High Performance Computing - 2: fasteR

Contents

Simple things to make R go fasteR

This section will review a couple of simple things one can do to make R go fasteR.

Better Computer

How fast is your computer may be a sensitive question for many, yet this is one of the most obvious ways to make R functions run faster. A good starting point is to figure out whether your computer is slow or fast. Of course, there is a package for that.

library(benchmarkme)

This library will not only pull up hardware specs but also compare your computer to others machines.

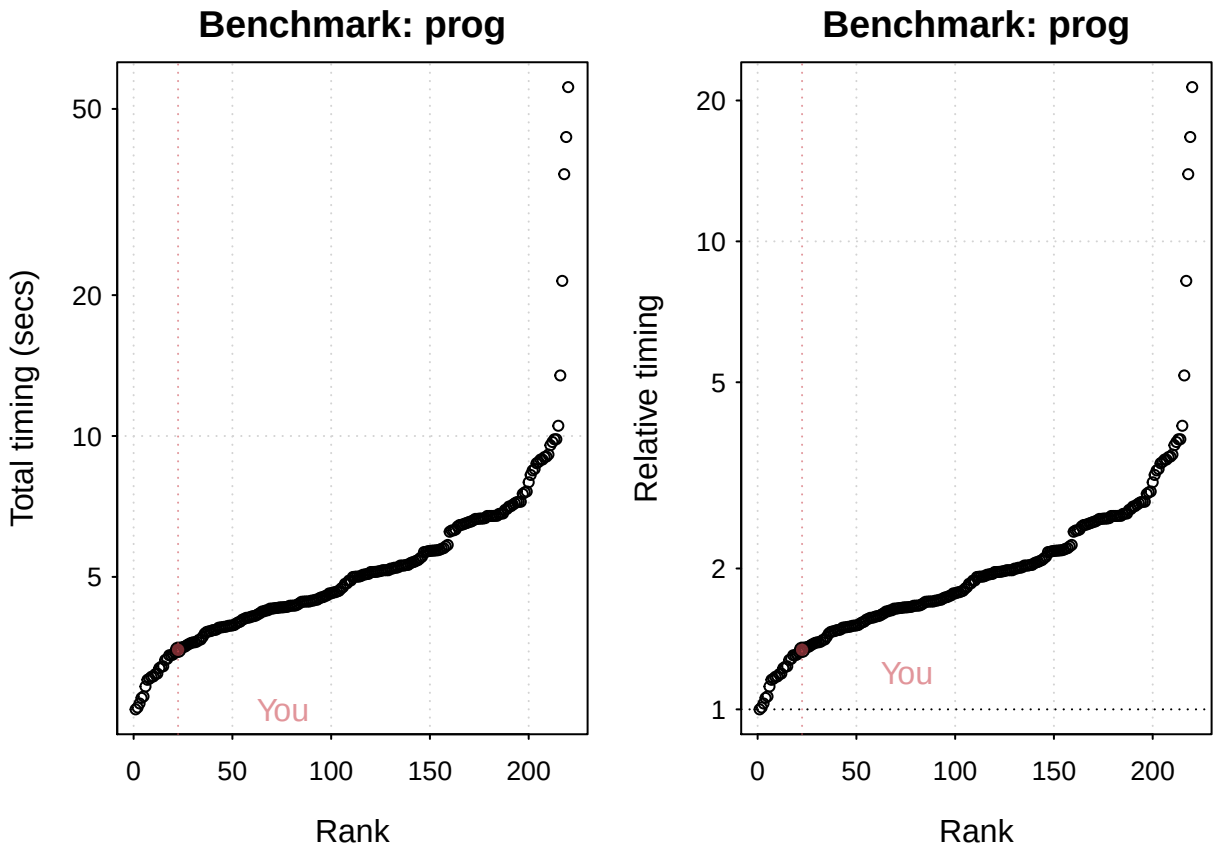
```
library(benchmarkme)
get_ram()
```

```
## NA B
```

