Should we parallelize?

PARALLEL PROGRAMMING IN R

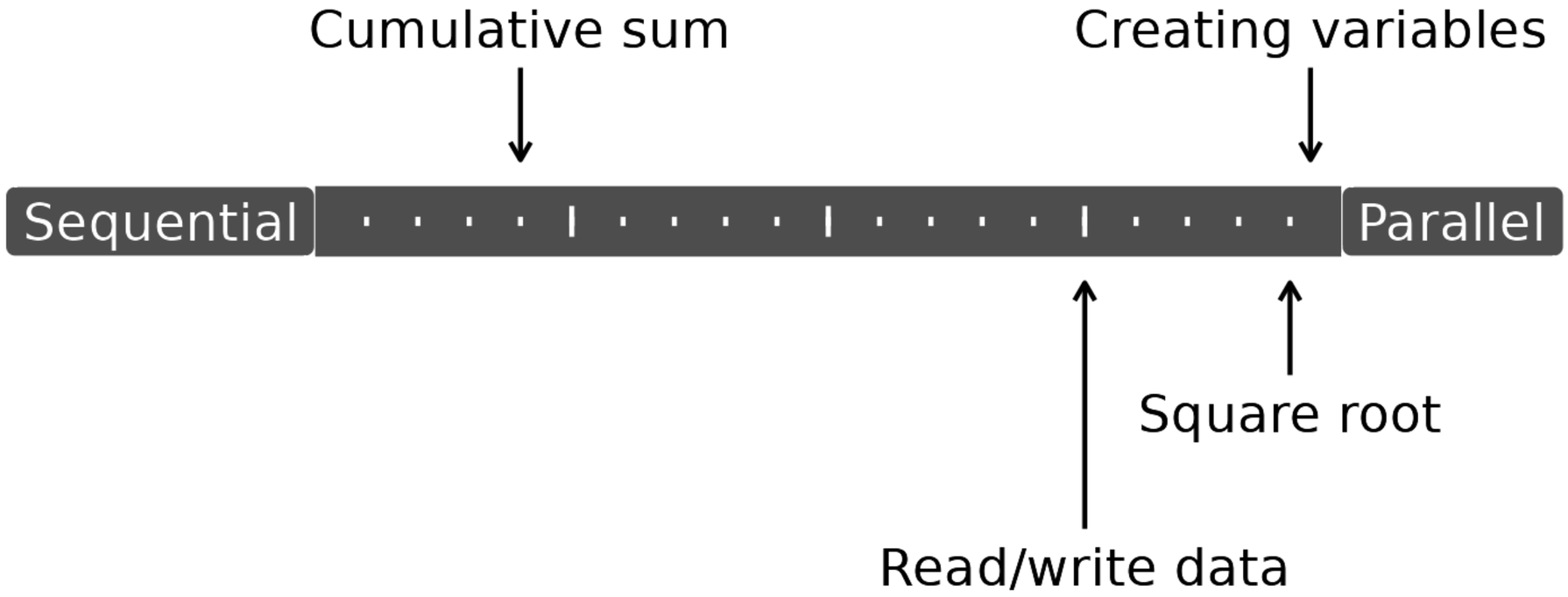
Nabeel Imam
Data Scientist

Let's construct a building

Building a floor on top of the last one:
sequential

Installing windows to finished structure:
parallel

The sequential-parallel scale



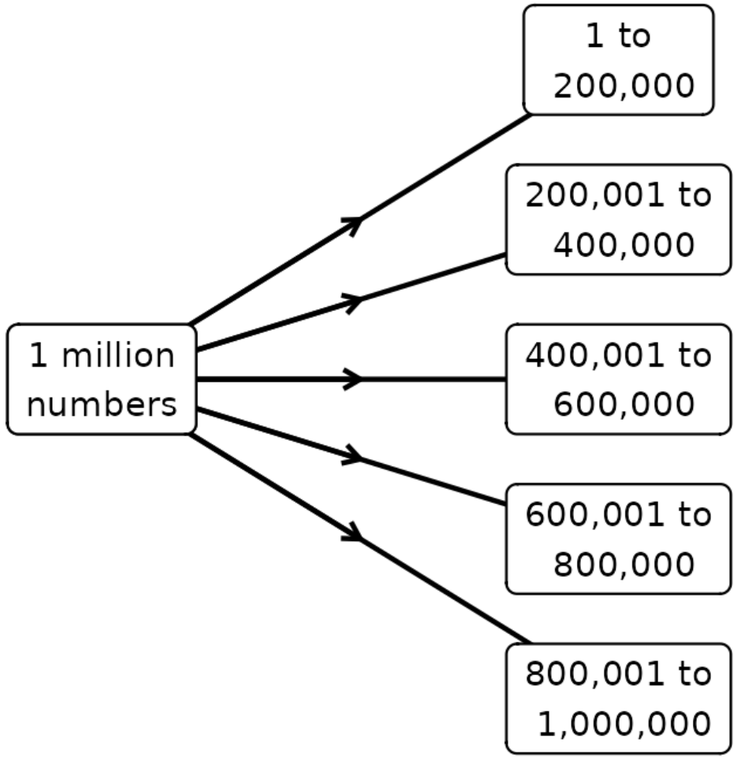
A classic numerical operation

