Efficient R Programming

Lecture 16

Dirk Eddelbuettel

STAT 430: Data Science Programming Methods (Fall 2019)
Department of Statistics, University of Illinois

Gillespie & Lovelace, "Efficient R Programming", O'Reilly, 2016

- Chapter 3: Efficient Programming
- Chapter 5: Efficient I/O
- Chapter 6: Efficient Data Carpentry
- Chapter 7: Efficient Optimization

Very handy book, we are not going to cover each and every section of the chapters we select. But an excellent introduction and overview.

Efficient Programming

Chapter 3: ‘Five key tips’

1. Be careful to never grow vectors
2. Vectorise whenever possible
3. Use `factor` when appropriate
4. Avoid unneccessary computation by caching
5. Byte-compile package for easy performance boost

Be careful to never grow vectors

- Very important: When in a loop, do not do this:

  `x <- c(x, new_result)`
- New copies are being made on each iteration
- Copy gets more expensive as `x` grows
- Rather: allocate `n` elements of `x` as start
- assign: `x[i] <- new_result`

Consider two possible implementation of a task done in a loop.

For illustration, the task itself is simple.

```r
f1 <- function(N) {
    # empty
    res <- c()
    for (i in seq_len(N)) {
        res <- c(res, 1/log10(i))
    }
    res
}
```

```r
f2 <- function(N) {
    # preallocated
    res <- vector(length=N)
    for (i in seq_len(N)) {
        res[i] <- 1/log10(i)
    }
    res
}
```

Benchmarking:

```
##        test replications elapsed relative
## 1 f1(100)          100   0.007     1.75
## 2 f2(100)          100   0.004     1.00

##        test replications elapsed relative
## 1 f1(1000)          100   0.148     5.92
## 2 f2(1000)          100   0.025     1.00

##         test replications elapsed relative
## 1 f1(10000)          100   7.003   32.572
## 2 f2(10000)          100   0.215    1.000
```

Lessons

- Growing objects is expensive
- Growing objects gets more expensive as N
- Each newly created object requires full copy of all previous ones
- Better: *Pre-allocate* and insert
- Our simple example showed a 32-fold gain

Vectorise code whenever possible

- This can avoid loops, one of the slowest parts of R
- Vectorised operations often go directly to compiled code
- Examples can be simple yet way more efficient:
- Don't: `s <- 0; for (i in 1:n) s <- s + x[i]`
- Do: `s <- sum(x)`
- But: *don't obsess over it.*
- Correctness comes first.

Benchmarking with

```r
f3 <- function(N) { 1 / log10( seq(1, N) ) }
```

```
##         test replications elapsed relative
## 1 f1(1000)          100   0.114       57
## 2 f2(1000)          100   0.054       27
## 3 f3(1000)          100   0.002        1
```

```
##          test replications elapsed relative
## 1 f1(10000)          100   7.231  482.067
## 2 f2(10000)          100   0.524   34.933
## 3 f3(10000)          100   0.015    1.000
```

Lessons

- Vectorising is very powerful as loops have overhead
- Gains from vectorising likely dominates gains from better loops
- Key to speed in vectorising: very few calls, very little overhead
- Our simple example showed a 380-fold gain (!!)
- Again: don't obsess over it: Correctness first

Use `factor` variables when appropriate

- It can make code simpler and clearer
- Which in turn may make it easier to
    - understand and reason with
    - maintain and modify
- May make it more efficient too

Avoid unnecessary computation by caching

- Fastest way to compute something is to … not compute it!
- Sometimes we can cache and store explicitly
- The `memoise` package can make this semi-automatic

Example

Consider a function doing something complicated taking one second:

```r
slowFun <- function(x) {
    Sys.sleep(1)  # imagine real work done here
    x^2
}
system.time(replicate(5, slowFun(1)))
```

```
##    user  system elapsed
##   0.001   0.000   5.005
```

The aggregate takes five seconds, give or take.

Example with `memoise`

```r
slowFun <- function(x) { Sys.sleep(1); x^2 }
library(memoise)
memoisedFun <- memoise(slowFun)
system.time(replicate(5, memoisedFun(1)))
```

```
##    user  system elapsed
##   0.039   0.000   1.040
```

Now it takes one second.

memoise

- `memoise()` hashes the function call *and arguments*
- on subsequent call, arguments are checked and …
    - if 'inventory of answers' has one for these arguments
    - it is returned immediately rather than being recomputed
- checking argument and lookup creates small overhead but …
    - if function is 'expensive enough' it pays off

ILLINOIS

Digression: Fibonacci with memoise

```
## memoization solution courtesy of Pat Burns
mfibR <- local({
    memo <- c(1, 1, rep(NA, 1000))
    f <- function(x) {
        if (x == 0) return(0)
        if (x < 0) return(NA)
        if (x > length(memo))
            stop("'x' too big for implementation")
        if (!is.na(memo[x])) return(memo[x])
        ans <- f(x-2) + f(x-1)
        memo[x] <<- ans
        ans
    }
})
```

Uses a function generator (as `local({...})` returns a function) with encapsulated environment to hold the **memo** hash

This is rather advanced R use (and outside of the scope of the course) but you may enjoy benchmarking it.