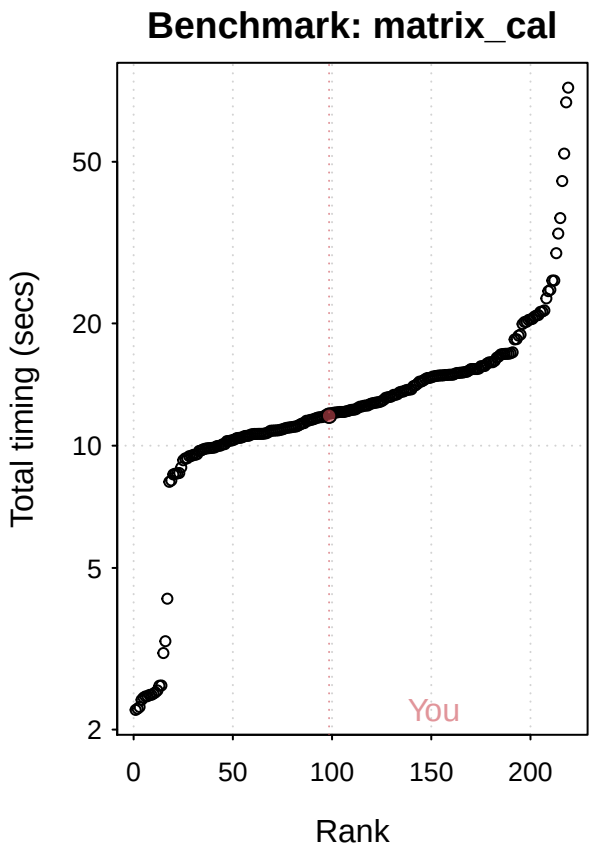
```
get_cpu()
```

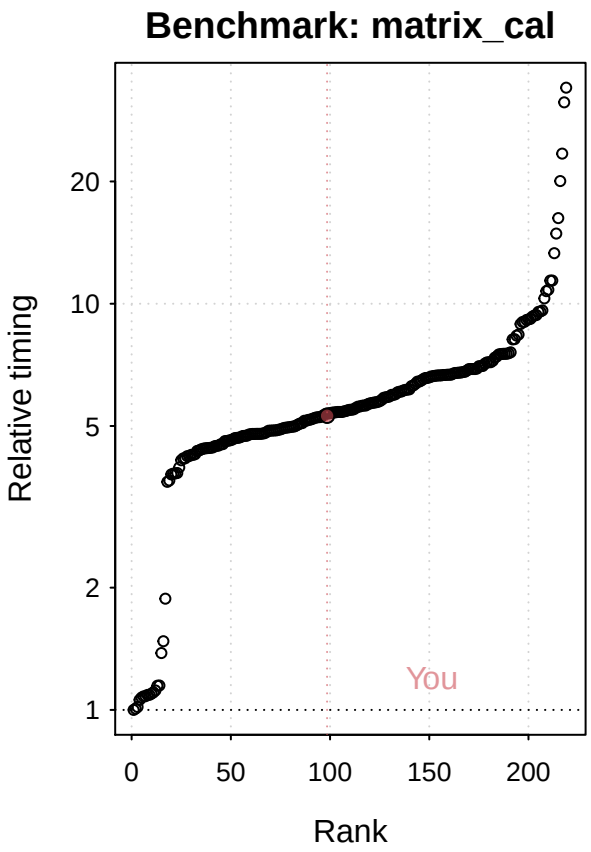
```
## $vendor_id
## [1] "GenuineIntel"
##
## $model_name
## [1] "Intel(R) Core(TM) i7-8650U CPU @ 1.90GHz"
##
## $no_of_cores
## [1] 8
```

```
speedResult = benchmark_std(runs = 3)
plot(speedResult)
```

