Hertie School

Parallel programming with future

Maria Fernanda Ortega
Danial Riaz

Date

Hertie School

Contents

Hertie School

Sequential: People in a line

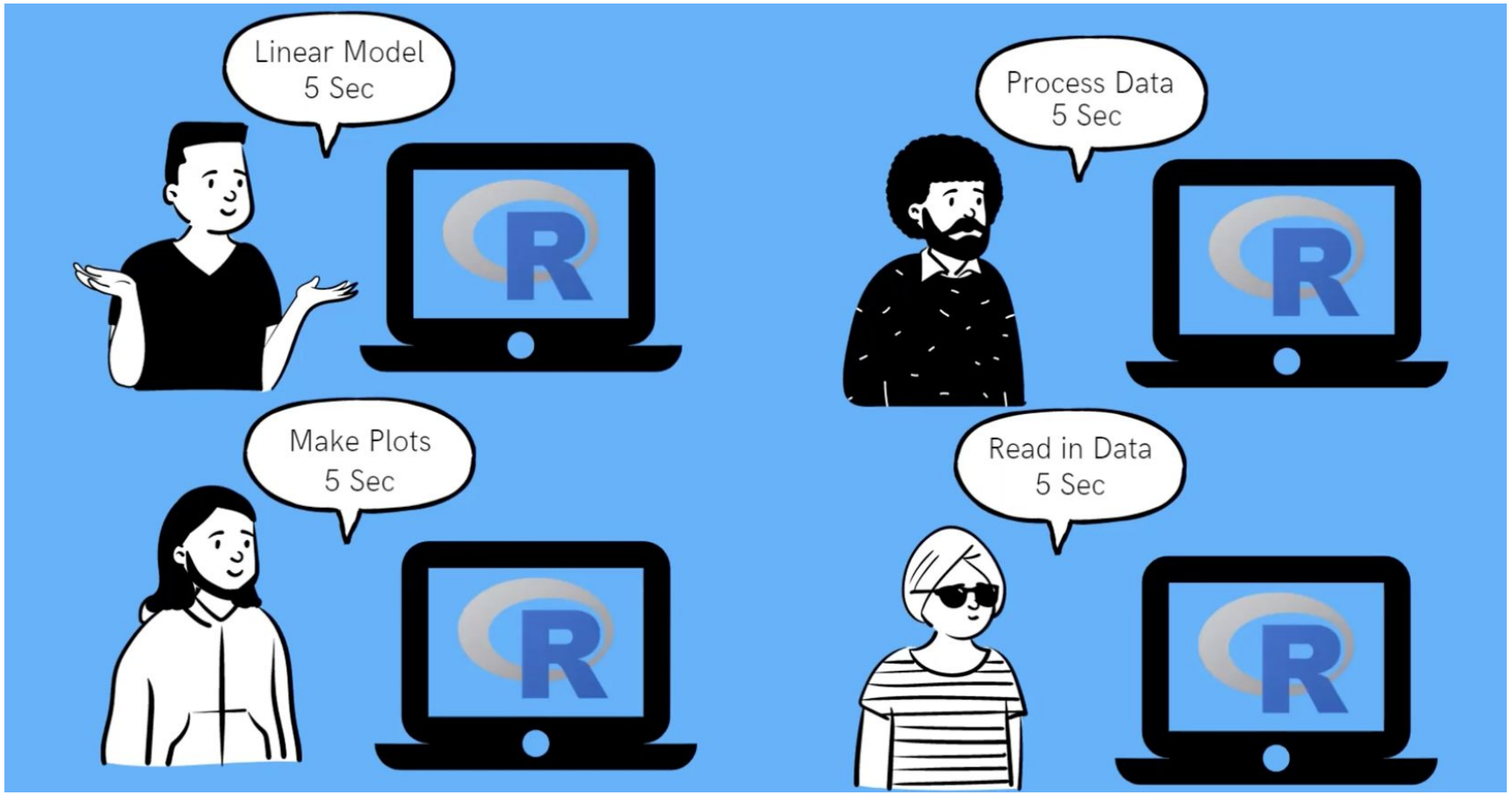
Total time 20 seconds