Calculating the square roots of a million numbers

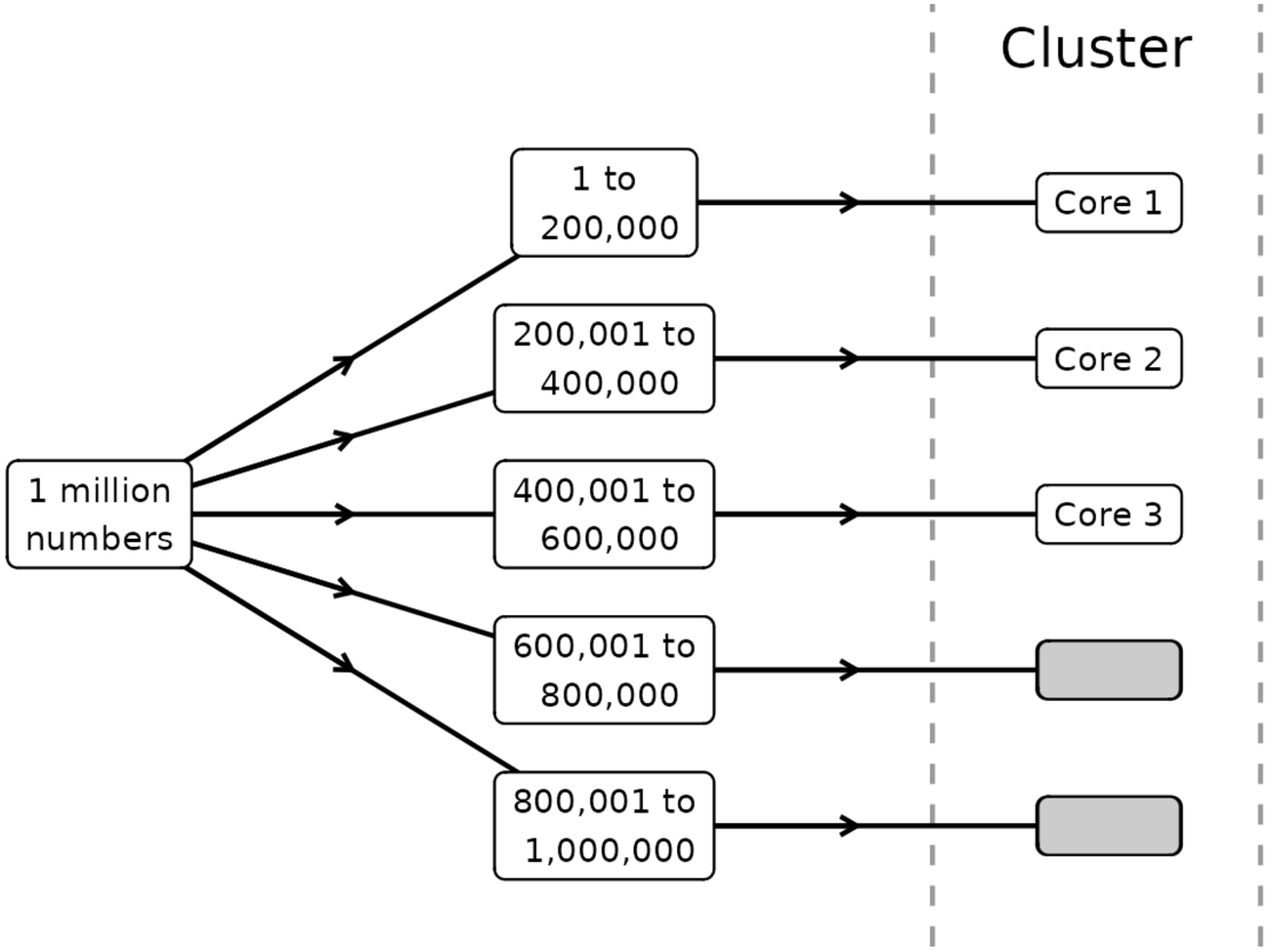
```
numbers <- 1:1000000

start <- Sys.time()
sq_roots <- lapply(numbers, sqrt)
end <- Sys.time()

end - start
```

```
Time difference of 1.044573 secs
```

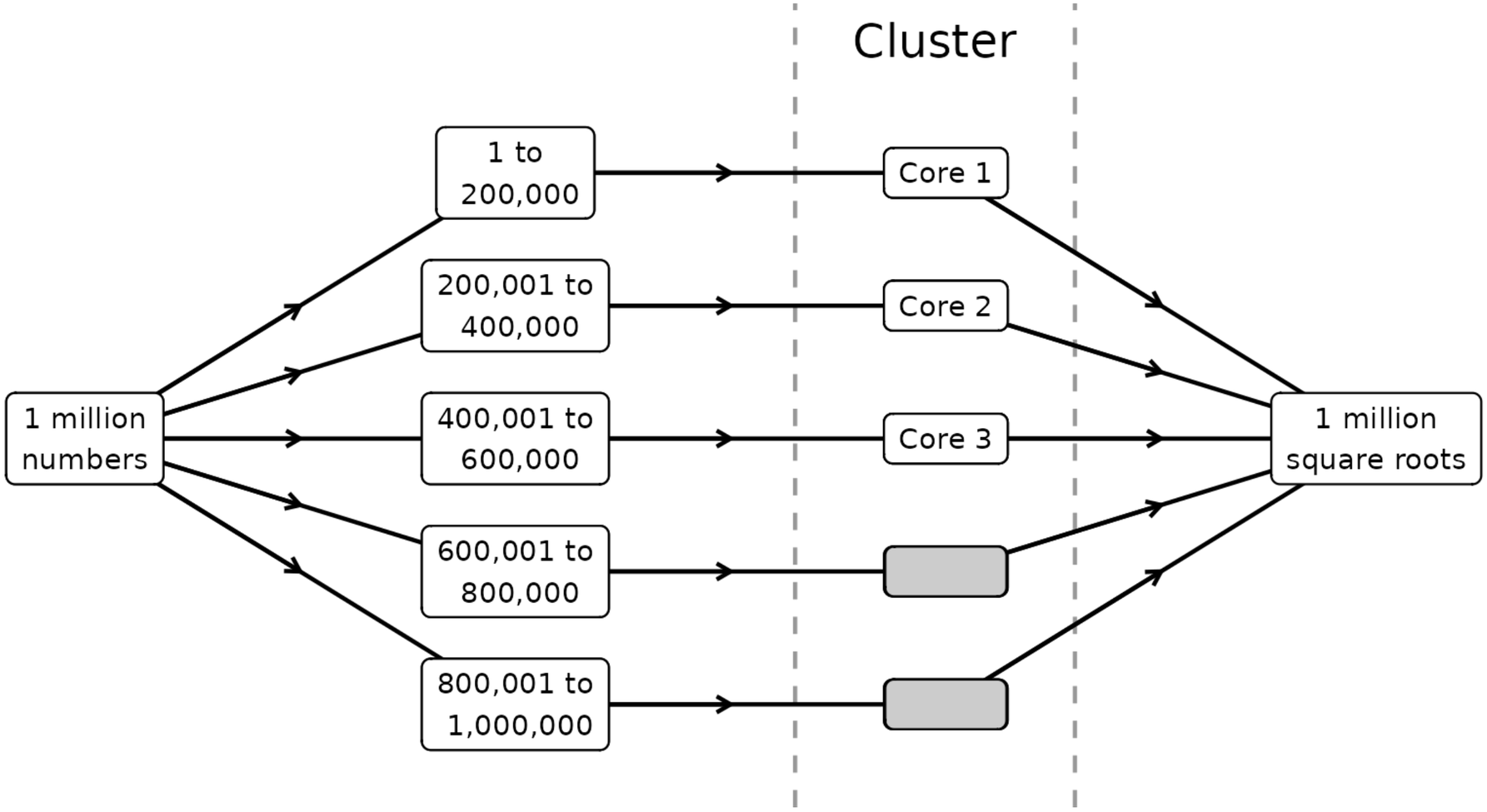
How could we parallelize the square root?



