# Efficient R

## Scientific workflows: Tools and Tips 🛠️

Dr. Selina Baldauf

2023-11-16

# What is this lecture series?

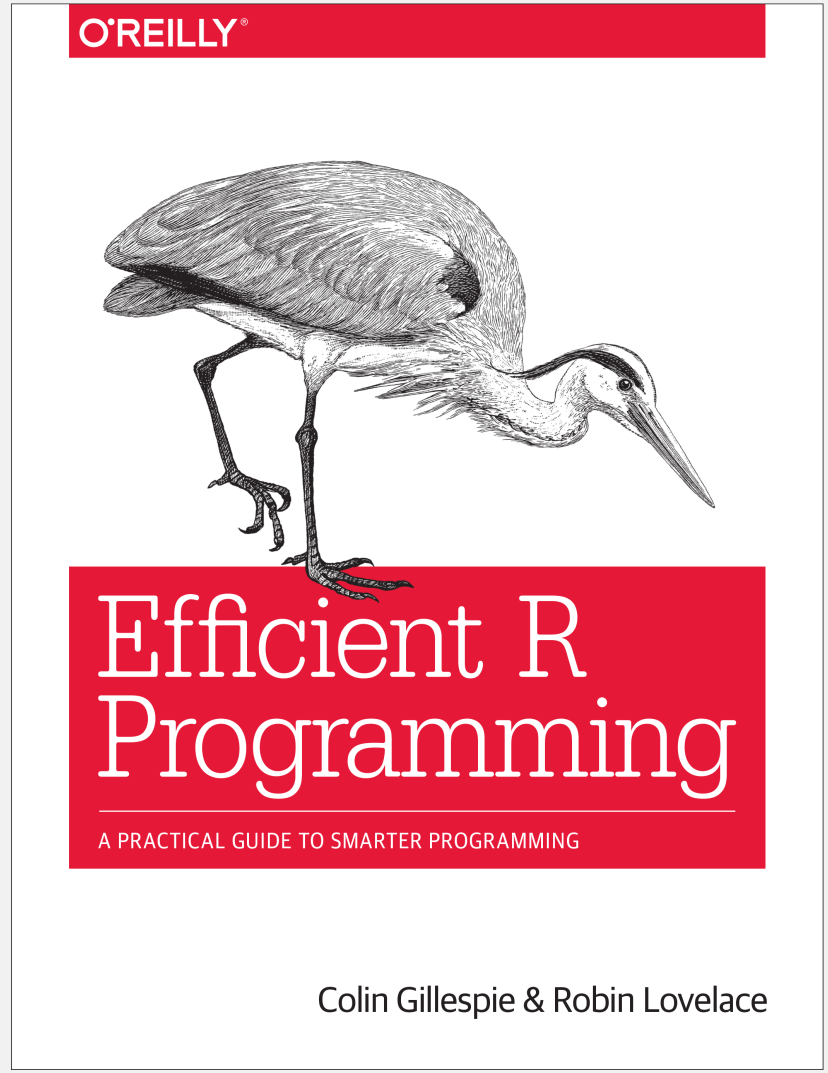
## Scientific workflows: Tools and Tips 🛠️

