R code optimization and profiling

Alex Shlemov

a.shlemov@spbu.ru
eodus

Saint Petersburg State University, Russia
Faculty of Mathematics and Mechanics
Department of Statistical Modelling

March 29, 2015

Outline

Introduction

R is very good:

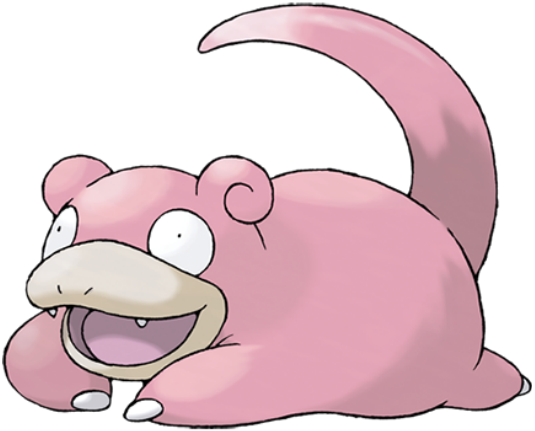
- High-level language with a lot of useful features
- Friendly, smooth learning curve
- Much cool out-of-box stuff
- Many packages
- Broad community
- Open and free
- Cross-platform (really cross-platform!)
- Easily extendable
- …

Introduction

R is very good:

- High-level language with a lot of useful features
- Friendly, smooth learning curve
- Much cool out-of-box stuff
- Many packages
- Broad community
- Open and free
- Cross-platform (really cross-platform!)
- Easily extendable
- …

…but sometimes it's very slow!



Example

C++ (via 'Rcpp'):

```
#include <Rcpp.h>
#include <cmath>

// [[Rcpp::export]]
double fcpp(const int n) {
  double s = 0.;
  for (int i = 1; i <= n; ++i)
    s += sin(i);

  return s;
}
```

```
fcpp(1e6)
```

```
## [1] -0.1171095
```

```
system.time(fcpp(1e6))
```

```
##    user  system elapsed
##   0.055   0.000   0.055
```

R (Revolution):

```
f <- function(n) {
  s <- 0
  for (i in 1:n)
    s <- s + sin(i)

  s
}
```

```
f(1e6)
```

```
## [1] -0.1171095
```

```
system.time(f(1e6))
```

```
##    user  system elapsed
##   0.521   0.004   0.525
```

‘microbenchmark’ timing package

`system.time()` works, but for fast calls it may be unprecise

`microbenchmark` from the package ‘microbenchmark’ is better alternative:

```
library(microbenchmark)

microbenchmark(f(1e6), fcpp(1e6), times = 25)

## Unit: milliseconds
##          expr       min        lq      mean    median        uq       max neval
##      f(1e+06) 518.58249 526.68754 548.77125 531.17251 551.81352 721.70462    25
##   fcpp(1e+06)  54.78589  54.90963  55.35176  55.00016  55.25229  60.18847    25
```

`benchmark()` from ‘rbenchmark’ is quite good too:

```
library(rbenchmark)
benchmark(f(1e6), fcpp(1e6), replications = 25)

##          test replications elapsed relative user.self sys.self user.child sys.child
## 1    f(1e+06)           25  13.252    9.659    13.238    0.012          0         0
## 2 fcpp(1e+06)           25   1.372    1.000     1.372    0.000          0         0
```

Motivation

Why is R code so slow?.. Main three causes:

- Often **R** code is poor written (nothing to add)
- **R** implementation is not optimized
- **R** is extremely *dynamic* language (all types are polymorphic, we can write to every environment in any time, etc)

See `http://adv-r.had.co.nz/Performance.html` for comprehensive explanation

How to speedup R code?

- Write better code following the **R** way
- Use optimized **R** implementations
- (If we really need) Reduce “dynamism” and/or use more low-level approaches (give up polymorphism, use more low-level functions or more low-level languages and so on)

Epigraph

"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

Outline

Outline

Vectorization

Loops are ugly and slow. Vectorized operations are elegant and fast

Let's try:

```
library(microbenchmark)
fvec <- function(n) {
  sum(sin(1:n))
}

microbenchmark(f(1e5), fvec(1e5), fcpp(1e5), times = 10)

## Unit: milliseconds
##         expr       min        lq      mean    median        uq       max neval
##      f(1e+05) 50.186745 51.183919 56.390626 52.899876 55.087063 88.405376    10
##   fvec(1e+05)  5.928014  5.942935  6.106567  6.019084  6.313066  6.349409    10
##   fcpp(1e+05)  5.478093  5.490812  5.509308  5.500083  5.511001  5.602193    10
```

Not bad!

