# Efficient R

Selina Baldauf

selina.baldauf@fu-berlin.de @selina-b

# Welcome

- Ecology/Theoretical Ecology
- Scientific programmer at Freie Universität Berlin
- Research + Coding + Teaching

# What is efficiency?

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

## Computational efficiency

💻 Computation time

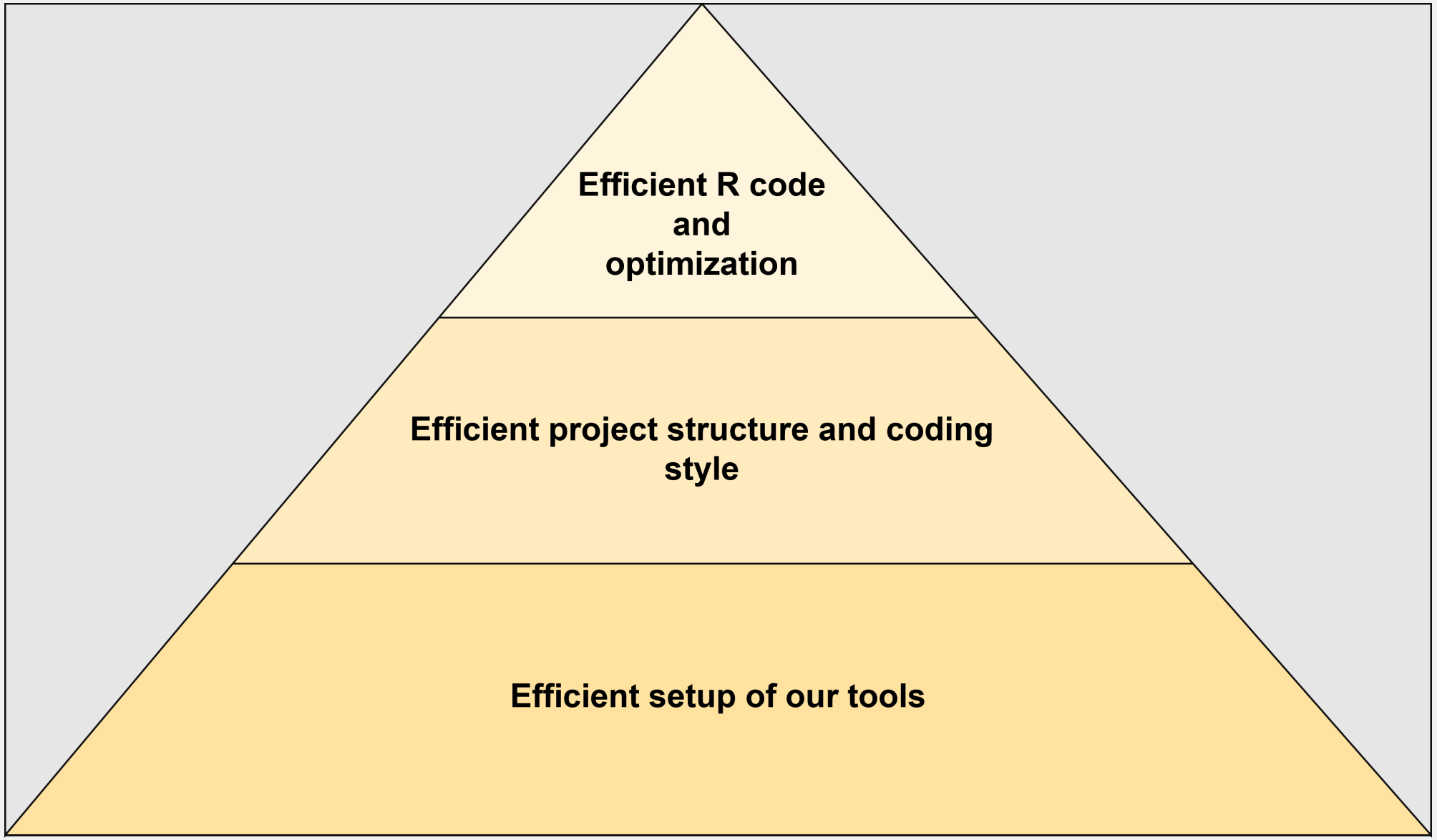
💾 Memory usage

## 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




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

# Today

- How to **measure speed** of your code
- **Basics** of efficient R programming
- Efficient **data analysis**
- Advanced optimization
  - **Parallelization**
  - Integrating **C++**

📦 Material (Slides, Code, References & Resources) on Github

❓ Question - just ask in the Chat or unmute yourself!

# Is R slow?

- R is slow compared to other programming languages (e.g. C++, Julia).
- R is the not the most memory efficient language
- R is designed to make statistical programming & data analysis **easy** and **interactive**, not fast
- 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** optimizing?
- **How often** do I run the code?
- Trade-offs between **readability** and **efficiency**

# How to optimize

Don't optimize from top to bottom!

1. **Identify bottlenecks** (i.e. slow parts) of your code
2. **Optimize** only those 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

