Functional programming

Hadley Wickham

Chief Scientist

RStudio

August 2015

1. Motivation
2. Warmups
3. Functionals
4. Friends of `lapply()`
    1. Vector output
    2. Parallel computing

Motivation

DRY principle:
Don’t Repeat Yourself

Every piece of knowledge must have a single, unambiguous, authoritative representation within a system

Popularised by the “Pragmatic Programmers”

```
# Fix missing values
df$a[df$a == -99] <- NA
df$b[df$b == -99] <- NA
df$c[df$c == -99] <- NA
df$d[df$d == -99] <- NA
df$e[df$e == -99] <- NA
df$f[df$f == -99] <- NA
df$g[df$g == -98] <- NA
df$h[df$h == -99] <- NA
df$i[df$i == -99] <- NA
df$i[df$j == -99] <- NA
df$k[df$k == -99] <- NA
```

DRY principle prevents inconsistency

```
fix_missing <- function(x) {
  x[x == -99] <- NA
  x
}
df$a <- fix_missing(df$a)
df$b <- fix_missing(df$b)
df$c <- fix_missing(df$c)
df$d <- fix_missing(df$d)
df$e <- fix_missing(df$e)
df$f <- fix_missing(df$f)
df$g <- fix_missing(df$g)
df$h <- fix_missing(df$h)
df$h <- fix_missing(df$i)
df$j <- fix_missing(df$j)
df$k <- fix_missing(df$k)
```

More powerful abstractions lead to less repetition

```
fix_missing <- function(x) {
  x[x == -99] <- NA
  x
}

df[] <- lapply(df, fix_missing)
```

And easier generalisation

```
fix_missing <- function(x) {
  x[x == -99] <- NA
  x
}

numeric <- vapply(df, is.numeric, logical(1))
df[numeric] <- lapply(df[numeric], fix_missing)
```

And easier generalisation

```
missing_fixer <- function(missing) {
  function(x) {
    x[x == missing] <- NA
    x
  }
}

numeric <- vapply(df, is.numeric, logical(1))
df[numeric] <- lapply(df[numeric], missing_fixer(-99))
```

Warmups

Your turn

Why should you avoid for loops in R?

When should you prefer for loops?

```
exp_smooth <- function(x, alpha) {
  s <- numeric(length(x) + 1)
  for (i in seq_along(s)) {
    if (i == 1) {
      s[i] <- x[i]
    } else {
      s[i] <- alpha * x[i - 1] + (1 - alpha) * s[i - 1]
    }
  }
  s
}
```

```
i <- 0
while(TRUE) {
  if (runif(1) > 0.9) break
  i <- i + 1
}
```

```
# What does this code do?

trans <- list(
  disp = function(x) x * 0.0163871,
  am = function(x) {
    factor(x, levels = c("auto", "manual")
  })
)
for(var in names(trans)) {
  mtcars[[var]] <- trans[[var]](mtcars[[var]])
}
```

```r
# What's special about these function calls?

plot(sqrt)

integrate(sin, 0, pi)

f <- function(x) exp(x) * (x ^ 3 + 4 * x - 2)
uniroot(f, c(0, 1))
```

Your turn

What's the difference between `x[] <- y` and `x <- y` ?

```
x <- factor(c("a", "b", "c"))
x <- c("c", "b", "a")
```

```
x <- factor(c("a", "b", "c"))
x[] <- c("c", "b", "a")
```

Functionals

Functionals

- A **functional** is a function that takes a function as input and returns a vector.
- Functionals are used to abstract over common patterns of looping.
- Common functions are `lapply()`, `apply()`, `tapply()`, ...
- Reduce bugs by better communicating intent.

```
set.seed(1014)

# Create some random output:
# 20 random vectors with random lengths
l <- replicate(20, runif(sample(1:10, 1)),
  simplify = FALSE)
str(l)
l
```

```r
# Extract length of each element
lengths <- vector("list", length(l))
for (i in seq_along(l)) {
  lengths[[i]] <- length(l[[i]])
}
lengths
```

Preallocating space for output saves a lot of time

```
# Extract length of each element
lengths <- vector("list", length(l))
for (i in seq_along(l)) {
  lengths[[i]]
}
lengths
```

Safe shortcut for `1:length(l)`

How would you change this to compute the mean of each element?

```
compute_length <- function(x) {
  out <- vector("list", length(l))
  for (i in seq_along(l)) {
    out[[i]] <- length(l[[i]])
  }
  out
}
```