📅 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

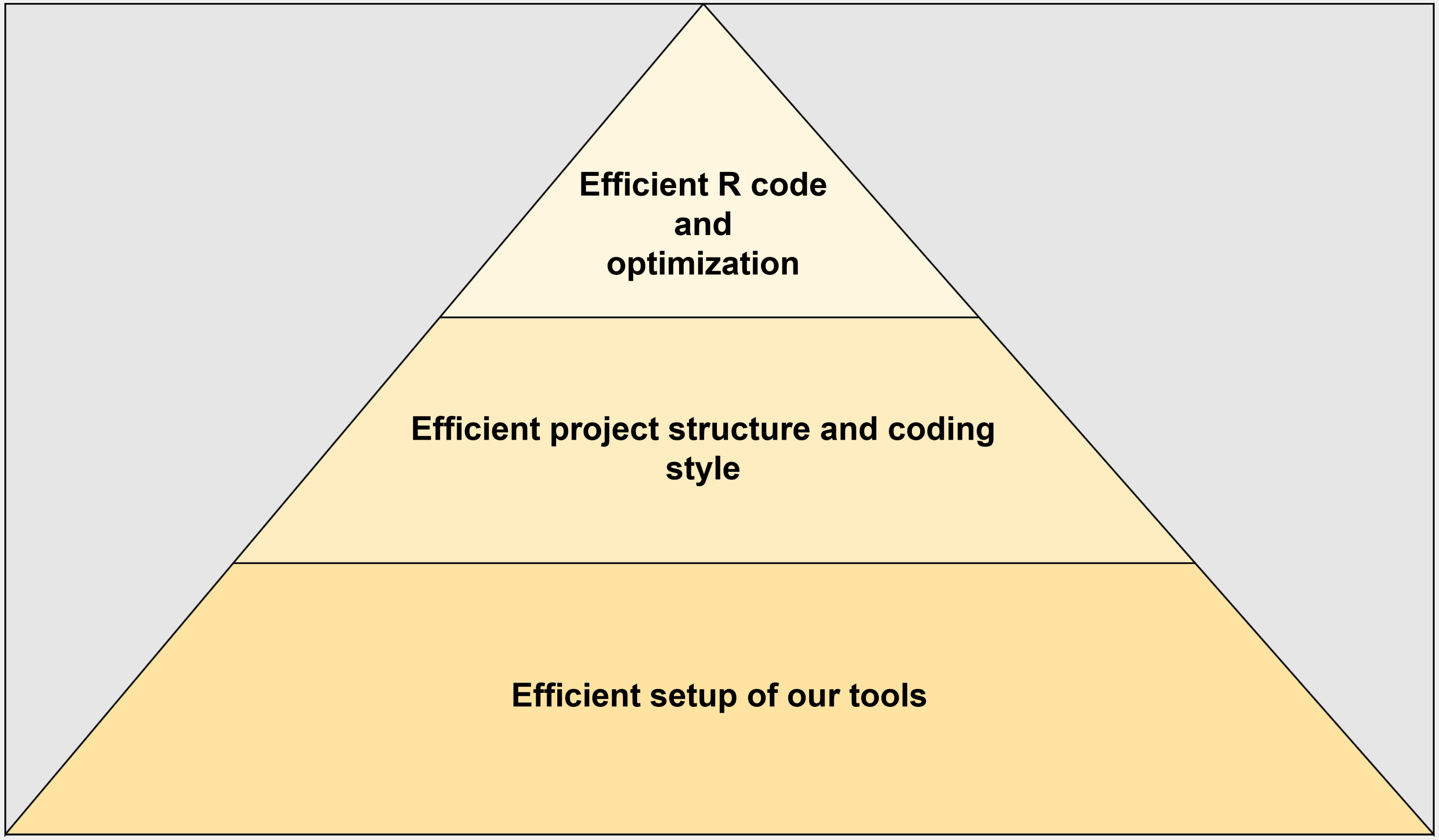
### Programmer efficiency

🧍 How long does it take to

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

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

# Today

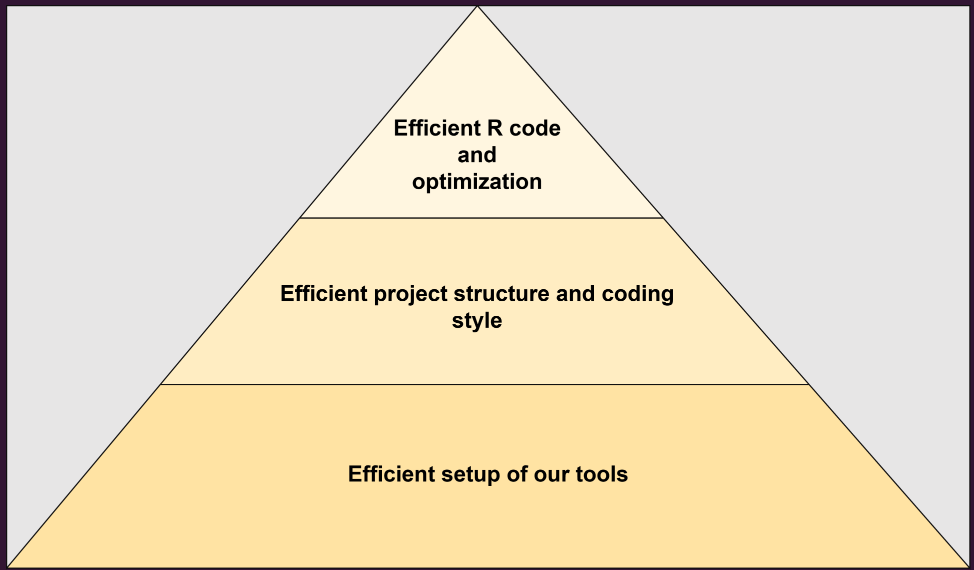



**Principles** and **tools** to make R programming more effiicent for 💻

Check out my talk "What they forgot to teach you about R" for first two levels

# Efficient R code and optimization

> How can I make my R code faster?



# Is R slow?

- R is slow compared to other programming languages (e.g. C++).
  - R is designed to make programming easy, not fast
- R is not designed to be memory efficient
- 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** by optimizing?
- **How often** do I run the code?
- How much time do I **spend** optimizing?