Due to Pat Burns; see Section 1.2.6 of Rcpp book (Eddelbuettel, 2013).

Byte compile packages for an easy performance boost

- This is now an automatically turned-on feature in R
    - Every package installation non byte-compiles
- We can still use the `compiler` package on non-package code
- It offers ‘just in time compilation’

```r
f2 <- function(N) {
    res <- vector(length=N)
    for (i in seq_len(N)) res[i] <- 1/log10(i)
    res
}
f2cmp <- compiler::cmpfun(f2)
rbenchmark::benchmark(f2(1000), f2cmp(1000))[,1:4]
```

```
##           test replications elapsed relative
## 1     f2(1000)          100   0.053    2.524
## 2 f2cmp(1000)          100   0.021    1.000
```

So no real gain here, and generally not worth it as R has improved.

Chapter 3: Benchmarking

- The hint about `microbenchmark` (or `rbenchmark`) is good
- Consider the examples about computing a sum
- We construct two functions, and a vector
- We submit both to `microbenchmark()` which runs them
- By default 100 runs, summarizes in appropriate time unit
- Reports min, 1st quartile, mean, median, 3rd quartile and max
- We also use `rbenchmark` which stresses *relative* comparison

Chapter 3: Benchmarking with microbenchmark

```r
library(microbenchmark)
loopfun <- function(x) {
    s <- 0
    for (i in 1:length(x)) s <- s + x[i]
    s
}
sumfun <- function(x) { sum(x) }
X <- seq(1,100000) # hundred thousand
microbenchmark(loop=loopfun(X), sum=sumfun(X))
```

```
## Unit: nanoseconds
##   expr     min      lq        mean  median      uq      max neval cld
##   loop 2998549 3056174 3271757.97 3082599 3125312 19527839   100   b
##    sum     288     300     639.88     364     420    18245   100  a
```

Chapter 3: Benchmarking with `rbenchmark`

```
library(rbenchmark)
loopfun <- function(x) {
    s <- 0
    for (i in 1:length(x)) s <- s + x[i]
    s
}
sumfun <- function(x) { sum(x) }
X <- seq(1,1000000) # one million elements
benchmark(loop=loopfun(X), sum=sumfun(X))[,1:4]
```

```
##   test replications elapsed relative
## 1 loop          100  19.511       NA
## 2  sum          100   0.000       NA
```

Possible Chapter 3 Extension: Benchmarking with **bench**

```
library(bench)
loopfun <- function(x) {
    s <- 0; for (i in 1:length(x)) s <- s + x[i]; s
}
sumfun <- function(x) { sum(x) }
X <- seq(1,1000000) # one million elements
mark(loop=loopfun(X), sum=sumfun(X))
```

```
## # A tibble: 2 x 6
##   expression      min   median `itr/sec` mem_alloc `gc/sec`
##   <bch:expr> <bch:tm> <bch:tm>     <dbl> <bch:byt>    <dbl>
## 1 loop         30.3ms   30.6ms      32.4        0B        0
## 2 sum           316ns  342.8ns 2443895.         0B        0
```

Chapter 3: Benchmarking

- Benchmarking can be a complicated topic
- There are issues about the right tasksize
- And whether task are representative of the real world tasks
- The previous example was clearly extreme
- But benchmarking is *very* useful. So experiment!
- (And no, we don't often see 3000:1 comparisons.)

Efficient Input/Output

Overview

- Reading and writing files can also be made performant
- Generally speaking *binary* files will be more efficient
- … while text files are more portable
- The internal R format `.rds` is fairly very efficient
    - and compressed by default with tuneable settings
- Highly recommended for *repeated* access to large files
- (The internal `.RData` or `.rda` format is less convenient.)

