COLUMBIA | MAILMAN SCHOOL of PUBLIC HEALTH

Lawrence G. Chillrud
Environmental Health Sciences
Mailman School of Public Health
Columbia University

Parallel Computation in R with the `foreach` package

A Brief Introduction

April 7, 2022

Outline

1. Loops in R
2. The `foreach` package
3. Parallel Computing
   a) When to parallelize
   b) How to parallelize in R (FORKs vs. SOCKs)
   c) Packages for parallelization

Loops in R: the usual suspects

`apply` functions

- `apply(X, MARGIN, FUN, ...)`
- `lapply(X, FUN, ...)`
- `sapply(X, FUN, ...)`
- `vapply(X, FUN, ...)`
- Etc.

`purrr` functions

- `map(.x, .f, ...)`
- `walk(.x, .f, ...)`
- `reduce(.x, .f, ...)`
- `accumulate(.x, .f, ...)`
- Etc.

Loops in R: the unfashionable crowd

Control-flow constructs

- `for (variable in sequence) do_cool_stuff()`
- `while (condition) do_cool_stuff()`

Enter the foreach package

```
foreach(variable = sequence, ...) %do% {
    cool_stuff()
}
```

Some useful arguments:

- .combine
- .init
- .final
- .errorhandling
- .verbose

Some useful operators

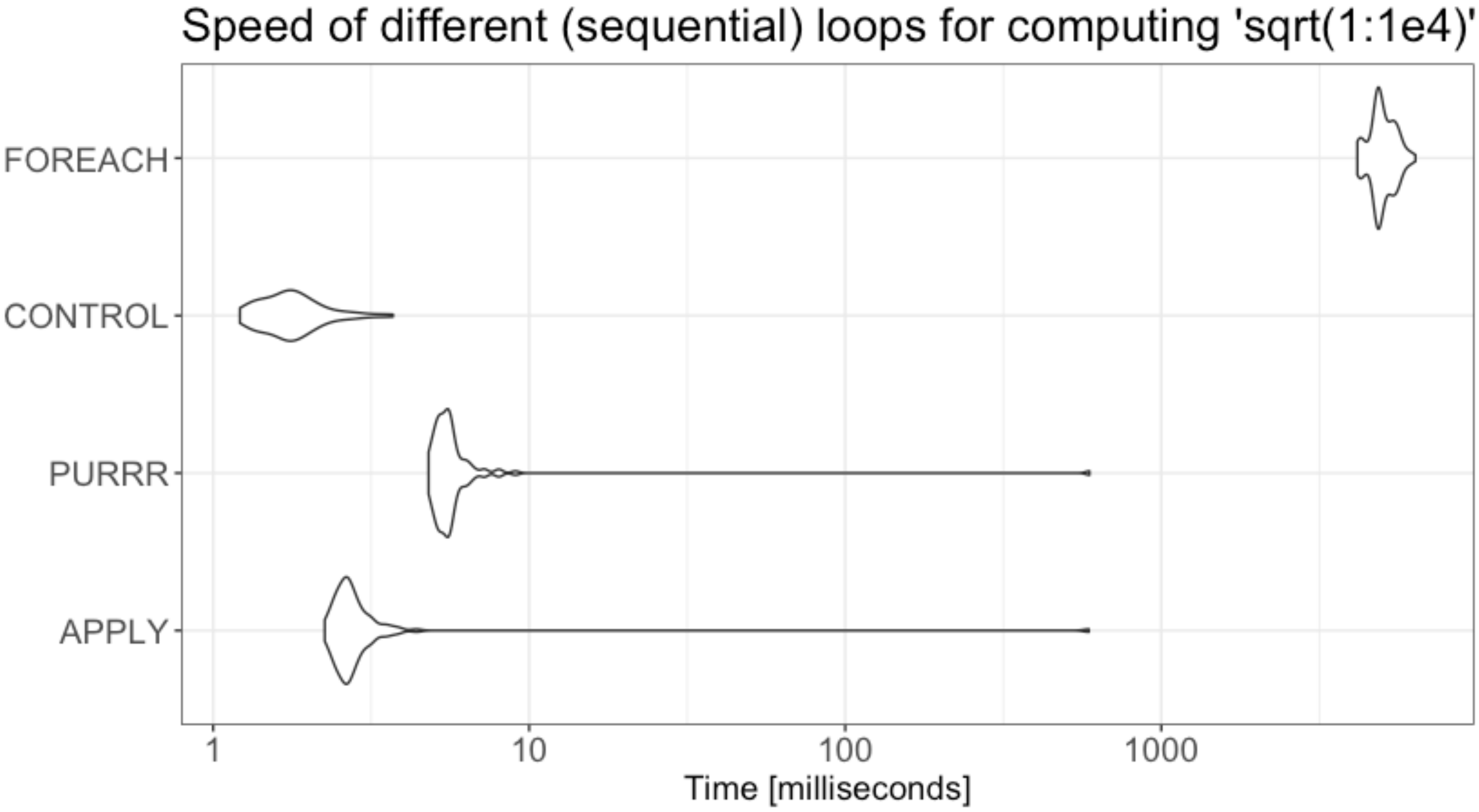
- %do%
- %:%
- when(condition)

