**WRITING EFFICIENT R CODE**

Version – prints the r version you are using

Benchmarking: Finding the fastest code that provides the solution to a problem.

system.time() – function that returns the time it took to process the R expression in the brackets

Reading in RDS files is much faster than reading in csv files

Using system.time() is convenient, but it does have its drawbacks when comparing multiple function calls. The **[microbenchmark](https://www.rdocumentation.org/packages/microbenchmark" \t "_blank)** package solves this problem with the **[microbenchmark()](https://www.rdocumentation.org/packages/microbenchmark/versions/1.4-2.1/topics/microbenchmark" \t "_blank)** function.

library(microbenchmark)

# Compare the two functions

compare <- microbenchmark(read.csv("movies.csv"),

readRDS("movies.rds"),

times = 10)

compare

Measuring the performance of your computer is called benchmarking, and you can do that with the **[benchmarkme](https://www.rdocumentation.org/packages/benchmarkme" \t "_blank)** package.

get\_ram() / get\_cpu() – how much ram or cpu the computer has

The **[benchmarkme](https://www.rdocumentation.org/packages/benchmarkme" \t "_blank)** package allows you to run a set of standardized benchmarks and compare your results to other users. One set of benchmarks tests is reading and writing speeds.

The function call

res = benchmark\_io(runs = 1, size = 5)

records the length of time it takes to read and write a 5MB file.

If we minimize **memory allocation** (e.g. variable assignment) we gain speed

Define a function that grows a vector constantly with numbers form a standard normal distribution with mean 0 and std 1. (never grow a vector, it is slow).

n <- 30000

# Slow code

growing <- function(n) {

x <- NULL

for(i in 1:n)

x <- c(x, rnorm(1))

x

}

How long does it take for 30000 numbers?

system.time(res.grow <- growing(30000))

preallocate values with a function without growing the vector afterwards.

n <- 30000

# Fast code

pre\_allocate <- function(n) {

x <- numeric(n) # Pre-allocate

for(i in 1:n)

x[i] <- rnorm(1)

x

}

n <- 30000 # test again how long it takes.

system.time(res\_allocate <- pre\_allocate(n))

Vetorized code is fast. (C or fortan language) 🡪 use a vectorized version whenever possible instead of slow for loops.

Dataframes are the key datastructure in R. Columns are the variables and rows different entries.

Difference in Dataframe and Matrix: A Matrix can have only one singular data type 🡪 Using matrices can provide a massive speed boost because variable selection is much faster. Use a matrix whenever possible.

# Which is faster, mat[, 1] or df[, 1]?

microbenchmark(mat[,1], df[,1])

# Which is faster, mat[1, ] or df[1, ]?

microbenchmark(mat[1, ], df[1, ])

How to find the speed bottleneck of a code: Code profiling.

The function [**profvis()**](https://www.rdocumentation.org/packages/profvis/topics/profvis) from the same named package can give an idea of the speed of each line of code: z.B.

data(movies, package = "ggplot2movies")

# Load the profvis package

library(profvis)

# Profile the following code with the profvis function

profvis({

# Load and select data

comedies <- movies[movies$Comedy == 1, ]

# Plot data of interest

plot(comedies$year, comedies$rating)

# Loess regression line

model <- loess(rating ~ year, data = comedies)

j <- order(comedies$year)

# Add fitted line to the plot

lines(comedies$year[j], model$fitted[j], col = "red")

}) ## Remember the closing brackets!

* Once a bottleneck is identified you can try to improve the function. Eg the dice throwing from the monopoly example:

# Load the microbenchmark package

library(microbenchmark)

# The previous data frame solution is defined

# d() Simulates 6 dices rolls

d <- function() {

data.frame(

d1 = sample(1:6, 3, replace = TRUE),

d2 = sample(1:6, 3, replace = TRUE)

)}

# Complete the matrix solution

m <- function() {

matrix(sample(1:6, 6, replace = TRUE), ncol = 2)

}

# Use microbenchmark to time m() and d()

microbenchmark(

data.frame\_solution = d(),

matrix\_solution = m()

)

Computers have multiple CPU (cores to calculate) but R usually uses only one -> can be improved by using parallel computing

The [**parallel**](https://www.rdocumentation.org/packages/parallel) package has a function **[detectCores()](https://www.rdocumentation.org/packages/parallel/topics/detectCores" \t "_blank)** that determines the number of cores in a machine.

> library(parallel)

> no\_of\_cores <- detectCores()

> no\_of\_cores

Example: Calculate multiple Monte Carlo simulations -> one calculation of the for loop per core

Only works for loops that can be run forward and backwards.

U can switch to parallel computations within your script (However, communication between cores also takes time 🡪 compare speed of single and multiple code calculations).

See how many cores you have, create a cluster with the maximum (minus the number of other stuff you are doing on your computer), parallelize your function (eg. parApply instead of apply) to be calculated faster, close clustrer (parallel computing) again:

# Determine the number of available cores

detectCores()

# Create a cluster via makeCluster

cl <- makeCluster(2)

# Parallelize this code

parApply(cl, dd, 2, median)

# Stop the cluster

stopCluster(cl)

Or with the sapply function: Applying a function to each element of a vector. Substitute with parSapply 🡪 load package, make a cluster, switch to parSapply(), Stop

library("parallel")

# Create a cluster via makeCluster (2 cores)

cl <- makeCluster(2)

# Export the play() function to the cluster

clusterExport(cl, "play")

# Re-write sapply as parSapply

res <- parSapply(cl, 1:100, function(i) play())

# Stop the cluster

stopCluster(cl)

Now see if new version is faster with system time or microbenchmark

# Set the number of games to play

no\_of\_games <- 1e5

## Time serial version

system.time(serial <- sapply(1:no\_of\_games, function(i) play()))

## Set up cluster

cl <- makeCluster(4)

clusterExport(cl, "play")

## Time parallel version

system.time(par <- parSapply(cl, 1:no\_of\_games, function(i) play()))

## Stop cluster

stopCluster(cl)