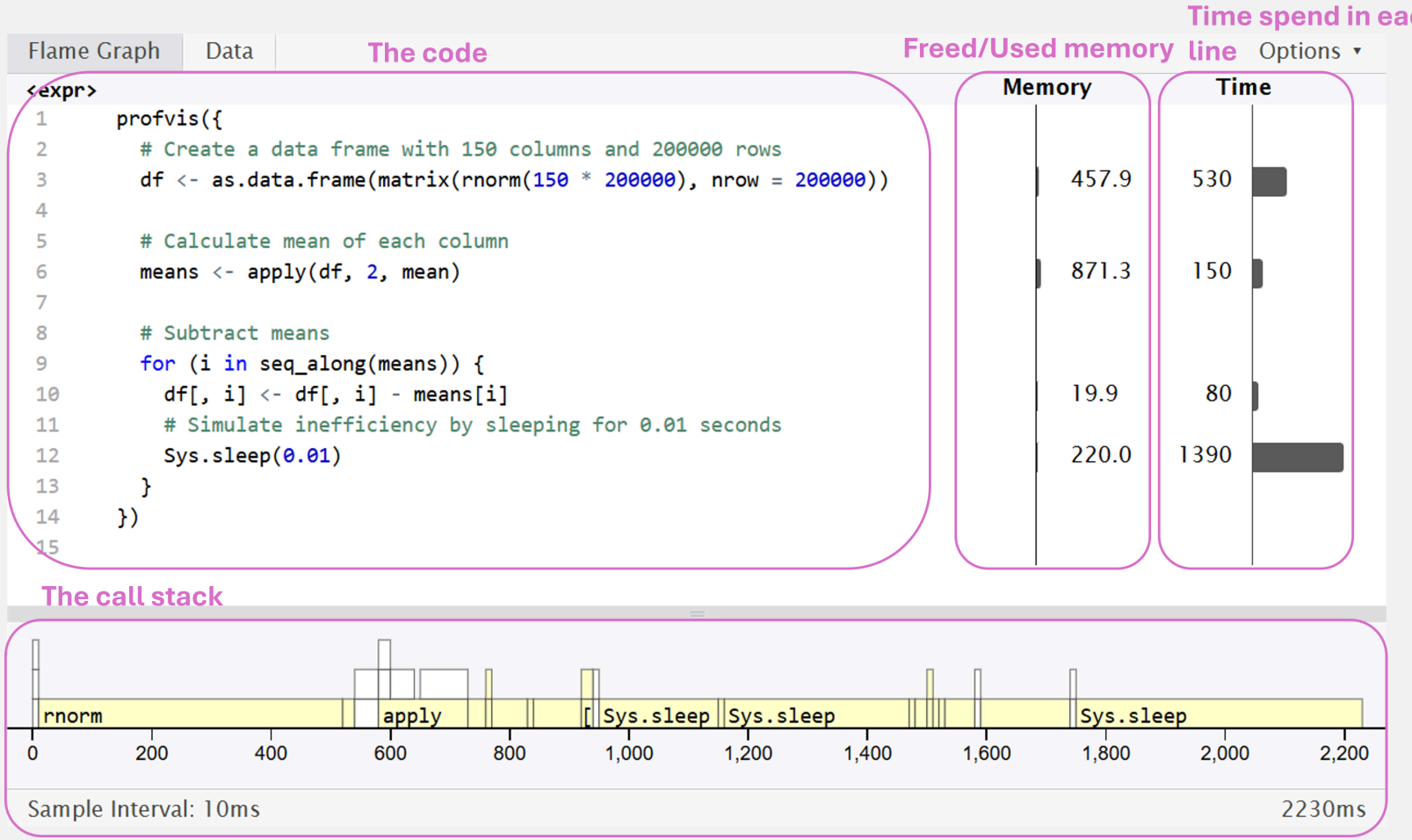
# Profiling R code

You can profile a section of code like this:

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

profvis({
  # Create a data frame with 150 columns and 200000 rows
  df <- as.data.frame(matrix(rnorm(150 * 200000), nrow = 200000))

  # Calculate mean of each column
  means <- apply(df, 2, mean)

  # Subtract means
  for (i in seq_along(means)) {
    df[, i] <- df[, i] - means[i]
    # Simulate inefficiency by sleeping for 0.01 seconds
    Sys.sleep(0.01)
  }
})
```

# Profiling R code

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



# Profiling R code

Profvis **data view** for details of the call stack.

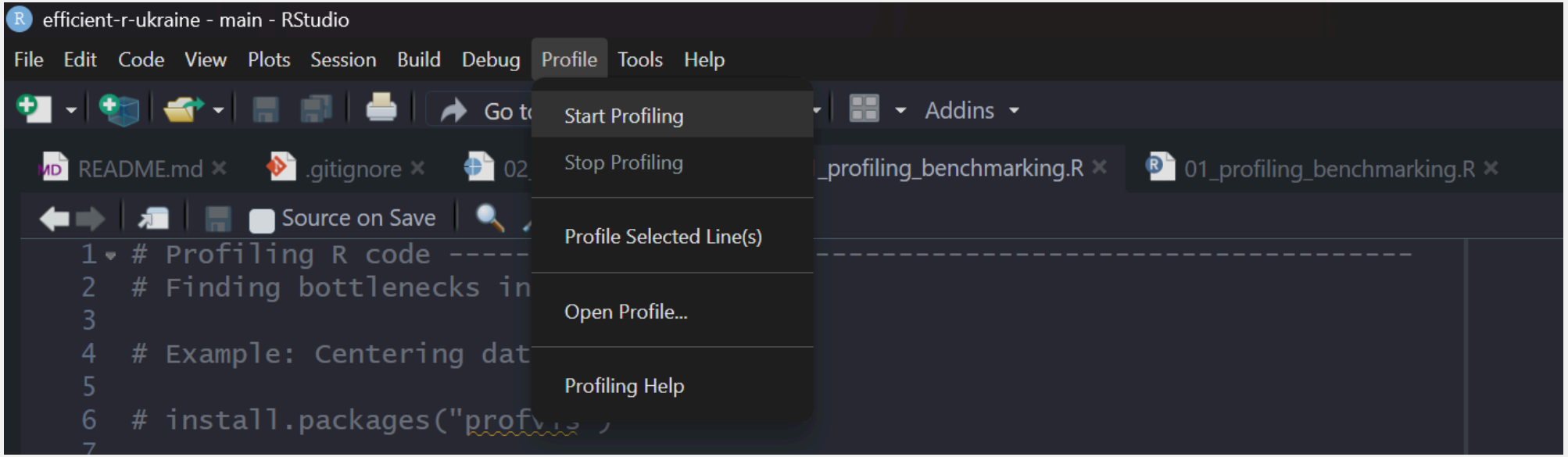
Flame Graph | Data | Options

| Code | File | Memory (MB) | | Time (ms) |
|---|---|---|---|---|
| Sys.sleep | <expr> | 0 | 220.0 | 1390 |
| rnorm | <expr> | 0 | 229.0 | 510 |
| ▼ apply | <expr> | 0 | 871.3 | 150 |
| mean.default | | 0 | 184.6 | 80 |
| aperm.default | | 0 | 228.9 | 40 |
| as.matrix.data.frame | | 0 | 308.2 | 20 |
| ▶ profvis | | 0 | 4.6 | 40 |
| as.data.frame.matrix | | 0 | 264.0 | 40 |
| [ | <expr> | 0 | 6.1 | 40 |
| matrix | <expr> | 0 | 228.9 | 20 |
| [.data.frame | <expr> | 0 | 3.1 | 20 |
| df[, i] <- df[, i] - means[i] | <expr> | 0 | 4.6 | 20 |

# 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?

