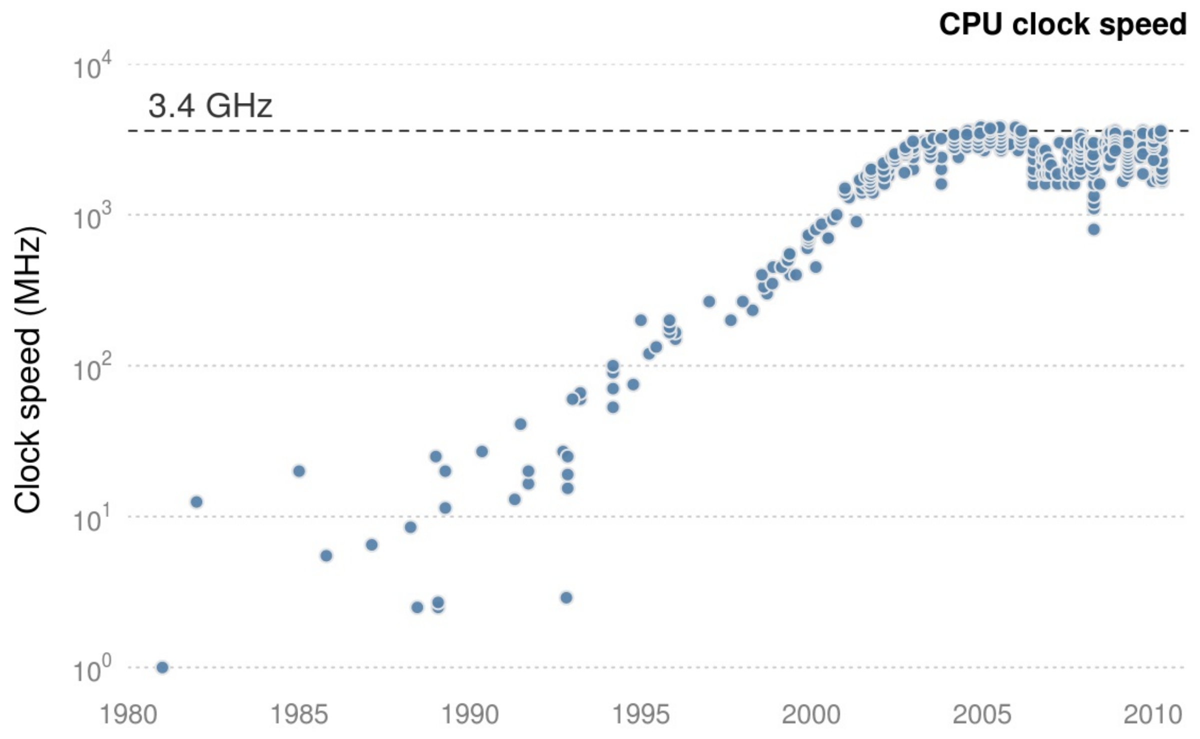
WRITING EFFICIENT R CODE

CPUs - why do we have more than one?

Colin Gillespie
Jumping Rivers & Newcastle University

CPUs

- CPU: brains of the computer
- Speed has slowly stabilized
  - CPUs were getting too hot
- Multi-core CPUs
- But R only uses 1 core :(



Your CPU

```
> library("parallel")
> detectCores()
[1] 8
```

```
> library("benchmarkme")
> get_cpu()
# $vendor_id
# "GenuineIntel"
#
# $model_name
# "Intel(R) Core(TM) i7-4702HQ CPU
#
# $no_of_cores
# 8
```

WRITING EFFICIENT R CODE

Let's practice!

