



SERVICES AND CODE-PARALLELIZATION IN R

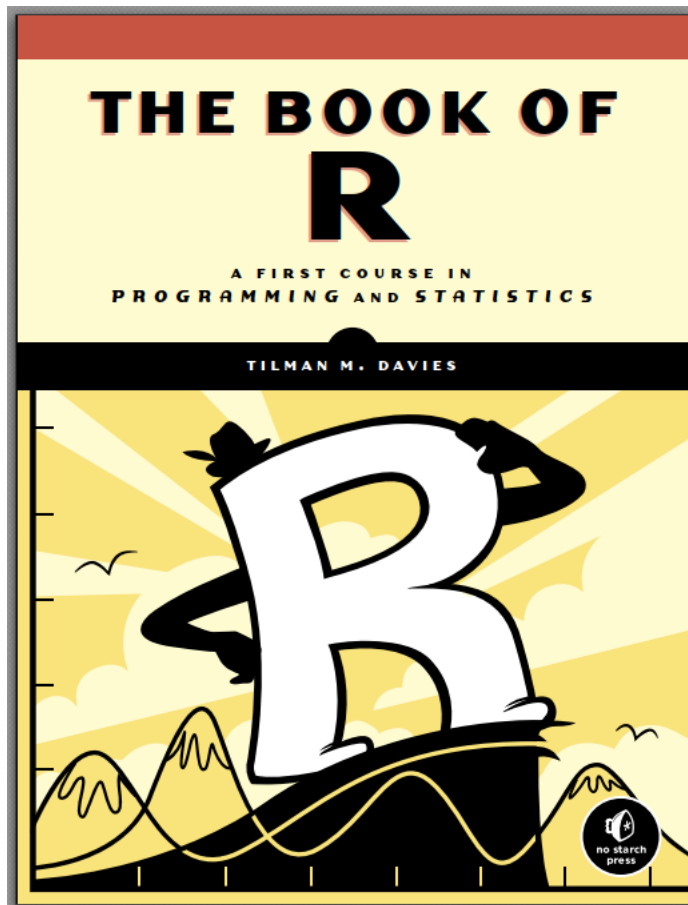
DR. SOFIA GIL-CLAVEL

1. Recap of the previous session
2. Intro to parallelization
3. Parallelizing in R
4. Servers: JupyterHub
5. When parallelizing is a bad/good idea

1. RECAP OF THE PREVIOUS SESSION



WE FOLLOWED:



Davies, Tilman M. *The Book of R: A First Course in Programming and Statistics*. No Starch Press, 2016.



CONTROL SEQUENCES



To write more sophisticated programs with R, you'll need to control the flow and order of execution in your code. One fundamental way to do this is to make the execution of certain sections of code dependent on a *condition*. Another basic control mechanism is the *loop*, which repeats a block of code a certain number of times. In this chapter, we'll explore these core programming techniques using if-else statements, for and while loops, and other control structures.

IN SUMMARY

The control sequences are divided into:

➤ Bifurcation statements:

- If/else - The if statement is the key to controlling exactly which operations are carried out in a given chunk of code.
- Loops:
 - for: Runs a loop a finite number of times.
 - while: Runs a loop as long as a condition is true
- Other control mechanisms:
 - break/next:
 - repeat (break): The repeat structure executes a loop infinitely. The only way to end the loop is to call the break function inside it.

Control structures describe the order in which statements or groups of statements are executed.

Function/operator	Brief description
if(){ }	Conditional check
if(){ } else { }	Check and alternative
ifelse	Element-wise if-else check
break	Exit explicit loop
next	Skip to next loop iteration
repeat{ }	Repeat code until break



FROM PROBLEMS TO PSEUDOCODE AND TO R-CODE

Useful for:

- Writing algorithms
- Writing functions



ANALYSIS AND SPECIFICATION OF THE PROBLEM

- Analysis of a problem is required to ensure that the problem is well defined and clearly understood.
- Formulating a problem specification requires an accurate and as complete description as possible of the problem input (information available for problem resolution) and the required output.
- A good problem specification needs to answer the following questions:
 - ✓ What should the program do?
 - ✓ What output should it produce?
 - ✓ What inputs are required to get the desired outputs?