Benchmarking test...

```
inputs <- 1:1e4

mbm <- microbenchmark(times = 100,
  "APPLY" = sapply(inputs, sqrt),
  "PURRR" = map_dbl(inputs, sqrt),
  "CONTROL" = for (i in inputs) sqrt(i),
  "FOREACH" = foreach(i = inputs, .combine = c) %do% (sqrt(i))
)
```

Benchmarking results!

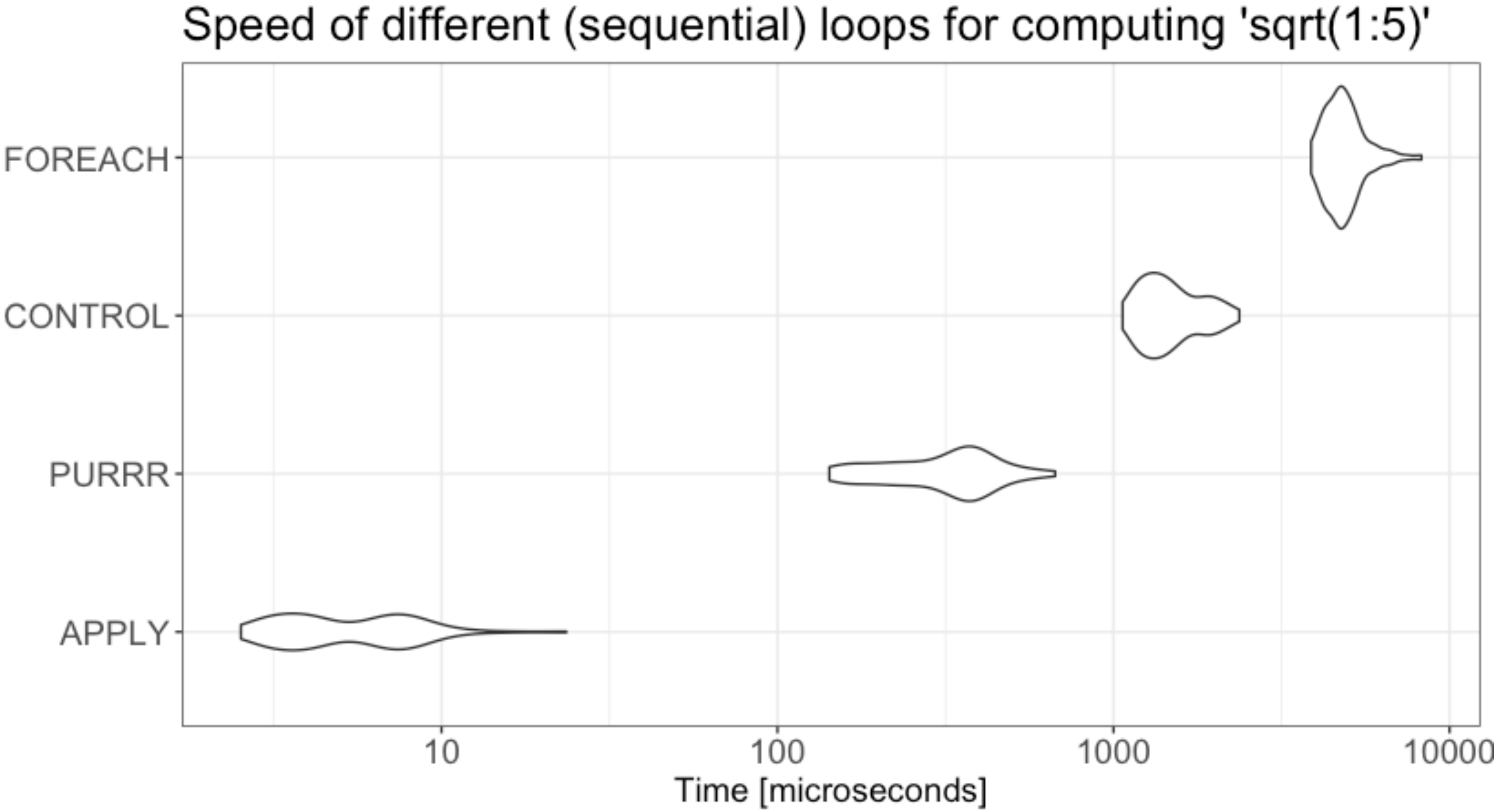


Another benchmark test...

```
inputs <- 1:5

mbm <- microbenchmark(times = 100,
   "APPLY" = sapply(inputs, sqrt),
   "PURRR" = map_dbl(inputs, sqrt),
   "CONTROL" = for (i in inputs) sqrt(i),