Often: Trade-off between **readability** and **efficiency**

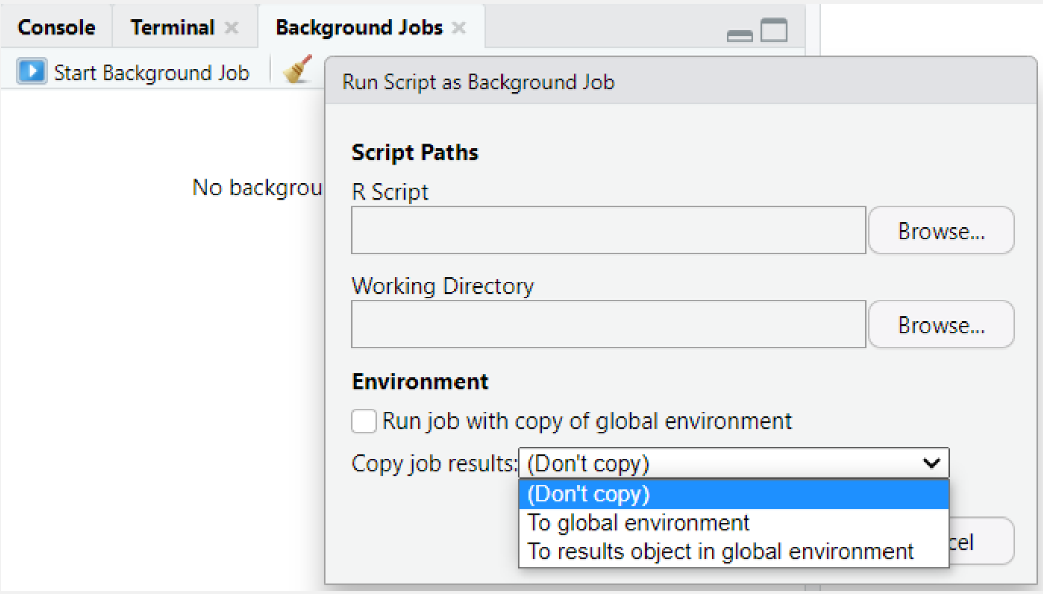
# Should I optimize?

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

1. If possible, **run the code somewhere else**

# Run the code somewhere else

- For this, RStudio has **background jobs**




- Or: run it on a cluster (e.g. FU Curta)

# Should I optimize?

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

1. If possible, **run the code somewhere else**
2. **Identify the critical** (slow) **parts** of your code
3. Then **optimize only the bottlenecks**

# Identify critical parts of your code

> **Profiling** & **Benchmarking** to measure the speed and memory use of your code

# Profiling R code

What are the speed & memory bottlenecks in my code?