Credit: Victor Feagins

Hertie School

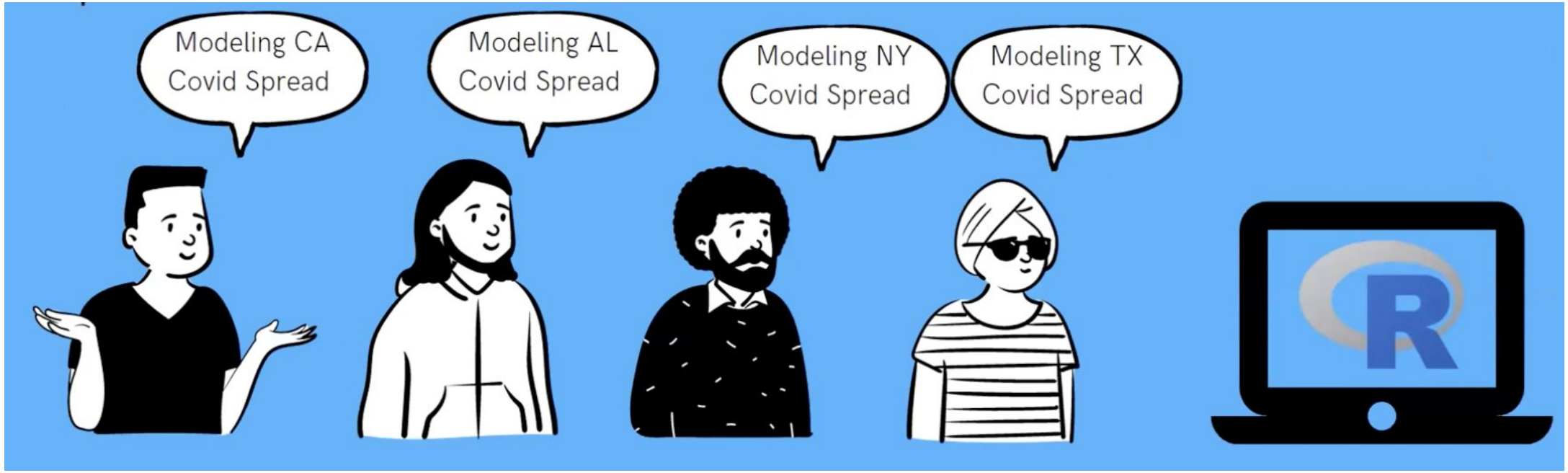
Working in Parallel

Total time 5 seconds



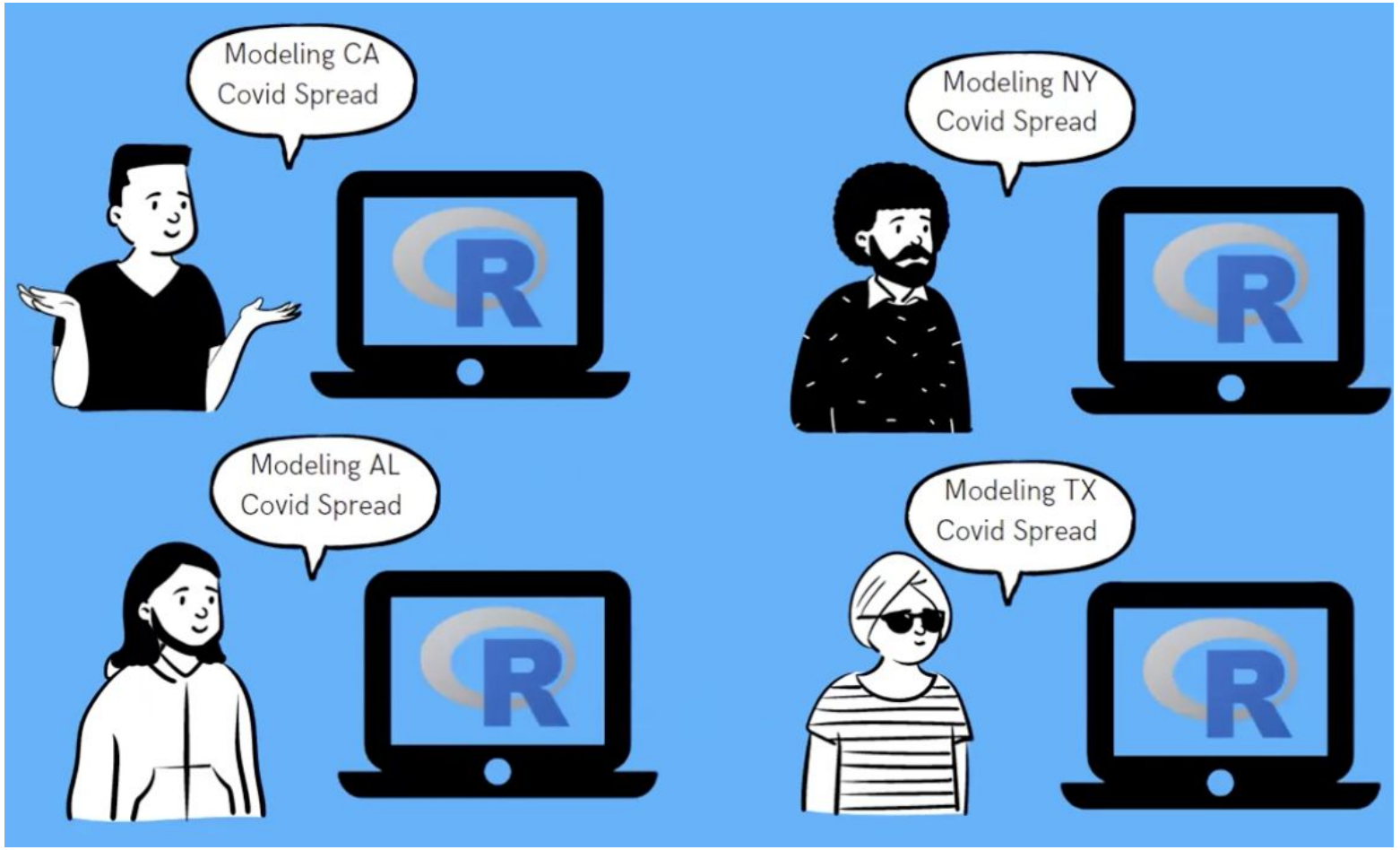
Ideal parallelizable problem

AKA 'Embarrassingly Parallel'