DESIGN

- The solution design phase begins once the program specification is in place and consists of formulating the steps to solve the problem.
- Designing a solution requires the use of algorithms. An algorithm is a set of instructions or steps to solve a problem. The algorithms must process the data necessary for the resolution of the problem.
- Programs will be composed of algorithms that manipulate or process information.
- A good technique for designing an algorithm is to break down the problem into smaller, simpler subtasks or subtasks, until you get to subtasks that are easier to program.



ALGORITHMS & PSEUDOCODE

Algorithms are usually written in pseudocode, which consists of a set of words that represent tasks to be performed and a syntax of use like any language.



WRITING FUNCTIONS

When the same algorithm must be applied to different values, it is better to encapsulate the code into a function.

function {base}

R Documentation

Function Definition

Description

These functions provide the base mechanisms for defining new functions in the **R** language.

Usage

```
function( arglist ) expr
\ ( arglist ) expr
return(value)
```

Arguments

<code>arglist</code>	empty or one or more (comma-separated) 'name' or 'name = expression' terms and/or the special token <u><code>...</code></u> .
<code>expr</code>	an expression.
<code>value</code>	an expression.



WRITING FUNCTIONS

When the same algorithm must be applied to different values, it is better to encapsulate the code into a function.

function {base}

R Documentation

Function Definition

Description

These functions provide the base mechanisms for defining new functions in the **R** language.

Usage

```
function( arglist ) expr  
\( arglist ) expr  
return(value)
```

Arguments

`arglist` empty or one or more (comma-separated) 'name' or 'name = expression' terms and/or the special token

`expr` an expression.

`value` an expression.

This is an ellipsis!



R PROPERTIES TO OPTIMIZE LOOPS

Using vectors to optimize loops:

- Colon operator “:”
- Sequence generation “seq()”
- Repetition “rep()”

R is vector oriented:

- Vector oriented behavior
- Vector oriented functions



COMMON OPERATORS

Operators					
Aritmetic		Comparative		Logical	
+	addition	<	less than	! x	logic NO
-	subtraction	>	more than	x & y	element-wise AND
*	multiplication	<=	less or equal than	x && y	single comparison AND
/	division	>=	more or equal than	x y	element-wise OR
^	power	==	equal than	x y	single comparison OR
%%	integer division	!=	different than	xor(x,y)	exclusive OR

VECTOR-ORIENTED FUNCTIONS



<code>sum(x)</code>
<code>prod(x)</code>
<code>max(x)</code>
<code>min(x)</code>
<code>which.max(x)</code>
<code>which.min(x)</code>
<code>range(x)</code>
<code>length(x)</code>
<code>mean(x)</code>
<code>median(x)</code>
<code>var(x) o cov(x)</code>
<code>cor(x)</code>
<code>var(x, y) o cov(x, y)</code>
<code>cor(x, y)</code>

<code>round(x, n)</code>
<code>rev(x)</code>
<code>sort(x)</code>
<code>rank(x)</code>
<code>log(x, base)</code>
<code>scale(x)</code>
<code>pmin(x, y, ...)</code>
<code>pmax(x, y, ...)</code>
<code>cumsum(x)</code>
<code>cumprod(x)</code>
<code>cummin(x)</code>
<code>cummax(x)</code>

<code>match(x, y)</code>
<code>which(x == a)</code>
<code>choose(n, k)</code>
<code>na.omit(x)</code>
<code>na.fail(x)</code>
<code>unique(x)</code>
<code>table(x)</code>
<code>subset(x, ...)</code>
<code>sample(x, size)</code>



SPEED-UP CODE USING APPLY, LAPPLY, ETC.

In some situations, especially for loops, you can avoid some of the details associated with explicit looping by using the apply function and other similar functions.



APPLY

The apply function is the most basic form of implicit looping—it takes a function and applies it to each margin of an array.

Arguments

`apply {base}`

R Documentation

Apply Functions Over Array Margins

Description

Returns a vector or array or list of values obtained by applying a function to margins of an array or matrix.