   "FOREACH" = foreach(i = inputs, .combine = c) %do% (sqrt(i))
)
```

And the new results



Pros vs. cons of looping via `foreach`?

Pros:

- Finer control
- Flexible
- Enhanced readability
- Easier to debug
- Extremely parallelizable

Cons:

- Computationally slower*
- Not very “R-like” ???

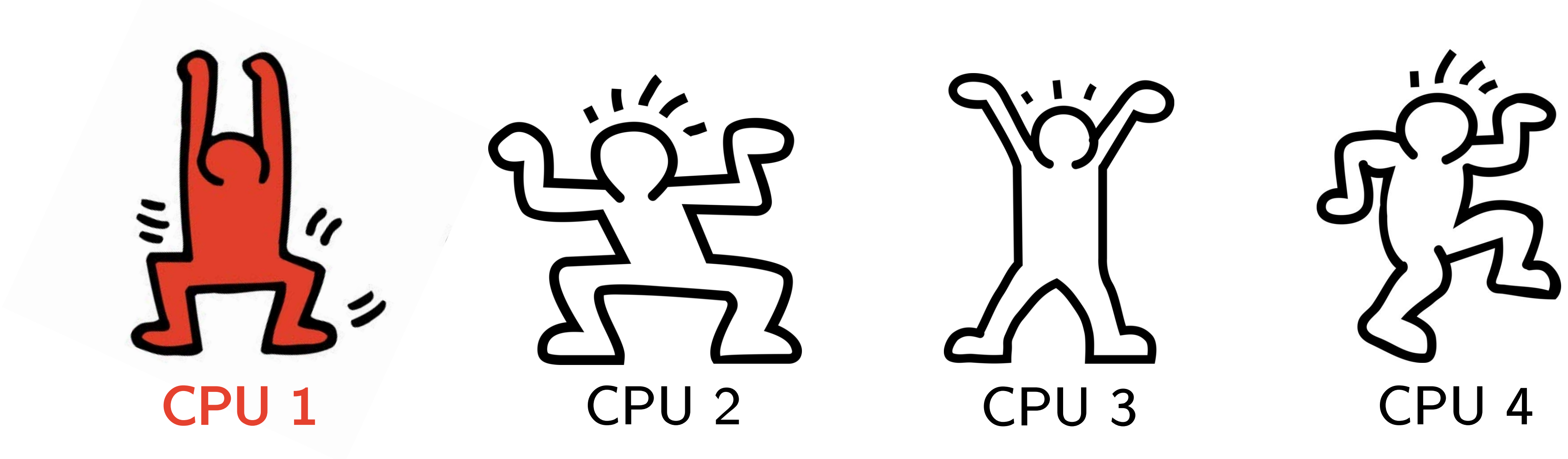
Use when you have a relatively few # of expensive and complex tasks!

It’s just another tool for your toolkit 

What is parallel computing?