```r
center_data_slow <- function() {
  # Create data with 150 columns and 100000 rows
  df <- as.data.frame(
    matrix(rnorm(150 * 100000), nrow = 100000)
  )

  # Calculate mean of each column
  means <- apply(df, 2, mean)

  # Subtract means
  for (i in seq_along(means)) {
    df[, i] <- df[, i] - means[i]
    # Simulate inefficiency by sleeping 0.01 seconds
    Sys.sleep(0.01)
  }
  return(df)
}
```

```r
center_data_fast <- function() {
  # Create data with 150 columns and 100000 rows
  df <- matrix(rnorm(150 * 100000), nrow = 100000)

  # Calculate mean of each column
  means <- colMeans(df)

  # Subtract means
  for (i in seq_along(means)) {
    df[, i] <- df[, i] - means[i]
  }
  return(df)
}
```

# Benchmarking R code - the easy way

Use the `tictoc` package also for longer code sections:

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

tic()
slow_data <- center_data_slow()
toc()
#> 3.06 sec elapsed

tic()
fast_data <- center_data_fast()
toc()
#> 0.55 sec elapsed
```

# Benchmarking R code

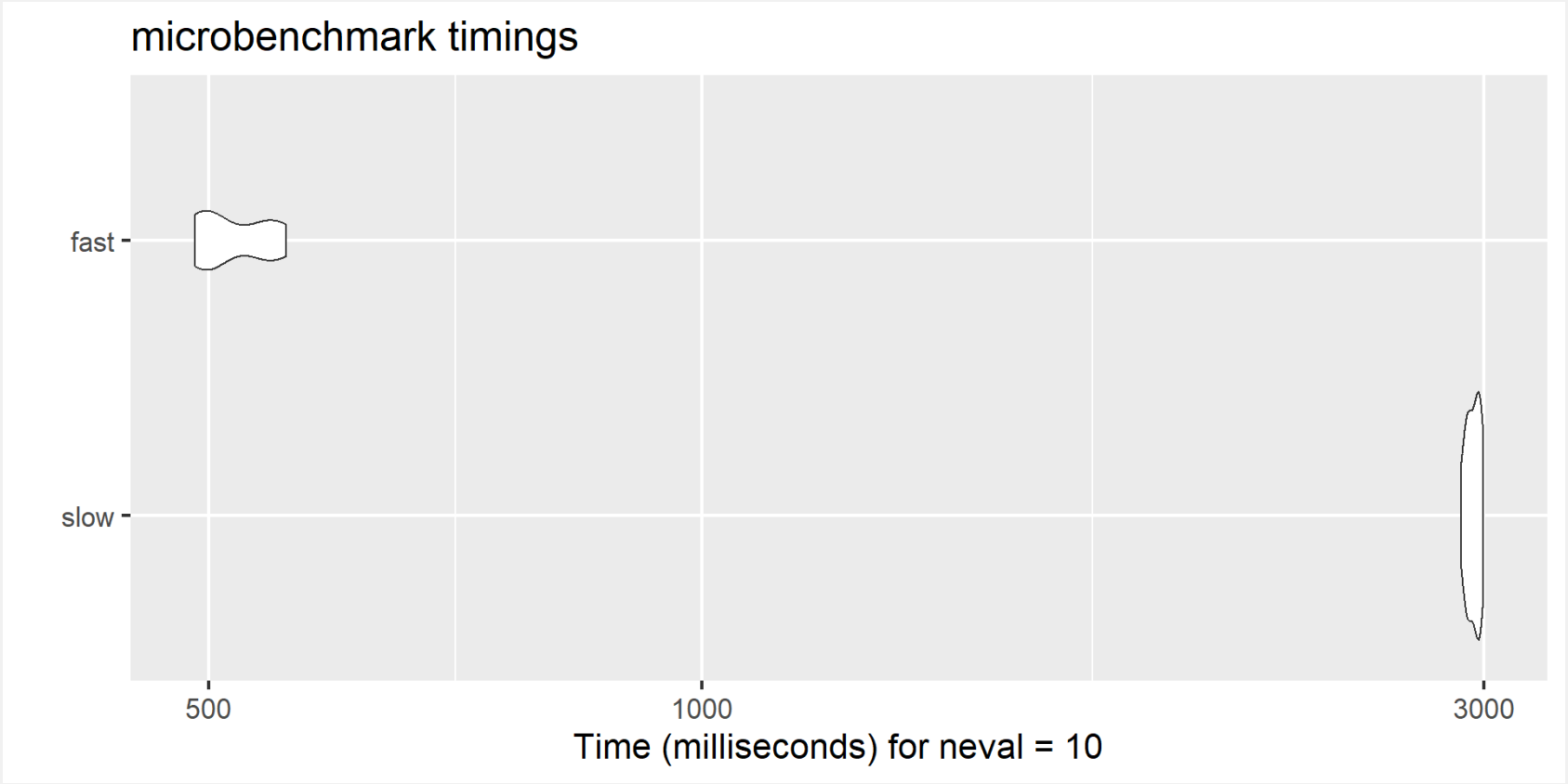
Use the `microbenchmark` package to compare the functions:

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

runtime_comp <- microbenchmark(
  slow = center_data_slow(),
  fast = center_data_fast(),
  times = 10 # the default is 100 but we are impatient
)

runtime_comp
#> Unit: milliseconds
#>  expr       min        lq      mean    median        uq       max neval cld
#>  slow 2907.6015 2930.4255 2956.9953 2960.0167 2981.8489 2997.6368    10  a
#>  fast  490.1377  496.1393  517.5706  505.7072  542.5332  557.1841    10   b

# look at relative runtime comparison
summary(runtime_comp, unit = "relative")
#>   expr      min       lq     mean   median       uq      max neval cld
#> 1 slow 5.932214 5.906457 5.713222 5.853222 5.496159 5.379975    10  a
#> 2 fast 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000    10   b
```

# Benchmarking R code

We can look at benchmarking results using ggplot

```
library(ggplot2)
autoplot(runtime_comp)
```



# Optimize your R 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

# Vectorized functions in R

Some examples of base-R vectorized functions:

- Arithmetic & Math: `+`, `-`, `*`, `/`, `log()`, `exp()`, `sqrt()`, `mean()`, `sd()`, `rowSums()`, `colMeans()`
- Logical & Comparison: `<`, `>`, `==`, `!=`, `&`, `|`, `all()`, `any()`, `%in%`
- Data Manipulation: `[`, `ifelse()`, `cut()`, `cumsum()`, `which()`, `match()`, `order()`
- String operations: `paste()`, `paste0()`, `grep()`, `gsub()`

# Vectorize your code

Vector arithmetic vs. for-loops | Example 2: Calculating cumulative values

```
x <- 1:1e6
y <- 1:1e6

microbenchmark(
  for_loop = {
    result <- numeric()
    for (i in seq_along(x)) {
      result[i] <- x[i] * 2 + y[i] / 2
    }
  },
  vectorized = x * 2 + y / 2,
  times = 10
)
#> Unit: milliseconds
#>        expr      min       lq      mean    median       uq      max neval cld
#>    for_loop 121.2896 125.0952 139.65615 129.18735 136.9957 190.5542    10  a
#>  vectorized   1.8256   1.8668   2.89729   3.22085   3.6280   3.6639    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.