Usage

```
apply(X, MARGIN, FUN, ..., simplify = TRUE)
```

`X`

an array, including a matrix.

`MARGIN`

a vector giving the subscripts which the function will be applied over. E.g., for a matrix 1 indicates rows, 2 indicates columns, `c(1, 2)` indicates rows and columns. Where `X` has named dimnames, it can be a character vector selecting dimension names.

`FUN`

the function to be applied: see ‘Details’. In the case of functions like `+`, `%*%`, etc., the function name must be backquoted or quoted.

`...`

optional arguments to `FUN`.

`simplify`

a logical indicating whether results should be simplified if possible.



LAPPLY AND SAPPLY

Using `lapply`, you can apply the function to each element of a list. The returned value is itself a list. Another variant, `sapply`, returns the same results as `lapply` but in an array form.

You can think of a data frame as a list with values (columns) consisting of vectors of the same length.



EXTRA ARGUMENTS?

All of R's apply functions allow for additional arguments to be passed to FUN; most of them do this via an ellipsis, i.e. the “...” at the end of the function's argument.

2. INTRODUCTION TO PARALLELIZATION



WE ARE GOING TO FOLLOW:

A tutorial I wrote some years ago:

https://github.com/SofiaG11/GWDG_MPIDR/blob/master/part2.md

Hands-on lab (Second Part)

Sofia Gil [info](#)

January 15, 2019

- [Introduction](#)
- [1. Writing efficient loops](#)
- [2. Paralellizing](#)
- [3. Submitting R scripts into the cluster](#)
- [References](#)

Introduction

It is well-known that when we are using *R* we always have to avoid **for** loops. Due to the calls that *R* does to *C* functions, therefore it is better to vectorize everything.

WHY DO WE NEED TO PARALLELIZE?



Imagine you're a restaurant owner and the only employee there. Guests are coming in and it's only your responsibility to show them to the table, take their order, prepare the food, and serve it. The problem is - **there's only so much you can do**. The majority of the guests will be waiting a long time since you're busy fulfilling earlier orders.

On the other hand, if you employ two chefs and three waiters, you'll drastically reduce the wait time. This will also allow you to serve more customers at once and prepare more meals simultaneously.

The first approach (when you do everything by yourself) is what we refer to as **single-threaded execution** in computer science, while the second one is known as **multi-threaded execution**. The ultimate goal is to find the best approach for your specific use case. If you're only serving 10 customers per day, maybe it makes sense to do everything yourself. But if you have a striving business, you don't want to leave your customers waiting.

You should also note that just because the second approach has 6 workers in total (two chefs, three waiters, and you), it doesn't mean you'll be exactly six times faster than when doing everything by yourself. **Managing workers has some overhead time, just like managing CPU cores does.**



EXAMPLE: CALCULATE THE ADDITION OF ALL THE NUMBERS FROM 1 TO N.

Pseudocode:

Input: n

Output: The addition

Pseudocode:

- Start the variables i and sum
- $i=1$
- $sum=0$
- **while**($i \leq n$)
 - $sum = sum + i$
 - $i = i + 1$
- **return** sum



EXAMPLE: CALCULATE THE ADDITION OF ALL THE NUMBERS FROM 1 TO N.

Pseudocode:

Input: n

Output: The addition

Pseudocode:

- Start the variables i and sum
- $i=1$
- $sum=0$
- $while(i \leq n)$
 - $sum = sum + i$
 - $i = i + 1$
- return sum

How would you parallelize this problem?



EXAMPLE: CALCULATE THE ADDITION OF ALL THE NUMBERS FROM 1 TO N.

Pseudocode:

Input: n

Output: The addition

Pseudocode:

- Start the variables i and sum
- $i=1$
- $sum=0$
- ```
while(i<=n)
 • $sum=sum + i$
 • $i=i+1$
```
- return sum

How would you parallelize this problem?



The secret is to focus on the loop. If you pass it as a loop parameter, then you can parallelize it.