Total time: 20 sec



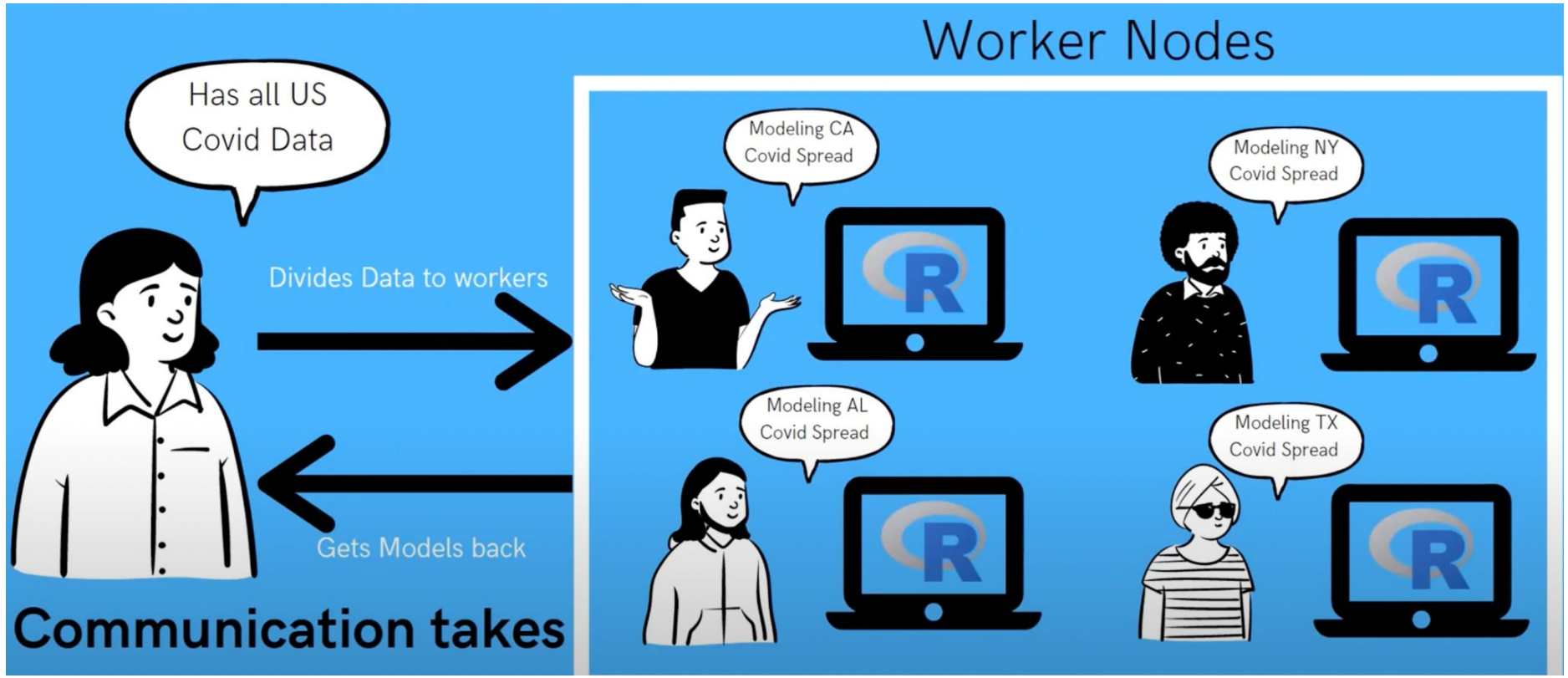
Ideal parallelizable problem

AKA 'Embarrassingly Parallel'
Total time: 7 sec



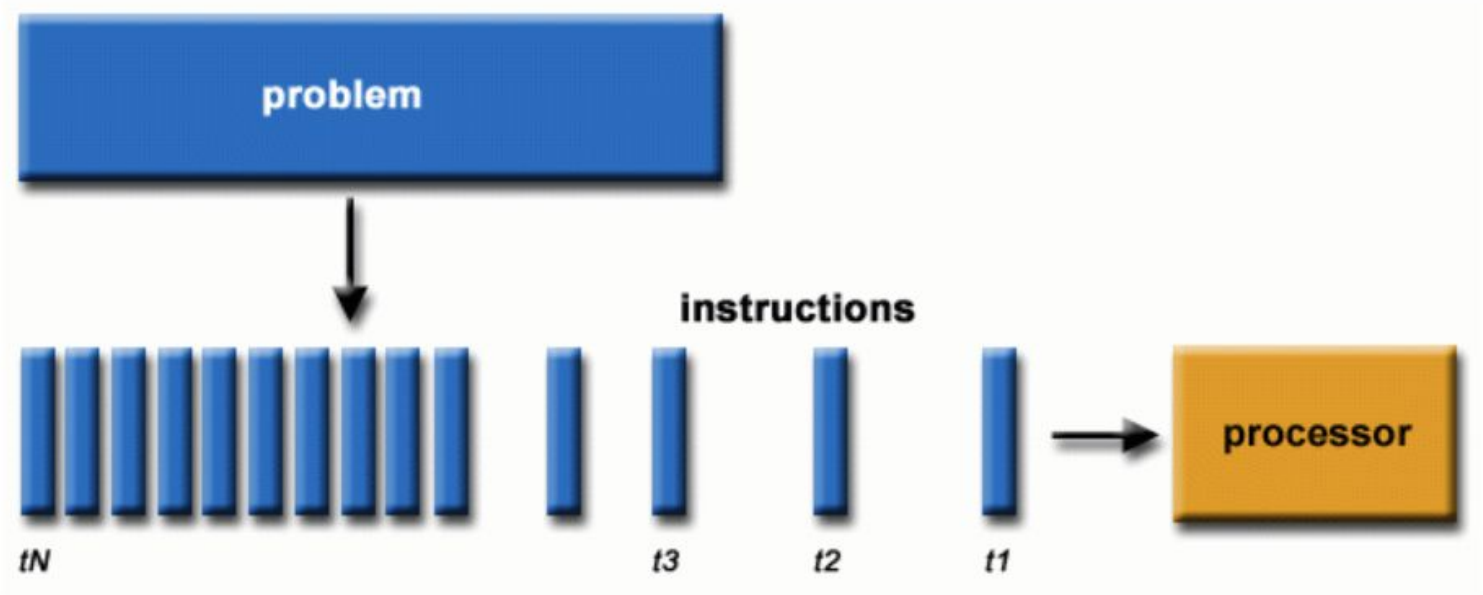
Ideal parallelizable problem

AKA 'Embarrassingly Parallel'
Total time: 7 sec



Serial processing vs Parallel processing

Serial processing: *a problem is divided into a series of instructions, which are executed sequentially, one after another, on a single processor (CPU).*