- Use the `profvis` package

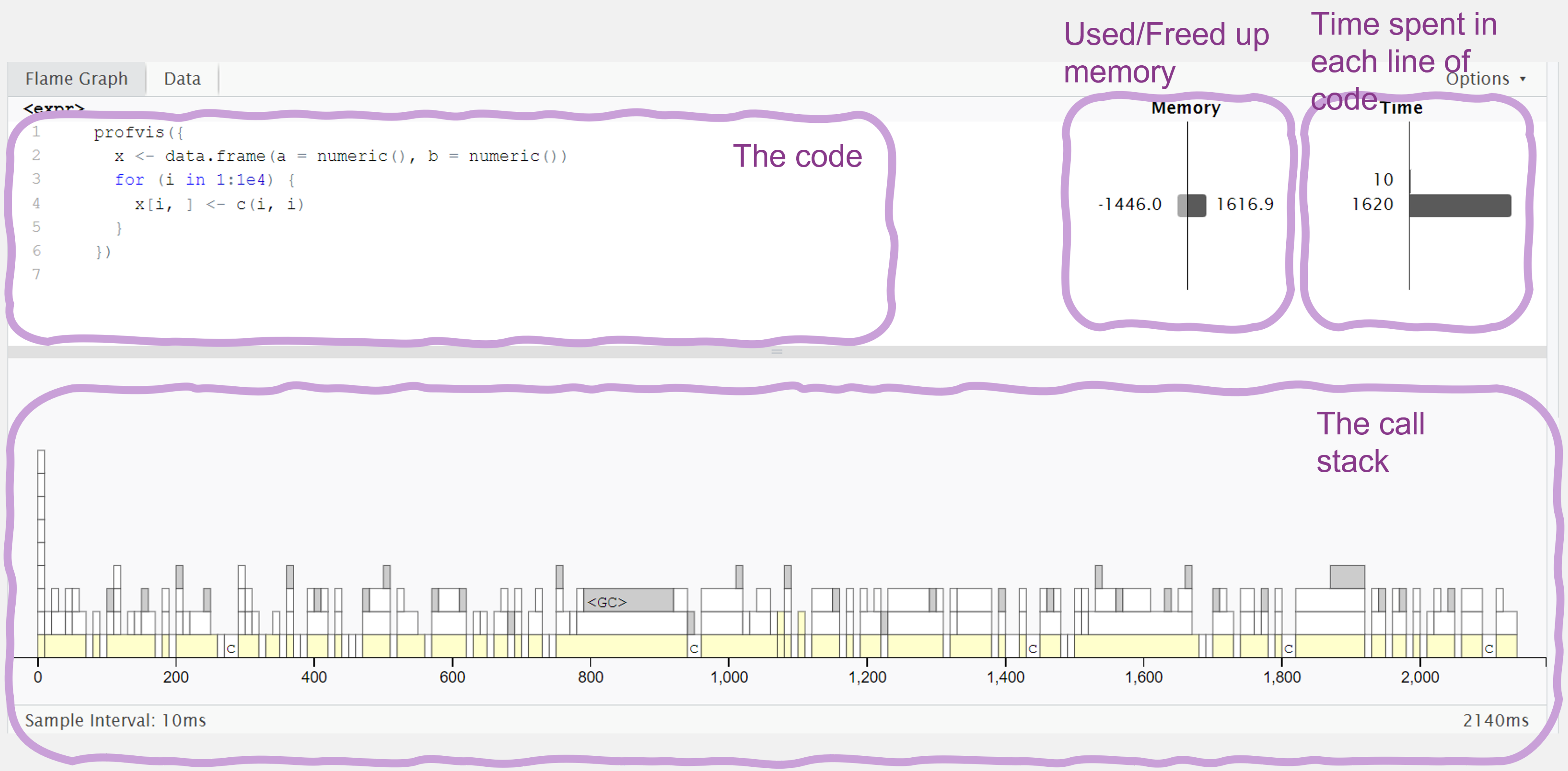
# Profiling R code

You can profile a section of code like this:

```r
# install.packages("profvis")
library(profvis)

# Create a data frame with 150 columns and 400000 rows
df <- data.frame(matrix(rnorm(150 * 400000), nrow = 400000))

profvis({
  # Calculate mean of each column and put it in a vector
  means <- apply(df, 2, mean)

  # Subtract mean from each value in the table
  for (i in seq_along(means)) {
    df[, i] <- df[, i] - means[i]
  }
})
```

# 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.

Flame Graph | Data | Options ▾

| Code | File | Memory (MB) | | Time (ms) |
|---|---|---|---|---|
| ▶ [<-.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

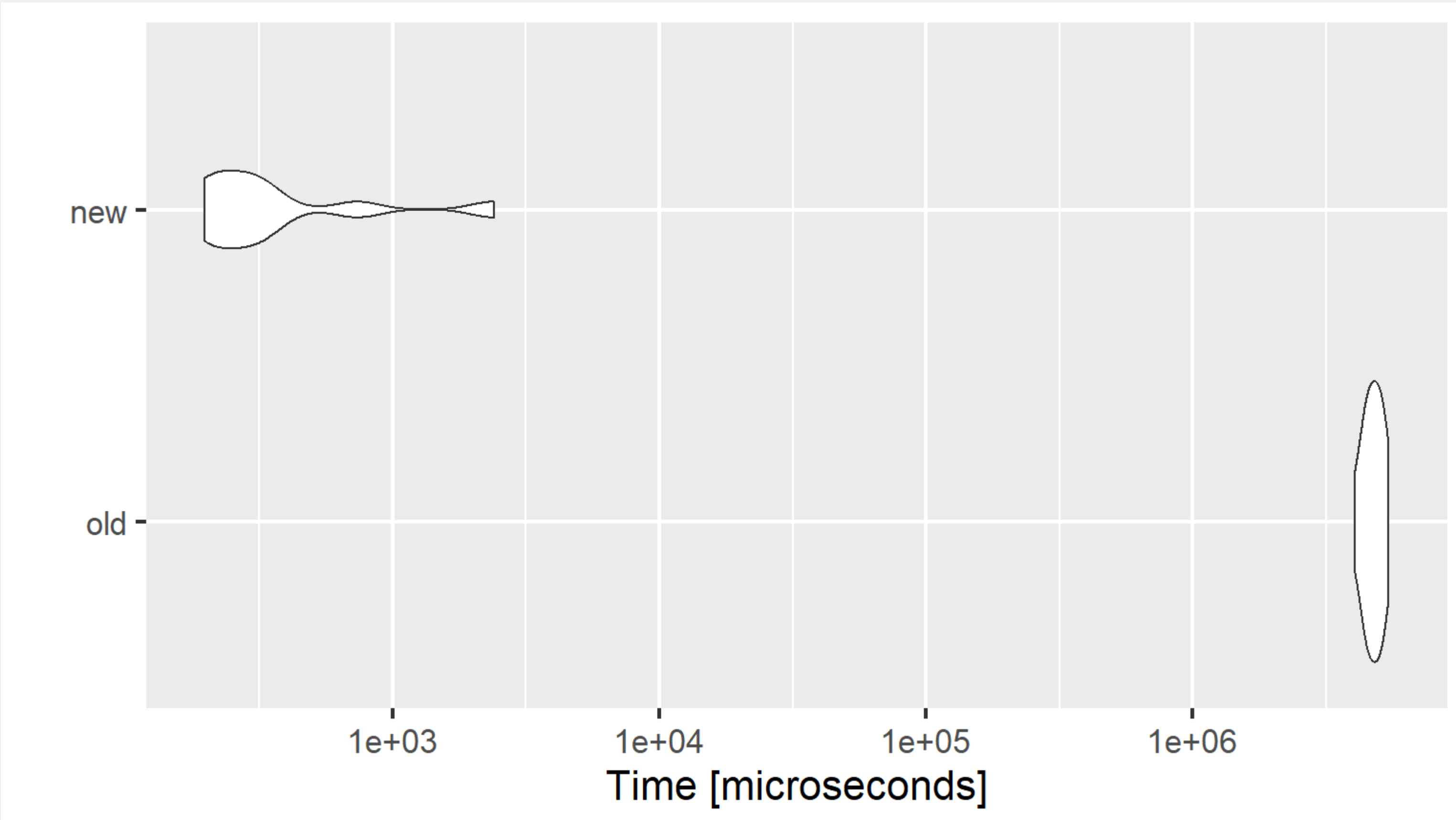
Use the `microbenchmark` package to compare the functions:

```
# install.packages("microbenchmark")
library(microbenchmark)

compare_functions <- microbenchmark(
  old = f1(),
  new = f2(),
  times = 10 # default is 100
)

compare_functions
#> Unit: microseconds
#>  expr       min         lq       mean      median       uq       max neval cld
#>   old 3020299.9 3148224.8 3477195.5 3330624.60 3881289 4231654.5    10   a
#>   new     204.8     296.9     718.7     335.85     516    4034.3    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