How would you change this to compute the median of each element?

```
compute_mean <- function(x) {
  out <- vector("list", length(l))
  for (i in seq_along(l)) {
    out[[i]] <- mean(l[[i]])
  }
  out
}
```

How would you change this to compute the median of each element?

```
compute_median <- function(x) {
  out <- vector("list", length(l))
  for (i in seq_along(l)) {
    out[[i]] <- median(l[[i]])
  }
  out
}
```

```
# How would you reduce the duplication here?
f1 <- function(x) x + 1
f2 <- function(x) x + 2
f3 <- function(x) x + 3
```

Your turn

How could you reduce the duplication between `compute_length()` and `compute_mean()` and `compute_median()`?

```
compute <- function(?, ?) {
   ...
}
```

```
compute(l, length)
compute(l, mean)
compute(l, median)
```

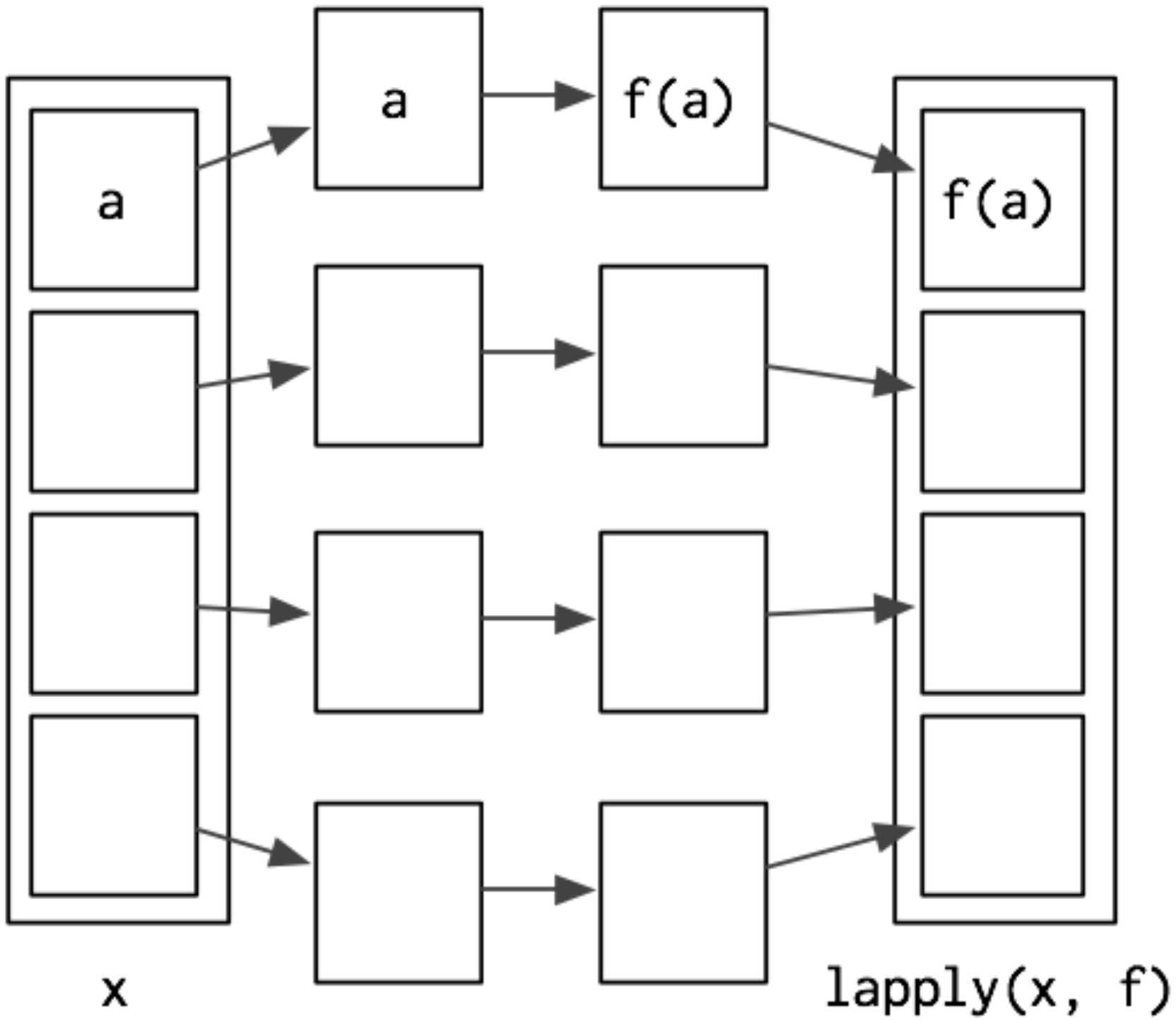
No peeking until you've made an attempt!

