Lecture 22: Profiling and microbenchmarking

An order of operations for programming

1. Make it run
2. Make it right
3. Make it fast

When speed matters

- You are working with very large data
- You are running a process (a simulation, a data analysis, etc.) many times
- A piece of code will be called many times (e.g., choosing a split in a decision tree)

Goals

- Learn how to identify bottlenecks in code
- Learn approaches for more efficient code in R
- Time permitting: learn how to use C++ to make code faster

Example: timing code

Suppose we want to compute the mean of each column of a data frame:

```
n <- 100000
cols <- 150
data_mat <- matrix(rnorm(n * cols, mean = 5), ncol = cols)
data <- as.data.frame(data_mat)

means <- rep(NA, cols)
for(i in 1:cols){
  means[i] <- mean(data[,i])
}
```

Example: timing code

Suppose we want to compute the mean of each column of a data frame:

```
n <- 100000
cols <- 150
data_mat <- matrix(rnorm(n * cols, mean = 5), ncol = cols)
data <- as.data.frame(data_mat)
                                  timing   code
system.time({
  means <- rep(NA, cols)
  for(i in 1:cols){
    means[i] <- mean(data[,i])
  }
})
```

```
 user  system elapsed        (in seconds)
1.930   0.017   1.960
```

Alternatives

```
means <- rep(NA, cols)
for(i in 1:cols){
  means[i] <- mean(data[,i])
}
```

What are the alternatives to this for-loop approach?

colMeans : function for computing column means

apply : apply a function to the margins of a matrix or data frame

apply(data, 2, mean)

Alternatives

```
# Option 1: for loop
for_loop_means <- function(data){
  cols <- ncol(data)
  means <- rep(NA, cols)
  for(i in 1:cols){
    means[i] <- mean(data[,i])
  }
  return(means)
}
means <- for_loop_means(data)

# Option 2: apply
means <- apply(data, 2, mean)

# Option 3: colMeans
means <- colMeans(data)
```

Comparing performance

Microbenchmarking: Evaluating the performance of a small piece of code

```
bench::mark(
  means <- for_loop_means(data),
  means <- apply(data, 2, mean),
  means <- colMeans(data),
  check = F
)
```

comparing three different approaches

```
# A tibble: 3 × 6
  expression                              min   median `itr/sec` mem_alloc
`gc/sec`
  <bch:expr>                         <bch:tm> <bch:tm>     <dbl> <bch:byt>
<dbl>
1 means <- for_loop_means(data)         1.93s    1.93s     0.519    1.85KB
0
2 means <- apply(data, 2, mean)         2.03s    2.03s     0.493  400.57MB
0.987
3 means <- colMeans(data)             461.4ms 469.34ms     2.13   114.45MB
1.07
```

