Speeding Up R: When is it worthwhile?

Marco D. Visser Sean McMahon Caspar Hallmann

October 1, 2015

Workshop schedule

1. General introduction
2. *Identifying whether and what to optimize*
3. Break out groups

Obligatory Donald Knuth quote

"We should forget about small inefficiencies, say about 97% of the time: premature optimization is the root of all evil"

- Donald Knuth (1974). "Structured Programming with go to Statements". ACM Journal Computing Surveys 6 (4): 268.

Best coding practices

1. Correct code. Start with easy to debug R code, don't worry about efficiency.
2. Too slow? Start profiling your program and calculate/guesstimate your expected gains (*aprof*).
3. Found a bottleneck? Confirm optimizing is worthwhile, then look for a pure R solution (greatest returns).
   - Confirm new version is correct.
4. Still not satisfied? Can your program run in parallel?
   - Confirm new version is correct.
5. Only when all else fails: refactor. Rewrite certain key parts in C, C++ or Fortran.
   - Confirm new version is correct.

This presentation, with all **knitr** code examples is available at github.com/MarcoDVisser/SpeedUpR

How fast is my code?

- Simple function

```
IQ1 <- function(N,r) {
  for (i in 1:r) {
    for (j in 1:N) { N/(1+N) }
    }
 }
```

- How long does it take (time in seconds)?

```
system.time(IQ1(100000,10))
```

```
##    user  system elapsed
##   0.432   0.004   0.437
```

- This is calculated from difference of proc.time() output

```
proc.time()

##    user  system elapsed
##   0.792   0.040   0.907

IQ1(100000,10)
proc.time()

##    user  system elapsed
##   1.272   0.040   1.387
```

- When comparing functions note: timing is random, for various reasons.

```
system.time(IQ1(100000,10))

##    user  system elapsed
##   0.464   0.000   0.466

system.time(IQ1(100000,10))

##    user  system elapsed
##   0.452   0.000   0.453
```

More precise timing

- microbenchmark() in microbenchmark package
  - Runs the expression multiple times to reduce variation
  - default is 100 times

```
 IQ2 <- function(N,r) {
   for (i in 1:r) {
     for (j in 1:N) { ((((((N/(1+N)))))) }
     }
  }

require(microbenchmark)

## Loading required package:  microbenchmark

microbenchmark(IQ1(100000,10),IQ2(100000,10), times=5)

## Unit: milliseconds
##            expr      min       lq     mean   median       uq      max
##  IQ1(1e+05, 10) 423.7159 427.4448 439.2324 434.6596 452.1753 458.1666
##  IQ2(1e+05, 10) 807.1674 809.8396 821.6858 820.7307 833.0454 837.6462
##  neval
##      5
##      5
```

Basic code tests

- Basic function: all.equal() & identical()
- Test driven development "testthat" or "RUnit".

```
x<-c(1,2,3,4,5)
identical(mean(x),sum(x)/length(x))

## [1] TRUE

y<-data.frame(x=x,y=x^2)

all.equal(colMeans(y),apply(y,2,mean))

## [1] TRUE
```

Exercise 1: A bootstrap problem

We have a moderately large biodiversity dataset, with 750 000 records (modelled after Hennekens & Schaminee, 2001) of species richness.

- 750 plots at 1000 different sites

Goal: calculate 10 000 bootstrapped estimates of site-specific species richness v.s. overall mean species richness.

```
## OnlineRcode17.R
N<-7.5*10^2 #Number of records per site

S<-1000 # Number of sites
BioData<-data.frame(S=rpois(N*S,15),
                    site=as.factor(rep(1:S,N)))
```

Using the package boot (in base).

```
## OnlineRcode18.R
require(boot)
R<-10000 # number of bootstrap resamples
SiteMeans<-function(x,d,){
  tapply(x$S[d],x$site[d],mean)-mean(x)
}
BtResults<-boot(BioData,SiteMeans,R)
```

Error (R 3.1.0.):

```
Error: cannot allocate vector of size 55.9 Gb
```

Starting code:

```
## OnlineRcode19.R

##Naive bootstrap function in R
NaiveBoot<-function(x,R){
  results<-NULL
        for(i in 1:R){
        index<-sample(seq_len(nrow(x)),replace=TRUE)
          results<-rbind(results,
          tapply(x$S[index],x$site[index],
          function(X) mean(X)-mean(x)))
        }
        return(results)
}
```

Tip: always optimize a smaller problem.

```
## OnlineRcode20.R

## make a small subset of the data to work with
subBioData<-BioData[1:7.5e4,]

# 10% of the 10 000 resamples
subR<-1000
```

Exercise: Optimize the function *NaiveBoot*

1. Profile the function
   - use ?Rprof
   - use ?aprof from the aprof package
   - read section 1.3 (page 6) of the tutorial
2. Identify the largest bottleneck
3. determine if optimization is worthwhile
4. Improve only the largest bottleneck

(if more speed is needed repeat step 1 - 4)

Amdahl' s law

$$T_i = (1 - \alpha) T_o + \frac{\alpha T_o}{I} \tag{1}$$

where T_i is run time after optimization improves speed of a section of code by factor I, when this code section took a fraction α of the original run time. Note that equation 1 reduces to:

$$S = \frac{1}{(1 - \alpha) + \alpha / I} \tag{2}$$

where S is the maximal theoretical speed-up at current scaling.