```r
# A vector with 1 million values
x <- 1:1e6

microbenchmark(
  for_loop = {
    log_sum <- 0
    for (i in 1:length(x)) {
      log_sum <- log_sum + log(x[i])
    }
  },
  sum = sum(log(x)),
  times = 10
)
#> Unit: milliseconds
#>      expr      min       lq      mean    median       uq      max neval cld
#>  for_loop 116.5736 124.5762 138.09591 139.41500 144.2552 175.4058    10  a
#>       sum  39.0255  50.0998  55.92386  51.99125  70.2841  73.6352    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

```r
# A matrix with 1000 columns
x <- matrix(rnorm(10000), ncol = 1000)

microbenchmark(
  no_cache = apply(x, 2, function(i) mean(i) / sd(x)),
  cache = {
    sd_x <- sd(x)
    apply(x, 2, function(i) mean(i) / sd_x)
  }
)
#> Unit: milliseconds
#>      expr      min       lq      mean    median        uq      max neval cld
#>  no_cache 130.3633 147.2735 166.20257 158.29205 179.40650 333.9536   100  a
#>     cache   8.2358   9.6942  11.86932  11.06015  12.80075  24.3593   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).

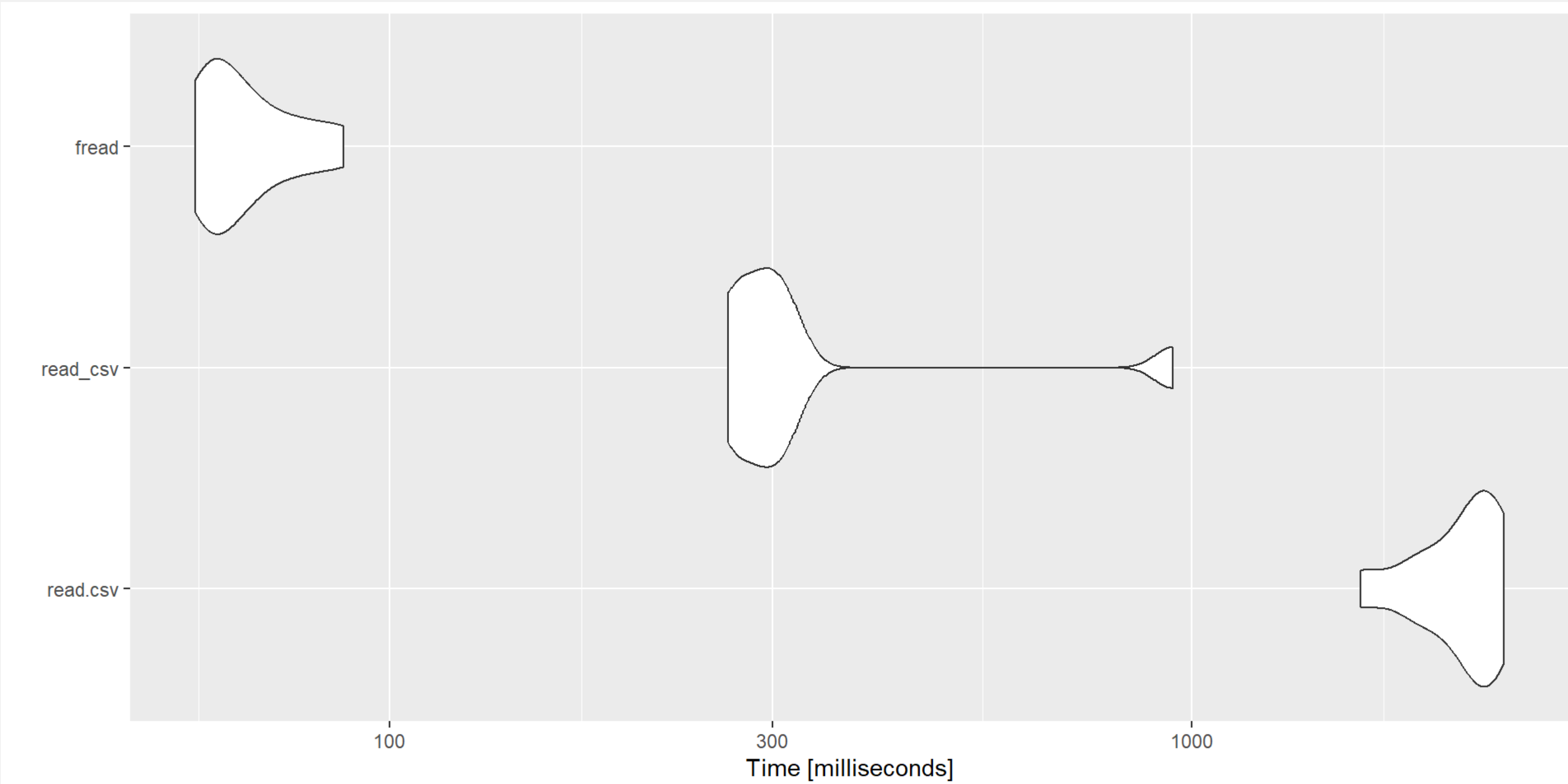
- Base-R functions to read csv files are:
  - `read.table`
  - `read.csv`
- There are many alternatives to read data, e.g.:
  - `read_csv` from the `readr` package (tidyverse)
  - `fread` from the `data.table` package

# Read data

## Compare some alternative reading functions