`data.table` and `fread` – and also `fwrite`

- These have gotten even better since the book was written
- Use multithreading to read / write multiple chunks at once
- Fastest for text files and generally recommended
- Alternatives:
    - `readr` for text files
    - `rio` for general purpose reading utilties

ILLINOIS

Binary formats

- The R format `.rds` is good but more difficult to share
- The `feather` package provides Python and R access
- The `arrow` package extends this for Parquet files
- The `fst` package provides the fastest read/write access
- See the chapter by Gillespie & Lovelace, it contains good tips
- (But the `data.table` function `fwrite` is now much better; is was not yet parallelised when the book was written.)

`rds` files

- Uses an efficient base R function; serializes any R data structure
- The easiest way to make read/write more efficient are
    - `saveRDS(x, file=filename)` to write `x` to `fname`
    - `x <- readRDS(filename)` to read it
- Portable between R installations and OSs
- Not as easily portable to other languages
    - there is librdata for C-level access

`fst` files

- `fst` is a more recent package by Mark Klik
- The fastest way to read and write binary files
- An aggressively optmized parallel version of the `rds` functions
- Worth exploring if you need to read/write a lot
- Very cleverly used by the `disk.frame` package for larger-than-memory analysis

Binary versus text

Text files are

- portable: anything can read `csv`
- human-readable: easier to check
- may be of lower precision: generally fewer than 16 decimals
- will generally be slower as text needs to be converted to binary
- may create larger files (but may compress well)

Rough summary: “easier” versus “better”

Binary files are

- not always portable across systems (though there are portable solutions)
- generally of higher read-write speed
- generally of higher precision

Efficient Data Carpentry

Overview of Chapter 6

- Tidy data with `tidyr` (or `data.table` or ...)
- Process data with `dplyr`
- Working with databases
- Data processing with `data.table`

Used below:

```r
library(tibble)
library(tidyr)
library(stringr)
library(readr)
library(dplyr)
library(data.table)
```

This brings in some tidyverse packages – and we generally applaud its aim for more unified and consistent interfaces.

However, these also makes distinctions between "packages for interactive work" and "packages for writing code" which to us departs from the very R philosophy of "turning users into programmers" using one consistent approach.

But you should use what *you* are comfortable with.

For me, this often means packages like `data.table` along with a few selected tidyverse applications such as `ggplot2`.

Five Tips From Chapter 6 (slightly edited)

1. Time spent preparing your data can save hours in the long run.
2. Tidy data provides a concept for organizing data, package `tidyr` provides some functions for this work.
3. The `data_frame` class defined by the `tibble` package makes datasets efficient to print and easy to work with.
4. `dplyr` provides [...] intuitive data processing functions; `data.table` has unmatched speed for some [...] applications.
5. The `%>%` pipe operator can clarify complex data processing workflows.

Tibbles

```r
library("tibble")
tibble(x = 1:3,
       y = c("A", "B", "C"))
```

```
## # A tibble: 3 x 2
##       x y
##   <int> <chr>
## 1     1 A
## 2     2 B
## 3     3 C
```

Note that:

- Dimensions and types are printed
- Character vector *not* coerced to `factor` type
- Printing limits to ten rows …
- as many columns as currently fit

Long and wide examples

```
#remotes::install_github("csgillespie/efficient")
library("efficient")
data(pew)         # see ?pew - data from efficient package
pew[1:3, 1:4]     # we suppressing many more columns here
```

```
## # A tibble: 3 x 4
##   religion `<$10k` `$10--20k` `$20--30k`
##   <chr>      <int>      <int>      <int>
## 1 Agnostic      27         34         60
## 2 Atheist       12         27         37
## 3 Buddhist      27         21         30
```

Long and wide examples: Full pew data set once

```
spew <- pew; spew[,1] <- substr(pew[,1,drop=TRUE],1,5); spew   # shortened column 1 for  display
```

```
## # A tibble: 18 x 10
##    religion `<$10k` `$10--20k` `$20--30k` `$30--40k` `$40--50k` `$50--75k` `$75--100k` `$100--150k` `>150k`
##    <chr>      <int>      <int>      <int>      <int>      <int>      <int>       <int>        <int>   <int>
##  1 Agnos         27         34         60         81         76        137         122          109      84
##  2 Athei         12         27         37         52         35         70          73           59      74
##  3 Buddh         27         21         30         34         33         58          62           39      53
##  4 Catho        418        617        732        670        638       1116         949          792     633
##  5 Don't         15         14         15         11         10         35          21           17      18
##  6 Evang        575        869       1064        982        881       1486         949          723     414
##  7 Hindu          1          9          7          9         11         34          47           48      54
##  8 Histo        228        244        236        238        197        223         131           81      78
##  9 Jehov         20         27         24         24         21         30          15           11       6
## 10 Jewis         19         19         25         25         30         95          69           87     151
## 11 Mainl        289        495        619        655        651       1107         939          753     634
## 12 Mormo         29         40         48         51         56        112          85           49      42
## 13 Musli          6          7          9         10          9         23          16            8       6
## 14 Ortho         13         17         23         32         32         47          38           42      46
## 15 Other          9          7         11         13         13         14          18           14      12
## 16 Other         20         33         40         46         49         63          46           40      41
## 17 Other          5          2          3          4          2          7           3            4       4
## 18 Unaff        217        299        374        365        341        528         407          321     258
```