# EXAMPLE: CALCULATE THE ADDITION OF ALL THE NUMBERS FROM 1 TO N.

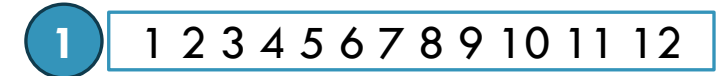
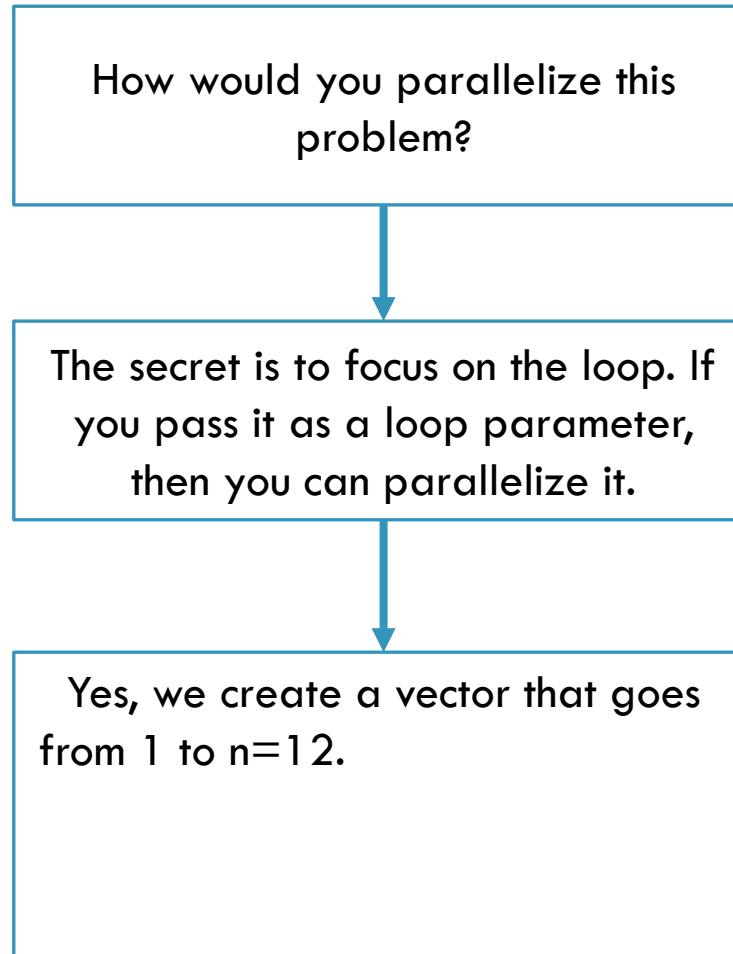
## Pseudocode:

**Input:** n

**Output:** The addition

## Pseudocode:

- Start the variables i and sum
- $i=1$
- $sum=0$
- **while**( $i \leq n$ )
  - $sum = sum + i$
  - $i = i + 1$
- **return** sum





