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.

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 wrapper for the innermost C code of `lm()` which computes OLS using QR decomposition
   - Exploiting that `X` is the only regressor allows us to use a more specific formula for $SE(\widehat{\beta}_1)$

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

   A disadvantage is that the fastest approach `d()` is error prone: an unexperienced user is likely to supply inputs which will cause the function to crash.

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

   `rowSums()` is a wrapper for the internal `.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 same-type data frames where we don't care about names, rowSums2 will be faster

Slide 13: 2. Do as Little as Possible — Case Study

```
n <- 1e6
df <- data.frame(a = rnorm(n), b = rnorm(n))

cor_df <- function(df, n) {
  i <- sample(seq(n), n, replace = TRUE)
  cor(df[i, , drop = FALSE])[2, 1]
}
```

Solution

- `:` is a primitive and is faster than `seq()`
- `sample.int(n, n)` is more specific (an thus faster) than `sample()`
- Passing vectors using `$` is faster than look-up of the correct [method
- `cor()` runs faster on vectors than on a matrix

We thus end up with:

```
cor_df2 <- function(x, n) {
  i <- sample.int(n, n, replace = T)
  cor(df$a[i], df$b[i])
}

bench::mark(
  cor_df(df, n),
  cor_df2(df, n),
  check = F
)
```

Slide 22: 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

```r
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.

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

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

sum(a * w)
```

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

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

   (c) Let's benchmark these guys:

```r
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 vectorized matrix computation done by the internal function used by `crossprod()`.

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

```r
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.

Slide 30: Vectorise your code — Case Study

Case Study: Monte Carlo Integration

1. (a)

```r
A_loop <- function(N) {
  counts <- numeric(N)
  for(i in 1:N) {
    U <- runif(2)
    if(U[2] < U[1]^2) counts[i] <- 1
  }
  sum(counts)/N
}
```

(b)

```r
A_loop(5e5)
```

2. (a)

```
```r
A_vec <- function(N) {
  U <- matrix(runif(2 * N), ncol = 2)
  sum(ifelse(U[,1] < U[,2]^2, 1, 0))/N
}
```
```

(b)

```r
bench::mark(
  A_loop(N),
  A_vec(N),
  check = F
)
```

(c)

```r
res <- bench::press(

  N = c(1e2, 1e3, 1e4, 1e5),

  {

  bench::mark(
    A_loop(N),
    A_vec(N),
    check = F
  )

  }

)

plot(res)
```

The difference in speed between both approaches does not seem to depend much on `N` (for large `N`).