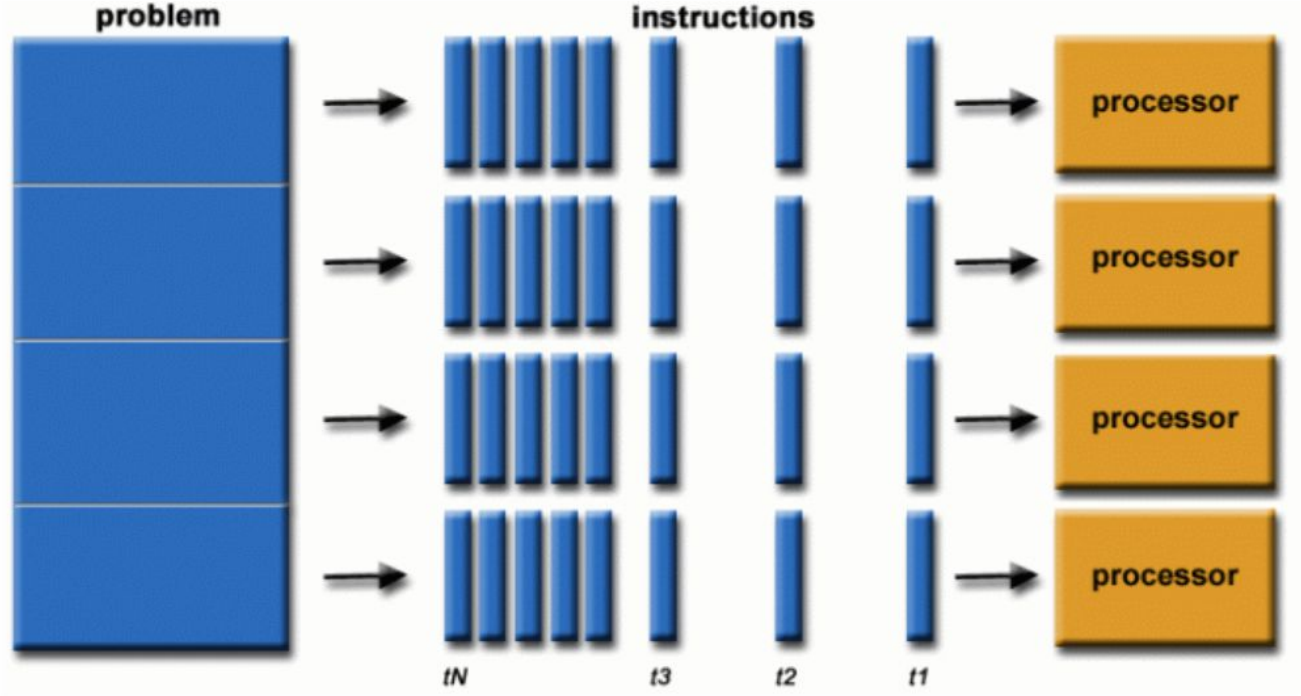


Source: Barney, B. (2010)

Serial processing vs Parallel processing

Hertie School

Parallel processing: *a problem is divided into discrete parts to which a series of instructions can be applied simultaneously on different processors(CPUs).*