Tidy data characteristics (Wickham, 2014, JSS)

- Each variable forms a column
- Each observation forms a row
- Each type of obversational unit forms a table

We should add that it has been pointed out that this owes a lot to Codd (1979) and the relational data model popularized by SQL.

Creating long data: gather()

```
library(tidyr)
# dim(pew)     ## from 18 x 10
pewt <- gather(data=pew, key=Income, value=Count, -religion)
# dim(pewt)    ## to 162 x 3
pewt[c(1:2, 50), ]  # rows 1 to 2 and 50
```

```
## # A tibble: 3 x 3
##   religion Income   Count
##   <chr>    <chr>    <int>
## 1 Agnostic <$10k       27
## 2 Atheist  <$10k       12
## 3 Orthodox $20--30k    23
```

Creating wide data: spread

```
pewtb <- spread(data=pewt,
                key=Income,
                value=Count)


pewtb[1:4, 1:5]
```

```
## # A tibble: 4 x 5
##   religion `<$10k` `>150k` `$10--20k` `$100--150k`
##   <chr>      <int>   <int>      <int>        <int>
## 1 Agnostic      27      84         34          109
## 2 Atheist       12      74         27           59
## 3 Buddhist      27      53         21           39
## 4 Catholic     418     633        617          792
```

Creating long data with **data.table::melt()**

```
suppressMessages(library(data.table))     # suppr. one-liner
pewdt <- data.table(pew)                  # create data.table
pewlong <- melt(pewdt, id="religion")     # very simple call
pewlong[,religion:=substr(religion,1,12)] # shorten col1
pewlong[c(1:3, 51:53), ]                  # select six rows
```

```
##          religion variable value
## 1:      Agnostic     <$10k    27
## 2:       Atheist     <$10k    12
## 3:      Buddhist     <$10k    27
## 4: Other Christ $20--30k    11
## 5: Other Faiths $20--30k    40
## 6: Other World  $20--30k     3
```

Creating wide data with `data.table::dcast()`

```
pewwide <- dcast(pewlong, religion ~ variable)
pewwide[,religion := substr(religion,1,12)] # shorten col1
pewwide[1:12, 1:8]                          # subset for display
```

```
##          religion <$10k $10--20k $20--30k $30--40k $40--50k $50--75k $75--100k
##  1:      Agnostic    27       34       60       81       76      137       122
##  2:       Atheist    12       27       37       52       35       70        73
##  3:      Buddhist    27       21       30       34       33       58        62
##  4:      Catholic   418      617      732      670      638     1116       949
##  5: Don’t know/r    15       14       15       11       10       35        21
##  6: Evangelical    575      869     1064      982      881     1486       949
##  7:         Hindu     1        9        7        9       11       34        47
##  8: Historically   228      244      236      238      197      223       131
##  9: Jehovah's Wi    20       27       24       24       21       30        15
## 10:        Jewish    19       19       25       25       30       95        69
## 11: Mainline Pro   289      495      619      655      651     1107       939
## 12:        Mormon    29       40       48       51       56      112        85
```

Comparison

tidyr

```
pewlong <- gather(data=pew, key=Income,
                  value=Count, -religion)
pewwide <- spread(data=pewlong, key=Income,
                  value=Count)

## and now
pewlong <- pivot_longer(pew,
                        -religion,
                        names_to = "income",
                        values_to = "count")
pewwide <- pivot_wider(pewllong,
                       names_from=income,
                       values_from=count)
```

data.table

```
## make sure sure object is a data.table
pewlong <- melt(pewdt, id="religion")
pewwide <- dcast(pewlong,
                 religion ~ variable)
```

Comparison: tidyr

```
pewlong <- gather(data=pew, key=Income, value=Count,-religion)
pewwide <- spread(data=pewlong, key=Income, value=Count)
```

Comparison: data.table

```
pewlong <- melt(pew, id="religion")
pewwide <- dcast(pewlong, religion ~ variable)
```

