Efficient Coding in R

Adam Kaplan / Tomoya Sasaki

April 01, 2022

Efficiency in `R`

- Why is some code “fast” and other code “slow” in `R`?
- Two main culprits:
    - Bad algorithms,
    - Inefficient memory usage.
- Does not suffice to use and understand `apply()` family of functions.
- $\Longrightarrow$ Need to understand basics of computational complexity and memory allocation.

Roadmap

- Basic of computational complexity
- Memory allocation
- Memory prescriptions
- Vectorization: `apply` and `purrr`
- Parallelization: `future` and `foreach`
- Random number generator

Computational Complexity

Basics of Computational Complexity

- A theoretically rigorous way to measure “speed” of an algorithm as a function of input size.
- Put simply: How “fast” can $f(x_1, \ldots, x_n)$ be evaluated as a function of n.
- Before we go any deeper, what do we mean by speed?
    - How many inputs x_i does f need to go over.
- Why can’t we just measure time from start to end?
    - Many other factors impact the speed with which your code runs; harder to pin down issues.
- What can you do?
    1. Find more appropriate algorithms,
    2. Vectorize whenever possible (why? stay tuned),
    3. If possible, parallelize.

Computational Complexity Examples

```r
# Block 1: Simple for loop
for (i in 1:n) {
  foo()
}

# Block 2: Sequential for loops
for (i in 1:n) {
  foo()
}
for (i in 1:n) {
  bar()
}
```

```r
# Block 3: Nested for-loops
for (i in 1:n) {
  foo()
  for (i in 1:n) {
    bar()
  }
}

# Block 4: Simple bootstrap
for (i in 1:b) {
  # Hint: mle with BFGS takes O(n^2)
  estimate <- mle(X_1, ..., X_n, method = BFGS)
}
```

Memory Allocation in R

Memory Allocation in R

- Now we are algorithm experts and will measure the complexity of every piece of code.
- Is this enough? Unfortunately no.
- Even if the algorithms are efficient, we can lose real-life performance due to inefficient memory usage.
- Will go over basics of memory (in R), data types, and vectorization to motivate efficiency prescriptions.

What is memory?

- Computer consists of three main parts: CPU core(s) (different circuits), Memory (RAM, Hard drive), GPU (can ignore for this session)
- CPU is wicked fast.
- Many different types of memory: Volatile (RAM, Caches), Non-volatile (Hard drive, USB, CDs, . . .).
- Every R session lives on RAM memory.
- RAM is actually very very fast for memory standards, but painstakingly slow compared to CPUs, particularly saving new things to it (writes).
- To write efficient code, we **need to avoid unnecessary memory writes!**

Memory Usage in R

- What happens when I delete an object with `rm()`?
  - You tell R that this reference to an object is no longer needed.
  - If there are no more, it will be "garbage collected" *automatically*.
- **Big culprit:** Copying of objects.
  - R uses a "copy if modified" framework so if `y <- x`, and `y[1] <- 0`, `y` will no longer point to `x`, but will allocate a new chunk of memory and point to that.
  - $\implies$ Avoid implicit and explicit copying of objects whenever possible.
  - $\implies$ "Vectorizing" is mostly faster because it avoids this implicit copying.

Implicit copying of objects

```r
# Example from: http://adv-r.had.co.nz/memory.html
library(pryr)
x <- data.frame(matrix(runif(100 * 1e4), ncol = 100))
medians <- vapply(x, median, numeric(1))

for (i in seq_along(medians)) {
  x[, i] <- x[, i] - medians[i]
  print(c(address(x), refs(x)))
}
```

Explicit copying objects

- Define the size and type of object to avoid copying object.
- Example: creating a vector of ones.

```r
f1 <- function() { # adding ones one by one
  x <- 1.0
  for (i in 2:10000) {
    x <- c(x, 1.0)
  }
  return(x)
}
f2 <- function() { # define length + type of vector
  x <- numeric(10000) # can be character(), interger() etc
  for (i in 1:10000) {
    x[i] <- 1.0
  }
  return(x)
}
f3 <- function() { # vectorization
  x <- rep(1.0, 10000)
  return(x)
}
```

```r
bench::mark( # sum two columns for each row
  f1(), f2(), f3()
) %>% select(expression, median)
```

```
## # A tibble: 3 x 2
##   expression   median
##   <bch:expr> <bch:tm>
## 1 f1()        195.4ms
## 2 f2()        527.3us
## 3 f3()         17.7us
```

Functions in R

- From last time, arguments are not copied until modified, yay!
- What if we return a function in a function?
- **Turns out:** Functions save their surrounding environment!
- So what? Imagine you create a large variable (`x <- 1:1e+50`) before you create your function, then your function will also copy the large variable into memory.
- Now on to more applied examples of why data types matter and how to vectorize in R.