How could we parallelize the square root?



How could we parallelize the square root?



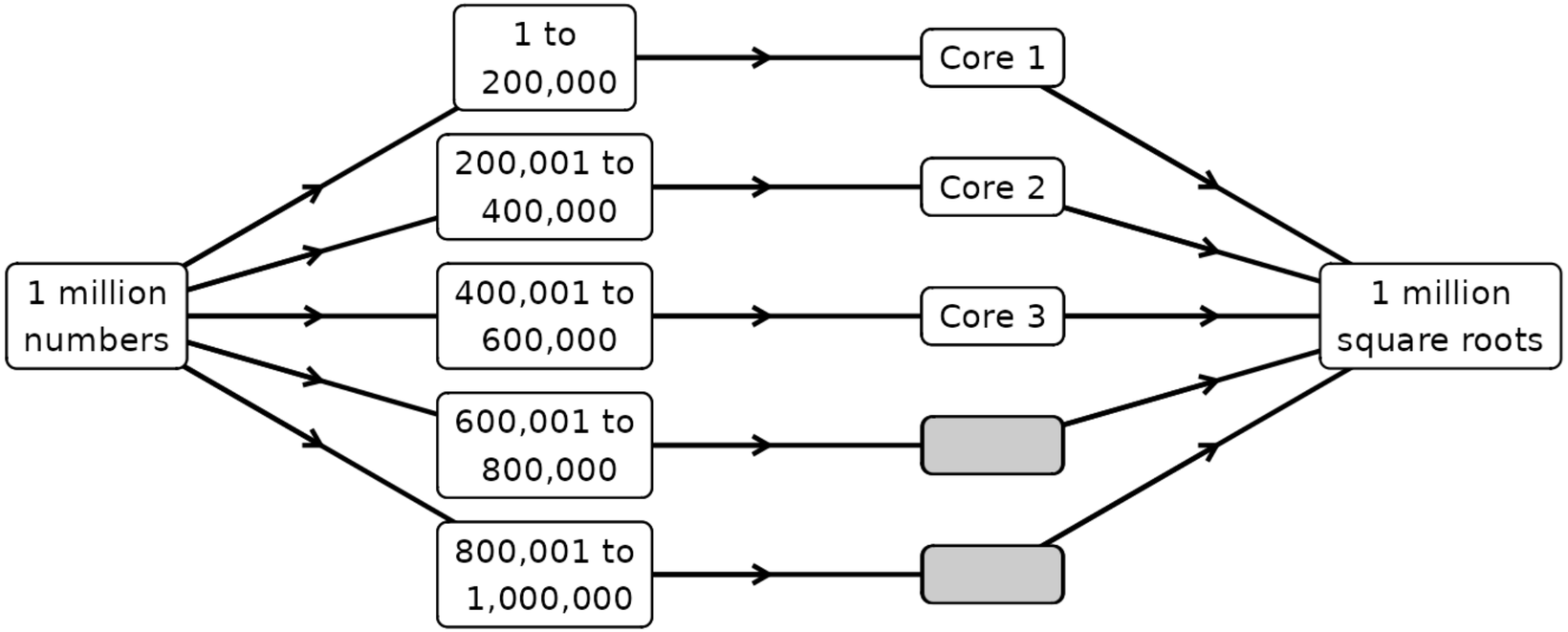
A parallelized numerical operation

The square roots of a million numbers in parallel

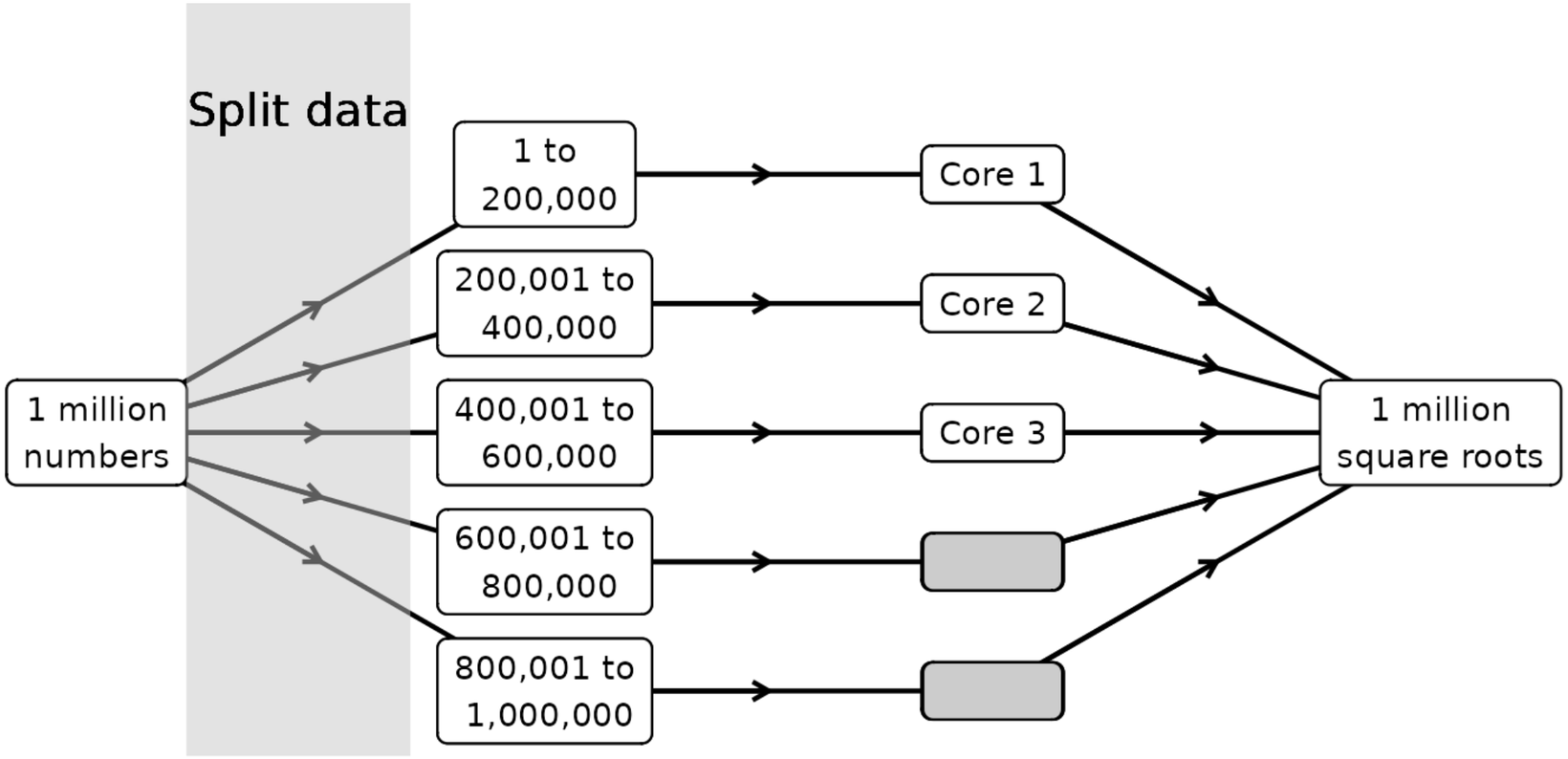
```
library(parallel)

my_cluster <- makeCluster(3)

start <- Sys.time()
sq_roots <- parLapply(my_cluster, numbers, sqrt)
end <- Sys.time()

stopCluster(my_cluster)

end - start
```

```
Time difference of 0.8416824 secs
```