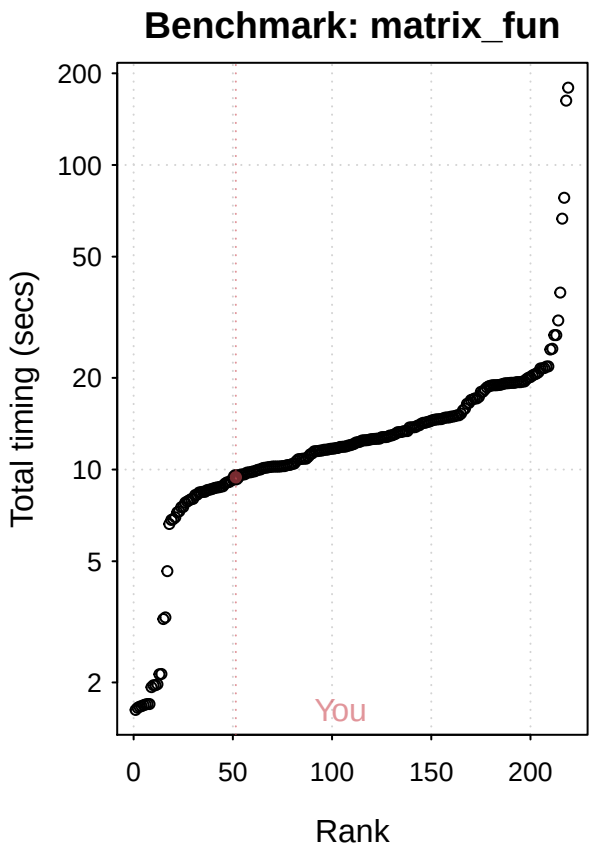


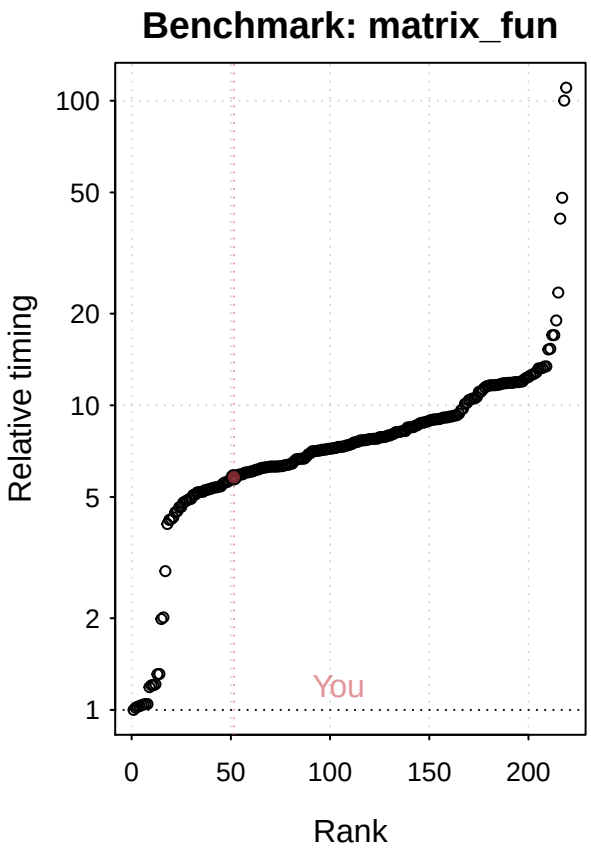

```
## Press return to get next plot
```