Vectorized functions

Use:

- vectorized mathematical functions like `sin()`, `log()`, etc;
- `ifelse()` as "vectorized if";
- `colSums()` instead of `apply(X, 2, sum)`, `rowMeans()` instead of `apply(X, 1, mean)` and so on;
- `max.col(X)` rather than `apply(X, 1, which.max)`;
- vectorized subscripting, e.g. `x[is.na(x)] <- 0` or `x <- x[x %in% S]`;
- vectorized subscripting for matrices and arrays like `m[1:10, ]`, `m[cbind(1:10, 10:1)]` and `m[1:10]`

`match()` and `%in%` are also vectorized.

Be aware of some vectorized functions like `diff()` and `cumsum()`

Combine! Write your own vectorized functions

`*apply()` and their friends

`*apply()`, `vectorize()`, `replicate()`, `aggregate()`, etc don't provide any real vectorization!

They are less efficient than vectorized functions but usually much better than ordinary loops because of it's easy to parallelize them

Package '`foreach`' is good alternative for loops too (by the same reason)

Be aware of `tapply()` (and `by()`). `tapply()` is *grouping* apply (like SQL' `GROUP BY`):

```
h <- data.frame(sex = sample(c("male", "female"), size = 10, replace = TRUE),
                height = rnorm(10, mean = 170, sd = 10))
tapply(h$height, h$sex, mean)
```

```
##   female     male
## 165.8786 168.7190
```

`rapply()` is also sometimes useful

Memory preallocation

R vectors are not like C++ `std::vector`'s, **R** don't reserve any space for "appending". Each resizing invokes reallocation with $\mathcal{O}(l)$ complexity.
Thus, such code:

```
x <- c()
for (i in 1:n) {
  x <- c(x, f(i))
}
```

has $\mathcal{O}(n^2)$ time complexity! Avoid growing objects!
Always try to use vectorization. If it's impossible:

```
# x <- numeric(n) # NA-filling is better because of preventing some errors
x <- rep(NA_real_, n) # Preallocation, suppose that all results are numeric
for (i in seq_len(n)) { # seq_len is a bit faster
  x[i] <- f(i)
}
```

`*apply()` function family almost always provides better alternative, e.g.:

```
x <- sapply(1:n, f) # Shorter
x <- vapply(seq_len(n), f, f(1)) # Litte bit faster because of more proper preallocation
```

If you really need growing object…

…Use preallocation like in `std::vector`

Bad:

```
res <- c()
i <- 1
while(TRUE) {
  cur <- f(i)
  if (g(cur)) { # Some stop condition
    break
  } else {
    res <- c(res, cur)
  }
  i <- i + 1
}
```

Much better:

```
res <- c()
i <- 1
while(TRUE) {
  cur <- f(i)
  if (g(cur)) { # Some stop condition
    break
  } else {
    if (length(res) < i)
      res[2 * i] <- NA_real_
    res[i] <-  cur
  }
  i <- i + 1
}
res <- res[seq_len(i - 1)] # Truncate
```

Prefer in-place operations

In-place operations are often much faster:

```
squish_ife <- function(x, a, b) {
  ifelse(x <= a, a, ifelse(x >= b, b, x))
}
squish_p <- function(x, a, b) {
  pmax(pmin(x, b), a)
}
squish_in_place <- function(x, a, b) {
  x[x <= a] <- a
  x[x >= b] <- b
  x
}

x <- runif(100, -1.5, 1.5)
microbenchmark(squish_ife(x, -1, 1), squish_p(x, -1, 1), squish_in_place(x, -1, 1))

## Unit: microseconds
##                       expr    min     lq      mean  median       uq     max neval
##       squish_ife(x, -1, 1) 96.712 98.409 104.45655 99.6835 106.1625 156.126   100
##         squish_p(x, -1, 1) 28.579 31.392  36.16638 32.7905  34.3600 269.708   100
##  squish_in_place(x, -1, 1) 11.336 12.717  13.79473 13.2785  13.7605  33.630   100
```

