Improving Performance — Notes and Examples

Slide 6: 2. Do as Little as Possible

Exercise: coercion of inputs / robustness checks

```
X <- matrix(1:1000, ncol = 10)
Y <- as.data.frame(X)

bench::mark(
  apply(X, 1, sum),
  apply(Y, 1, sum)
)
```

- `apply()` accepts various inputs and outputs which requires coercion which is slow.
- `apply()` coerces `Y` to matrix which triggers a *copy*. You can check this using `lobstr::tracemem()`.

Slide 7: 2. Do as Little as Possible

Exercise: coercion of inputs / robustness checks — ctd.

```
bench::mark(
  rowSums(X),
  apply(X, 1, sum)
)
```

- Using `apply()` yields a much longer call stack than `rowSums()` which is much more specific and hence faster.
- Also note that `rowSums()` is a wrapper for faster internal functions. This is seen from the source code.

Slide 8: 2. Do as Little as Possible

Exercise: Searching a vector

```
x <- 1:100

bench::mark(
  any(x == 10),
  10 %in% x
)
```

Testing equality (using `any()`) is faster than testing inclusion in a set with `%in%`.

Slide 9: 2. Do as Little as Possible

Exercise: Linear Regression — computation of $SE(\widehat{\beta})$

- `a()` is what we teach undergraduates which is totally fine — except if you want (only) $SE(\widehat{\beta})$ and fast.
  - `a()` runs lot of interpretation and robustness checks and produces a long call stack.
  - also note that many (in this case superfluous) components are computed

- `b()` is rather focused on the essentials but is also less flexible.

Let's compare both approaches in a microbenchmark.

```
bench::mark(
  a(),
  b()
)
```

The difference is indeed huge!

Slide 12: Exercises

Solutions

1. Can you come up with an even faster implementation of `b()` in the linear regression example?

   ```
   d <- function() {
    fit <- .lm.fit(X, Y)
    sqrt(1/(nrow(X)-1) * sum(fit$residuals^2) * 1/sum(X^2))
   }
   ```

   - `.lm.fit(X, Y)` is a 'bare bones wrapper' for the innermost C code of `lm()` which computes OLS using QR decomposition
   - Exploiting that `X` is the only regressor allows us to replace the regressor matrix product and use a more specific approch to compute $SE(\widehat{\beta}_1)$ in `d()`

   ```
   bench::mark(
     a(),
     b(),
     d()
   )
   ```

   A disadvantage is that the faster approaches `b()` and `d()` are unflexible and error prone: an unexperienced user is likely to supply inputs which will cause the function to crash or return false results–note that we don't do any checks of the input!

2. What's the difference between `rowSums()` and `.rowSums()`?

   `rowSums()` is a wrapper for the `.rowSums()`, an internal C function. `rowSums()` does robustness checks and performs coercion before calling `.rowSums()`

3. `rowSums2()` is an alternative implementation of `rowSums()`. Is it faster for the input `df`? Why?

   ```
   rowSums2 <- function(df) {
     out <- df[[1L]]
     if (ncol(df) == 1) return(out)
       for (i in 2:ncol(df)) {
         out <- out + df[[i]]
       }
     out
   }

   df <- as.data.frame(
     replicate(1e3, sample(100, 1e4, replace = TRUE))
   )

   bench::mark(
     rowSums2(df),
   ```

```
  rowSums(df)
)
```

- Note that `rowSums()` converts the data frame to a matrix (ensuring all types are the same) and *handles more than two dimensions* and names
- For two-dimensional dataframes where we don't care about names, `rowSums2()` will be faster than `rowSums()`

Slide 21: Vectorise your Code

Example: Avoid growing objects

```
# grow
vec <- numeric(0)
for(i in 1:n) vec <- c(vec, i)

# fill
vec <- numeric(n)
for(i in 1:n) vec[i] <- i

# primitive
vec <- 1:n
```

Technically this does not directly relate to vectorisation but it yet again demonstrates that growing objects using loops is a bad idea: a vectorised approach is often faster.

Slide 27: Vectorise your Code — Exercises

Solutions:

1. Compare the speed of `apply(X, 1, sum)` with the vectorised `rowSums(X)` for varying sizes of the square matrix `X` using `bench::mark()`. Consider the dimensions `1`, `1e1`, `1e2`, `1e3`, `0.5e4` and `1e5`. Visualize the results using a violin plot.

   We compare different sizes of square matrices

   ```
   library(ggplot2)
   b <- bench::press(
     dim = c(1, 1e2, 1e3, 0.5e4, 1e4),

     {
       X <- matrix(runif(dim*dim), ncol = dim)

       bench::mark(
         apply(X, 1, sum),
         rowSums(X),
         relative = T
       )

     }

   )

   plot(b)
   ```

   Note that `apply()` which is not 'vectorised for performance' cannot keep up with `rowSums()`: it is clearly outperformed by the *C* internals, especially if dimensions are large. This is because the `rowSums()` and it's internal *C* functions need less (costly) function calls and is more memory efficient.

2. (a) We may simply use `sum()` here:

```
a <- rnorm(100)
w <- rnorm(100)

sum(a * w)
```

   (b) `crossprod()` computes the dot product which is also a weighted sum:

```
sum(a * w) - crossprod(a, w)[1]
```

   We subset because the return value is a matrix, see`?crossprod`.

   (c) Let's benchmark these guys:

```
res <- bench::press(
  dim = c(1, 1e2, 1e3, 0.5e4, 1e4, 1e5, 1e6),

  {
    a <- rnorm(dim)
    w <- rnorm(dim)

    bench::mark(
     sum(a * w),
     crossprod(a, w),
     check = F,
      relative = T
    )

  }

)
```

   - There's a turning point at `dim = 0.5e4` where `crossprod()` takes the lead: this is due to the advantage of vectorised matrix computation done by the internal function used by `crossprod()`.
   - Also `sum(a * w)` triggers garbage collection as dimensions increase.

3. One way is to use `split()`.

```
X <- matrix(rnorm(1000),
            ncol = 100)

X_list <- split(X, col(X))

bench::mark(
sapply(X_list, max),
apply(X, 2, max),
check = F
)
```

Applying `max()` on list elements is faster than iterating over the columns of a numeric matrix.