More functionality

- Efficient R Programming discusses `separate()`, string functions and regular expression
- We will not go there now
- One word of caution: the tidyverse can change a lot
  - original paper on package `reshape` with `plyr` examples
  - replaced by packages `reshape2` and `dplyr`
  - current favourite `tidyr` already needing an internal update
  - `spread` & `gather` have just been replaced by `pivot_wider` & `pivot_longer`

Data processing with `dplyr`

- `dplyr` is pretty fast due to an efficient C++ backend
- *i.e.* faster than `plyr` (yet still slower than `data.table`)
- `dplyr` is widely used and described in many tutorials
- core idea of `dplyr` is use of set of “verbs” to transform data
- `dplyr` works well with the “pipe” from `magrittr`
- `dplyr` has database integration

dplyr verbs

- `filter()` and `slice()` to subset rows
- `arrange()` to (re-)order data by variable
- `select()` to subset columns
- `rename()` to rename columns
- `distinct()` to return unique tows
- `mutate()` to create new variables
- `summarize()` to collapse into single row
- `sample_n()` to extract a sample

Efficient Optimization

Top Five Tips (from the book)

1. Before you start to optimize you code, ensure that you know where the bottleneck lies; use a code profiler.
2. If the data in your data frame is all of the same type, consider converting it to a matrix for a speed boost.
3. Use specialized row and column functions whenever possible.
4. The `parallel` package is ideal for Monte Carlo simulations.
5. For optimal performance, consider rewriting key parts of your code in C++. (*Note:* We should get there later in the course.)

Profiling

- R has a built-in facility called `Rprof()` to profile
- Several add-on package facilitate analysing its output.
- Easiest: `profviz`, particularly from RStudio

First profiling example

```
library(profvis)
profvis({
  data(movies, package = "ggplot2movies") # Load data
  movies = movies[movies$Comedy == 1,]
  plot(movies$year, movies$rating)
  # loess regression line
  model = loess(rating ~ year, data = movies)
  j = order(movies$year)
  # Add line to the plot
  lines(movies$year[j], model$fitted[j])
})
```

Second profiling example

```
## ensure package is installed with source and
## without byte compilation so that profviz can
## analyse; force to allow reinstallation
remotes::install_github("csgillespie/efficient",
                        args="--with-keep.source --no-byte-compile",
                        force=TRUE) # if already installed
library(efficient)
profvis(simulate_monopoly(10000))
```

Optimisation consideration

- `ifelse()` versus explicit `if(...) else` (when arg of length 1)
- Sorting and ordering: `radix` option to `sort()` a good suggestion
- `which.min()` + `which.max()` faster than `which(x == min(x))`
- factors to numerics: `as.numeric(levels(f))[f]`
- Row and column operations:
    - `apply(data, 1, f)` sweeps `f()` over rows of `data`
    - similarly, `apply(data, 2, f)` goes across columns
    - `rowSums()`, `colSums()`, `rowMeans()`, `colMeans()`
- Matrix subsetting generally faster than `data.frame` so if all columns are of the same type, converting to matrix can speed up a lot too

But: Most important is still to get the *correct* result, not the fastest!

Optimising `move_square`

- Several small changes:
  - `matrix(sample(1:6, 6, replace=TRUE), ncol=2)` uses an integer matrix
  - `rowSums()` instead of `apply()`
  - Single-element boolean `&&`
- Account for a ~ 25x speedup (!!)
- Remainder of chapter discusses
  - `parallel` (which we have seen) and
  - `Rcpp` (which we may get to at the end of the term)

Optimising `move_square`

```r
move_square <- function(current) {
    rolls <- matrix(sample(1:6, 6, replace=TRUE), ncol=2)
    Total <- rowSums(rolls)
    isDouble <- rolls[,1] == rolls[,2]
    if (isDouble[1] && isDouble[2] && isDouble[3]) {
        current <- 11
    } else if (isDouble[1] && isDouble[2]) {
        current <- current + sum(Total[1:3])
    } else if (isDouble[1]) {
        current <- current + sum(Total[1:2])
    } else {
        current <- Total[1]
    }
    current
}
```

Optimising move_square

```
R> rbenchmark::benchmark(new=move_square(37),
+                        old=efficient::move_square(37),
+                        replications=10000)[,1:5]
  test replications elapsed relative user.self
1  new        10000   0.092    1.000     0.092
2  old        10000   2.308   25.087     2.308
R>
```

Efficient Programming

- Keep a few key ideas in mind
- Measure! Benchmarking gives you data

Efficient I/O

- Know which input/output tools exists
- Know which tradeoffs exist

Efficient Data Carpentry

- Recap of *long* versus *wide*
- Comparing `tidyr` and `data.table` & recap of `dplyr` functions

Efficient Optimization

- Introduction to profiling
- Two examples