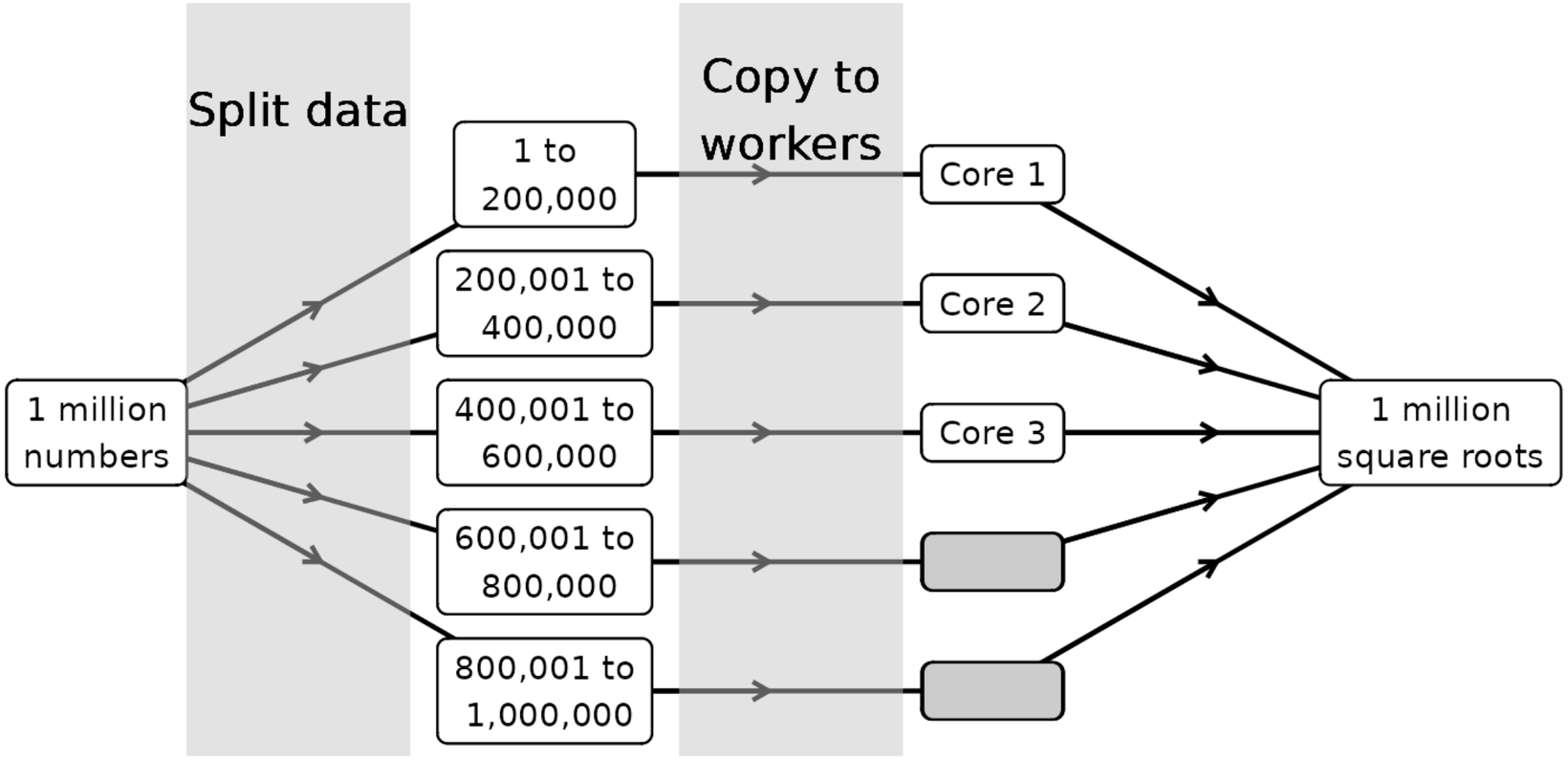
Not as fast as we expected



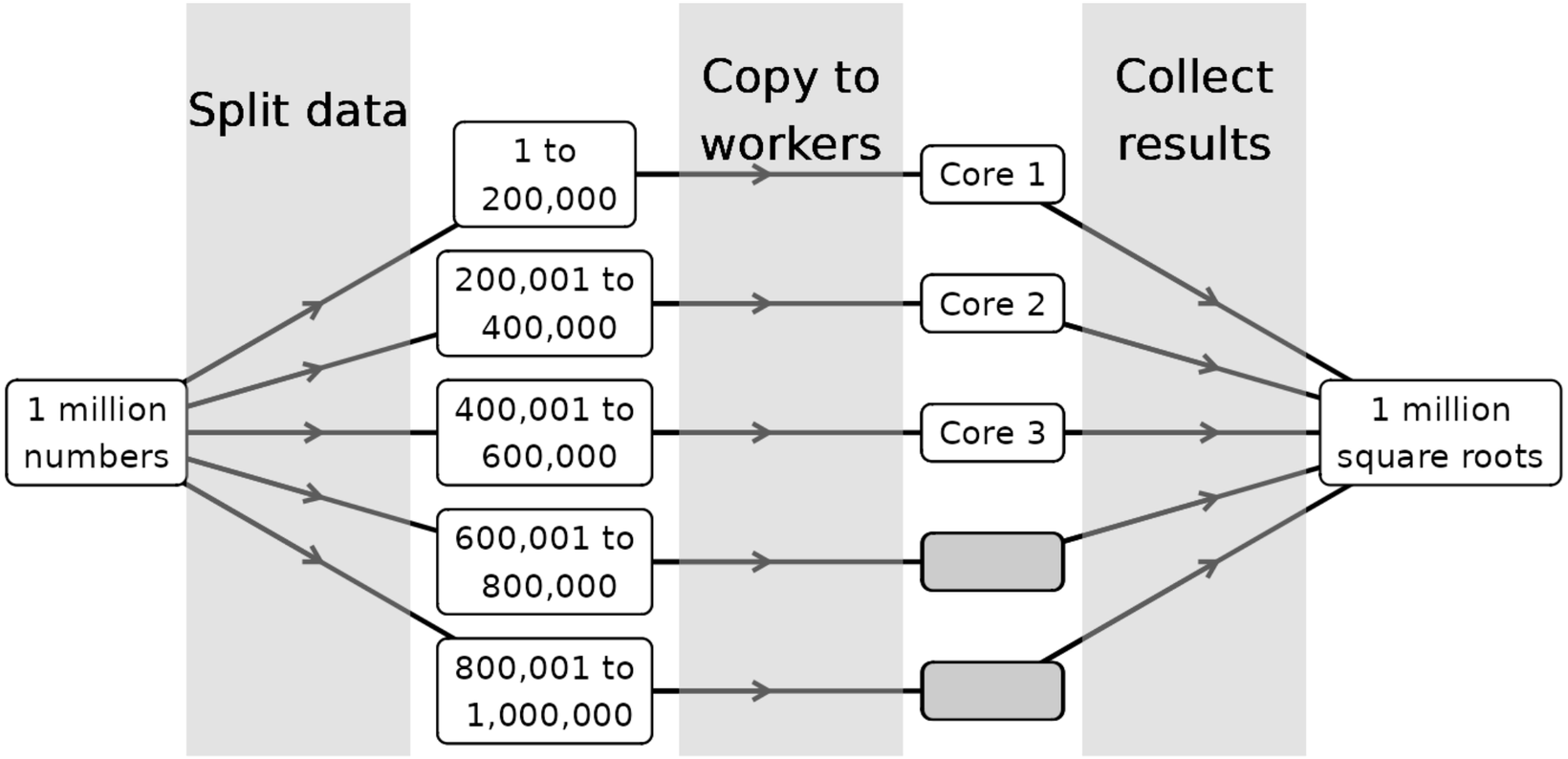
Not as fast as we expected



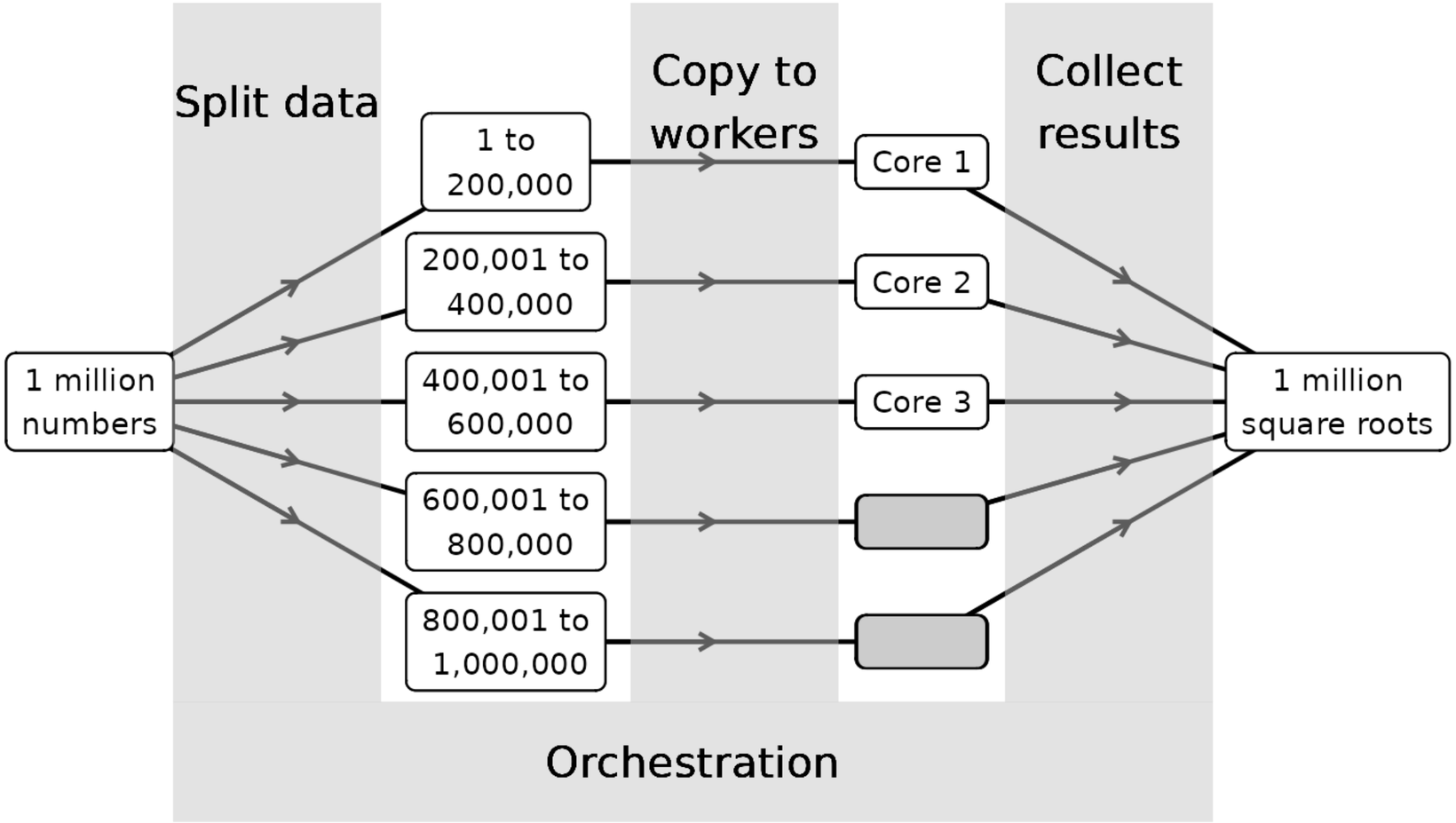
Not as fast as we expected



Not as fast as we expected



Not as fast as we expected



So, should we parallelize?

For a sufficiently complex task, consider:

Pros

- Faster than sequential
- More cost-efficient in the long run

Cons

- Requires special programming skills (but *you* are all set!)
- High memory usage

Let's practice!