# EXAMPLE: CALCULATE THE ADDITION OF ALL THE NUMBERS FROM 1 TO N.

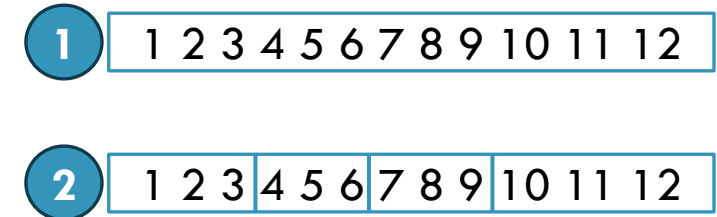
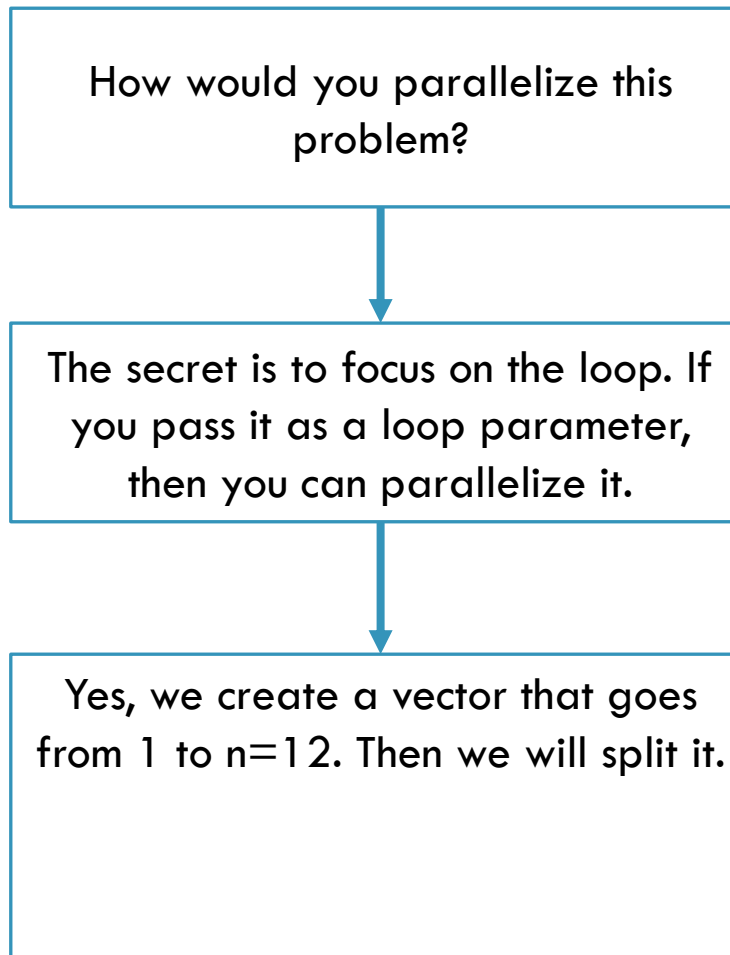
## Pseudocode:

**Input:** n

**Output:** The addition

## Pseudocode:

- Start the variables i and sum
- $i=1$
- $sum=0$
- **while**( $i \leq n$ )
  - $sum = sum + i$
  - $i = i + 1$
- **return** sum





# EXAMPLE: CALCULATE THE ADDITION OF ALL THE NUMBERS FROM 1 TO N.

## Pseudocode:

**Input:** n

**Output:** The addition

## Pseudocode:

- Start the variables i and sum
- $i=1$
- $sum=0$
- **while**( $i \leq n$ )
  - $sum = sum + i$
  - $i = i + 1$
- **return** sum

How would you parallelize this problem?

The secret is to focus on the loop. If you pass it as a loop parameter, then you can parallelize it.

Yes, we create a vector that goes from 1 to  $n=12$ . Then we will split it. Each section value will be calculated independently.

1 1 2 3 4 5 6 7 8 9 10 11 12

2 1 2 3 4 5 6 7 8 9 10 11 12

3  $sum(1\ 2\ 3) \rightarrow 6$   
 $sum(4\ 5\ 6) \rightarrow 15$   
 $sum(7\ 8\ 9) \rightarrow 24$   
 $sum(10\ 11\ 12) \rightarrow 33$