similar performance

least memory used

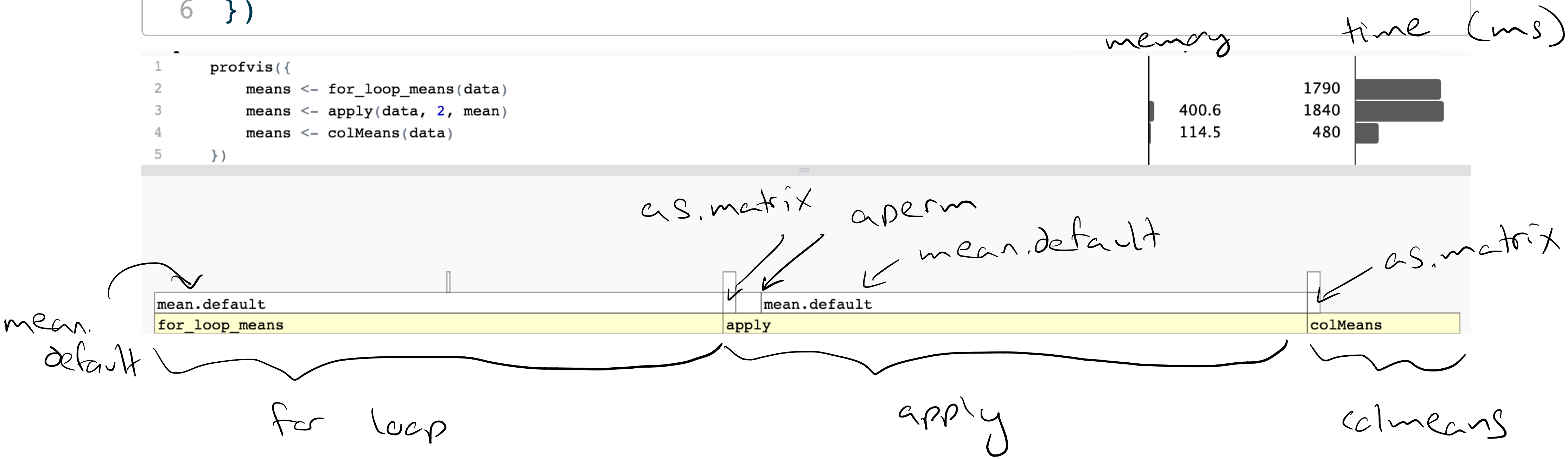
most memory used

about 4 times faster

time it took the code to run once

Profiling

```r
library(profvis)
profvis({
  means <- for_loop_means(data)
  means <- apply(data, 2, mean)
  means <- colMeans(data)
})
```



Space for efficiency increases?

```
colMeans
```

Checking inputs, doing some transformations

```
function (x, na.rm = FALSE, dims = 1L)
{
    if (is.data.frame(x))
        x <- as.matrix(x)
    if (!is.array(x) || length(dn <- dim(x)) < 2L)
        stop("'x' must be an array of at least two dimensions")
    if (dims < 1L || dims > length(dn) - 1L)
        stop("invalid 'dims'")
    n <- prod(dn[id <- seq_len(dims)])
    dn <- dn[-id]
    z <- if (is.complex(x))
        .Internal(colMeans(Re(x), n, prod(dn), na.rm)) + (0+1i) *
            .Internal(colMeans(Im(x), n, prod(dn), na.rm))
    else .Internal(colMeans(x, n, prod(dn), na.rm))
```

Increase efficiency by avoiding extraneous steps

```r
n <- 100000
cols <- 150
data_mat <- matrix(rnorm(n * cols, mean = 5), ncol = cols)
data <- as.data.frame(data_mat)

bench::mark(
  means <- colMeans(data_mat),
  means <- colMeans(data),
  check = F
)
```

matrix

data frame

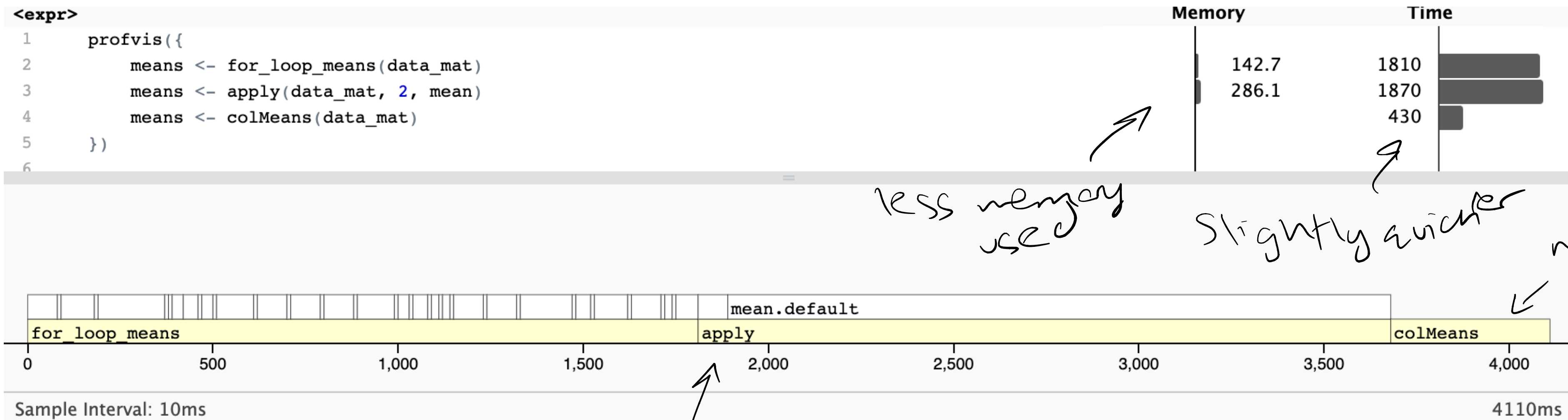
```
# A tibble: 2 × 6
  expression                              min   median `itr/sec` mem_alloc
`gc/sec`
  <bch:expr>                         <bch:tm> <bch:tm>     <dbl> <bch:byt>
<dbl>
1 means <- colMeans(data_mat)           437ms    437ms      2.29    1.22KB
0
2 means <- colMeans(data)               453ms    453ms      2.21  114.45MB
2.21
```

less memory used

slightly faster

Profiling

```r
profvis({
  means <- for_loop_means(data_mat)
  means <- apply(data_mat, 2, mean)
  means <- colMeans(data_mat)
})
```