PARALLEL PROGRAMMING IN R

Parallelization in R

PARALLEL PROGRAMMING IN R

Nabeel Imam
Data Scientist

datacamp

A practical example

The data

```
print(file_list)
```

```
 [1] "./uni_data_country/Argentina.csv"
 [2] "./uni_data_country/Armenia.csv"
 [3] "./uni_data_country/Australia.csv"
 [4] "./uni_data_country/Austria.csv"
 [5] "./uni_data_country/Azerbaijan.csv"
 [6] "./uni_data_country/Bahrain.csv"
 [7] "./uni_data_country/Bangladesh.csv"
 [8] "./uni_data_country/Belarus.csv"
 [9] "./uni_data_country/Belgium.csv"
[10] "./uni_data_country/Bolivia.csv"
...
```

Add a column

```
for (file in file_list) {

  df <- read.csv(file)

  df$top100 <- NA

  for (r in 1:nrow(df)) {
    df$top100[r] <- df$world_rank[r] <= 100
  }

  write.csv(df, file)
}
```

Profiling

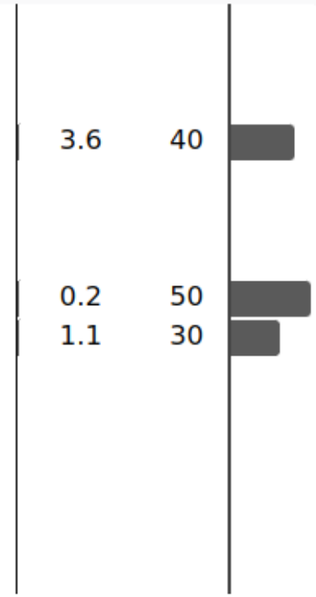
Code

```
library(profvis)

profvis({
  for (file in file_list) {

    df <- read.csv(file)
    df$top100 <- NA

    for (r in 1:nrow(df)) {
      df$top100[r] <- df$Rank[r] <= 100
    }
    write.csv(df, file)
  }
})
```