WRITING EFFICIENT R CODE

What sort of problems benefit from parallel computing

Colin Gillespie
Jumping Rivers & Newcastle University

Cooking

AN EXTRA HAND

TOO MANY COOKS

Running in parallel

- Not every analysis can make use of multiple cores
- Many statistical algorithms can only use a single core

> *So where can parallel computing help?*

Monte-Carlo simulations

```
> for(i in 1:8)
+     sims[i] <- monte_carlo()
```

```
> combine(sims)
```

- 8 core machine
- One simulation per core
- Combine the results at the end
- *Embarrassingly parallel*

Not everything runs in parallel

```
> x <- 1:8
> for(i in 2:8)
+     x[i] <- x[i-1]
```

```
> x[8] = x[7] = ... x[2] = x[1] = 1
```

- Can we run this in parallel?
  - NO
- But order of evaluation in parallel computing *can't* be predicted
- We'll get the wrong answer, since `x[3]` may get evaluated before `x[2]`

Rule of thumb

Can the loop be run forward and backwards?

```
> for(i in 1:8)
+     sim[i] <- monte_carlo_simulation()

> for(i in 8:1)
+     sim[i] <- monte_carlo_simulation()
```

- Both loops give the same result
- So we can run the loops in parallel

Rule of thumb

Can the loop be run forward and backwards?

```
> x <- 1:8
> for(i in 2:8)
+     x[i] <- x[i-1]

> for(i in 8:2)
+     x[i] <- x[i-1]
```

- The loops give different answers
  - The first: x[8] = x[7] = ... = 1
  - The second: x[8] = x[7] = 7
- Can't use parallel computing

WRITING EFFICIENT R CODE

Let's practice!

WRITING EFFICIENT R CODE

The parallel package - parApply

Colin Gillespie
Jumping Rivers & Newcastle University

The parallel package

- Part of R since 2011

```r
library("parallel")
```

- Cross platform: Code works under Windows, Linux, Mac
- Has parallel versions of standard functions

The apply() function

- apply() is similar to a for loop
  - We *apply* a function to each row/column of a matrix
- A 10 column, 10,000 row matrix:

```
> m <- matrix(rnorm(100000), ncol = 10)
```

- apply is neater than a for loop

```
> res <- apply(m, 1, median)
```

Converting to parallel

- Load the package
- Specify the number of cores
- Create a cluster object
- Swap to parApply()
- Stop!

```
> library("parallel")
```

```
> copies_of_r <- 7
```

```
> cl <- makeCluster(copies_of_r)
```

```
> parApply(cl, m, 1, median)
```

```
> stopCluster(cl)
```

The bad news

As Lewis Caroll said

The hurrier I go, the behinder I get.

- Sometimes running in parllel is slower due to thread communication

```
# Serial version
> apply(m, 1, median)
```

```
# Parallel version
> parApply(cl, m, 1, median)
```

- Benchmark both solutions

WRITING EFFICIENT R CODE

Let's practice!

WRITING EFFICIENT R CODE

The parallel package - parSapply

Colin Gillespie
Jumping Rivers & Newcastle University

The apply family

There are parallel versions of

- apply() - parApply()
- sapply() - parSapply()
  - applying a function to a vector, i.e. a for loop
- lapply() - parLapply()
  - applying a function to a list