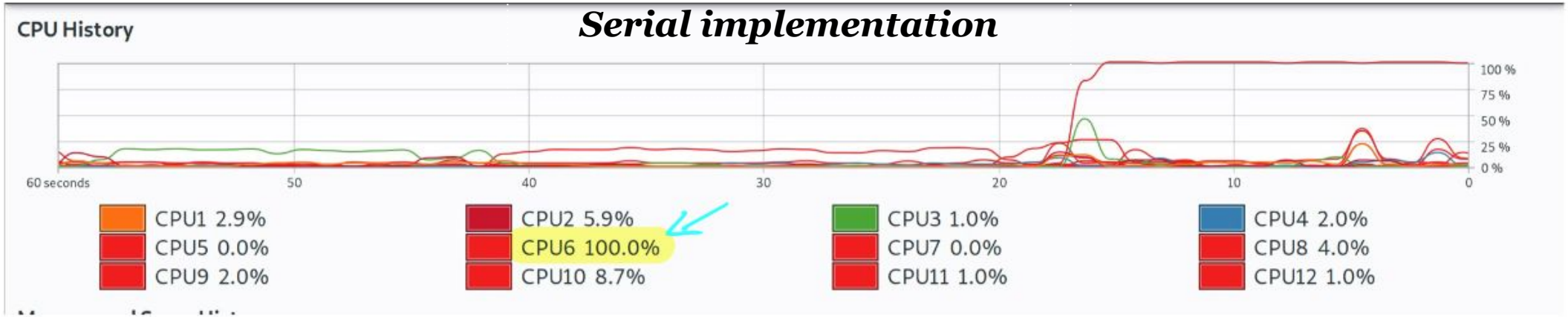


Source: Barney, B. (2010)