Memory Prescriptions

Memory Prescription 1: Data Types Matter

- List vs. Matrix vs. Dataframe (or Tibble)
  - Matrix algebra is heavily optimized!
  - Neither lists nor matrices have memory overhead.
  - Dataframes have many specialized functions such as grouping that are much faster than DIY approaches on lists or matrices.

```r
x <- runif(10000)
y <- rnorm(10000)
DF <- data.frame(x = x, y = y) # dataframe
TIB <- tibble::as_tibble(DF) # tidyverse tibble
MAT <- cbind(x, y) # matrix
```

```r
bench::mark( # sum two columns for each row
  apply(DF, 2, sum),
  apply(TIB, 2, sum),
  apply(MAT, 2, sum),
  colSums(MAT) # Implemented in C
) %>% select(expression, median)
```

```
## # A tibble: 4 x 2
##   expression             median
##   <bch:expr>           <bch:tm>
## 1 apply(DF, 2, sum)     334.5us
## 2 apply(TIB, 2, sum)    322.5us
## 3 apply(MAT, 2, sum)      193us
## 4 colSums(MAT)           30.3us
```

Memory Prescription 2: Vectorize

- "Vectorize" your codes as much as you can.
- Most R functions allow you to vectorize by default.

Vectorization: apply and purrr

Vectorization: apply and purrr

- apply, lapply, sapply, ... come built-in with R and allow you to apply *any* function in a vectorized way to a matrix/dataframe, list, or vector.
- purrr's map, map_dbl, map_dfc, ... are the tidyverse equivalents and extensions to the apply() family.

```r
library(tidyverse)
d <- list(
  data.frame(quant = c("danny", "insong", "teppei")),
  data.frame(schools = c("MIT", "Harvard"))
)
# We want a list as an output
lapply(d, function(x) nrow(x))

# default map returns a list
map(d, function(x) nrow(x))

# The tidy version
d %>% map(~ nrow(.x))
```

```
library(tidyverse)
v <- 1:10

# We want a list as an output
sapply(v, function(x) x * x)

# The tidy version
v %>% map_dbl(~ .x * .x)
```

Vectorization using purrr

- `map_*` return different object type and will fail if it is not appropriate.
- Makes the code more predictable, and thus easier to debug.
- Of particular interest are `map_dfr` and `map_dfc`, which run a function on the input and then row/column bind the outputs together.
- Inspired popular parallelization package `furrr` which we will introduce you to shortly.

Parallelize: future and foreach

Why and when to parallelize

- Enable multiple computations to take place at the same time
- Useful when you have time-consuming, unordered tasks
  - Data cleaning
  - Bootstrap
  - Monte Carlo simulations
  - Any tasks with lots of loops, apply, or maps

Common issues in parallel computing

- Often need two different codes for parallel and non-parallel computing
  - Increase potential bugs
- Methods for parallel computing differ across operating systems and different packages implements different methods
  - `mclapply`. `parallel`, `doParallel` etc
  - Parallelization with `mclapply` (forking) does not work in Windows
  - Your choice of package would decide how to run in parallel

future package

- future provides a simple and uniform tool for async. parallel, and distributed processing in R
- Same coding style between sequential and parallel tasks
- Users decide how to parallelize: the code does not depend on how to run in parallel
    - Use the same code for parallel computing in different operating systems

```
f <- future(expr) ## evaluate in parallel
r <- resolved(f) ## check if done
v <- value(f) ## get result
```

Same coding style

```r
f3 <- function() {
  tmp <- rnorm(10000)
  for (i in 1:10000) {
    tmp <- tmp + i
  }
  sum(tmp)
}

library(future)
future_obj <- future(f3()) ## evaluate in parallel
```

Choose how to parallelize

- Sequential: `plan(sequential)`
- Parallelize on local machine: `plan(multisession)`
- “Fork” your task (not fore Windows): `plan(multicore)`
- Multiple local or remote computers `plan(cluster)`