# Don't grow objects in a loop

If you know how big your object will be, pre-allocate it.

```
f1 <- function() {
  x <- numeric() # no pre-allocation
  for (i in 1:1e6) {
    x[i] <- i
  }
}
```

```
f2 <- function() {
  x <- numeric(1e6) # pre-allocate vector
  for (i in 1:1e6) {
    x[i] <- i
  }
}
```

```
compare_alloc <- microbenchmark(
  no_alloc = f1(),
  pre_alloc = f2(),
  times = 10
)

summary(compare_alloc, unit = "relative")
#>        expr       min        lq      mean   median       uq       max neval cld
#> 1  no_alloc 1.0000000 1.0000000 1.0000000 1.000000 1.000000 1.0000000    10   a
#> 2 pre_alloc 0.9650891 0.9896431 0.9831439 1.009073 1.017852 0.8558446    10   a
```

# Don’t grow objects in a loop

## Why?

R makes a copy of the object each time it grows, which is inefficient.

# Don’t grow objects in a loop

```
x <- numeric()
y <- numeric(10)

for (i in 1:10) {
  x[i] <- i
  y[i] <- i

  # Print current memory location after each assignment
  cat(
    "After iteration ",
    i,
    ": x is in ",
    pryr::address(x),
    "and y is in ",
    pryr::address(y),
    "\n"
  )
}
#> After iteration  1 : x is in  0x1c07a517498 and y is in  0x1c06627cc38
#> After iteration  2 : x is in  0x1c065e5f5c8 and y is in  0x1c06627cc38
#> After iteration  3 : x is in  0x1c0928940a8 and y is in  0x1c06627cc38
```

# 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

```
# 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 52.1793 56.61065 68.90352 72.38950 75.22725 94.6167   100  a
#>     cache  3.3721  3.58560  3.97894  3.65755  3.77115 19.7137   100   b
```

# Efficient data analysis

# Efficient workflow

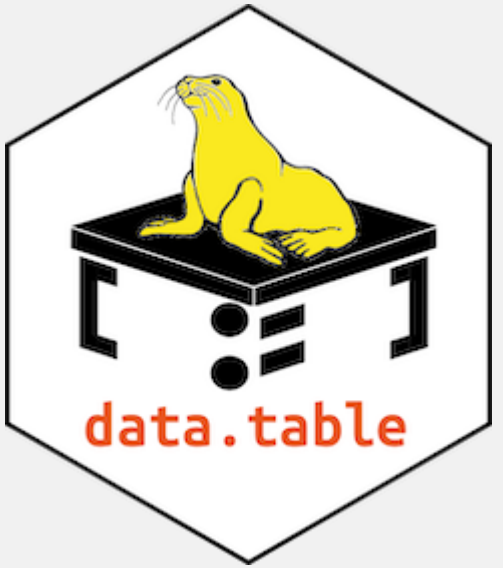
- Filter early: Remove unnecessary rows/columns at the beginning
- Avoid redundant calculations
  - Calculate values once and store them in variables
  - Save intermediate results
- Use efficient data formats
  - Text files instead of Excel files
  - Binary formats like Parquet and feather for large datasets
  - Consider databases for very large datasets
- Use the right packages and functions for your tasks

# tidyverse

- Collection of packages for data analysis
- Consistent syntax, works well with pipes (`|>`, `%>%`)
- Main data structure `tbl_df`
- Universe of efficient packages build around the core tidyverse, e.g.
  - `dbplyr` for database and `dtplyr` for data.table backends
  - `futureverse` packages for parallelization
- Often fast enough but can be slower than alternatives

# data.table

- Functions for I/O, data manipulation and more
- Popular alternative to the `tidyverse`
- Designed for fast and memory-efficient data analysis
- Syntax quite different from tidyverse:




```
dt[i, j, by]
```

**Take** data.table `dt`, **subset rows** using `i` and **manipulate columns** with `j`, **grouped** according to `by`.

# collapse

> collapse is written in C and C++, with algorithms much faster than base R's, has extremely low evaluation overheads, scales well, and excels on complex statistical tasks.


- Data manipulation and statistical computing
- Easy to integrate with existing R code
  - works well with both `data.table` and `dplyr` workflows/data structures
  - often prefixes functions with `f` like `fmean`, `fmutate`, `fsummarize`, ...

# Arrow

- Access to features of Apache Arrow C++ library
- Functions for I/O and data manipulation
- Access to efficient binary data formats like parquet, feather, etc.
- Can also operate on larger-than-RAM data
- Provide Arrow C++ backend to `dplyr`

# Other packages

Of course there are so many more packages like:

- `polars`: Fast DataFrame library implemented in Rust
- `fst`: Fast and efficient data format with high read/write speed
- `memoise`: Cache function calls
- ...