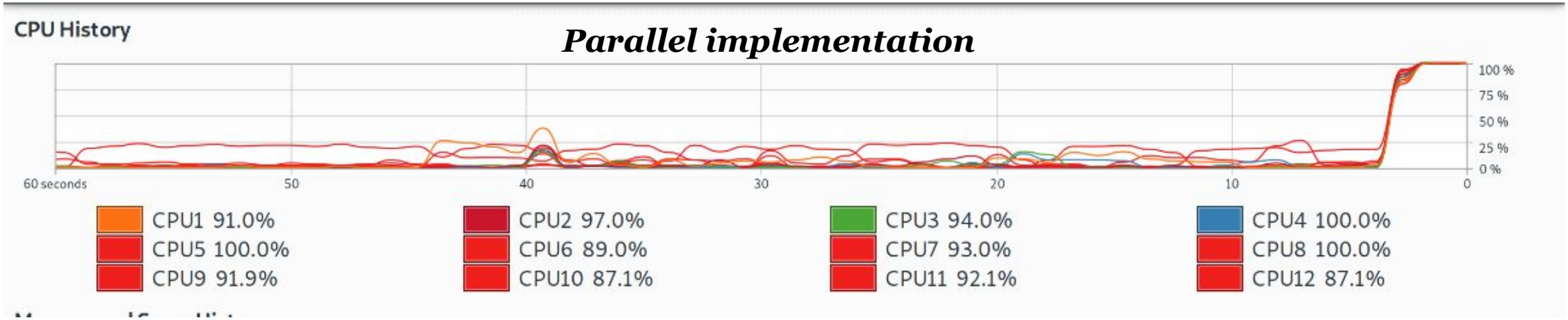
CPU History

Hertie School

Serial implementation



Parallel implementation

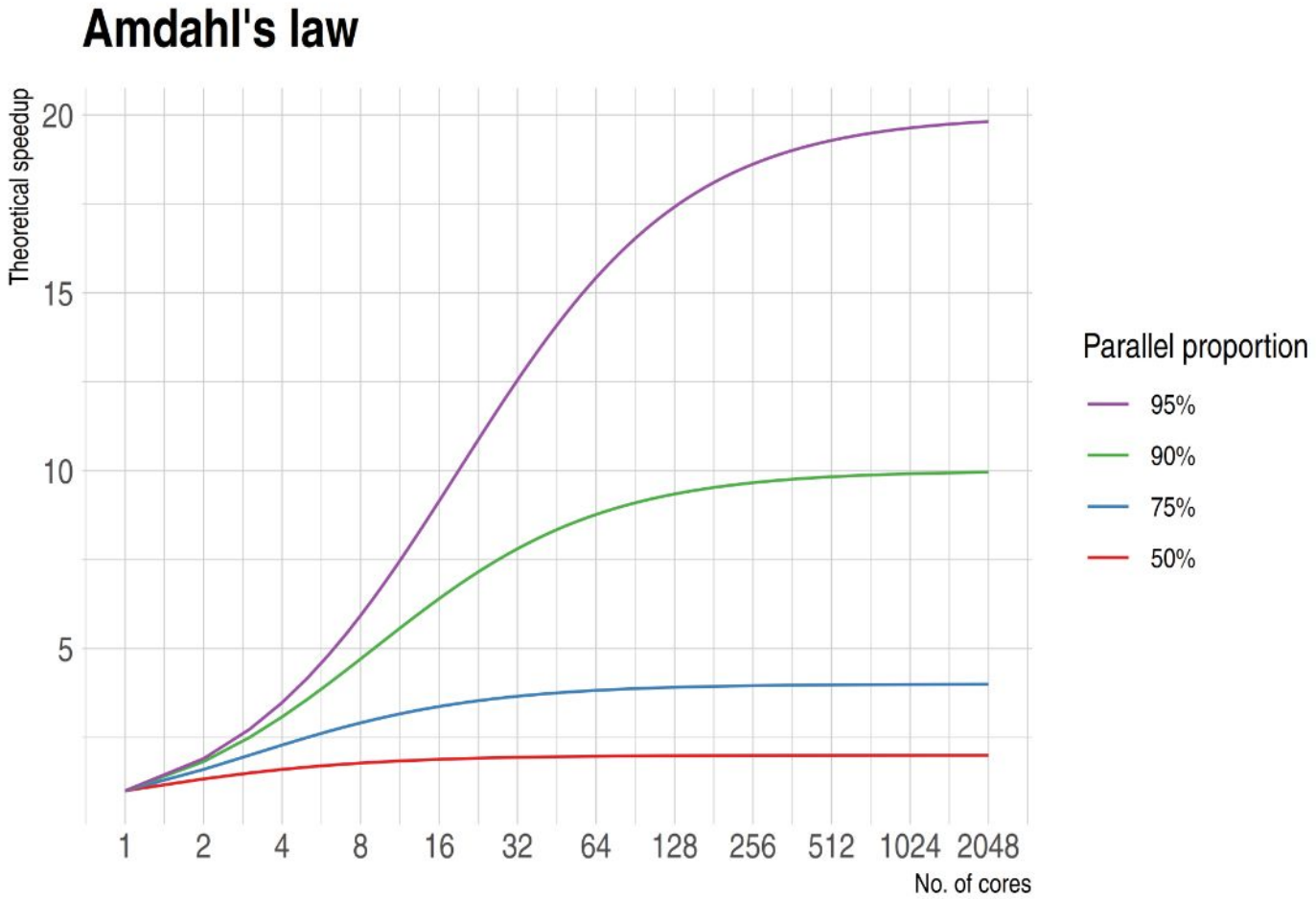


Source: (Eddelbuettel, 2021)

When should I go parallel?

- *Computational problems are easy to break up into smaller chunks (code chunks are independent and do not need to communicate in any way.)*

- *Thus, there are diminishing returns to parallelization, depending on the proportion of your code that can be run in parallel.*



Source: (Eddelbuettel, 2021)

Hertie School

Future package

- *The purpose of the **future package** is to allow users to switch effortlessly between evaluating code in serial or asynchronously (i.e. in parallel).*
- *"In programming, a future is an abstraction for a value that may be available at some point in the future."*
- *The state of a future can be unresolved or resolved, depending on which strategy is used to evaluate them.*

Strategies to evaluate futures

- ***Sequential:*** *Resolves futures sequentially in the current R process.*

- ***Multisession:*** *Resolves futures asynchronously (in parallel) in separate R sessions running in the background on the same machine.*

- ***Multicore:*** *Resolves futures asynchronously (in parallel) in separate forked R processes running in the background on the same machine. Not supported on Windows, and can also cause problems in an IDE or GUI like RStudio.*

- ***Cluster:*** *Resolves futures asynchronously (in parallel) in separate R sessions running typically on one or more machines.*

Hertie School

Explicit and implicit Futures

```
library(future)
plan(multisession)
a %<-% sum( 1:50 )
print(a)
```

```
## [1] 1275
```

Implicit

```
library(future)
plan(multisession)
a <- future({sum( 1:50 )})
b<- value(a)
print(b)
```

```
## [1] 1275
```

Explicit

Hertie School

Future Ecosystem

Future provides the framework for other packages to implement parallel versions of their functions.