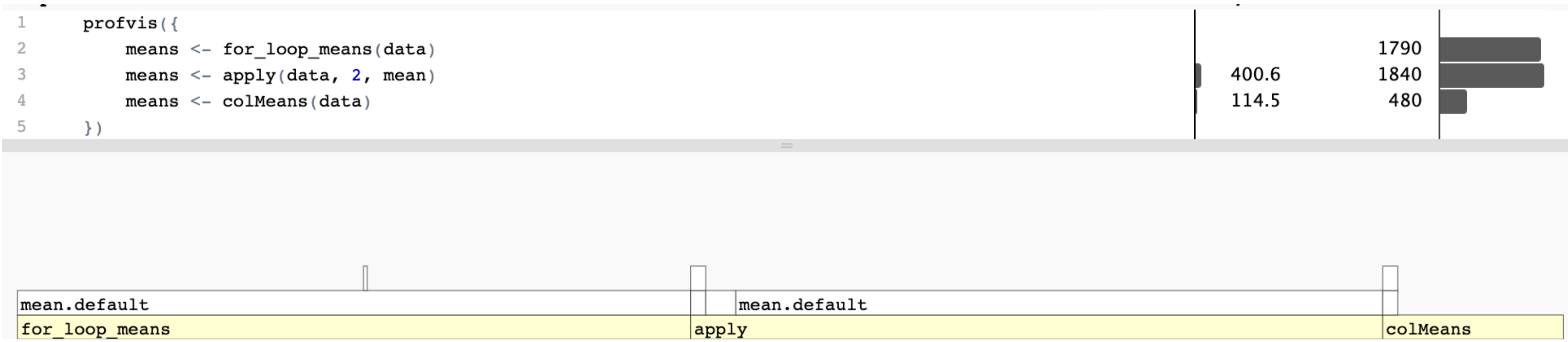
less memory used

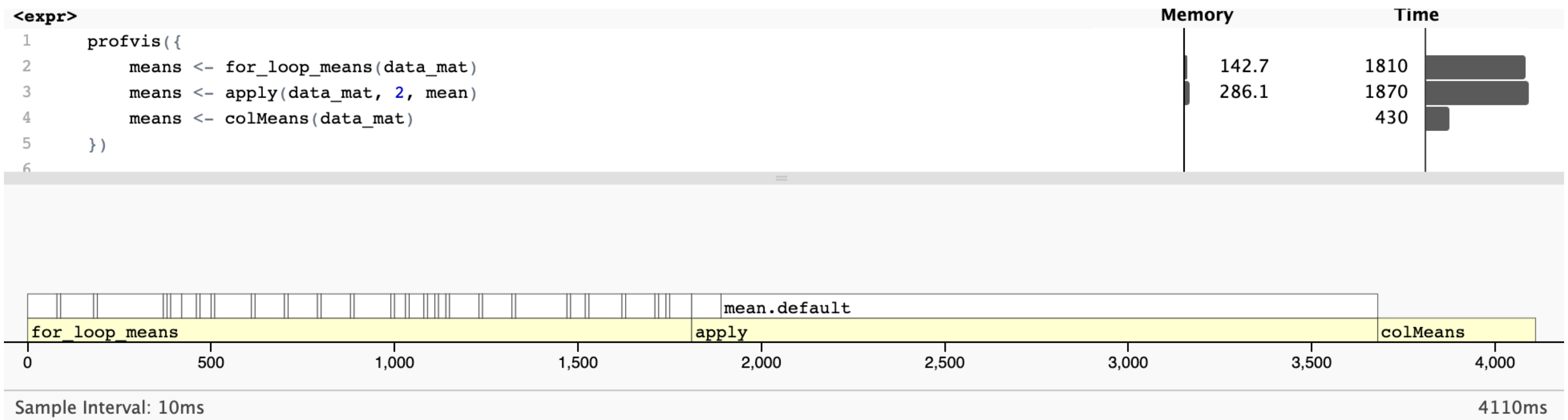
Slightly quicker

no data frame conversion

no data frame conversion

Profiling


profvis({
means <- for_loop_means(data)
means <- apply(data, 2, mean)
means <- colMeans(data)
})
1790
400.6
1840
114.5
480
mean.default
mean.default
for_loop_means
apply
colMeans


<expr>
Memory
Time
profvis({
means <- for_loop_means(data_mat)
means <- apply(data_mat, 2, mean)
means <- colMeans(data_mat)
})
142.7
1810
286.1
1870
430
mean.default
for_loop_means
apply
colMeans
0
500
1,000
1,500
2,000
2,500
3,000
3,500
4,000
Sample Interval: 10ms
4110ms

Class activity

https://sta279-f23.github.io/class_activities/ca_lecture_24.html