Typically, the code we write is executed sequentially (in serial)

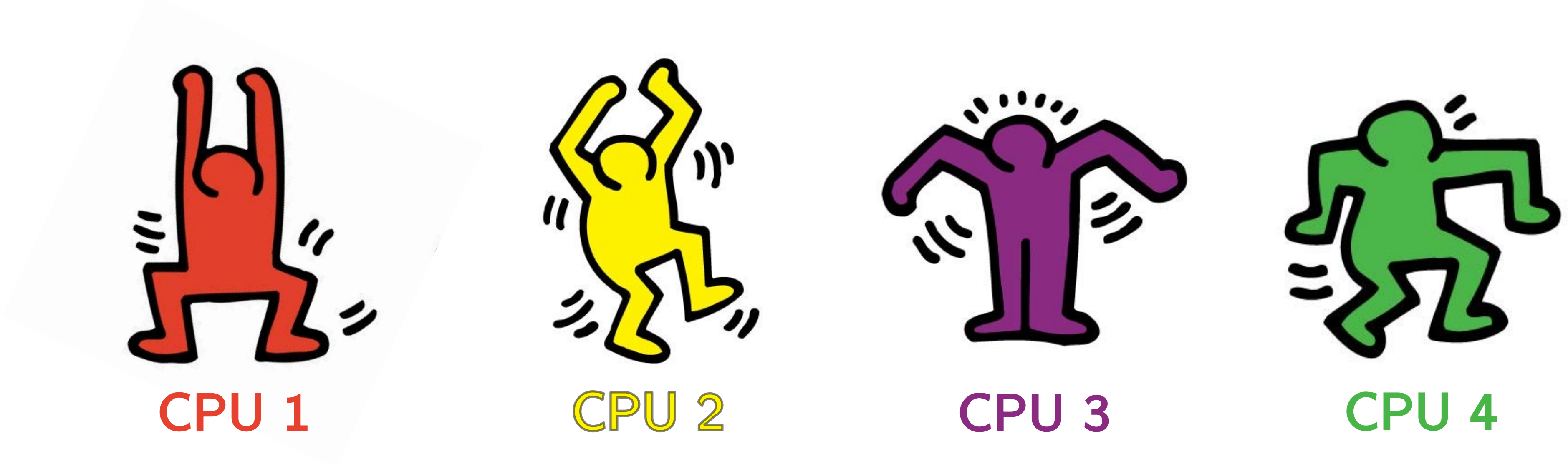
- This is done by a single CPU:



- What about our other CPUs???

What is parallel computing?

When possible, ideally we'd like to leverage all our available CPUs to speed things up:



This is known as parallel processing/programming/computing

What can be parallelized?

"Embarrassingly" parallel loops:

- Repeated independent tasks
  - `purrr::map()`
  - Sensitivity analyses
  - Parameter tuning / searching

Inherently sequential loops:

- Chains of dependent tasks
  - `purrr::accumulate()`
  - Gaussian processes
  - Markov decision processes