Chunking for large structures

Remember our “sum of sines”. Let n be very big:

```
fvec <- function(n) {
  sum(sin(seq_len(n))) # <= Huge vector constructed here
}

f(1e10) # CRASH!

## Error: cannot allocate vector of size 74.5 Gb
```

Try to devectorize (chunk):

```
fchunk <- function(n, chunk = 1e7) {
  sum(vapply(seq(from = 1, by = chunk, length.out = ceiling(n / chunk)),
             function(i) sum(sin(seq(i, min(i + chunk - 1, n)))),
             0.))
}
fchunk(1e10)

## [1] -0.1276265
```

Use proper methods, functions, data structures and packages

Where are many **R** packages…
…and some of them are better than others

Find proper packages for your problem!

Try to use, e.g:

- ‘`data.table`’ and ‘`sqldf`’ for big dataframes;
- ‘`zoo`’, ‘`xts`’ for time series;
- ‘`mboost`’, ‘`flare`’ for linear regression;
- ‘`optimx`’ for optimization;
- ‘`cluster`’ for clustering;
- ‘`Matrix`’ for large sparse matrices;
- ‘`svd`’ for fast truncated SVD (used e.g. for PCA and related methods);
- ‘`FFTW`’ for Fourier transform (rather than slow `fft()`);
- …

Parallelize

K-means clustering with multistart (single-thread and parallel versions):

```
best.kmeans <- function(x, centers, N) {
cls <- lapply(seq_len(N),
              function(i) kmeans(x, centers = centers))
cls[[which.min(sapply(cls, function(x) x$tot.withinss))]]
}

best.kmeans.par <- function(x, centers, N, cluster) {
cls <- parLapplyLB(cluster,
                   seq_len(N),
                   function(i) kmeans(x, centers = centers))
cls[[which.min(sapply(cls, function(x) x$tot.withinss))]]
}
```

Package ‘`foreach`’ provides more friendly interface

Parallelize

```
library(parallel)
cores <- detectCores()
cores

## [1] 4

cluster <- makePSOCKcluster(cores)

USA.scaled <- scale(USArrests)

clusterExport(cluster, "USA.scaled")

microbenchmark(singe = best.kmeans(USA.scaled, 5, 1000),
               parallel = best.kmeans.par(USA.scaled, 5, 1000, cluster), times = 10)

## Unit: milliseconds
##      expr      min       lq     mean  median       uq      max neval
##     singe 487.1285 498.6937 512.6206 511.520 529.3923 536.3230    10
##  parallel 234.5809 237.4995 263.4459 242.605 288.8327 353.0925    10

stopCluster(cluster)
```

Outline

Byte-compilation and JIT

R is *interpreter.* Originally **R** considered a function as a raw text and parsed it on each call. Byte-compilation can reduce time costs for expression parsing:

```
f <- function(n) {
  s <- 0
  for (i in 1:n) s <- s + sin(i)

  s
}

library(compiler)
fcomp <- cmpfun(f)

microbenchmark(f(1e5), fcomp(1e5), times = 100)

## Unit: milliseconds
##           expr      min       lq     mean   median       uq      max neval
##       f(1e+05) 47.44344 48.63555 50.74597 48.86885 50.06414 97.74632   100
##   fcomp(1e+05) 25.86107 26.31081 27.87162 27.29993 27.91099 40.95275   100
```

Byte-compilation and JIT

JIT means *Just-In-Time* [compilation]. If JIT is enabled, all functions are automatically compiled before their first use

```r
enableJIT(3) # 3 means the most aggressive compilation
```

Pro:

- Something for nothing, easy to try, easy to give up
- Sometimes significantly speedups complicated code

Contra:

- Byte-compiled code cannot to be profiled
- Much speedup only for loops; almost no speedup for vectorized code
- Sometimes slows up code (a little bit, but…)

Alternative math libraries

`blas` and `LAPACK` are common used libraries for matrix operations and linear algebra