The sapply() function

sapply() is just another way of writing a for loop

The loop

```
> for(i in 1:10)
+     x[i] <- simulate(i)
```

Can be written as

```
> sapply(1:10, simulate)
```

We are applying a function to each value of a vector

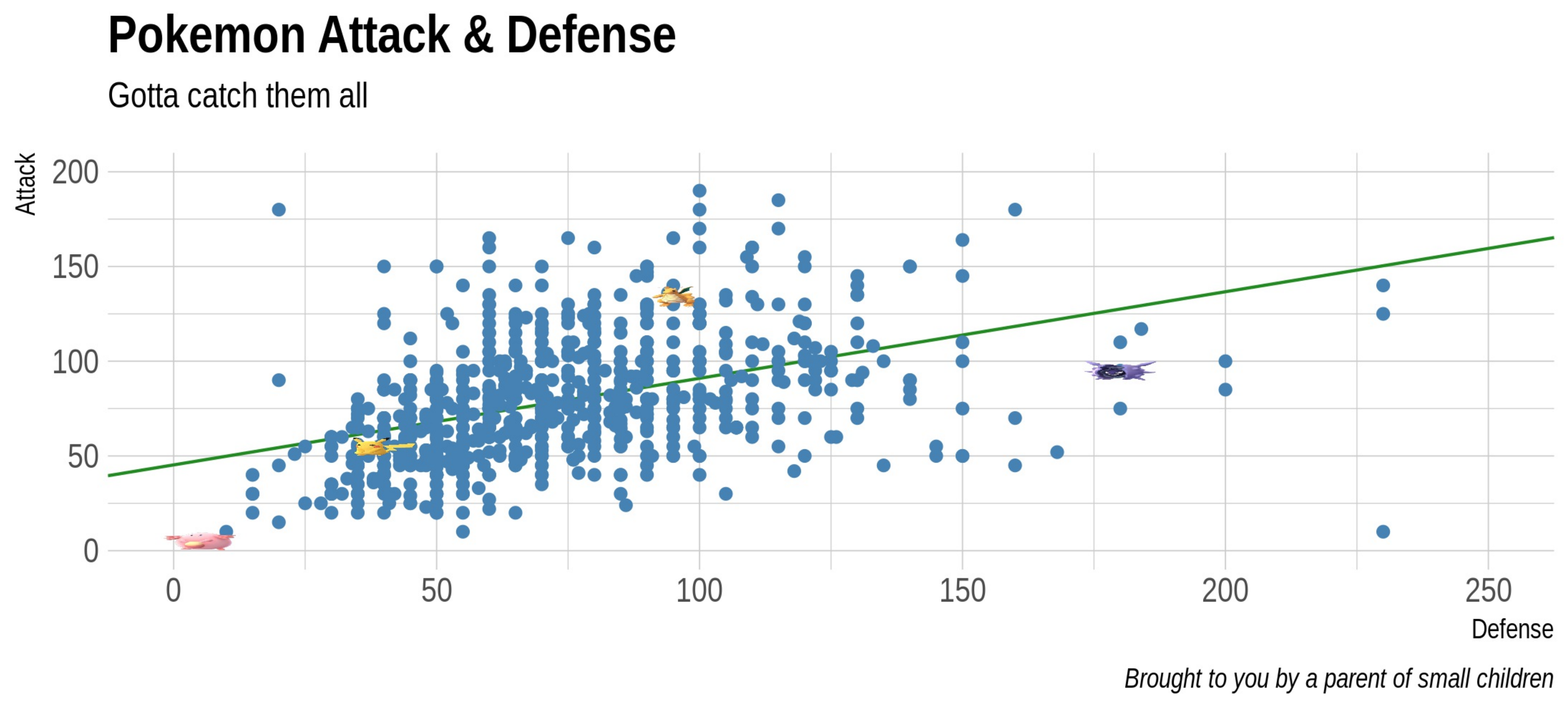
Switching to `parSapply()`

It's the same recipe!

1. Load the package
2. Make a cluster
3. Switch to `parSapply()`
4. Stop!

Example: Pokemon battles

```
> plot(pokemon$Defense, pokemon$Attack)
> abline(lm(pokemon$Attack ~ pokemon$Defense), col = 2)
> cor(pokemon$Attack, pokemon$Defense)
[1] 0.437
```



Bootstrapping

In a perfect world, we would resample from the *population*; but we can't

Instead, we assume the original sample is representative of the population

1. Sample with *replacement* from your data
    - The same point could appear multiple times
2. Calculate the correlation statistics from your new sample
3. Repeat

A single bootstrap

```
> bootstrap <- function(data_set) {
+     # Sample with replacement
+     s <- sample(1:nrow(data_set), replace = TRUE)
+     new_data <- data_set[s,]
+
+     # Calculate the correlation
+     cor(new_data$Attack, new_data$Defense)
+ }
```

```
# 100 independent bootstrap simulations
> sapply(1:100, function(i) bootstrap(pokemon))
```

Converting to parallel

- Load the package
- Specify the number of cores
- Create a cluster object
- Export functions/data
- Swap to parApply()
- Stop!

```
> library("parallel")
```

```
> no_of_cores <- 7
```