Parallelizability exists on a spectrum

Amdahl's Law

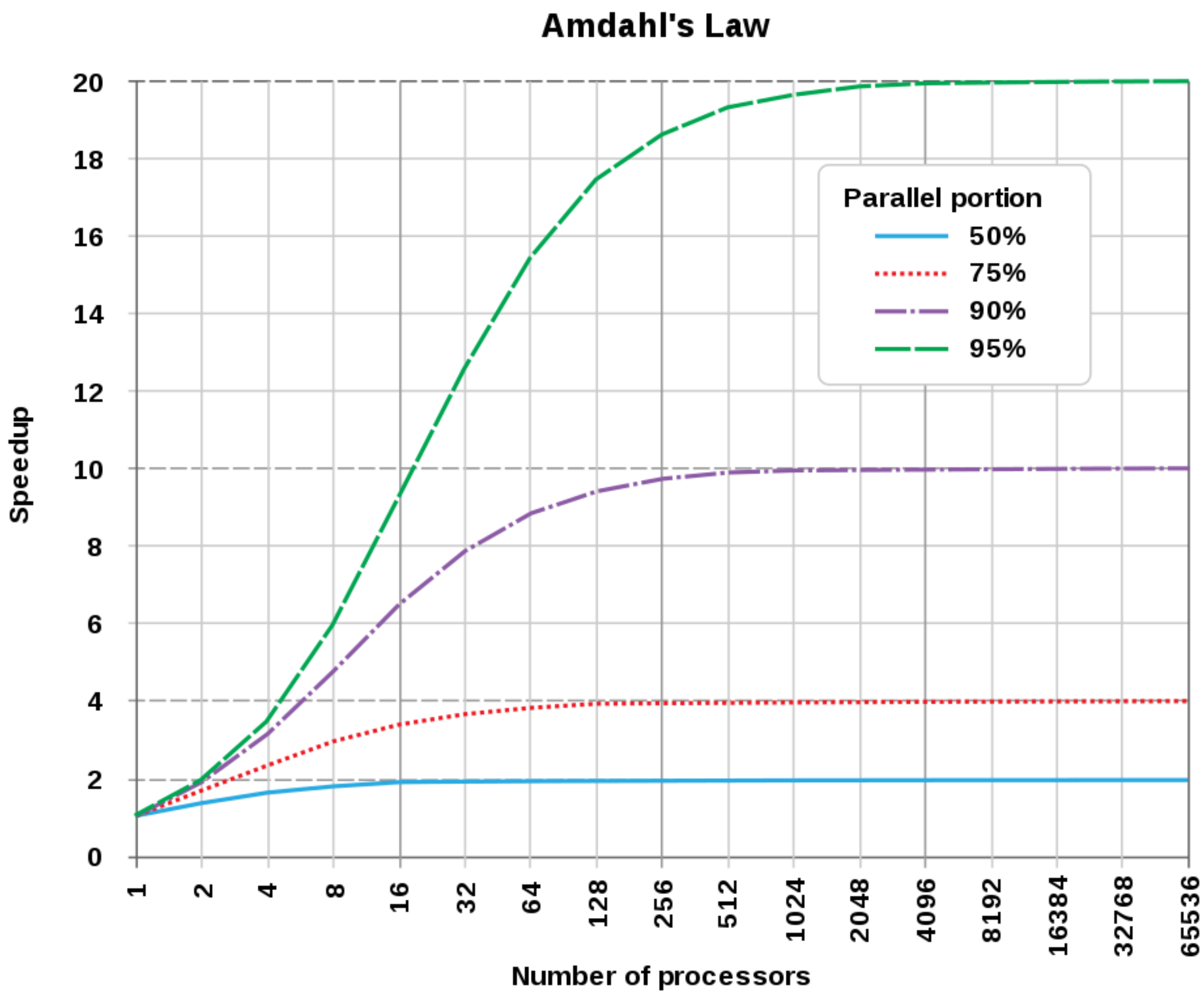


Image Source: [Wikipedia]

When should we parallelize?

```
> independent_task(x) == expensive
[1] TRUE

> length(independent_tasks) == large
[1] TRUE
```

How should we parallelize in R?

FORKs

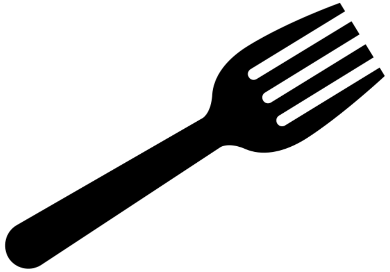
- Duplicates main process to each core
- Keyword: multicore
- Pro: Each core shares same workspace
- Pro: Very fast
- Pro: Easy to implement
- Con: RNG & GUI issues
- Con: Does ***not*** work on Windows

SOCKETs

- Launches new R session on each core
- Keyword: multisession
- Con: Cores need workspaces set up
- Con: Relatively slow
- Con: Harder to implement
- Pro: Unique threads mean no issues
- Pro: Works on any system

How should we parallelize in R?

FORKs



SOCKETs

The `parallel` package: Overview

- Comes bundled with R
- Made by combining best bits of two older packages:
    - `multicore` – parallelism via FORKs
    - `snow` – parallelism via SOCKETs
- Basically all parallel computing in R relies on `parallel` under-the-hood

The parallel package: using FORKs

```
library(parallel)
num_cores <- detectCores(logical = F)
mclapply(1:5, sqrt, mc.cores = num_cores) # same as lapply(1:5, sqrt)
pvec(1:5, sqrt, mc.cores = num_cores)     # same as sapply(1:5, sqrt)
```