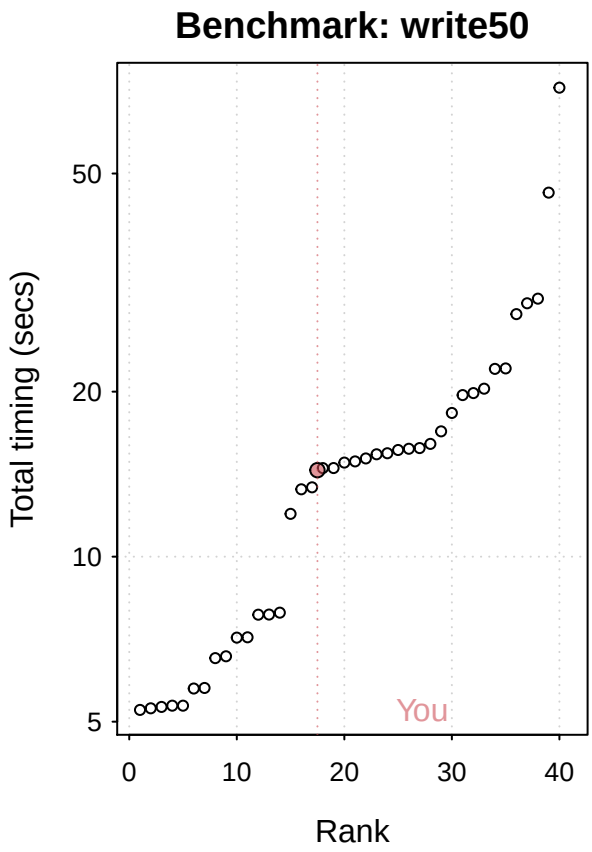


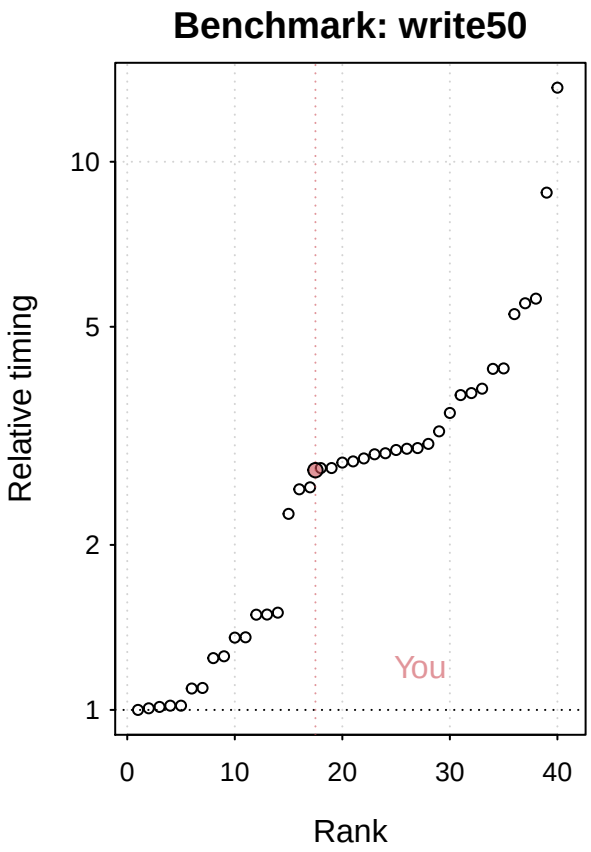

```
## Press return to get next plot
```