Do you know others? Let's collect them in the chat

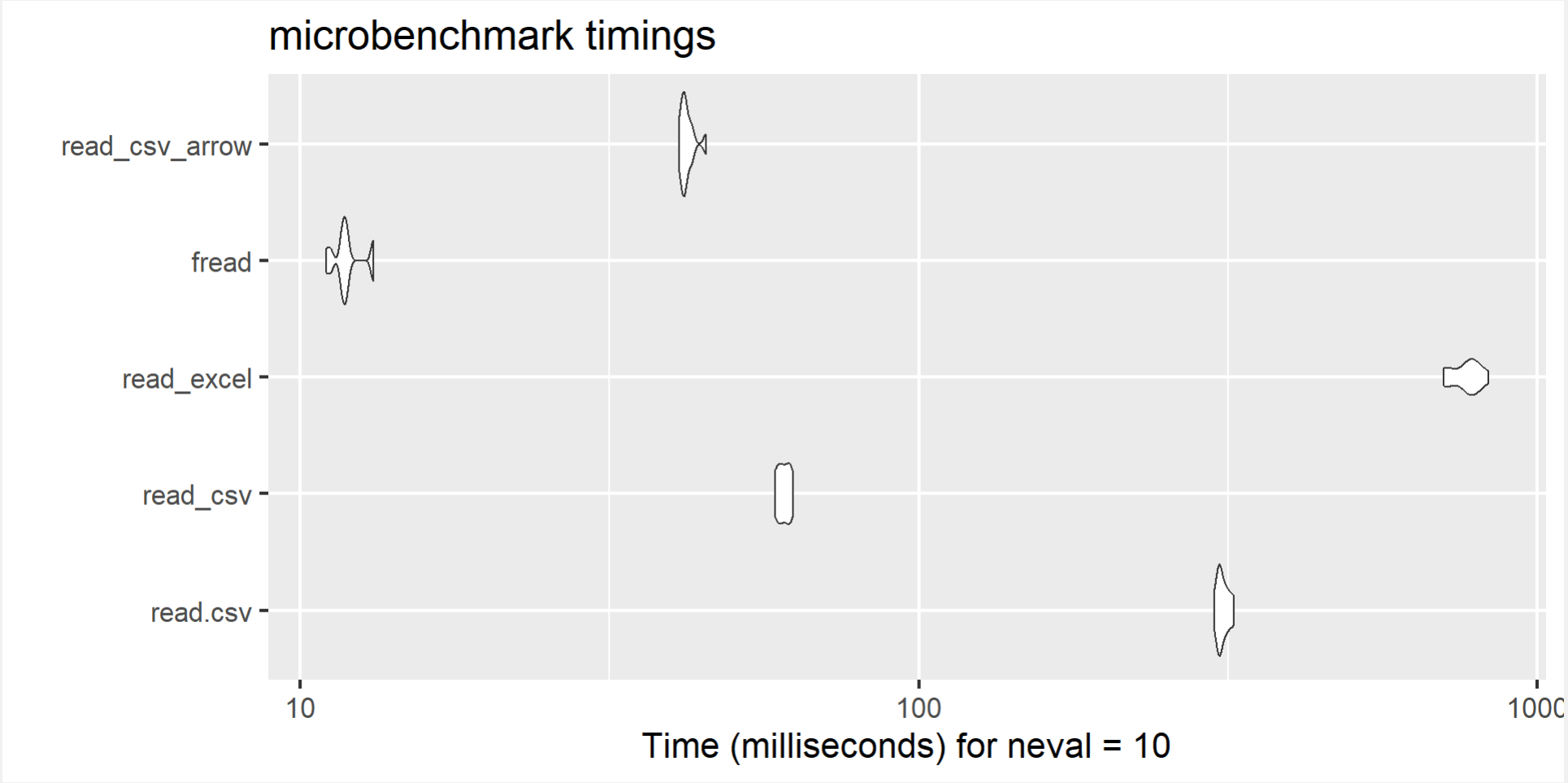
Check the resources section of the README for all links.

# Read data - text and excel

**Example:** Read data on global greenhouse gas emissions (~320000 rows, 8 cols).

```
file_path_csv <- here::here("data/ghg_ems.csv")
file_path_xlsx <- here::here("data/ghg_ems.xlsx")

compare_read <- microbenchmark(
  # base R
  read.csv = read.csv(file_path_csv),

  # tidyverse
  read_csv = readr::read_csv(file_path_csv, show_col_types = FALSE),
  read_excel = readxl::read_excel(file_path_xlsx),

  # data.table
  fread = data.table::fread(file_path_csv, showProgress = FALSE),

  # arrow
  read_csv_arrow = arrow::read_csv_arrow(file_path_csv),
  times = 10
)

autoplot(compare_read)
```

