1000, ])

# open cluster
require(parallel)
clu <- makeCluster(5)

# my cluster fun
my_cluster_fun <- function(split_data){

  # apply cluster function
  out <- clusterApplyLB(clu, split_data, colMeans)

  # combine results
  colMeans(do.call(rbind, out))
  }

# microbenchmark
microbenchmark(vectorz = colMeans(data),
               cluster = my_cluster_fun(split_data))
```

```
## Unit: milliseconds
##     expr  min   lq mean median   uq  max neval cld
##  vectorz 7.71 8.18 8.83   8.47 9.27 11.7   100   b
##  cluster 2.44 2.74 3.15   3.00 3.34 11.1   100  a
```

Rcpp

Another very effective, but highly advanced option is to write the essential code using Rcpp - **R's C++ interface**.

As many functions are already implemented in C++ or Fortran in the background, this works well **only for non-standard operations**.

Quick-Guide

```
# define data
my_data <- matrix(rnorm(10000000),ncol = 10)

# define function
my_Rcpp_fun = "
NumericVector colMeans_c(NumericMatrix x){
  NumericVector out(x.ncol());
  for(int j = 0; j < x.ncol(); ++j){
    double m = 0;
    for(int i = 0; i < x.nrow(); ++i) m += x(i, j);
  out[j] = m / x.nrow();
  }
return out;
}"

# compile function
require(Rcpp) ; cppFunction(my_Rcpp_fun)

# microbenchmark
microbenchmark(vectorz = colMeans(my_data),
               Rcpp = colMeans_c(my_data))
```

```
## Unit: milliseconds
##     expr  min   lq mean median   uq  max neval cld
##  vectorz 7.67 7.91 8.71   8.27 9.12 13.0   100   a
##     Rcpp 7.61 7.83 8.46   8.01 8.97 11.6   100   a
```

Alternative R implementations

from **Advanced R**

R implementation	Author	Description
`pqR`	Radford Neal	Built on top of R 2.15.0, it fixes many obvious performance issues, and provides better memory management and some support for automatic multithreading.
`Renjin`	BeDataDriven	Renjin uses the Java virtual machine, and has an extensive test suite.
`FastR`	Purdue University	FastR is similar to Renjin, but it makes more ambitious optimisations and is somewhat less mature.
`Riposte`	Justin Talbot and Zachary DeVito	Riposte is experimental and ambitious. For the parts of R it implements, it is extremely fast. Riposte is described in more detail in Riposte: A Trace-Driven Compiler and Parallel VM for Vector Code in R.

Practical

Link to practical