Step 1: profiling

A possible solution:

```
## OnlineRcode21.R

## reload our program so everything matches up exactly
source("NaiveBoot.R")

## Switch on R's profiler
Rprof(file="NaiveBoot.out",line.profiling =TRUE)

## set the random seed so we can compare results
## later.
set.seed(123)

## Run NaiveBoot on a subset of data 1000 times
ResultsNB<-NaiveBoot(subBioData,subR)

## stop profiling
Rprof(append=F)
```

Step 1: profiling

```
summaryRprof("NaiveBoot.out")
```

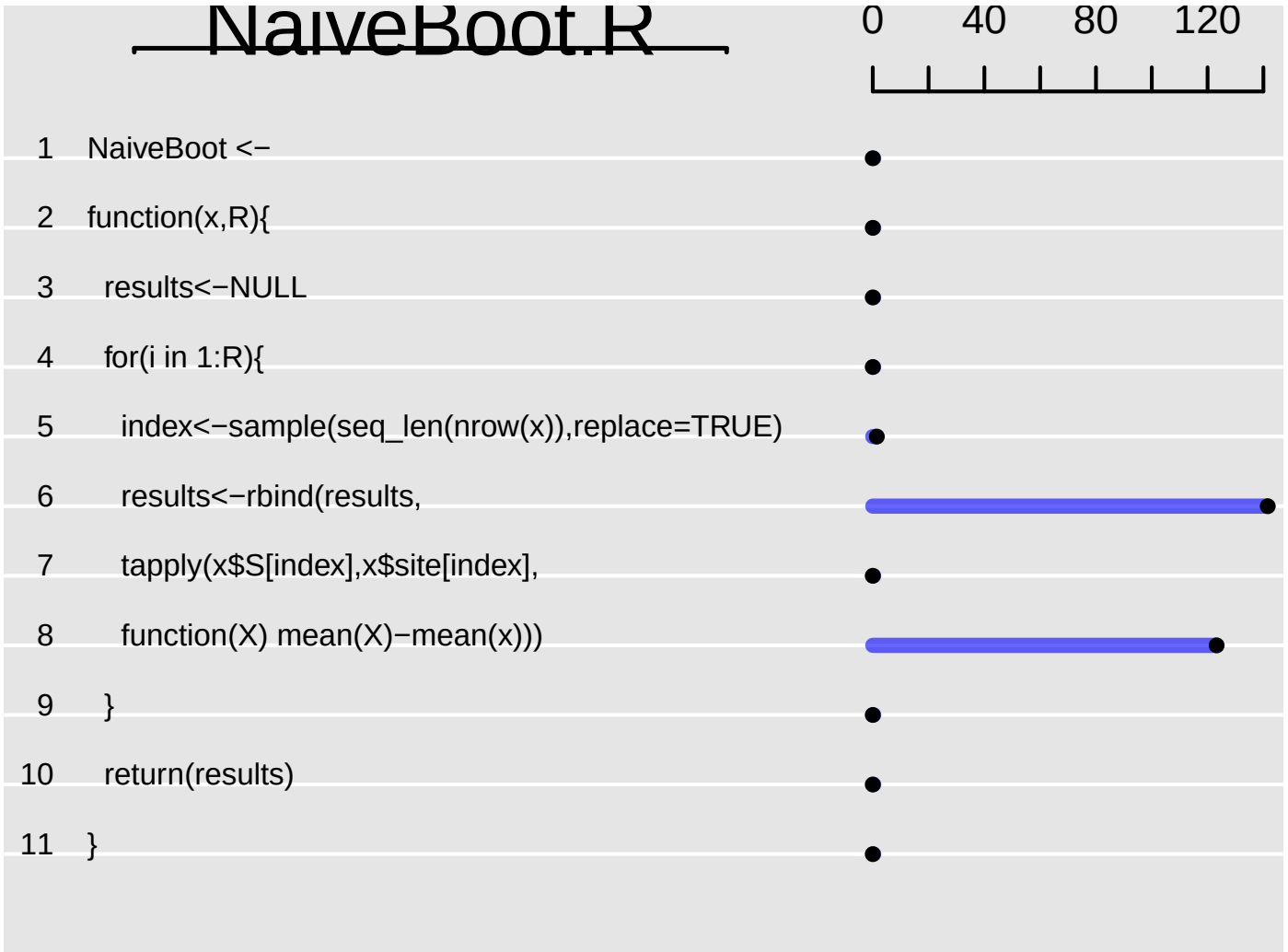
```
## $by.self
##                         self.time self.pct total.time total.pct
## ".makeMessage"              23.36    16.34      32.72     22.88
## "structure"                 17.98    12.58      35.00     24.48
## "mean"                      12.36     8.64     121.92     85.27
## "match"                     10.02     7.01      10.34      7.23
## "rbind"                      8.94     6.25     141.54     98.99
## "makeRestartList"            8.66     6.06      32.06     22.42
## "doWithOneRestart"           5.50     3.85      25.86     18.09
## "warning"                    5.34     3.73     101.24     70.81
## "mean.default"               5.08     3.55     109.56     76.63
## "$<-"                        5.08     3.55       5.08      3.55
## "lapply"                     3.48     2.43     128.82     90.10
## "paste"                      3.34     2.34       3.34      2.34
## "unlist"                     3.30     2.31       3.86      2.70
## "is.numeric"                 2.98     2.08       2.98      2.08
## "c"                          2.66     1.86       2.66      1.86
## "withRestarts"               2.62     1.83      62.04     43.39
## "%in%"                       2.36     1.65       7.36      5.15
## "split.default"              2.10     1.47       4.48      3.13
## "FUN"                        1.52     1.06     123.54     86.40
## "tapply"                     1.46     1.02     132.60     92.74
## "makeRestart"                1.40     0.98      20.12     14.07
## "simpleWarning"              1.30     0.91      17.58     12.30
## "withOneRestart"             1.26     0.88      27.12     18.97
## "list"                       1.04     0.73       1.04      0.73
## ".signalSimpleWarning"       1.02     0.71      63.06     44.10
## "length"                     1.02     0.71       1.02      0.71
```