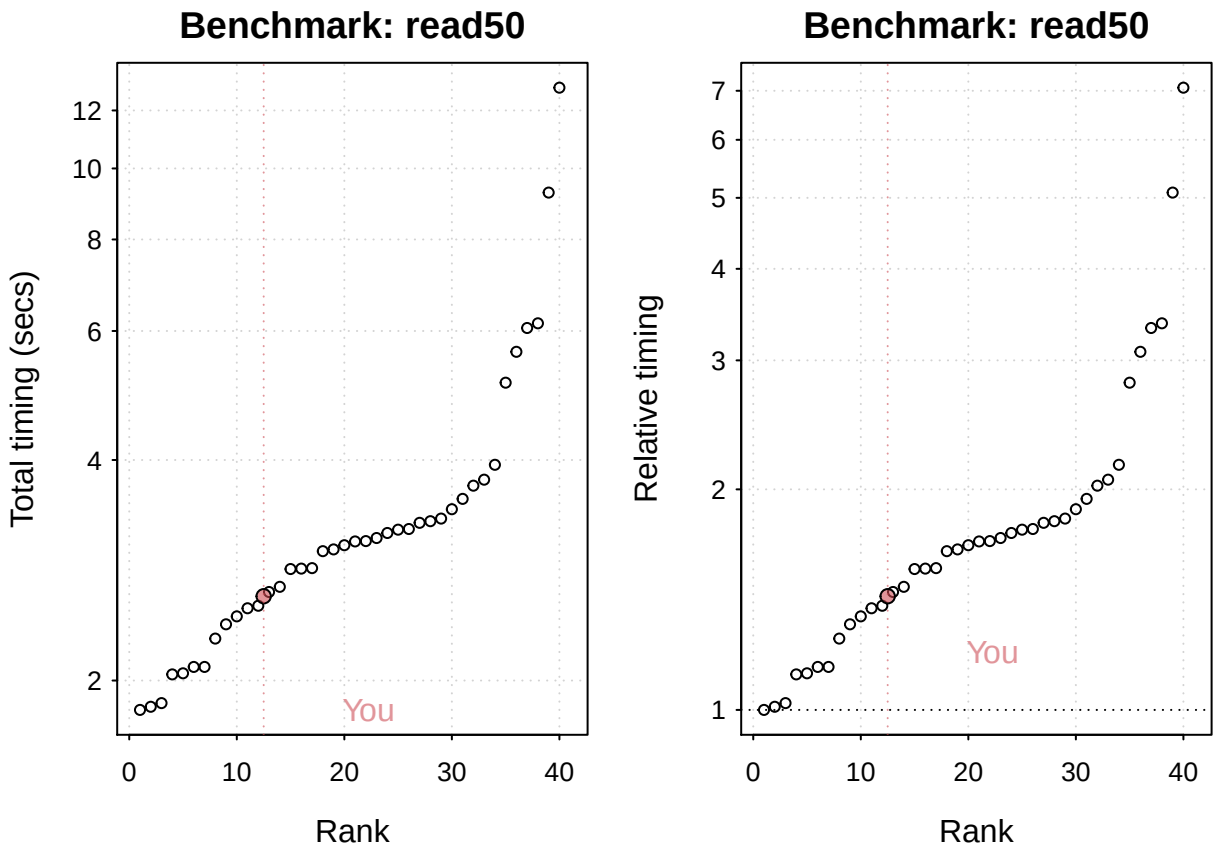



```
speed_io = benchmark_io(runs=3,size = 50)
plot(speed_io)
```






```
## Press return to get next plot
```



Vectorization (over loops)

Vectorized operations involve applying a function once to an entire vector. On the other hand, a loop will apply the function to each element n times.

Loop

```
num = sample(1:10,size=1e7,replace=T)
num_square = integer(length(num))

system.time(for (i in 1:length(num)){
  num_square[i] = num[i]^2
})
```

```
##    user  system elapsed
##    0.58    0.00    0.58
```

Vectorized operation

```
system.time(num_squared <- num^2)
```

```
##    user  system elapsed
##    0.03    0.00    0.03
```

Use built-in functions

Many R functions and packages are implemented in compiled languages like C/C++ These will always run faster than functions written in R, an interpreted language

```
data = sample(1:10,size = 1e7,replace=T)
dim(data) = c(100000,100)

system.time(apply(X=data,MARGIN = 2,FUN = sum))
```

```
##    user  system elapsed
##     0.1     0.0     0.1
```

```
system.time(colSums(data))
```

```
##    user  system elapsed
##    0.02    0.00    0.01
```

```
library(microbenchmark)
microbenchmark(apply = apply(X=data,MARGIN = 2,FUN = sum),colSums = colSums(data))
```

```
## Unit: milliseconds
##     expr       min         lq       mean     median         uq      max
##    apply 93.209801 101.849801 111.881589 105.674051 109.600551 217.7091
##  colSums  7.531101   8.192101   8.868389   8.848602   9.362202  11.0057
##  neval
##    100
##    100
```