The `parallel` package: using SOCKETs

```
# The following is equivalent to lapply(X, FUN, ...)
library(parallel)
num_cores <- detectCores(logical = F)      # get num of physical cores
cl <- makeCluster(num_cores)               # create a cluster
base <- 4                                  # define local variable
clusterExport(cl, “base”)                  # send variable to cluster
parLapply(cl, 1:5, function(exp) base^exp) # parallelize lapply
stopCluster(cl)                            # shut down cluster
```

The `future` and `furrr` packages: Overview

- `future` is the parallel backend allowing the director and workers to communicate
- `furrr` (short for `future purrr`) provides functions from `purrr` that can be parallelized
  - `map(.x, .f, ...)` → `future_map(.x, .f, ...)`

The future and furrr packages: using FORKs

```r
base <- 4
purrr::map(1:5, ~base^.x)
```

```r
base <- 4
num_cores <- parallel::detectCores(logical = F)
future::plan(“multicore”, workers = num_cores)
furrr::future_map(1:5, ~base^.x)
future::plan(“sequential”)
```

The `future` and `furrr` packages: using SOCKETs

```
base <- 4
purrr::map(1:5, ~base^.x)
```

```
base <- 4
num_cores <- parallel::detectCores(logical = F)
future::plan("multisession", workers = num_cores)
furrr::future_map(1:5, ~base^.x)
future::plan("sequential")
```

The `doParallel` and `foreach` packages: Overview

- Here, `doParallel` is the parallel backend allowing the director and workers to communicate
- `foreach` comes with a `%dopar%` operator which then parallelizes your loop once you've registered your backend in `doParallel`
- `foreach` is also compatible with other parallel backends such as:
  - `doFuture`
  - `doSNOW`
- `foreach` comes with many useful options when using it with the `%dopar%` operator:
  - `.inorder`, `.packages`, `.export`, etc.

The doParallel and foreach packages: FORKs

```
library(foreach)
base <- 4
num_cores <- parallel::detectCores(logical = F)
cl <- parallel::makeCluster(num_cores, type = "FORK")
doParallel::registerDoParallel(cl)
foreach(exp = 1:5) %dopar% {
   base^exp
}
parallel::stopCluster(cl)
```

The doParallel and foreach packages: SOCKETs

```
library(foreach)
base <- 4
num_cores <- parallel::detectCores(logical = F)
cl <- parallel::makeCluster(num_cores, type = "PSOCK")
doParallel::registerDoParallel(cl)
foreach(exp = 1:5) %dopar% {
   base^exp
}
parallel::stopCluster(cl)
```

The doFuture and foreach packages: SOCKETs

```
library(foreach)
base <- 4
num_cores <- parallel::detectCores(logical = F)
doFuture::registerDoFuture()
future::plan(“multisession”, workers = num_cores)
foreach(exp = 1:5) %dopar% {
   base^exp
}
```

Keep an eye out for `multidplyr`

```
library(multidplyr)   # parallel backend for dplyr package
library(nycflights13) # loads the flights dataset for us
library(tidyverse)    # general data wrangling (includes dplyr)

num_cores <- parallel::detectCores(logical = F)
cl <- new_cluster(num_cores) # creating the cluster with multidplyr
cluster_library(cl, “dplyr”) # loading the dplyr package to each core in the cluster

flight_dest <- flights %>%
  group_by(dest) %>%
  partition(cl)  # multidplyr partitioning all the groups across all available cores

flight_dest %>%
  summarise(delay = mean(dep_delay, na.rm = T), n = n()) %>%
  collect() # multidplyr collecting all the groups back into the host session
```

Source

References

- Video lecture: Parallel computing with R using foreach, future, and other packages by Bryan Lewis (RStudio, 2020)
- Video lecture: Future: Simple Async, Parallel & Distributed Processing in R by Henrik Bengtsson (RStudio, 2020)
- Tutorial: Using the foreach package by Steve Weston
- Tutorial: Parallel Processing in R by Josh Errickson
- Tutorial: Parallelized loops with R by Blas Benito (2021)
- Tutorial: Quick Intro to Parallel Computing in R by Matt Jones (2017)
- Textbook: Parallel R by Q. Ethan McCallum and Stephen Weston (2011)
- Textbook: R Programming for Data Science, Chapter 22: Parallel Computation by Roger Peng (2020)

Thanks!

Contact:

✉ lgc2139@cumc.columbia.edu
🌎 lawrence-chillrud.github.io