

Efficient R (update)

Scientific workflows: Tools and Tips 

Dr. Selina Baldauf

2025-06-19

What is this lecture series?

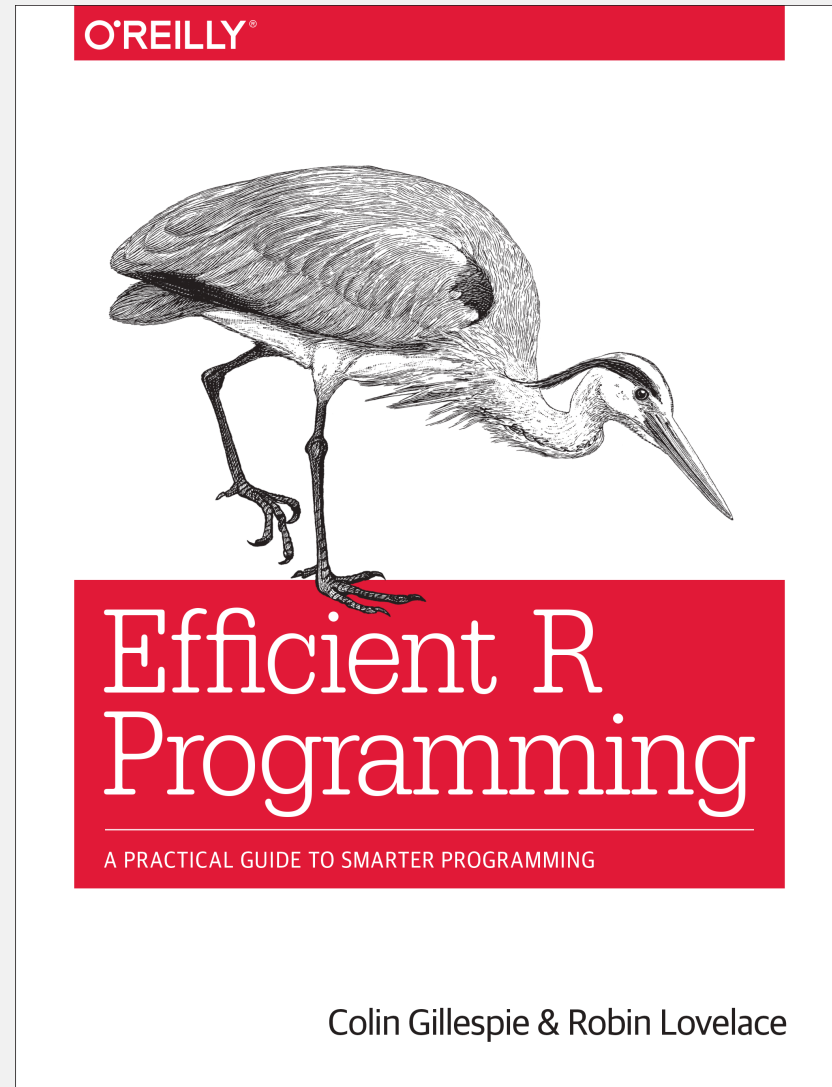
Scientific workflows: Tools and Tips

 Every 3rd Thursday  4-5 p.m.  Webex

- One topic from the world of scientific workflows
- Material provided [online](#)
- If you don't want to miss a lecture
 - [Subscribe to the mailing list](#)

Main reference

Efficient R book by Gillespie and Lovelace, read it [here](#)



What is efficiency?