Functions can be arguments!

```
compute <- function(x, f) {
  out <- vector("list", length(x))
  for(i in seq_along(x)) {
    out[[i]] <- f(x[[i]])
  }
  out
}

compute(l, length)
compute(l, mean)
compute(l, median)
```

```
# BUT WAIT...

lapply(l, length)
lapply(l, mean)
lapply(l, median)
```

```
# We've just reinvented lapply :)
# Two differences:
# * lapply() uses some C tricks to be faster
# * lapply() passes ... on to f
```


a
a
f(a)
f(a)
x
lapply(x, f)

Placeholder for “any other” arguments

```
compute <- function(x, f, ...) {
  out <- vector("list", length(x))
  for(i in seq_along(x)) {
    out[[i]] <- f(x[[i]], ...)
  }
  out
}

compute(l, mean, trim = 0.5)
compute(l, mean, na.rm = TRUE)
```

Your turn

The function below scales a vector so it falls in the range [0, 1]. How would you apply it to every column of a data frame?

```
scale01 <- function(x) {
  rng <- range(x, na.rm = TRUE)
  (x - rng[1]) / (rng[2] - rng[1])
}
```

```
mtcars <- lapply(mtcars, scale01)
mtcars # a list :(
rm(mtcars)

mtcars[] <- lapply(mtcars, scale01)
mtcars # a data frame :)
rm(mtcars)

for(i in seq_along(mtcars)) {
  mtcars[[i]] <- scale01(mtcars[[i]])
}
```

Friends of

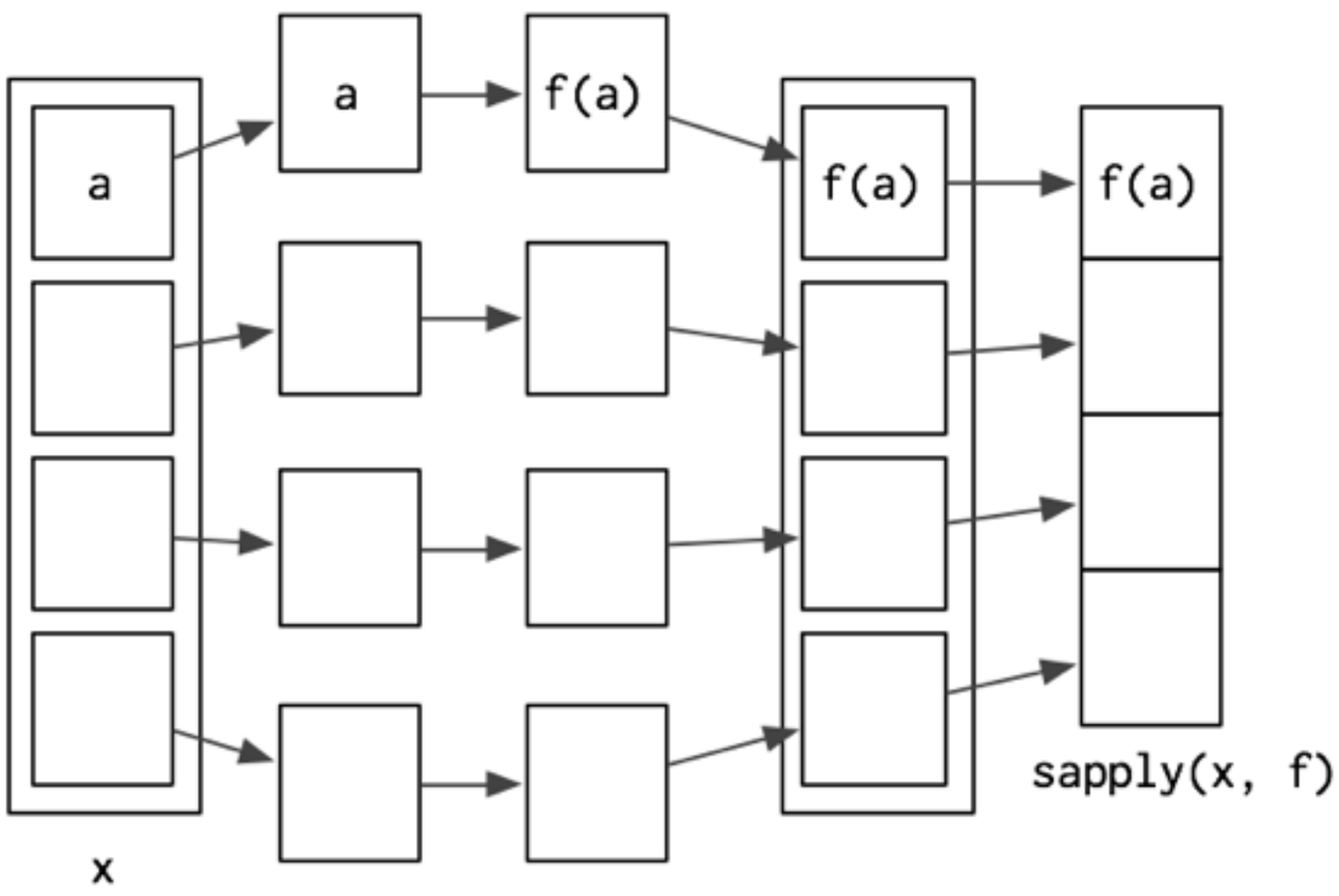
lapply()