Use faster functions

Certain functions are faster, possibly because they are optimized in C++ or because of more efficient ways processes. `aggregate`, `tapply`, `dplyr`, and `data.table` are all ways of summarizing data, but there are differences in performance.

```
library(ggplot2)
library(dplyr)
str(diamonds)
```

```
## Classes 'tbl_df', 'tbl' and 'data.frame':    53940 obs. of  10 variables:
##  $ carat  : num  0.23 0.21 0.23 0.29 0.31 0.24 0.24 0.26 0.22 0.23 ...
##  $ cut    : Ord.factor w/ 5 levels "Fair"<"Good"<..: 5 4 2 4 2 3 3 3 1 3 ...
##  $ color  : Ord.factor w/ 7 levels "D"<"E"<"F"<"G"<..: 2 2 2 6 7 7 6 5 2 5 ...
##  $ clarity: Ord.factor w/ 8 levels "I1"<"SI2"<"SI1"<..: 2 3 5 4 2 6 7 3 4 5 ...
##  $ depth  : num  61.5 59.8 56.9 62.4 63.3 62.8 62.3 61.9 65.1 59.4 ...
##  $ table  : num  55 61 65 58 58 57 57 55 61 61 ...
##  $ price  : int  326 326 327 334 335 336 336 337 337 338 ...
##  $ x      : num  3.95 3.89 4.05 4.2 4.34 3.94 3.95 4.07 3.87 4 ...
##  $ y      : num  3.98 3.84 4.07 4.23 4.35 3.96 3.98 4.11 3.78 4.05 ...
##  $ z      : num  2.43 2.31 2.31 2.63 2.75 2.48 2.47 2.53 2.49 2.39 ...
```

```
data = diamonds[sample(1:nrow(diamonds),size = 1e6,replace=T),]
system.time(aggregate(price~cut,data,mean))
```

```
##    user  system elapsed
##    0.22    0.04    0.25
```

```
system.time(tapply(data$price,data$cut,mean))
```

```
##    user  system elapsed
##    0.01    0.00    0.02
```

```
system.time(data%>%
              group_by(cut)%>%
              summarize(price=mean(price)))
```

```
##    user  system elapsed
##    0.03    0.00    0.03
```

```
library(data.table)
dt = data.table(data)
system.time(dt[,mean(price),by='cut'])
```

```
##    user  system elapsed
##    0.07    0.00    0.03
```

```
microbenchmark(aggregate = aggregate(price~cut,data,mean),
               tapply = tapply(data$price,data$cut,mean),
               dplyr = data%>% group_by(cut)%>% summarize(price=mean(price)),
               data.table = dt[,mean(price),by='cut'],times = 5)
```

```
## Unit: milliseconds
##        expr      min       lq      mean   median       uq      max neval
##   aggregate 248.5514 258.6974 279.93940 284.5857 288.6803 319.1822     5
##      tapply  14.8571  15.0157  16.00934  15.3419  16.8178  18.0142     5
##       dplyr  18.5646  19.5972  23.80064  20.4323  24.9480  35.4611     5
##  data.table  21.1708  21.1967  26.61130  21.3339  33.8206  35.5345     5
```

Pre-allocate memory

In programming languages like C, C++, or Java, a vector (or array) has to be declared prior to its use. Declaring in effect preallocates memory space.

In R, this happens automatically. But, if the memory allocated is not large enough, then R will have to create a larger space and move the data to the larger memory space. This reallocation based on need can slow things down, especially if it is done repeatedly for each additional element. Manually allocating memory can save R some time.

Memory not preallocated

```
numbers = sample(1:10,size=1e6,replace=T)
num_square =
  function(num){
  num_square = integer()
    for (i in 1:length(num)){
      num_square[i] = num[i]^2
  }
}
system.time(num_square(numbers))
```

```
##    user  system elapsed
##    0.25    0.00    0.25
```