$$\text{efficiency} = \frac{\text{work done}}{\text{unit of effort}}$$

Computational efficiency



Computation time



Memory usage

Programmer efficiency

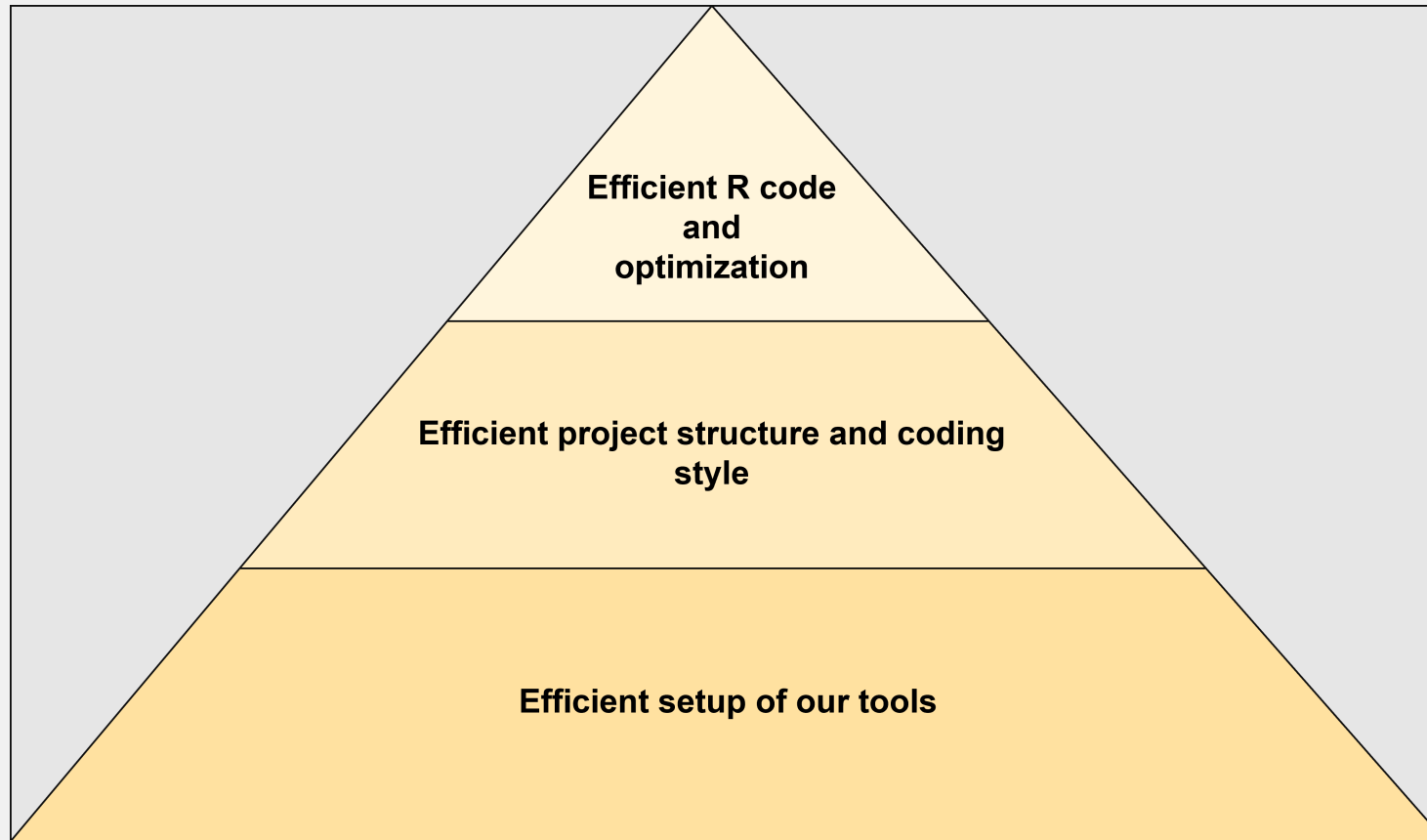


How long does it take to

- *write* code?
- *maintain* code?
- *read* and *understand* the code?

Tradeoffs and **Synergies** between these types of efficiencies

Today



Principles and **tools** to make R programming more efficient for the 

Check out my talk [“Write R code that lasts”](#) for basics

Is R slow?

- R is slow compared to other programming languages (e.g. C++, Julia).
 - R is designed to make statistical programming & data analysis **easy** and **interactive**, not fast
- R is not the most memory efficient language
- But: **R is fast and memory efficient enough** for most tasks.

Should I optimize?

It's easy to get caught up in trying to remove all bottlenecks. Don't! **Your time is valuable** and is **better spent analysing your data**, not eliminating possible inefficiencies in your code. **Be pragmatic**: don't spend hours of your time to save seconds of computer time.
(Hadley Wickham in [Advanced R](#))

Think about

- How much time do I **save** vs. **spend** with optimizing?
- **How often** do I run the code?
- Trade-offs between **readability** and **efficiency**

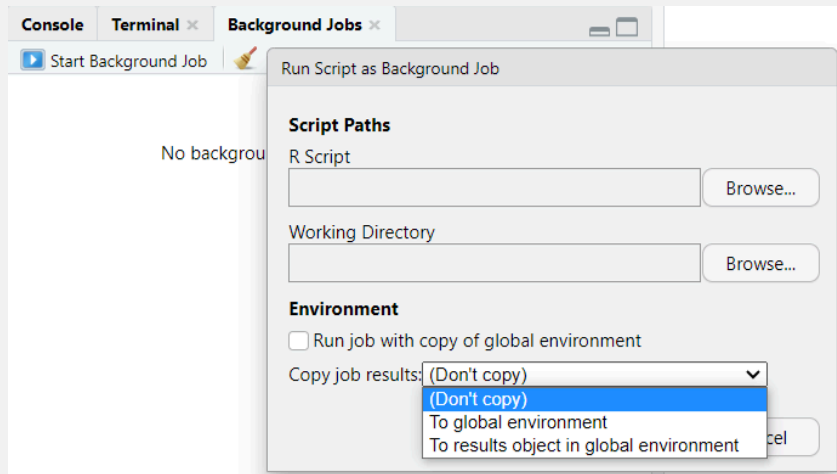
Should I optimize?

If your code is too slow for you, you can go through these steps:

1. Think if you can **run the code somewhere else**

Run the code somewhere else

- Use RStudio background jobs



- Start your R script from the command line

```
Rscript my_script.R
```

- Run it on a cluster (e.g. [FU Curta](#))

Should I optimize?

If your code is too slow for you, you can go through these steps:

1. Think if you can **run the code somewhere else**
2. **Identify the critical (slow) parts** of your code
3. Then **optimize only the bottlenecks**

Profiling and benchmarking

Measure the speed and memory usage of your code

Profiling R code

What are the speed & memory bottlenecks in my code?