Output

Flame Graph | Data

<expr>	Memory	Time
1 profvis({		
2 for (file in file_list) {		
3		
4 df <- read.csv(file)	3.6	40
5		
6 df$top100 <- NA		
7		
8 for (r in 1:nrow(df)) {	0.2	50
9 df$top100[r] <- df$Rank[r] <= 100	1.1	30
10 }		
11		
12 write.csv(df, file)		
13 }		
14 })		
15		



Let's parallelize

The loop

```
for (file in file_list) {

  df <- read.csv(file)
  df$top100 <- NA

  for (r in 1:nrow(df)) {
    df$top100[r] <- df$Rank[r] <= 100
  }
  write.csv(df, file)
}
```

Function

```
add_col <- function(file_path) {

  df <- read.csv(file_path)
  df$top100 <- NA

  for (r in 1:nrow(df)) {
    df$top100[r] <- df$Rank[r] <= 100
  }
  write.csv(df, file_path)
}

cl <- makeCluster(6)
dummy <- parLapply(cl, file_list, add_col)
stopCluster(cl)
```

Practical considerations: number of cores

Detecting cores

```
detectCores()
```

```
[1] 8
```

Parallelized code

```
cl <- makeCluster(detectCores() - 2)

dummy <- parLapply(cl, file_list, add_col)

stopCluster(cl)
```

Practical considerations: cluster type

PSOCK cluster (default)

```
cl <- makeCluster(detectCores() - 2)
```

- Creates copies of current R session
- Cores do not share memory
- Works on any OS (Windows, Mac, Linux)

FORK cluster

```
cl <- makeCluster(detectCores() - 2,
                  type = "FORK")
```

- Creates subprocesses from R session
- Cores share memory (faster than PSOCK)
- Does not work on Windows

Let's exercise!

PARALLEL PROGRAMMING IN R

Measuring the benefits

PARALLEL PROGRAMMING IN R

Nabeel Imam
Data Scientist

Toy example

```
numbers <- 1:1000000

# Sequential
sqroots <- lapply(numbers, sqrt)

# Parallel
cl <- makeCluster(4)
sqroots <- parLapply(cl, numbers, sqrt)
stopCluster(my_cluster)
```

Which will perform better?

Benchmarking performance

Run code several times to estimate average execution time

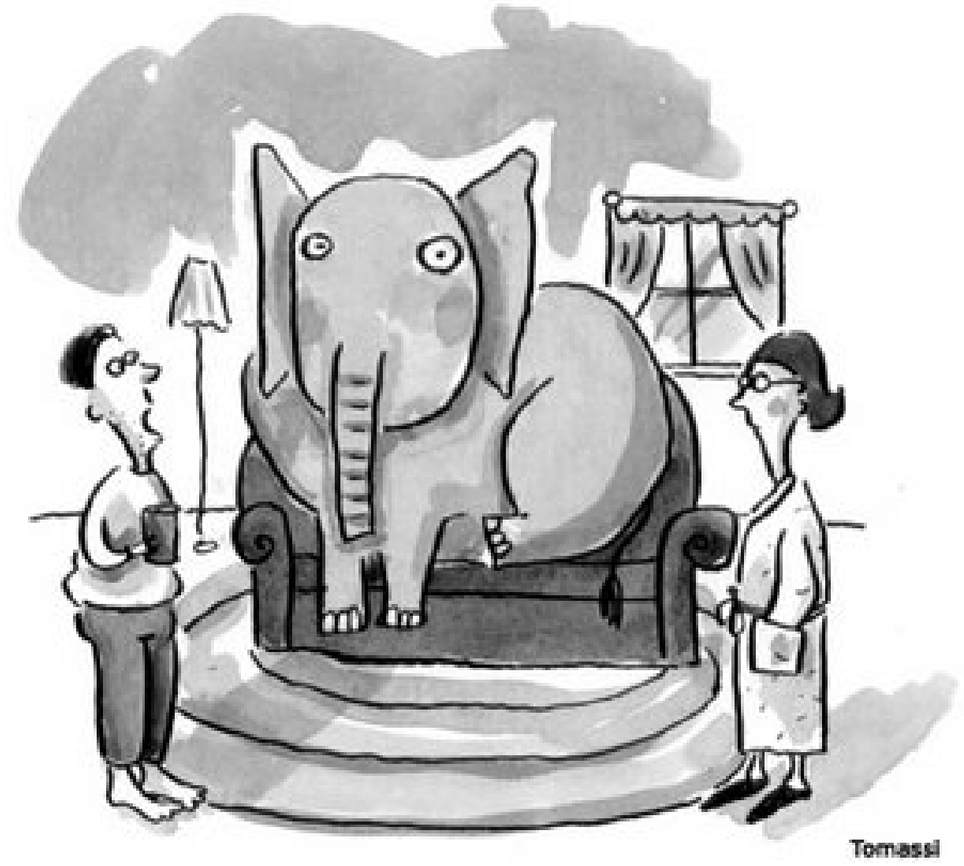
```
library(microbenchmark)

microbenchmark(
  "Sequential" = lapply(numbers, sqrt),
  "Parallel" = {
    cl <- makeCluster(4)
    parLapply(cl, numbers, sqrt)
    stopCluster(my_cluster)
  },
  times = 10
)
```

```
Unit: milliseconds
       expr     min    mean     max neval
 Sequential  633.96  838.09  993.59    10
   Parallel 1136.95 1247.29 1557.58    10
```