```r
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),
  times = 10
)

autoplot(compare_input)
```

# Read data

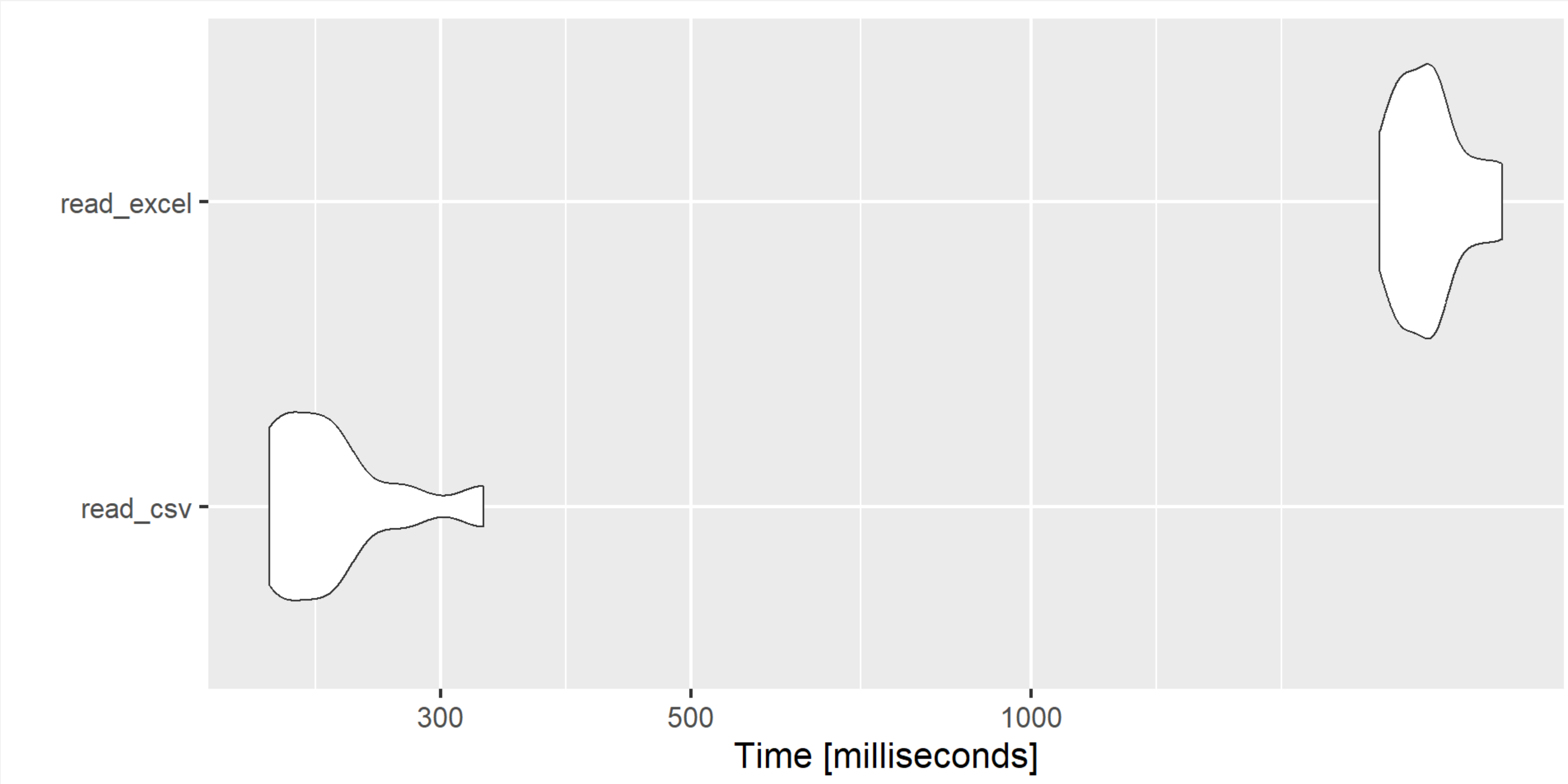


# Use plain text data

## 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, progress = FALSE, show_col_types = FALSE),
  read_excel = readxl::read_excel(file_path_xlsx),
  times = 10
)

autoplot(compare_excel)
```