- Use the `profvis` package

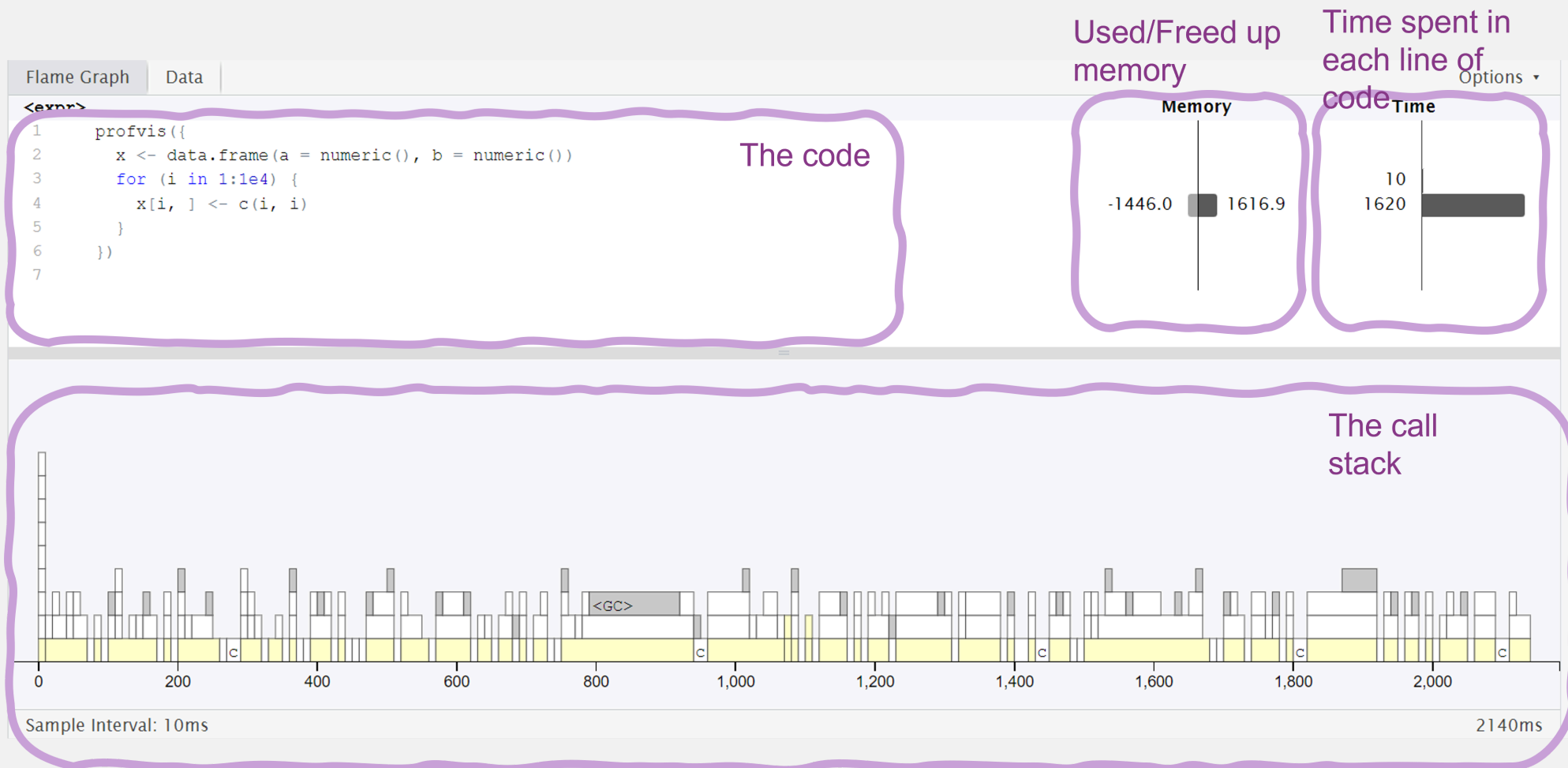
Profiling R code

You can profile a section of code like this:

```
1 # install.packages("profvis")
2 library(profvis)
3
4 # Create a data frame with 150 columns and 400000 rows
5 df <- data.frame(matrix(rnorm(150 * 400000), nrow = 400000))
6
7 profvis({
8   # Calculate mean of each column and put it in a vector
9   means <- apply(df, 2, mean)
10
11   # Subtract mean from each value in the table
12   for (i in seq_along(means)) {
13     df[, i] <- df[, i] - means[i]
14   }
15 })
```














Profiling R code

Profvis flame graph shows time and memory spent in each line of code.



Profiling R code

Profvis data view for details on time spent in each function in the call stack.

Code	File	Memory (MB)	Time (ms)
▶ [<code><-</code>].data.frame	<expr>	-1446.0  1547.2	1600 
c		-445.2  355.7	360 
any		-69.9  83.5	70 
length		-78.5  34.4	40
[<-	<expr>	0  34.9	20
▶ compiler:::tryCompile	<expr>	0  0	10
attr		-44.4  0	10
as.character		0  16.7	10
dim		0  17.7	10
all		0  10.9	10

Profiling R code

You can also interactively profile code in RStudio:

- Go to **Profile -> Start profiling**
- Now interactively run the code you want to profile
- Go to **Profile -> Stop profiling** to see the results

Benchmarking R code

Which version of the code is faster?

```
# Fill a data frame in a loop
f1 <- function() {
  x <- data.frame(a = numeric(), b = numeric())
  for (i in 1:1e4) {
    x[i, ] <- c(i, i)
  }
}

# Fill a data frame directly with vectors
f2 <- function() {
  x <- data.frame(a = 1:1e4, b = 1:1e4)
}
```

Benchmarking R code - the easy way

Use the `tictoc` package to get a quick overview of the time a section of code takes

```
1 # install.packages("tictoc")
2 library(tictoc)
3
4 tic()
5 f1()
6 toc()
7 #> 1.25 sec elapsed
8
9 tic()
10 f2()
11 toc()
12 #> 0 sec elapsed
```