- *Future.apply package (implementation of the apply() collection that can be resolved using Future)*
- *Furrr package (implementations of the family of map() functions from purrr that can be resolved using Future)*

Hertie School

Future package is not opinionated!

You can do the same thing with any syntax

- *forloops*
- *lapply* *e.g. future_lapply()*
- *map* *e.g. future_map()*
- *replicate*
- *No loop at all*

Summary: Key Benefits of Parallel Programming

1. ***IT MODELS THE REAL WORLD***
   The world around us isn't serial. Things don't happen one at a time, waiting for one event to finish before the next one starts.
2. ***SAVES TIME***
   Serial computing forces fast processors to do things inefficiently. It's like using a Ferrari to drive 20 oranges from Berlin to Bonn one orange at a time.
3. ***SAVES MONEY***
   By saving time, parallel computing makes things cheaper. When we scale up a system to billions of operations - bank software, for example - we see massive cost savings.
4. ***SOLVE MORE COMPLEX OR LARGER PROBLEMS***
   Computing is maturing. With AI and big data, a single web app may process millions of transactions every second.
5. ***LEVERAGE REMOTE RESOURCES***
   With parallel processing, multiple computers with several cores each can sift through many times more real-time data than serial computers working on their own.

Credits: www.hp.com

Hertie School

Multi Core Processors

- ***CPU on computers today have multiple cores on them***
- ***The first multicore processor invented in 2001***
- ***R language developed in 1993***

So..

How many cores do you have? (aka workers)
detectCores()

But before..

Hertie School

Packages to be familiar with

```
library(tictoc)
library(parallel)
library(future.apply)
library(furrr)
plan(multisession)# telling R that we want to implement the iteration in parallel
```

Hertie School

Assessing time saved

Assessing time saved through parallel programming

Lets assess the potential time saved through parallel programming by creating a test function called standard_function()

```{r}
standard_function =
  function(x) {
    x_sq = x^2
    d = tibble(value = x, value_squared = x_sq)
    Sys.sleep(2)
    return(d)
  }

```

Let's iterate over this function using the standard lapply() method that we're already familiar with by now. Note that this iteration will be execute in serial. We'll use the tictoc package to record timing.

```{r}
tic()
serial_func = lapply(1:12, standard_function) %>% bind_rows()
toc()
```

```
 24.087 sec elapsed
```

As expected, the iteration took about 24 seconds to run because of the enforced break after every sequential iteration (i.e. Sys.sleep(2)). On the other hand, this means that we can easily speed things up by iterating in parallel.

Hertie School

Assessing time saved

Parallel Programming with 'lapply'

Now lets implement the parallel iteration using the future.apply package. Note that the parameters of the problem are otherwise unchanged.

```{r}
tic()
future_func = future_lapply(1:12, standard_function) %>% bind_rows() #slightly amend the previous lapply() function
toc(log = TRUE)
```

```
8.213 sec elapsed
```

We see that only with a minor modification of our syntax and using the plan(multisession) command we have reduced the time taken by R to only a quarter of the original time.

Parallel Programming with 'purrr'

If you prefer the purrr::map() family of functions for iteration and are feeling left out; don't worry. The furrr package has you covered. Once again, the syntax for these parallel functions will be very little changed from their serial versions. We simply have to tell R that we want to run things in parallel with plan(multisession) and then slightly amend our map call to future_map_dfr().

```{r}
tic()
furrr_func = future_map_dfr(1:12, standard_function)
toc()
```

```
8.268 sec elapsed
```

Hertie School

References

- Eddelbuettel, Dirk. “Parallel Computing with R: A Brief Review.” *WIREs Computational Statistics* 13, no. 2 (March 2021). https://doi.org/10.1002/wics.1515.
- *A Future for R: A Comprehensive Overview*. Accessed November 15, 2022. https://cran.r-project.org/web/packages/future/vignettes/future-1-overview.html.
- “Apply Mapping Functions in Parallel Using Futures.” Accessed November 15, 2022. https://furrr.futureverse.org/index.html.
- R Core Team. “Package parallel,” June 8, 2022
- “How-to Go Parallel in R – Basics + Tips | G-Forge.” Accessed November 15, 2022. https://gforge.se/2015/02/how-to-go-parallel-in-r-basics-tips/.

And now we move to R for some examples...