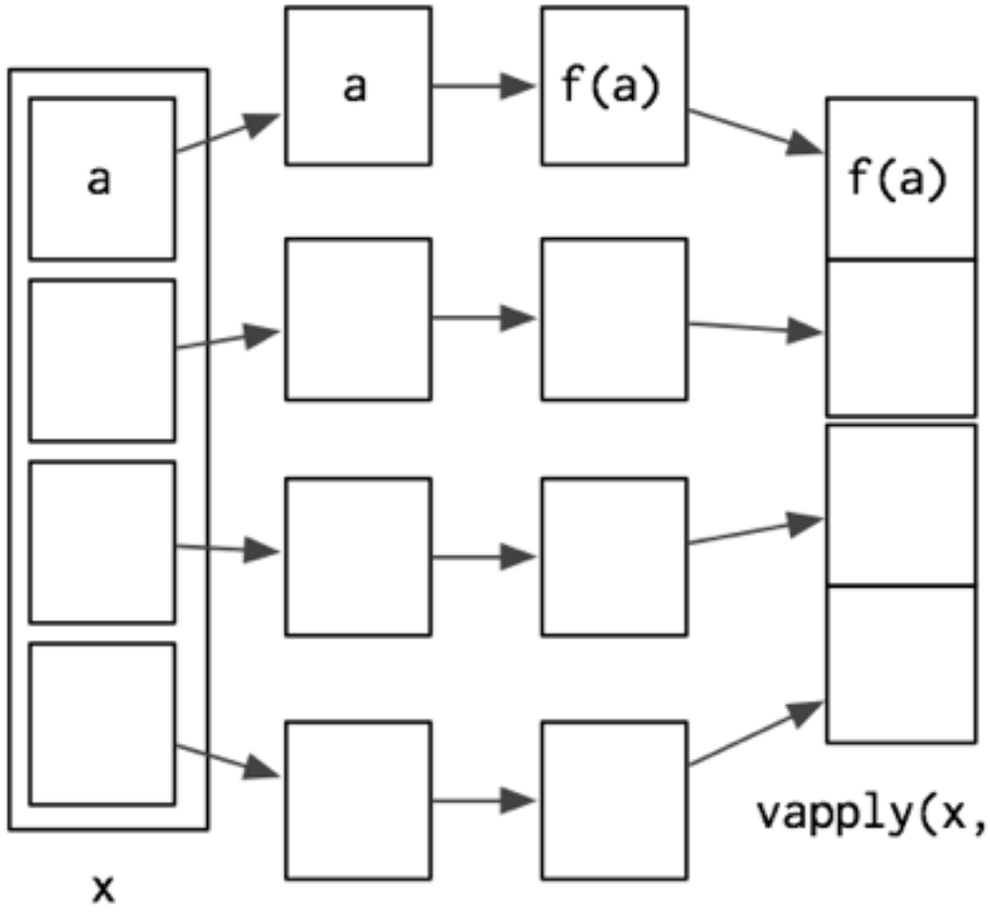
Variation	Function
I want an atomic vector, not a list	`sapply()`, `vapply()`
I have more than one input	`Map()`, `mapply()`
I have lots of computation to do	`mclapply()`

```
# Output is annoyingly long!
lapply(mtcars, is.numeric)
# We'd prefer a vector instead of a list

sapply(mtcars, is.numeric)
vapply(mtcars, is.numeric, logical(1))

# sapply() is useful interactively, but
# dangerous in a function because it has
# to guess the output type.
```

```
sapply(mtcars[,0], is.numeric)
vapply(mtcars[,0], is.numeric, logical(1))
```

```
sapply(mtcars, class)

mtcars$x <- Sys.now()
sapply(mtcars, class)
```