Combine together

```
## multisession using two cores: default setup uses the max #
## print with future::availableCores()
plan(multisession, workers = 2)

future_obj <- future(f3()) ## evaluate in parallel
output2 <- value(future_obj) ## get result

output <- f3()
all.equal(output, output2)
```

```
## [1] TRUE
```

```
plan() ## check current plan
```

```
## multisession:
## - args: function (..., workers = 2, envir = parent.frame()
```

User friendly functions

- Add `future_` in front of the function
- Evaluate, check, get result within one function

```
## apply / future.apply
lapply(x, your_fun)

plan(multisession, workers = 2)
future_lapply(x, your_fun)
```

- furrr package: integrate with purrr

```r
## purrr / furrr
x %>% map(your_fun)

library(furrr)
plan(sequential)
x %>% future_map(your_fun)
```

foreach for parallel computing

- A parallel/distributed computing framework for the R language
    - A tool for “what to parallelize”
- Enables computation across multiple CPU cores and computers
- `foreach` runs sequentially in the absence of a parallel adapter
- `doFuture` from `future` framework provides a useful adapter for `foreach`

Basic structure

```
library(foreach)
## by default return a list
## where each element is an output from each iteration
res  <- foreach (i = 1:3) %do% {
  i + 1
}
res
```

```
## [[1]]
## [1] 2
##
## [[2]]
## [1] 3
##
## [[3]]
## [1] 4
```

Basic structure 2

```
## you can iterate over two variables with the same length
res <- foreach(i = 1:3, j = 11:13) %do% {
  i + j
}
res
```

```
## [[1]]
## [1] 12
##
## [[2]]
## [1] 14
##
## [[3]]
## [1] 16
```

Basic structure 3

```r
## the output can be a vector
res <- foreach(i = 1:3, j = 11:13, .combine = "c") %do% {
  i + j
}
res
```

```
## [1] 12 14 16
```

```r
## the output can be pretty flexible
res <- foreach(i = 1:3, j = 11:13, .combine = "+") %do% {
  i + j
}
res
```

```
## [1] 42
```

Parallelization with foreach and future

```r
library(doFuture)
registerDoFuture() ## register cores
plan(multisession, workers = 2) ## multisession with 2 cores

## compute mean while trimming values less than 0.1
## dopar automatically detect parallel adapter
cutoff <- 0.1
y <- foreach(x = mtcars, .combine = "c") %dopar% {
  mean(x, trim = cutoff)
}
head(y)
```

```
## [1]  19.696154   6.230769 222.523077 141.192308   3.579231
```

Some notes

```
library(tidyverse)
registerDoFuture() ## register cores
plan(multisession, workers = 2) ## multisession with 2 cores

## call functions explictly
y <- foreach(x = mtcars, .packages = c("tidyverse")) %dopar% {
  data %>% filter(x > 0)
}
```

xvii and foreach

- Your computer usually does not have enough cores for parallel computing
    - # of your cores = # of processors you can run simultaneously
- Use xvii, which has more than 20 cores
- The above codes work with xvii
- Do not use too many cores

Random number generators

Random number generation with Futures

- **Motivating example:** Monte Carlo simulations.
- To ensure replicability we would usually set a random seed using `set.seed(02135)` before running the simulations.
- Unfortunately, when we parallelize, since each process is independent of each other, it is very likely that the random state *across* processes is no longer pseudo-random. In fact, it will likely be correlated.
- Even the naive solution of setting a different seed for each iteration does not guarantee the desired level of randomness.
- Thankfully, there are random number generation algorithms specifically designed for parallel computing. The default for `future` and most commonly used one is *L'Ecuyer CMRG algorithm*.
- Using it is as easy as in a sequential program.

Random number generation in future.apply

```r
library(future.apply)
# Set the seed
set.seed(02135)
# Specify that you want to use random numbers
out1 <- future_sapply(1:10, function(x) runif(1),
  future.seed = TRUE
)
```

Random number generation in furrr

```r
library(furrr)
# Set the seed
set.seed(02135)
# Specify that you want to use random numbers
out2 <- future_map_dbl(1:10, ~ runif(1),
  .options = furrr_options(seed = TRUE)
)
# Check that they are the same
all(out1 == out2)
```

```
## [1] TRUE
```

Conclusion

Key Takeaways

- Choose your data types appropriately.
- Use canned functions whenever possible and otherwise try to avoid nested for loops if possible.
- Vectorize whenever you can.
- Be on the lookout for explicit or implicit copying of objects.

Thank you for listening!