# Read data - text and excel

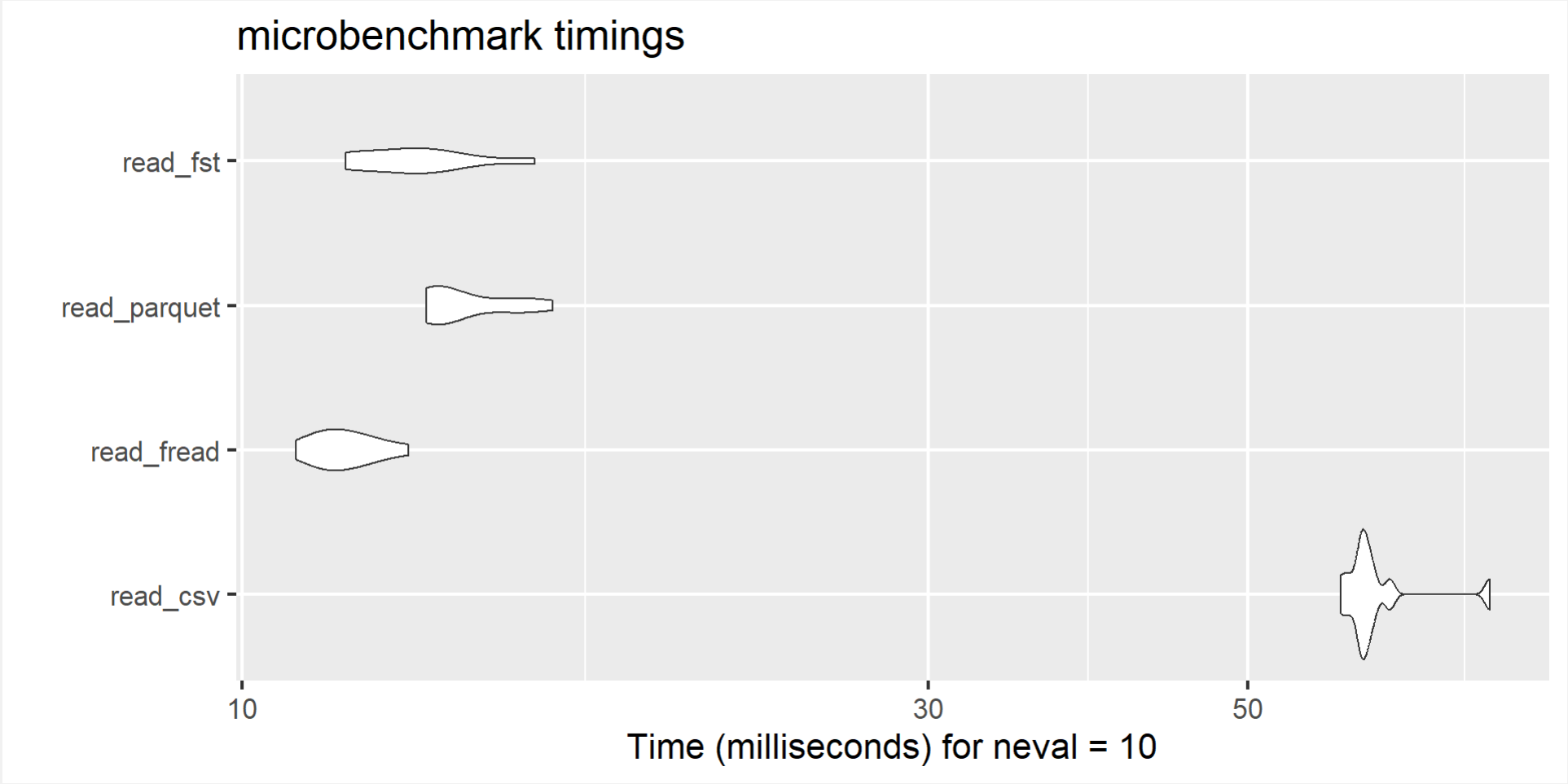


# Read data - binary

Binary file types can be faster (especially as file size increases)

```
file_path_parquet <- here::here("data/ghg_ems.parquet")
file_path_fst <- here::here("data/ghg_ems.fst")

compare_read_binary <- microbenchmark(
  # tidyverse
  read_csv = readr::read_csv(file_path_csv, show_col_types = FALSE),

  # data.table
  read_fread = data.table::fread(file_path_csv, showProgress = FALSE),

  # arrow parquet format
  read_parquet = arrow::read_parquet(file_path_parquet),

  # fst format (efficient R only format)
  read_fst = fst::read_fst(file_path_fst),

  times = 10
)

autoplot(compare_read_binary)
```

# Read data - binary



# Write data

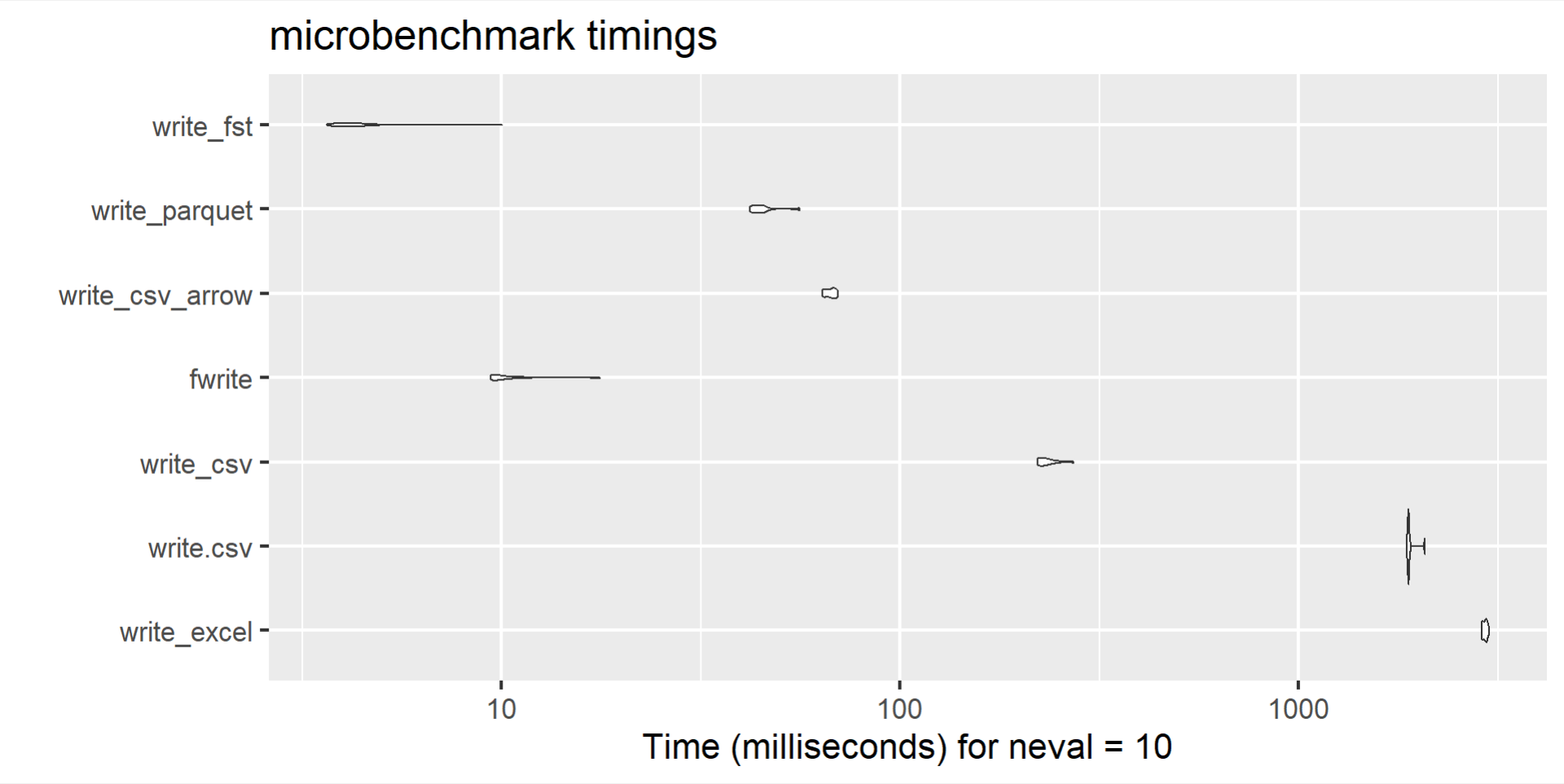
Every corresponding read function has a write counterpart:

- `write_csv` from the `readr` package (tidyverse)
- `fwrite` from the `data.table` package
- `write_csv_arrow`, `write_parquet` from the `arrow` package
- `write_fst` from the `fst` package

# Write data

```
# create an example with 1000000 rows to export
df <- data.frame(x = 1:1000000, y = 1:1000000, z = 1:1000000)

compare_output <- microbenchmark::microbenchmark(
  # write Excel
  write_excel = writexl::write_xlsx(df, "df.xlsx"),

  # write text
  write.csv = write.csv(df, "df.csv"),
  write_csv = readr::write_csv(df, "df.csv"),
  fwrite = data.table::fwrite(df, "df.csv"),
  write_csv_arrow = arrow::write_csv_arrow(df, "df.csv"),

  # write binary formats
  write_parquet = arrow::write_parquet(df, "data/df.parquet"),
  write_fst = fst::write_fst(df, "data/df.fst"),
  times = 10
)

autoplot(compare_output)
```

# Write data


microbenchmark timings
write_fst
write_parquet
write_csv_arrow
fwrite
write_csv
write.csv
write_excel
10
100
1000
Time (milliseconds) for neval = 10