- Simple numerical operations rarely benefit from parallelization
- Profiling gives line-by-line report, benchmarking gives overall execution times

The elephant in the room

```
sqroots <- sqrt(numbers)
```



Vectorization

```
sqroots <- sqrt(numbers)
```

- Base R functions, like `sqrt()` , are vectorized.
- Map a single function to many inputs
- Very fast but only applicable to simple operations

```
microbenchmark(
  "Vectorized" = sqrt(numbers),
  "Sequential" = lapply(numbers, sqrt),
  "Parallel" = {
    cl <- makeCluster(4)
    parLapply(cl, numbers, sqrt)
    stopCluster(my_cluster)
  },
  times = 10)
```

```
Unit: milliseconds
       expr       min       mean       max neval
 Vectorized    2.3904    9.2071   66.303    10
 Sequential  352.1166  771.7491 1004.753    10
   Parallel 1191.3176 1377.6926 1700.316    10
```

The bootstrap

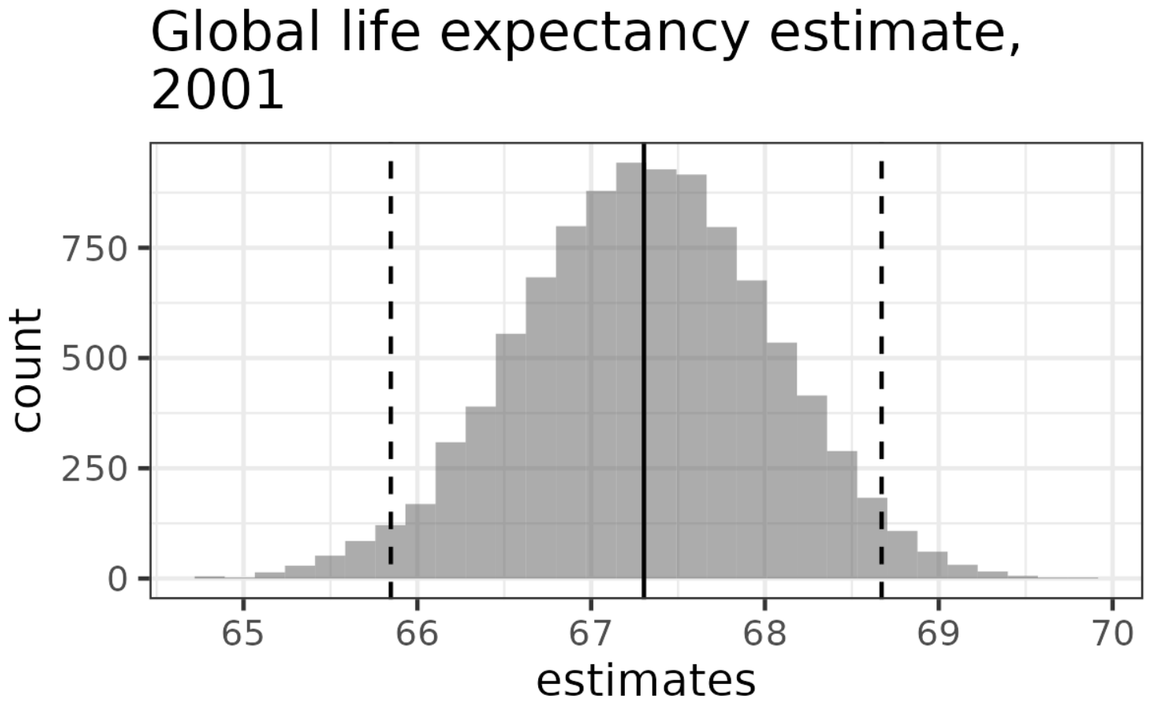
Sampling from the current data with replacement

```
print(ls_df)
```

```
$`2001`
   Country              Life_expectancy  Year
 1 Afghanistan                     56.3  2001
 2 Albania                         74.3  2001
 3 Algeria                         71.1  2001
...
$`2002`
   Country              Life_expectancy  Year
 1 Afghanistan                     56.8  2002
 2 Albania                         74.6  2002
 3 Algeria                         71.6  2002
...
```

Classic version

```
df <- ls_df$`2001`

estimates <- rep(0, 10000)

for (i in 1:10000) {

  b <- sample(df$Life_expectancy,
              replace = T)

  estimates[i] <- mean(b)
  }
```



- Confidence interval using quantiles:

```
quantile(estimates, c(0.025, 0.975))
```

The good news

Bootstraps can be parallelized

```
estimates <- rep(0, 10000)

for (i in 1:10000) {

  b <- sample(df$Life_expectancy,
              replace = T)

  estimates[i] <- mean(b)
  }
```

```
boot_dist <- function (df) {

  estimates <- rep(0, 10000)

  for (i in 1:10000) {
    b <- sample(df$Life_expectancy, replace = T)
    estimates[i] <- mean(b)
  }

  return(estimates)
}

cl <- makeCluster(4)
ls_dists <- parLapply(cl, ls_df, boot_dist)
stopCluster(cl)
```

The benefits

```
microbenchmark(
  "lapply" = lapply(ls_df, boot_dist),
  "parLapply" = {
    cl <- makeCluster(4)
    parLapply(cl, ls_df, boot_dist)
    stopCluster(cl)
  },
  times = 10
)
```

```
Unit: seconds
     expr    min   mean    max neval
   lapply 3.6938 4.2184 4.5267    10
parLapply 1.9006 2.5166 2.7292    10
```

How to get there:

- Profile existing code, identify slowest part
- Parallelize/optimize this step
- Benchmark and compare

Let's practice!

PARALLEL PROGRAMMING IN R