# EXAMPLE: CALCULATE THE ADDITION OF ALL THE NUMBERS FROM 1 TO N.

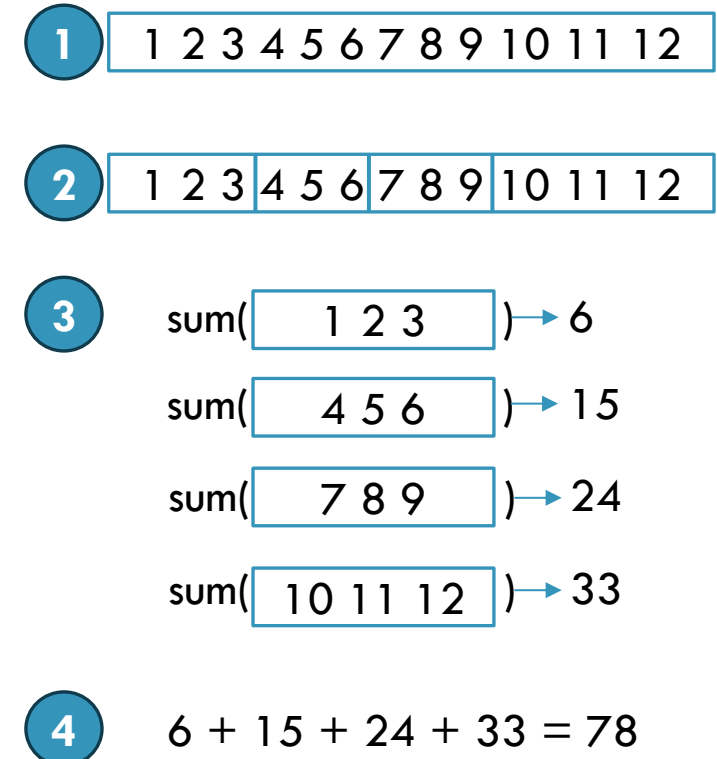
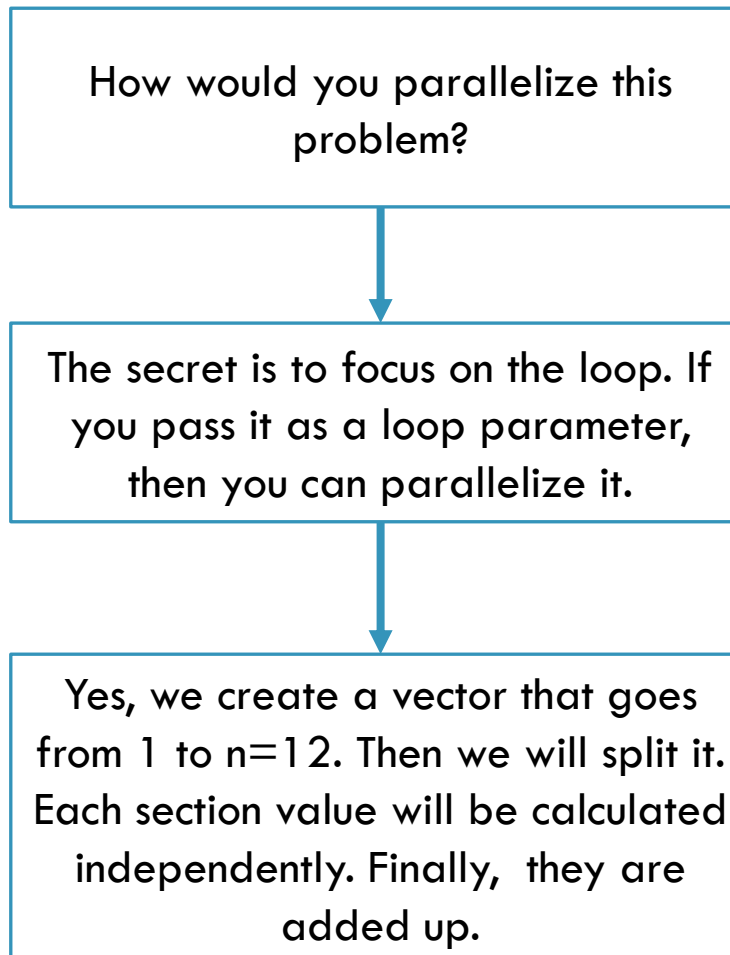
## Pseudocode:

**Input:** n

**Output:** The addition

## Pseudocode:

- Start the variables i and sum
- $i=1$
- $sum=0$
- while( $i \leq n$ )
  - $sum=sum + i$
  - $i=i+1$
- return sum





# NOTATION

1 1 2 3 4 5 6 7 8 9 10 11 12

2 1 2 3 4 5 6 7 8 9 10 11 12

3  $\text{sum}(1\ 2\ 3) \rightarrow 6$

$\text{sum}(4\ 5\ 6) \rightarrow 15$

$\text{sum}(7\ 8\ 9) \rightarrow 24$

$\text{sum}(10\ 11\ 12) \rightarrow 33$

4  $6 + 15 + 24 + 33 = 78$



# NOTATION

1 1 2 3 4 5 6 7 8 9 10 11 12

2 1 2 3 4 5 6 7 8 9 10 11 12

3

sum( 1 2 3 ) → 6

sum( 4 5 6 ) → 15

sum( 7 8 9 ) → 24

sum( 10 11 12 ) → 33

4  $6 + 15 + 24 + 33 = 78$



# NOTATION

1 1 2 3 4 5 6 7 8 9 10 11 12

2 1 2 3 4 5 6 7 8 9 10 11 12

3

|                 |      |
|-----------------|------|
| sum( 1 2 3 )    | → 6  |
| sum( 4 5 6 )    | → 15 |
| sum( 7 8 9 )    | → 24 |
| sum( 10 11 12 ) | → 33 |