Benchmarking R code

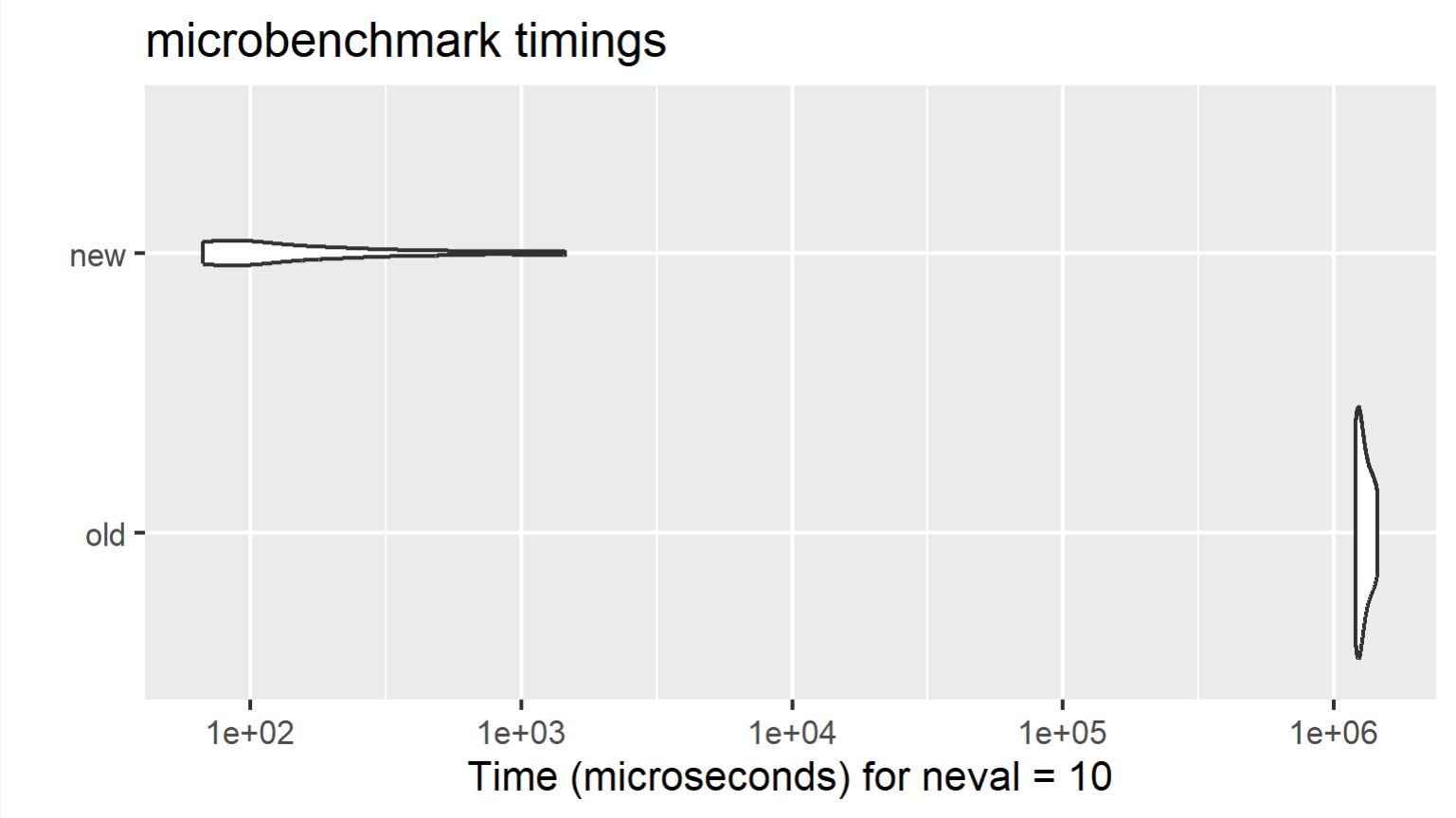
Use the `microbenchmark` package to compare the functions:

```
1 # install.packages("microbenchmark")
2 library(microbenchmark)
3
4 compare_functions <- microbenchmark(
5   old = f1(),
6   new = f2(),
7   times = 10 # default is 100
8 )
9
10 compare_functions
11 #> Unit: microseconds
12 #>   expr      min       lq      mean     median        uq      max  neval  cld
13 #>   old 1204274.4 1224502.9 1289691.2 1265086.30 1353718.5 1453146.0    10    a
14 #>   new    66.9     80.6     280.9     96.85     204.8     1446.8    10    b
```

We can look at benchmarking results using `ggplot`

```
library(ggplot2)
autoplot(compare_functions)
```

Benchmarking R code



Optimize your code

- Basic principles
- Data analysis bottlenecks
- Advanced optimization: Parallelization and C++

Basic principles

Vectorize your code

- Vectors are central to R programming
- R is optimized for vectorized code
 - Implemented directly in C/Fortran
- Vector operations can often replace for-loops in R
- If there is a vectorized version of a function: Use it

Vectorize your code

Example: Calculate the log of every value in a vector and sum up the result

```
1 # A vector with 1 million values
2 x <- 1:1e6
3
4 microbenchmark(
5   for_loop = {
6     log_sum <- 0
7     for (i in 1:length(x)) {
8       log_sum <- log_sum + log(x[i])
9     }
10  },
11  sum = sum(log(x)),
12  times = 10
13 )
14 #> Unit: milliseconds
15 #>      expr      min       lq      mean  median      uq      max  neval  cld
16 #> for_loop 39.2334 39.5070 40.35139 40.3330 41.0836 41.6231    10    a
17 #>      sum 19.9408 20.2146 21.28390 21.6083 22.0062 22.8566    10    b
```

For-loops in R

- For-loops are **relatively slow** and it is easy to make them even slower with bad design
- Often they are used when vectorized code would be better
- For loops can often be replaced, e.g. by
 - Functions from the apply family (e.g. `apply`, `lapply`, ...)
 - Vectorized functions (e.g. `sum`, `colMeans`, ...)
 - Vectorized functions from the `purrr` package (e.g. `map`)

But: For loops are not necessarily bad, **sometimes** they are the **best solution** and **more readable** than vectorized code.

Cache variables

If you use a value multiple times, store it in a variable to avoid re-calculation

Example: Calculate column means and normalize them by the standard deviation

```
1 # A matrix with 1000 columns
2 x <- matrix(rnorm(10000), ncol = 1000)
3
4 microbenchmark(
5   no_cache = apply(x, 2, function(i) mean(i) / sd(x)),
6   cache = {
7     sd_x <- sd(x)
8     apply(x, 2, function(i) mean(i) / sd_x)
9   }
10 )
11 #> Unit: milliseconds
12 #>      expr      min       lq      mean   median      uq      max  neval  cld
13 #> no_cache 38.3473 42.2739 45.597475 43.43775 45.20405 158.0655   100    a
14 #>   cache  3.1371  3.2574  3.858989  3.51440  4.20380   7.4205   100    b
```

Efficient data analysis

Efficient workflow

- Prepare the data to be clean and concise for analysis
 - Helps to avoid unnecessary calculations
- Save intermediate results
 - Don't re-run time-consuming steps if not necessary
- Use the right packages and functions

Read data

Example: Read csv data on worldwide emissions of greenhouse gases (~14000 rows, 7 cols).