Memory preallocated

```
num_square_preallocated =
  function(num){
    num_square = integer(length(num))
      for (i in 1:length(num)){
        num_square[i] = num[i]^2
    }
  }
system.time(num_square_preallocated(numbers))
```

```
##    user  system elapsed
##    0.07    0.00    0.07
```

```
microbenchmark(num_square(numbers),num_square_preallocated(numbers),times = 5)
```

```
## Unit: milliseconds
##                               expr      min       lq      mean   median
##               num_square(numbers) 212.1671 213.2324 230.73644 214.5849
##  num_square_preallocated(numbers)  54.4287  54.9023  65.33224  65.0815
##        uq      max neval
##  237.0752 276.6226     5
##   73.2437  79.0050     5
```

```
numbersX10 = sample(1:10,size=1e7,replace=T)
numbersX50 = sample(1:10,size=5e7,replace=T)
```

```
microbenchmark(num_square(numbers),num_square_preallocated(numbers),
               num_square(numbersX10),num_square_preallocated(numbersX10),
               num_square(numbersX50),num_square_preallocated(numbersX50), times = 5)
```

```
## Unit: milliseconds
##                                 expr        min         lq       mean
##                 num_square(numbers)   192.9677   198.2499   225.0252
##    num_square_preallocated(numbers)    50.4048    51.2434    54.1411
##               num_square(numbersX10)  2091.4580  2159.0479  2194.5514
##  num_square_preallocated(numbersX10)   517.3215   522.8447   548.1758
```

```
##               num_square(numbersX50) 10934.8037 10972.8561 11223.3313
##  num_square_preallocated(numbersX50)  2614.7162  2731.5347  2764.1884
##      median         uq        max neval
##    203.0666   213.6380   317.2039     5
##     52.8251    57.5290    58.7032     5
##   2162.5725  2235.3871  2324.2914     5
##    546.2083   555.4134   599.0913     5
##  11079.5498 11169.9413 11959.5057     5
##   2731.8380  2754.1040  2988.7489     5
```

Another Example

```
s3 = function(n) {
  sum = numeric()
  for (i in 1:n){
    sum = sum+i
  }
  sum
}
system.time(s3(1e6))
```

```
##    user  system elapsed
##    0.05    0.00    0.05
```

```
s3_preallocate = function(n) {
  sum = numeric(length(n))
  for (i in 1:n){
    sum = sum+i
  }
  sum
}
system.time(s3_preallocate(1e6))
```

```
##    user  system elapsed
##    0.01    0.00    0.02
```

```
microbenchmark(s3(1e6),s3_preallocate(1e6),times = 10)
```

```
## Unit: milliseconds
##                   expr     min      lq     mean  median      uq     max
##              s3(1e+06) 51.1861 51.9494 54.19305 54.4818 56.3459 57.1304
##  s3_preallocate(1e+06) 19.5551 20.1282 21.36295 20.6064 22.4966 25.1389
##  neval
##     10
##     10
```

Use simpler data structures

For instance, if all the data is of the same class, it is better to use a matrix rather than data.frame. Also, data.table may not always be the fastest

```
mat = sample(1:10,size = 1e8,replace=T)
dim(mat) = c(100000,1000)
system.time(colSums(mat))
```

```
##    user  system elapsed
##    0.14    0.09    0.23
```

```
df = data.frame(mat)
system.time(colSums(df))
```

```
##    user  system elapsed
##    0.24    0.03    0.26
```

```
microbenchmark(matrix = colSums(mat),data.frame = colSums(df),times = 10)
```

```
## Unit: milliseconds
##        expr      min       lq     mean    median       uq      max neval
##      matrix  78.5877  79.8753  81.7139  80.54365  82.9724  86.7583    10
##  data.frame 259.1037 264.0137 294.3046 290.09910 329.0335 346.7585    10
```

```
microbenchmark(matrix = mat[,100],   data.frame = df[,100],times=10) # Subsetting col 100
```

```
## Unit: microseconds
##        expr     min      lq     mean  median      uq     max neval
##      matrix 281.101 281.802 291.1109 285.001 293.001 314.601    10
##  data.frame   5.601   6.001  10.9710   7.301   8.401  43.501    10
```

```
microbenchmark(matrix = mat[100,],   data.frame = df[100,],times=10) # Subsetting row 100
```

```
## Unit: microseconds
##        expr      min       lq      mean    median       uq      max neval
##      matrix    5.201    5.901   19.7211   11.9515   30.601   50.902    10
##  data.frame 3830.402 3885.000 4237.5510 4102.6505 4441.001 4928.401    10
```

```
microbenchmark(matrix = mat[100,100],data.frame = df[100,100],times=10) # Subsetting row 100, col 100
```

```
## Unit: nanoseconds
##        expr   min    lq    mean  median    uq   max neval
##      matrix     1   101   611.0   250.5   401  4301    10
##  data.frame 15501 16501 25711.3 19001.0 30001 59602    10
```

```
library(dplyr)
library(data.table)
dt = data.table(mat)
system.time(colSums(dt))
```

```
##    user  system elapsed
##    0.25    0.02    0.28
```

```
dplyr_df = as_data_frame(mat)
system.time(colSums(dplyr_df))
```

```
##    user  system elapsed
##    0.25    0.02    0.26
```

```
microbenchmark(matrix = colSums(mat),data.frame = colSums(df),data.table = colSums(dt), dplyr = colSums
```

```
## Unit: milliseconds
##        expr      min       lq     mean   median       uq      max neval
##      matrix  80.5359  81.5133  82.5637  82.4817  83.6029  84.6010    10
##  data.frame 265.4383 269.9502 288.5585 271.6755 278.4160 429.4637    10
##  data.table 266.1023 270.7439 343.4811 359.7571 386.3648 460.1683    10
##       dplyr 266.4850 270.2058 296.0127 272.8927 340.7059 379.1870    10
```

```
microbenchmark(matrix = mat[100,100],data.frame = df[100,100],data.table = dt[100,100],dplyr = dplyr_df
```

```
## Unit: nanoseconds
##        expr    min     lq     mean   median     uq     max neval
##      matrix    201    202    800.9    750.5   1201    2201    10
##  data.frame  14601  19301  27421.1  25451.5  32201   43602    10
##  data.table 320401 444801 725821.1 637401.5 798402 1756000    10
##       dplyr  31101  44101  74931.0  59201.0  62801  258001    10
```