# Data manipulation

**Example**: Summarize mean carbon emissions from Electricity by Country

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

# Use the right data formats
ghg_ems <- readr::read_csv(file_path_csv)
ghg_ems_dt <- setDT(ghg_ems) # to data table
ghg_ems_parquet <- read_parquet(file_path_parquet, as_data_frame = FALSE)
```

# Summarize data by group

```
# 1. The data table way
summarize_dt <- function() {
  ghg_ems_dt[, 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))
}

# 4. The arrow way
summarize_arrow <- function() {
```

# Efficient data manipulation

## **Example**: Summarize mean carbon emissions from Electricity by Country

```
# compare the speed of all versions
compare_summarize <- microbenchmark(
  dplyr = summarize_dplyr(),
  data_table = summarize_dt(),
  collapse = summarize_collapse(),
  arrow = summarize_arrow(),
  times = 10
)

summary(compare_summarize, unit = "relative")
#>         expr       min        lq      mean    median        uq       max neval
#> 1      dplyr  4.792294  4.924788  4.347089  4.578174  4.041363  4.221200    10
#> 2 data_table  2.238343  2.608554  2.469827  2.729258  2.494380  2.175025    10
#> 3   collapse  1.000000  1.000000  1.000000  1.000000  1.000000  1.000000    10
#> 4      arrow 16.002969 16.226567 15.801069 15.930190 14.333536 19.986203    10
#>   cld
#> 1 a
#> 2 ab
#> 3  b
#> 4   c
```

# Advanced optimization

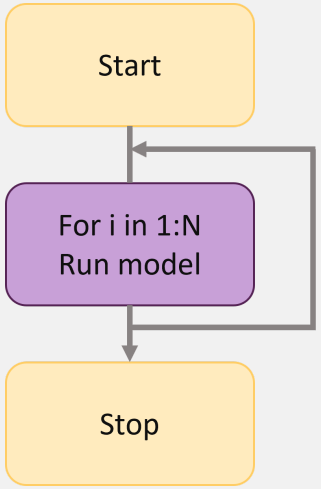
Parallelization and C++

# Parallelization

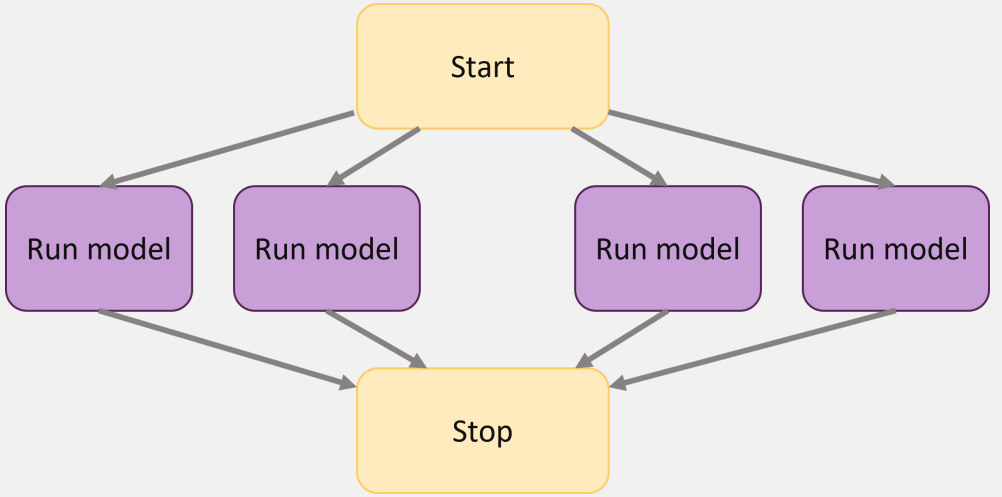
## By default, R works on one core but CPUs have multiple cores

```
# Find out how many cores you have with the parallel package
# install.packages("parallel")
parallel::detectCores()
#> [1] 32
```

## 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

# 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 5 cores
plan(multisession, workers = 5)
```

# 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()
#> 10.22 sec elapsed
```

## Parallel `future_lapply`

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

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

First, change into a `foreach`

```
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()
#> 10.17 sec elapsed
```

## The **parallel** version

```
library(doFuture)

tic()
z <- foreach(i = 1:10) %dofuture%
  {
    slow_sqrt(i)
  }
toc()
#> 3.11 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()
#> 10.11 sec elapsed
```

## The parallel version

```
library(furrr)