- Base-R functions to read csv files are:
 - `read.table`
 - `read.csv`
- There are many alternatives to read data, e.g.:
 - `read_csv` from the `readr` package (tidyverse)
 - `fread` from the `data.table` package
 - `read_csv_arrow` from the `arrow` package

Read data

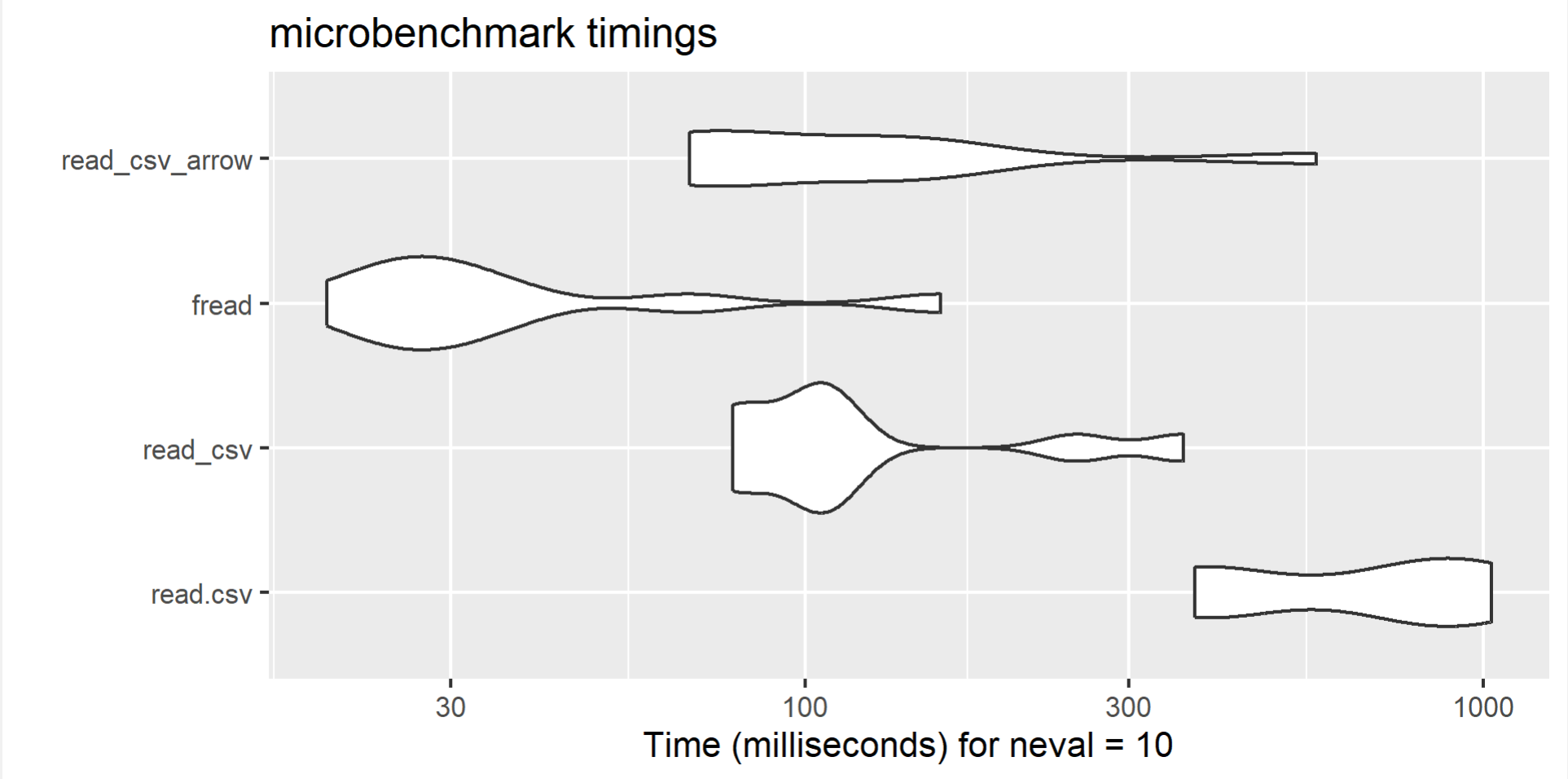
Compare some alternative reading functions

```
file_path_csv <- here::here("slides/data/ghg_ems_large.csv")

compare_input <- microbenchmark::microbenchmark(
  read_csv = read.csv(file_path_csv),
  read_csv = readr::read_csv(file_path_csv, progress = FALSE, show_col_types = FALSE),
  fread = data.table::fread(file_path_csv, showProgress = FALSE),
  read_csv_arrow = arrow::read_csv_arrow(file_path_csv),
  times = 10
)

autoplot(compare_input)
```

Read data



Use plain text data

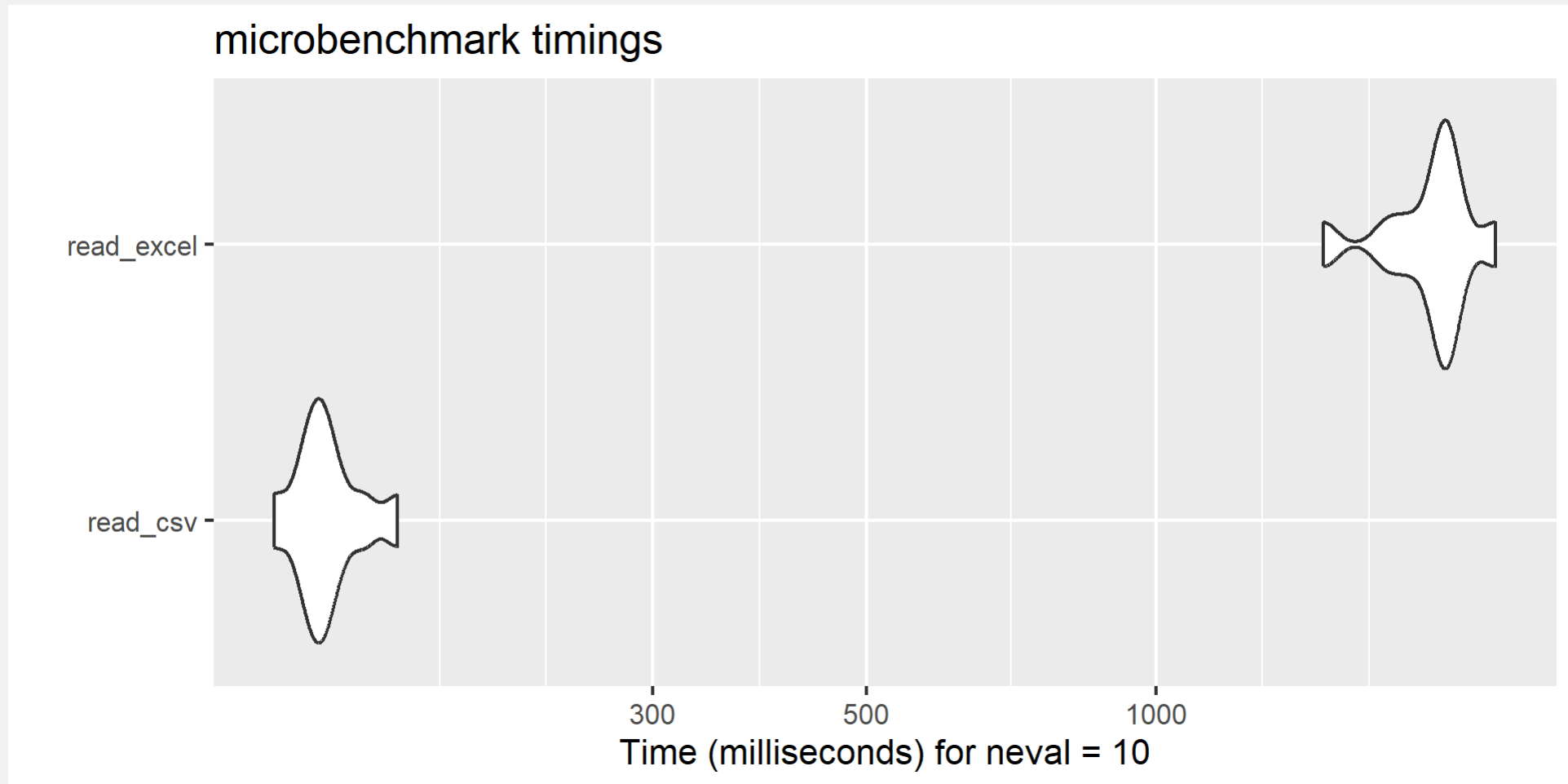
Reading plain text is faster than excel files

```
file_path_xlsx <- here::here("slides/data/ghg_ems_large.xlsx")

compare_excel <- microbenchmark(
  read_csv = readr::read_csv(file_path_csv),
  read_excel = readxl::read_excel(file_path_xlsx),
  times = 10
)

autoplot(compare_excel)
```