Use hash tables

For frequent lookups on large data, it is better to use Hash tables

```
data = rnorm(1E4)
data_ls = as.list(data)
names(data_ls) = paste("V", c(1:1E4), sep="")
index_rand = sample(1:1E4, size=1000, replace=T)
index = paste("V", index_rand, sep="")
list_comptime = sapply(index, FUN=function(x){
  system.time(data_ls[[x]])[3]})
sum(list_comptime)
```

```
## [1] 0.04
```

```
library(hash)
data_h = hash(names(data_ls), data)
hash_comptime = sapply(index, FUN=function(x){
  system.time(data_h[[x]])[3]})
sum(hash_comptime)
```

```
## [1] 0.05
```

Use faster, more efficient packages

In this example, you will note that fastcluster gives better performance than the clustering algorithm in the stats package. On the other hand, Rcppeigen did not do any better than lm.

```r
data = rnorm(1e4*100)
dim(data) = c(1e4,100)
dist_data = dist(data)

system.time(hclust(dist_data))

##    user  system elapsed
##    3.14    0.01    3.15

library(fastcluster)
system.time(hclust(dist_data))

##    user  system elapsed
##    1.41    0.10    1.51

data = rnorm(10000*100)
dim(data) = c(10000,100)
#princomp(data)
#prcomp(data)
microbenchmark(princomp(data),prcomp(data),times=5)

## Unit: milliseconds
##            expr      min       lq     mean   median       uq      max
##  princomp(data) 140.5168 140.8075 141.6153 141.3111 142.2221 143.2191
##    prcomp(data) 265.3858 266.5744 270.4801 266.5758 276.5318 277.3325
##  neval
##      5
##      5

library(ggplot2)
data = diamonds[sample(1:nrow(diamonds),size = 1e6,replace=T),]
system.time(lm(price~.,data))

##    user  system elapsed
##    0.97    0.05    1.03

library(RcppEigen)
x = model.matrix(price~.-1,data)
y = data$price

fast = function(dataframe){
  library(RcppEigen)
  x = model.matrix(price~.-1,data)
  y = data$price
  fastLm(x,y)
}

microbenchmark(lm(price~.,data),fast(data),times=5)

## Unit: seconds
##                 expr      min       lq     mean   median       uq      max
##  lm(price ~ ., data) 1.037521 1.041725 1.099751 1.043391 1.174409 1.201711
```

```
##           fast(data) 1.136776 1.208716 1.253311 1.281928 1.311353 1.327783
##  neval
##      5
##      5
```