a
a
f(a)
f(a)
f(a)
x
sapply(x, f)


a
a
f(a)
f(a)
x
vapply(x, f

```r
sapply2 <- function(x, f, ...) {
  res <- lapply2(x, f, ...)
  simplify2array(res)
}

vapply2 <- function(x, f, f.value, ...) {
  out <- matrix(rep(f.value, length(x)), nrow = length(x))
  for (i in seq_along(x)) {
    res <- f(x[[i]], ...)
    stopifnot(
      length(res) == length(f.value),
      typeof(res) == typeof(f.value)
    )
    out[i, ] <- res
  }
  out
}
```

sapply()

Never use inside a function!

Your turn

Take the function on the next page and make it work more reliably, or at least identifying why it is failing.

```r
col_means <- function(df) {
  numeric <- sapply(df, is.numeric)
  numeric_cols <- df[, numeric]

  data.frame(lapply(numeric_cols, mean))
}

col_means(mtcars)
col_means(mtcars[, 0])
col_means(mtcars[0, ])
col_means(mtcars[, "mpg", drop = F])
col_means(1:10)
col_means(as.matrix(mtcars))
col_means(as.list(mtcars))

mtcars2 <- mtcars
mtcars2[-1] <- lapply(mtcars2[-1], as.character)
col_means(mtcars2)
```

No peeking until you've made an attempt!

```r
col_means <- function(df) {
  stopifnot(is.data.frame(df))
  # OR
  # df <- as.data.frame(df)
  numeric <- vapply(df, is.numeric, logical(1))
  numeric_cols <- df[, numeric, drop = FALSE]

  data.frame(lapply(numeric_cols, mean))
}
```

Your turn

Generalise `col_means()` to allow you to apply any function to the numeric columns in a data frame

Multicore computing

```
# Using these two helper functions and lapply()
# bootstrap this model 500 times and collect
# the R^2 values

data("mpg", package = "ggplot2"
lm(hwy ~ class + displ, data = mpg)

boot_df <- function(x) x[sample(nrow(x), rep = T), ]
rsquared <- function(mod) summary(mod)$r.square
```

```r
# Do it in parallel
library(parallel)
options(mc.cores = 4L)

bootstraps <- lapply(1:500, function(i) boot_df(mtcars))

# Linux and Mac
bootstraps <- mclapply(1:500, function(i)
boot_df(mtcars))

# Windows, Linux, and Mac
cluster <- makePSOCKcluster(4)
clusterExport(cluster, c("boot_df", "rsquared"))
bootstraps <- parLapply(cluster, 1:500,
  function(i) boot_df(mtcars))
```

Costs

- How much time does it take to setup the cluster?
- How much time does it take transfer data?
- How much time does it take you to write the code?

```
my_boot <- function(i) boot_df(mpg)
```

```
system.time(lapply(1:500, my_boot))
```

Takes 0.15s

```
cluster <- makePSOCKcluster(2)
system.time(parLapply(cluster, 1:500, my_boot))
```

How long will this take?

```
cluster <- makePSOCKcluster(4)
system.time(parLapply(cluster, 1:500, my_boot))
```

How long will this take?

```
boot_rsq <- function(i) {
  rsquared(lm(hwy ~ class + displ, data = boot_df(mpg)))
}
system.time(lapply(1:500, boot_rsq))
```

Takes 1s

```
cluster <- makePSOCKcluster(2)
system.time(parLapply(cluster, 1:500, boot_rsq))
```

How long will this take?

```
cluster <- makePSOCKcluster(4)
system.time(parLapply(cluster, 1:500, boot_rsq))
```

How long will this take?

Tips

- Be aware of communication costs. If your task involves moving a lot of data around, parallel is likely to be slower.
- `parLapply()` is fiddlier to set up, but it's obvious what data you're sending to each computer. Cluster setup only incurred once.

Learning more

Advanced R

http://adv-r.had.co.nz/Functions.html

http://adv-r.had.co.nz/Functional-programming.html

http://adv-r.had.co.nz/Functionals.html

http://adv-r.had.co.nz/Function-operators.html