Use plain text data



Write data

- Base-R functions to write csv files are:
 - `write.table`
 - `write.csv`
- Faster alternatives are
 - `write_csv` from the `readr` package (tidyverse)
 - `fwrite` from the `data.table` package
 - `write_csv_arrow` from the `arrow` package

Write data

Efficient data manipulation

Different packages offer fast and efficient data manipulation and analysis:

- `dplyr` package has a C++ backend and is often faster than base R
- `data.table` package is fast and memory efficiency
 - Syntax is quite different from base R and `tidyverse`
- `collapse` package is a C++ based and specifically developed for fast data analysis
 - Works together with both `tidyverse` and `data.table` workflows
 - Many functions similar to base R or `dplyr` just with prefix “f” (e.g. `fselect`, `fmean`, ...)
- `arrow` package for efficient reading, processing and writing of large datasets (even larger than RAM)

Summarize data by group

Example: Summarize mean carbon emissions from Electricity by Country

```
library(data.table)
library(dplyr)
library(collapse)
```

Summarize data by group

Example: Summarize mean carbon emissions from Electricity by Country

```
1 # 1. The data table way
2 # Convert the data to a data.table
3 setDT(ghg_ems)
4 summarize_dt <- function() {
5   ghg_ems[, mean(Electricity, na.rm = TRUE), by = Country]
6 }
7
8 # 2. The dplyr way
9 summarize_dplyr <- function() {
10  ghg_ems |>
11    group_by(Country) |>
12    summarize(mean_e = mean(Electricity, na.rm = TRUE))
13 }
14
15 # 3. The collapse way
16 summarize_collapse <- function() {
17  ghg_ems |>
18    fgroup_by(Country) |>
19    fsummarise(mean_e = fmean(Electricity))
20 }
```

Efficient data manipulation

Example: Summarize mean carbon emissions from Electricity by Country

```
# compare the speed of all versions
microbenchmark(
  dplyr = summarize_dplyr(),
  data_table = summarize_dt(),
  collapse = summarize_collapse(),
  times = 10
)
#> Unit: microseconds
#>      expr    min      lq   mean  median     uq    max  neval  cld
#>  dplyr 4853.2 5205.3 8037.60 7323.50 9931.3 13163.8    10    a
#> data_table  925.2 1026.5 2933.56 1648.55 4993.2  8628.7    10    b
#>  collapse  123.0  178.0  607.66  222.80  348.7  2670.0    10    b
```

Select columns

Example: Select columns Country, Year, Electricity, Transportation

```
1  microbenchmark(  
2    dplyr = select(ghg_ems, Country, Year, Electricity, Transportation),  
3    data_table = ghg_ems[, .(Country, Electricity, Transportation)],  
4    collapse = fselect(ghg_ems, Country, Electricity, Transportation),  
5    times = 10  
6  )  
7  #> Unit: microseconds  
8  #>      expr      min      lq     mean  median      uq     max  neval  cld  
9  #>      dplyr 1068.6 1164.6 2101.62 1728.55 2833.6 4012.4    10    a  
10 #>  data_table  436.2  469.3 1034.74 1192.65 1495.8 1572.6    10    b  
11 #>   collapse    5.6    8.8   38.45   14.75   37.1  203.5    10    c
```