# Use plain text data



# Write data

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

# 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

```r
# 1. The data table way
# Convert the data to a data.table
setDT(ghg_ems)
summarize_dt <- function(){
  ghg_ems[, mean(Electricity, na.rm = TRUE), by = Country]
}

# 2. The dplyr way
summarize_dplyr <- function(){
  ghg_ems |>
      group_by(Country) |>
      summarize(mean_e = mean(Electricity, na.rm = TRUE))
}

# 3. The collapse way
summarize_collapse <- function(){
  ghg_ems |>
      fgroup_by(Country) |>
      fsummarise(mean_e = fmean(Electricity))
}
```

# Summarize data by group

**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 14790.6 15268.5 17610.55 15896.70 17849.3 31915.9    10  a
#>  data_table  1807.9  1896.3  3303.93  1978.45  2583.2 11115.6    10   b
#>    collapse   452.7   538.0  1595.99   576.20  1720.4  5854.5    10   b
```

# Select columns

**Example**: Select columns Country, Year, Electricity, Transportation

```
microbenchmark(
  dplyr = select(ghg_ems, Country, Year, Electricity, Transportation),
  data_table = ghg_ems[, .(Country, Electricity, Transportation)],
  collapse = fselect(ghg_ems, Country, Electricity, Transportation),
  times = 10
)
#> Unit: microseconds
#>        expr    min     lq    mean  median     uq     max neval cld
#>       dplyr 5286.3 5808.9 6946.64 5969.85 7321.2 12978.6    10  a
#>  data_table  761.3 1017.7 1132.57 1065.05 1178.8  1882.5    10   b
#>    collapse   16.2   18.1   51.58   35.50   36.6   255.5    10   b
```

# Advanced optimization

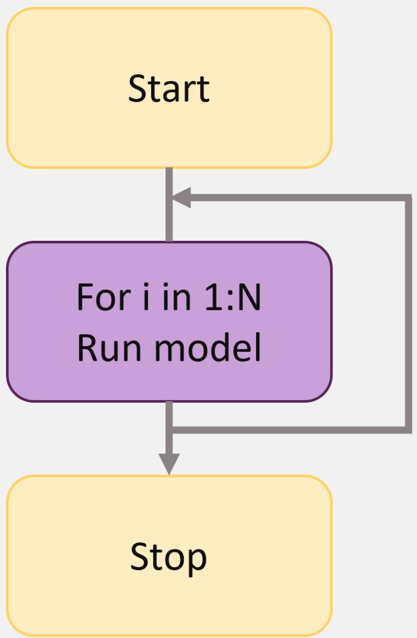
Parallelization and C++

# Parallelization

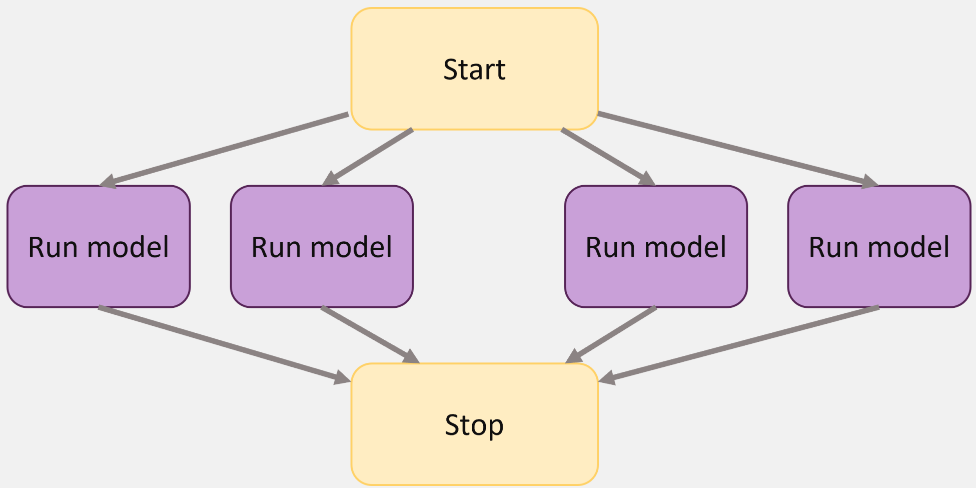
By default, R works on one core but CPUs have multiple cores

```
# Find out how many cores you have with the parallel package
# install.packages("parallel")
parallel::detectCores()
#> [1] 8
```

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:

```
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

## The **sequential** version

```r
# to measure the runtime
library(tictoc)

# create a vector of 10 numbers
x <- 1:10

tic()
result <- lapply(x, slow_sqrt)
toc()
#> 10.23 sec elapsed
```