`ATLAS`, `OpenBLAS`, `Intel MKL`, `AMD ACML` provide more efficient implementation with the same interface

On my Intel I3 laptop:

```r
mx <- matrix(rnorm(1000^2), 1000, 1000)
```

```r
#with default BLAS and LAPACK
system.time(svd(mx))
##    user  system elapsed
##   4.734   0.016   4.751
```

```r
# with OpenBLAS and ATLAS LAPACK
system.time(svd(mx))
##    user  system elapsed
##   3.391   0.348   1.942
```

Alternative math libraries

Pro:

- Speedup for all linear algebra procedures ($\mathbf{A}x = b$, PCA, LM, GLM, …)
- Easy to install, easy to uninstall (on Linux)
- Speedups not only R code (also Python '`numpy`', GNU Octave, …)

Contra:

- Significant speedup only for large linear algebra problems
- Possible bugs and incompatibilities
- Non-trivial to install on Windows
- Intel MKL and AMD ACML are the most efficient but both are commercial

R Revoluton

*"Revolution **R** Enterprise is the fastest, most cost effective enterprise-class big data big analytics platform available today"*

Revolution R is optimized **R** distribution, which already includes Intel MKL

Even faster than OpenBLAS:

```
mx <- matrix(rnorm(1000^2), 1000, 1000)
```

```
#with Revolution (Intel MKL) BLAS and LAPACK
system.time(svd(mx))
##     user  system elapsed
##    2.425   0.012   1.365
```

```
# with OpenBLAS and ATLAS LAPACK
system.time(svd(mx))
##    user  system elapsed
##   3.391   0.348   1.942
```

R Revolution

Pro:

- Open edition is free
- Easy to install (anywhere). Just execute installer
- Already provides Intel MKL (for R only) for all supported platforms
- Some common **R** packages are optimized too

Contra:

- Commercial-based; less community support; possible copyright/license issues
- ~~Possible~~ bugs and incompatibilities (e.g. with ‘`svd`’, ‘`devtools`’)
- Some packages may have old versions

Outline

Get rid of S3/S4 method dispatching

S3/S4 method dispatching (polymorphism) is very slow (R isn't C++ or Java)

E.g., function `mean()` is generic (S3 polymorphic function):

```
methods(mean)

## [1] mean.Date     mean.default  mean.difftime mean.POSIXct  mean.POSIXlt

x <- runif(1e2)

microbenchmark(mean(x), mean.default(x))

## Unit: microseconds
##             expr   min     lq    mean median     uq    max neval
##          mean(x) 6.942 7.1705 8.32311 7.3060 7.6225 60.489   100
##  mean.default(x) 1.929 2.1195 2.33768 2.1925 2.3645 10.059   100
```

Thus, for S3 specify function directly, for S4 use for S4 use `findMethod()` to find the method and cache it into variable

Work on low-level

R is friendly and it's functions contains a lot of checks and coercions. If we exclude it, we can significantly speedup our code

```
findInterval
## function (x, vec, rightmost.closed = FALSE, all.inside = FALSE)
## {
## if (anyNA(vec))
## stop("'vec' contains NAs")
## if (is.unsorted(vec))
## stop("'vec' must be sorted non-decreasingly")
## .Internal(findInterval(as.double(vec), as.double(x), rightmost.closed,
## all.inside))
## }
```

We can just write:

```
findInterval.fast <- function (x, vec, rightmost.closed = FALSE, all.inside = FALSE)
{
    .Internal(findInterval(as.double(vec), as.double(x), rightmost.closed,
        all.inside))
}
```

Work on low-level

For `mean()`:

```
mean.default
## function (x, trim = 0, na.rm = FALSE, ...)
## {
##
## if (!is.numeric(x) && !is.complex(x) && !is.logical(x)) {
##   warning("argument is not numeric or logical: returning NA")
##   return(NA_real_)
## }
## ...
```

```
x <- runif(1e2)

microbenchmark(mean(x), mean.default(x), sum(x) / length(x), .Internal(mean(x)))

## Unit: nanoseconds
##                expr  min     lq     mean median      uq    max neval
##             mean(x) 7369 8495.5 12142.92   9157 10382.5 215629   100
##     mean.default(x) 1967 2614.5  3278.45   2930  3353.0  27465   100
##    sum(x)/length(x)  692  986.5  1895.38   1196  1742.5  37745   100
##  .Internal(mean(x))  420  459.5   614.92    524   623.0   5811   100
```