Advanced optimization

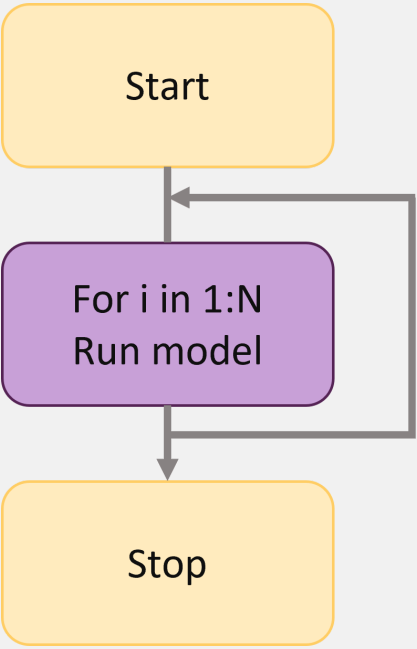
Parallelization and C++

Parallelization

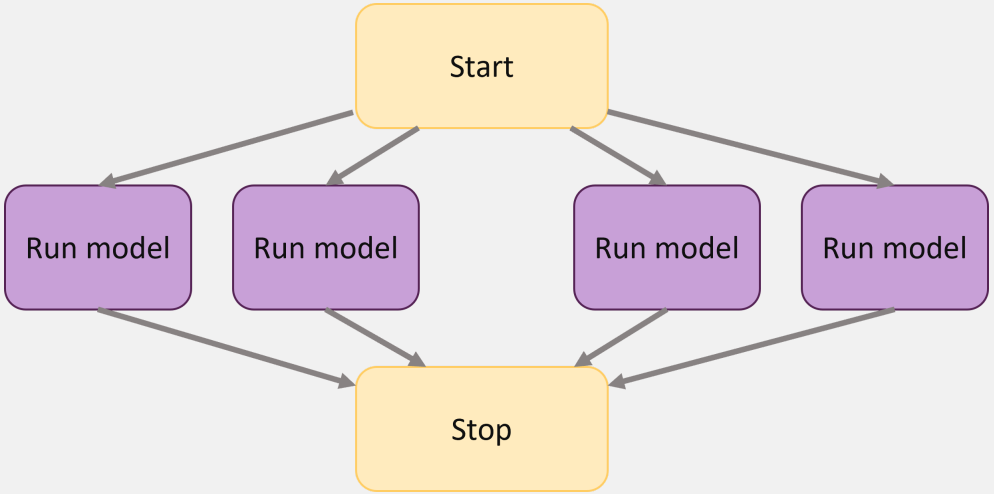
By default, R works on one core but CPUs have multiple cores

```
# Find out how many cores you have with the parallel package
# install.packages("parallel")
parallel::detectCores()
#> [1] 14
```

Sequential



Parallel



Parallelization with the futureverse

- `future` is a framework to help you parallelize existing R code
 - Parallel versions of base R apply family
 - Parallel versions of `purrr` functions
 - Parallel versions of `foreach` loops
- Find more details [here](#)
- Find a tutorial for different use cases [here](#)

A slow example

Let's create a very slow square root function

```
slow_sqrt <- function(x) {  
  Sys.sleep(1) # simulate 1 second of computation time  
  sqrt(x)  
}
```

Before you run anything in parallel, tell R how many cores to use:

```
# Load future package  
library(future)  
# Plan parallel session with 6 cores  
plan(multisession, workers = 6)
```

Parallel apply functions

To run the function on a vector of numbers we could use

Sequential `lapply`

```
# create a vector of 10 numbers
x <- 1:10
tic()
result <- lapply(x, slow_sqrt)
toc()
#> 10.18 sec elapsed
```

Parallel `future_lapply`

```
# Load future.apply package
library(future.apply)

tic()
result <- future_lapply(x, slow_sqrt)
toc()
#> 2.6 sec elapsed
```

Use `parallel::detectCores()` to find out how many cores you have.

Parallel apply functions

Selected base R apply functions and their future versions:

base	future.apply
<code>lapply</code>	<code>future_lapply</code>
<code>sapply</code>	<code>future_sapply</code>
<code>vapply</code>	<code>future_vapply</code>
<code>mapply</code>	<code>future_mapply</code>
<code>tapply</code>	<code>future_tapply</code>
<code>apply</code>	<code>future_apply</code>
<code>Map</code>	<code>future_Map</code>

Parallel for loops

A normal for loop:

```
z <- list()
for (i in 1:10) {
  z[i] <- slow_sqrt(i)
}
```

Use `foreach` to write the same loop

```
library(foreach)
z <- foreach(i = 1:10) %do%
{
  slow_sqrt(i)
}
```

Parallel for loops

Use `doFuture` and `foreach` package to parallelize for loops

The **sequential** version

```
library(foreach)

tic()
z <- foreach(i = 1:10) %do%
  {
    slow_sqrt(i)
  }
toc()
#> 10.2 sec elapsed
```

The **parallel** version

```
library(doFuture)

tic()
z <- foreach(i = 1:10) %dofuture%
  {
    slow_sqrt(i)
  }
toc()
#> 2.39 sec elapsed
```

Future purrr functions

The `furrr` package offers parallel versions of `purrr` functions

The **sequential** version

```
library(purrr)

# the purrr version
tic()
z <- map(x, slow_sqrt)
toc()
#> 10.19 sec elapsed
```

The **parallel** version

```
library(furrr)

# the furrr version
tic()
z <- future_map(x, slow_sqrt)
toc()
#> 2.45 sec elapsed
```

Close multisession

When you are done working in parallel, explicitly close your multisession:

```
# close the multisession plan  
plan(sequential)
```

Replace slow code with C++

- Use the [Rcpp package](#) to re-write R functions in C++
- [Rcpp](#) is also used internally by many R packages to make them faster
- Requirements:
 - C++ compiler installed
 - Some knowledge of C++
- See [this book chapter](#) and the [online documentation](#) for more info

Rewrite a function in C++

Example: R function to calculate Fibonacci numbers

```
# A function to calculate Fibonacci numbers
fibonacci_r <- function(n) {
  if (n < 2) {
    return(n)
  } else {
    return(fibonacci_r(n - 1) + fibonacci_r(n - 2))
  }
}
```

```
# Calculate the 30th Fibonacci number
fibonacci_r(30)
#> [1] 832040
```

Rewrite a function in C++

Use `cppFunction` to rewrite the function in C++:

```
library(Rcpp)

# Rewrite the fibonacci_r function in C++
fibonacci_cpp <- cppFunction(
  'int fibonacci_cpp(int n){
    if (n < 2){
      return(n);
    } else {
      return(fibonacci_cpp(n - 1) + fibonacci_cpp(n - 2));
    }
  }'
)
```

```
# calculate the 30th Fibonacci number
fibonacci_cpp(30)
#> [1] 832040
```

Rewrite a function in C++

You can also source C++ functions from C++ scripts.