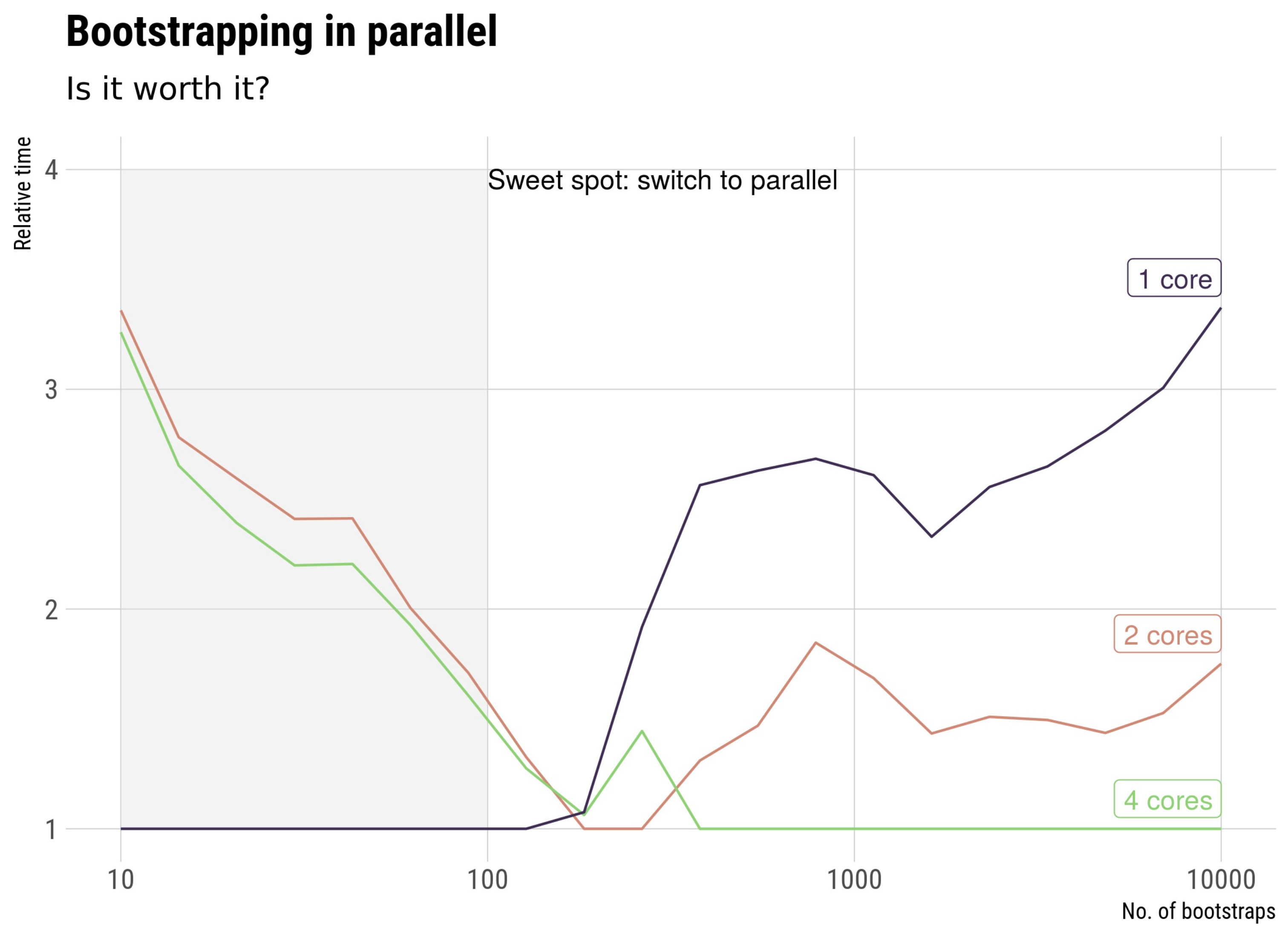
```
> cl <- makeCluster(no_of_cores)
```

```
> clusterExport(cl,
+               c("bootstrap", "pokemor
```

```
> parSapply(cl, 1:100,
+          function(i) bootstrap(poker
```

```
> stopCluster(cl)
```

Timings


Bootstrapping in parallel
Is it worth it?
Relative time
Sweet spot: switch to parallel
1 core
2 cores
4 cores
1
2
3
4
10
100
1000
10000
No. of bootstraps

WRITING EFFICIENT R CODE

Let's practice!

WRITING EFFICIENT R CODE

Congratulations!

Colin Gillespie
Jumping Rivers & Newcastle University