Call other code from **R**

Most simple:
You can combine R with other languages (using files, pipes, sockets, etc)

Most effective:
Also you can write your own package using Fortran, C or C++

Most comfortable:
use packages '`rJava`', '`rPython`' and especially '`Rcpp`'

Some advices

Avoid premature optimization ("root of all evil") and "optimization for optimization". Optimize only critical parts of code. Use profiler to find them

Optimized code may be erroneous. Verify optimized code by unit tests

Estimate actual speedup by timing. Don't use optimization which makes code much complicated but no significantly faster

Outline

Profiling

Code profiling is dynamic program analysis that measures space or time complexity of a program

Idea: Run program and evaluate time of each call

Functions and packages:

- `Rprof()`, `summaryRprof()` — standard, out-of-box
- `profr` from package '`profr`' (from CRAN)
- `lineprof()` from package '`lineprof`' (from Hadley's GitHub)

```
install.packages("devtools")
library(devtools)
install_github("hadley/lineprof")
library(lineprof)
```

Profiling example

```r
fchunk <- function(n, chunk = 1e7) {
  sum(vapply(seq(from = 1, by = chunk, length.out = ceiling(n / chunk)),
             function(i) sum(sin(seq(i, min(i + chunk - 1, n)))),
             0.))
}

Rprof(interval = 0.001)
fchunk(10^8)

## [1] 1.713649

Rprof(NULL)
head(summaryRprof()$by.self, 10)

##       self.time self.pct total.time total.pct
## "sin"     1.565    94.73      1.565     94.73
## ":"       0.050     3.03      0.050      3.03
## "sum"     0.037     2.24      0.037      2.24
```

Profiling example

```
head(summaryRprof()$by.total, 15)
```

```
##                         total.time total.pct self.time self.pct
## "<Anonymous>"                1.707    100.00         0        0
## "block_exec"                 1.707    100.00         0        0
## "call_block"                 1.707    100.00         0        0
## "evaluate"                   1.707    100.00         0        0
## "evaluate_call"              1.707    100.00         0        0
## "in_dir"                     1.707    100.00         0        0
## "process_file"               1.707    100.00         0        0
## "process_group"              1.707    100.00         0        0
## "process_group.block"        1.707    100.00         0        0
## "withCallingHandlers"        1.707    100.00         0        0
## "doTryCatch"                 1.706     99.94         0        0
## "eval"                       1.706     99.94         0        0
## "fchunk"                     1.706     99.94         0        0
## "FUN"                        1.706     99.94         0        0
## "handle"                     1.706     99.94         0        0
```

Profiling limitations

- Profiling does not extend to C code. You can see if your **R** code calls C/C++ code but not what functions are called inside of your C/C++ code
- Similarly, you can't see what's going on inside primitive functions or byte code compiled code
- If you're doing a lot of functional programming with anonymous functions, it can be hard to figure out exactly which function is being called. The easiest way to work around this is to name your functions
- Some calls are too fast to be traced by profiler
- Profiling may influence to code performance

Outline

Conclusions

- Use high-level optimization, avoid low-level one
- Avoid premature optimization, use profiler to find bottlenecks
- 'Rcpp' is good tool for low-level optimization
- Use timing for estimation of obtained speedup
- Use unit test for verification of optimized code
- Some **R** distributions are faster (but may be incompatible)

For further studying

Out-of-view subjects:

- '`Rcpp`', '`rPython`', '`rJava`'
- Writing own packages
- Memory tracing and profiling
- Profiling of compiled code
- Alternative **R** implementations ('`pqR`', '`Renjin`', '`FastR`', '`Riposte`')

Useful packages:

- '`sqldf`' and '`data.table`' for big dataframes
- '`testthat`' for unit tests
- '`foreach`' for effective loops paralleling

Useful links:

- Advanced **R** by Hadley Wickham
- The **R** Inferno by Patrick Burns
- **R** Packages by Hadley Wickham
- Writing **R** Extensions

Thanks for your attention!

Thanks for your attention!