C++ script `fibonacci.cpp`:

```
#include "Rcpp.h"

// [[Rcpp::export]]
int fibonacci_cpp(const int x) {
  if (x < 2) return(x);
  return (fibonacci(x - 1)) + fibonacci(x - 2);
}
```

Then source the function in your R script using `sourceCpp`:

```
sourceCpp("fibonacci.cpp")




# Use the function in your R script like you are used to
fibonacci_cpp(30)
```

How much faster is C++?

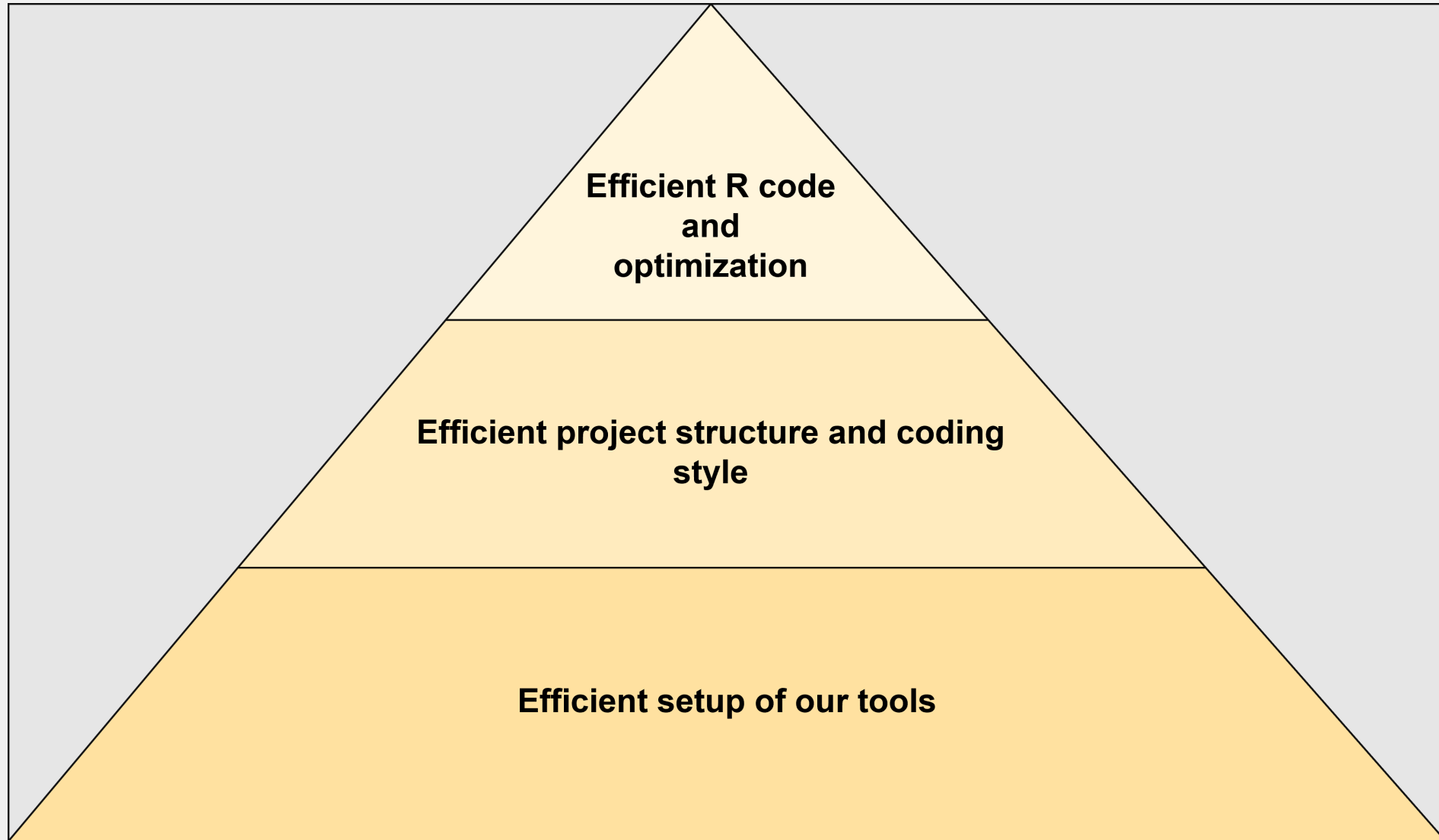
```
microbenchmark(  
  r = fibonacci_r(30),  
  rcpp = fibonacci_cpp(30),  
  times = 10  
)  
#> Unit: microseconds  
#>   expr      min       lq      mean     median      uq      max  neval  cld  
#>    r 368853.5 375472.8 384258.16 383497.70 387699.5 404987.4   10    a  
#>   rcpp    913.8   939.8  1058.99   947.75   983.7   1798.1   10    b
```

Summary

Efficient R code and optimization

- First: Can I run it somewhere else?
 -  Background job or cluster
- If not: Find bottlenecks in your code
 -  `profvis` package for profiling
 -  `microbenchmark` package for benchmarking
- Make the critical sections more efficient

Summary



Next lecture

Topic t.b.a.

 17th July  4-5 p.m.  Webex

 Subscribe to the mailing list

 For topic suggestions and/or feedback [send me an email](#)

Thank you for your attention :)

Questions?

Appendix

Cache function results

- Use the `memoise` package
- If functions are called many times with the same arguments
 - Avoids the recalculation
- Useful to e.g. improve the performance of a shiny app

Cache function results

Example: Create a plot on a subset of the `iris` data set

```
# Example of using memoise to cache results
library(memoise)
library(ggplot2)

# Remove rows from plotting function
select_iris_species <- function(rows_to_remove) {
  iris_subset <- iris[-rows_to_remove, ]
  # Do a plot on the subset
  p <- ggplot(
    iris_subset,
    aes(x = Sepal.Length, y = Sepal.Width, color = Species)
  ) +
    geom_point()
}

# Version of the function with memoise
select_iris_species_mem <- memoise(select_iris_species)
```

Cache function results

Example: Create a plot on a subset of the `iris` data set

```
# Compare the two versions
result <- microbenchmark(
  no_cache = select_iris_species(10),
  cache = select_iris_species_mem(10)
)

autoplot(result)
```