Step 1: profiling

```
## OnlineRcode22.R

## Load Amdahl's profiler
require(aprof)

## make an object of the aprof class
NaiveBootAprof <- aprof("NaiveBoot.R","NaiveBoot.out")

## Plot the execution time per line
plot(NaiveBootAprof)
```

Step 1: profiling

```
## Loading required package:  aprof
```



Step 1: Is it worthwhile

```
## OnlineRcode23.R
summary(NaiveBootAprof)


## Largest attainable speed-up factor for the entire program
##
##         when 1 line is sped-up with factor (S):
##
##   Speed up factor (S) of a line
##              1     2     4     8     16    S -> Inf**
## Line*: 6 :  1.00  1.36  1.66  1.87  1.99  2.13
## Line*: 8 :  1.00  1.30  1.53  1.68  1.77  1.86
## Line*: 5 :  1.00  1.00  1.00  1.00  1.01  1.01
##
## Lowest attainable execution time for the entire program when
##
##              lines are sped-up with factor (S):
##
##   Speed up factor (S) of a line
##              1      2      4      8      16
## All lines   266.3  133.1   66.6   33.3   16.6
## Line*: 6 :  266.3  195.5  160.1  142.4  133.6
## Line*: 8 :  266.3  204.7  173.9  158.4  150.7
## Line*: 5 :  266.3  265.6  265.2  265.0  264.9
##
##     Total sampling time:  266.28  seconds
##  *  Expected improvement at current scaling
##  ** Asymtotic max. improvement at current scaling
```

Step 1: need more detail?

```
L6<-targetedSummary(target = 6,NaiveBootAprof, findParent = TRUE)
L8<-targetedSummary(target = 8,NaiveBootAprof, findParent = TRUE)
head(L6)
```

```
##   Function Parent Calls   Time
## 1    rbind     L6  7077 141.54
## 2   lapply tapply  6636 132.72
## 3   tapply  rbind  6630 132.60
## 4       L8    FUN  6195 123.90
## 5      FUN lapply  6187 123.74
## 6     mean     L8  6096 121.92
```

```
head(L8)
```

```
##               Function               Parent Calls   Time
## 1                 mean                   L8  6063 121.26
## 2         mean.default                 mean  5478 109.56
## 3              warning         mean.default  5062 101.24
## 4 .signalSimpleWarning              warning  3153  63.06
## 5         withRestarts .signalSimpleWarning  3102  62.04
## 6            structure          makeRestart  1750  35.00
```

Potential solution 1

```
## OnlineRcode24.R
## Less naive
LessNaiveBoot<-function(x,R){
  avg<-mean(x$S)

  results<-array(dim=c(R,nlevels(x$site)))
        for(i in 1:R){
        index<-sample(seq_len(nrow(x)),replace=TRUE)
          results[i,]<-tapply(x$S[index],x$site[index],
          function(X) mean(X)-avg)
        }
        return(results)
}
```

Potential solution 2

```
## OnlineRcode31.R

## Even Less naive
fastBoot<-function(x,R){
  avg<-mean(x$S)

  results<-array(dim=c(R,nlevels(x$site)))
        for(i in 1:R){
        index<-sample(seq_len(nrow(x)),replace=TRUE)
          results[i,]<-tapply(x$S[index],x$site[index],
          function(X) mean.default(X)-avg)
        }
        return(results)
}
```

Workshop schedule

1. General introduction
2. Identifying whether and what to optimize
3. *Break out groups*