# the furrr version
tic()
z <- future_map(x, slow_sqrt)
toc()
#> 3.02 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:
  - Install C++ compiler (Rtools for Windows, Xcode for Mac, gcc for Linux, see here for instructions)
  - 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++:

```
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++?

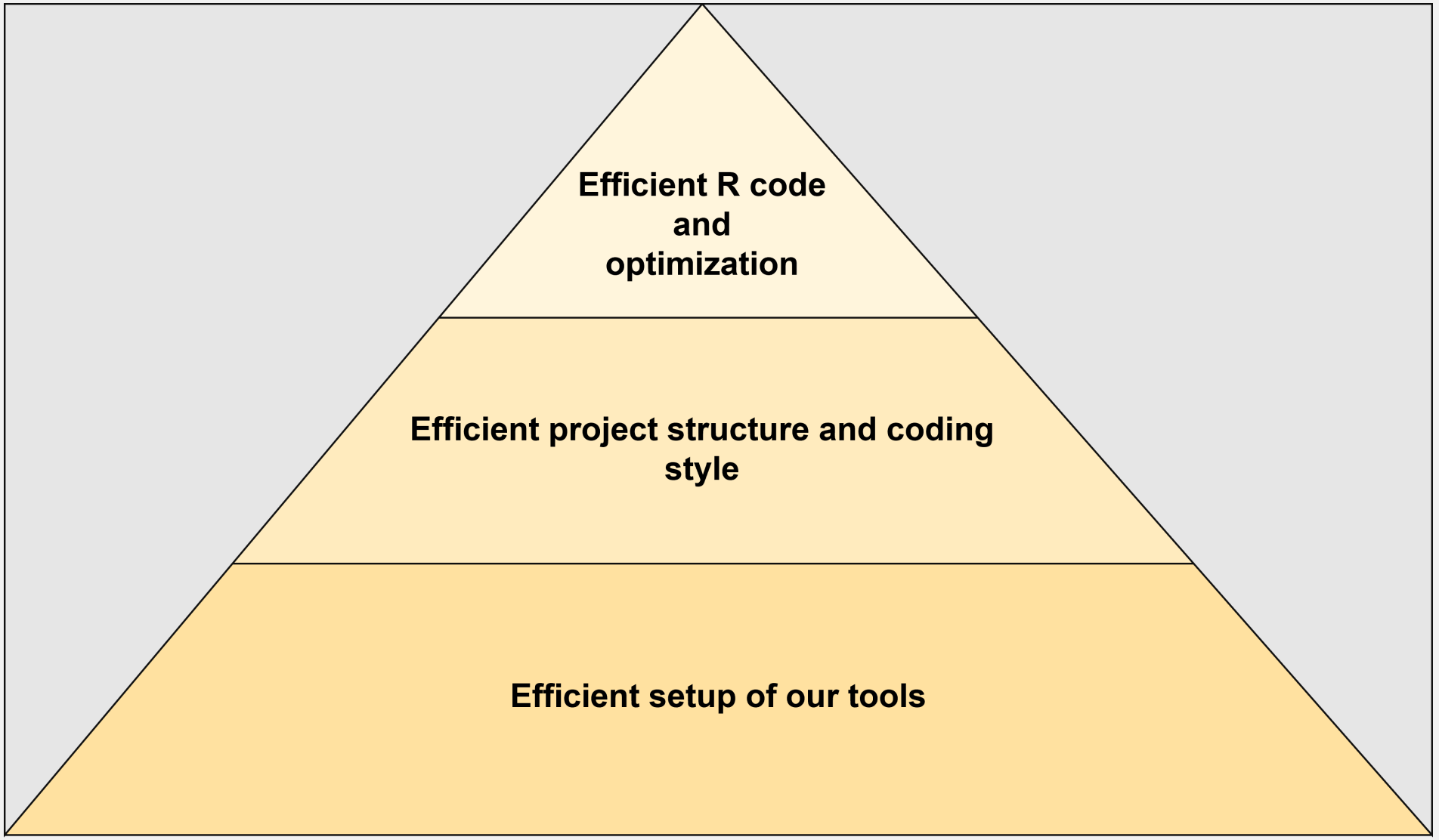
```
compare_rcpp <- microbenchmark(
  r = fibonacci_r(30),
  rcpp = fibonacci_cpp(30),
  times = 10
)

summary(compare_rcpp, unit = "relative")
#>   expr      min       lq     mean   median       uq      max neval cld
#> 1    r 457.9179 462.7536 448.4766 465.1011 467.8475 365.7407    10  a
#> 2 rcpp   1.0000   1.0000   1.0000   1.0000   1.0000   1.0000    10   b
```

# Summary

# Efficient R code and optimization

- Find your code bottlenecks and optimize those
  - Use `profvis`, `microbenchmark`, and `tictoc`
- Many quick wins, e.g.
  - Use `fread` instead of `read.csv`
  - Use `future_apply/future_map` instead of `apply/map`
  - Replace some tidyverse/base R functions with collapse functions
- If necessary: Advanced optimization techniques (e.g. C++ integration)
- For memory bottlenecks:
  - Consider data bases together with `dbplyr`
  - Use `arrow` to work on larger-than-RAM datasets

# Summary


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

# 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

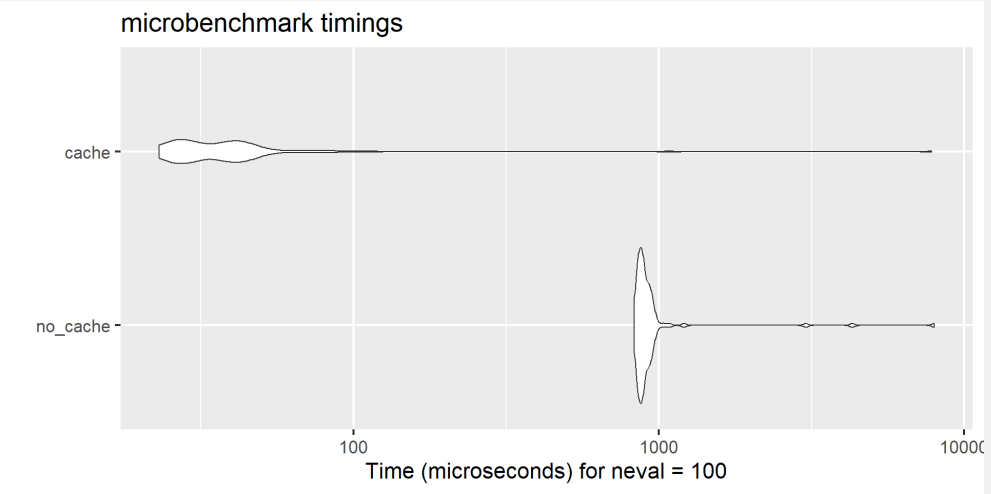
**Example**: Create a plot on a subset of the `iris` data set

```
# 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

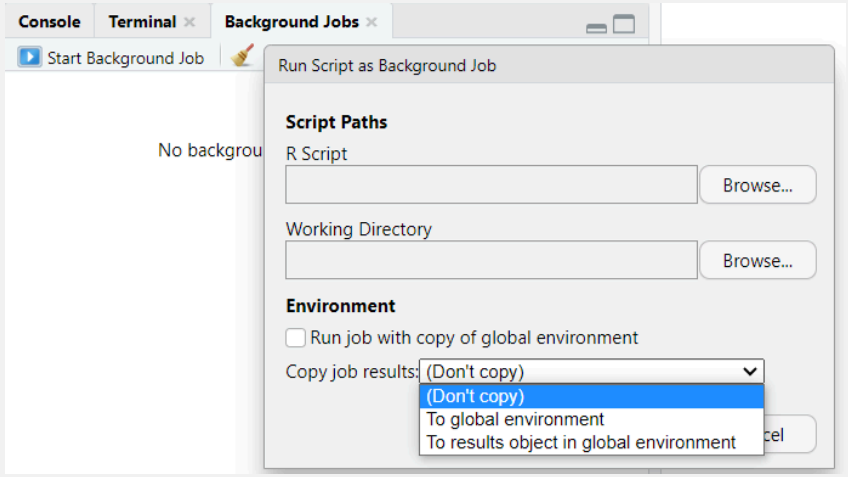
```
# Compare the two versions
result <- microbenchmark(
  no_cache = select_iris_species(10),
  cache = select_iris_species_mem(10)
)

autoplot(result)
```



# Run the code somewhere else

- Run it on a cluster
- Run your script in the background

## Use RStudio **background jobs**




## Start your R script from the command line

```
Rscript my_script.R
```