## The **parallel** version

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

tic()
result <- future_lapply(x, slow_sqrt)
toc()
#> 4.23 sec elapsed
```

# Parallel apply functions

Selected base R apply functions and their future versions:

| base | future.apply |
|---|---|
| `lapply` | `future_lapply` |
| `sapply` | `future_sapply` |
| `vapply` | `future_vapply` |
| `mapply` | `future_mapply` |
| `tapply` | `future_tapply` |
| `apply` | `future_apply` |
| `Map` | `future_Map` |

# 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()
#> 10.19 sec elapsed
```

The **parallel** version

```
library(doFuture)

tic()
z <- foreach(i = 1:10) %dofuture% {
  slow_sqrt(i)
}
toc()
#> 2.84 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

```r
# 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))
  }
}
```

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

# Rewrite a function in C++

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

```r
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));
    }
  }'
)
```

```r
# 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);
}
```

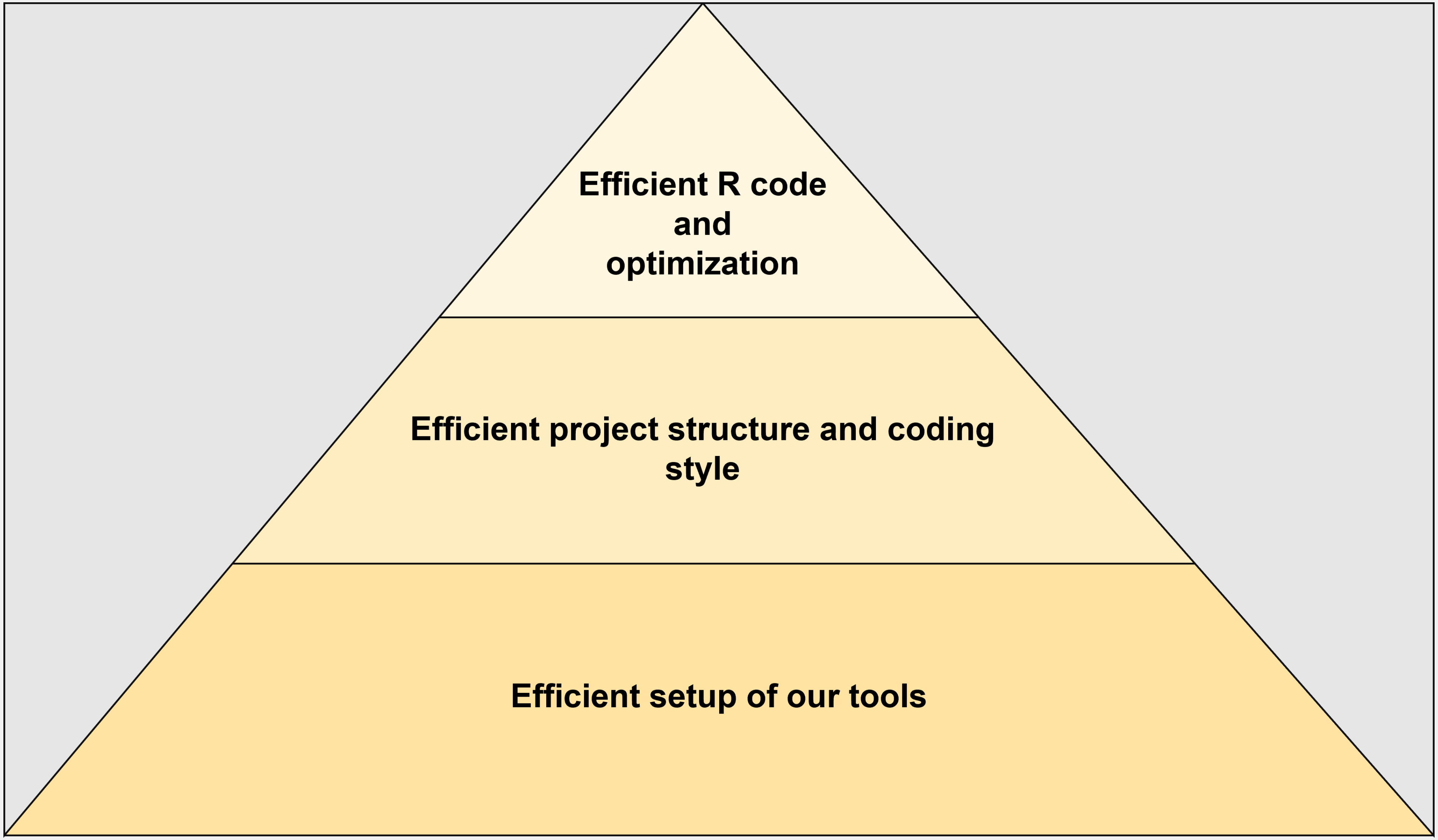
Then source the function in your R script using `sourceCpp`:

```
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: milliseconds
#>  expr       min       lq       mean    median        uq       max neval cld
#>     r 1500.8667 1565.980 1695.35756 1629.9862 1780.3685 2059.7378    10  a
#>  rcpp    2.3096    2.322    2.68697    2.5622    2.8446    3.5691    10   b
```

# Summary


Efficient R code
and
optimization
Efficient project structure and coding
style
Efficient setup of our tools

# Next lecture

## Topic t.b.a.

📆 18th January 🕓 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?