4  $6 + 15 + 24 + 33 = 78$

**Each of this is a thread** and **each is processed in a core**. A CPU **core** is a **physical processing unit within a processor**, while a thread is a virtual sequence of instructions that a core can execute.



# NOTATION

1 1 2 3 4 5 6 7 8 9 10 11 12

2 1 2 3 4 5 6 7 8 9 10 11 12

3

|                 |      |
|-----------------|------|
| sum( 1 2 3 )    | → 6  |
| sum( 4 5 6 )    | → 15 |
| sum( 7 8 9 )    | → 24 |
| sum( 10 11 12 ) | → 33 |

4  $6 + 15 + 24 + 33 = 78$

So, we split the problem  
into **4 threads** that were  
executed in **4 cores**.



**Each of this is a thread** and **each is processed in a core**. A CPU **core** is a **physical processing unit within a processor**, while a thread is a virtual sequence of instructions that a core can execute.





# NOTATION

1 1 2 3 4 5 6 7 8 9 10 11 12

2 1 2 3 4 5 6 7 8 9 10 11 12

3

|                 |      |
|-----------------|------|
| sum( 1 2 3 )    | → 6  |
| sum( 4 5 6 )    | → 15 |
| sum( 7 8 9 )    | → 24 |
| sum( 10 11 12 ) | → 33 |

4  $6 + 15 + 24 + 33 = 78$

So, we split the problem into **4 threads** that were executed in **4 cores**.

We could have split the problem into **4 threads** that were executed in **2 cores**.

Each of this is a **thread** and each is processed in a **core**. A CPU **core** is a physical processing unit within a processor, while a thread is a virtual sequence of instructions that a core can execute.



# NOTATION

1 1 2 3 4 5 6 7 8 9 10 11 12

2 1 2 3 4 5 6 7 8 9 10 11 12

3

|                 |      |
|-----------------|------|
| sum( 1 2 3 )    | → 6  |
| sum( 4 5 6 )    | → 15 |
| sum( 7 8 9 )    | → 24 |
| sum( 10 11 12 ) | → 33 |

4  $6 + 15 + 24 + 33 = 78$

So, we split the problem into **4 threads** that were executed in **4 cores**.

Each of this is a **thread** and each is processed in a **core**. A CPU **core** is a physical processing unit within a processor, while a thread is a virtual sequence of instructions that a core can execute.

We could have split the problem into **4 threads** that were executed in **2 cores**.

3

|                 |      |
|-----------------|------|
| sum( 1 2 3 )    | → 6  |
| sum( 4 5 6 )    | → 15 |
| sum( 7 8 9 )    | → 24 |
| sum( 10 11 12 ) | → 33 |



# NOTATION

1 1 2 3 4 5 6 7 8 9 10 11 12

2 1 2 3 4 5 6 7 8 9 10 11 12

3

|                 |      |
|-----------------|------|
| sum( 1 2 3 )    | → 6  |
| sum( 4 5 6 )    | → 15 |
| sum( 7 8 9 )    | → 24 |
| sum( 10 11 12 ) | → 33 |

4  $6 + 15 + 24 + 33 = 78$

So, we split the problem into **4 threads** that were executed in **4 cores**.

Each of this is a **thread** and each is processed in a **core**. A CPU **core** is a physical processing unit within a processor, while a thread is a virtual sequence of instructions that a core can execute.

We could have split the problem into **4 threads** that were executed in **2 cores**.

3

|                 |      |
|-----------------|------|
| sum( 1 2 3 )    | → 6  |
| sum( 4 5 6 )    | → 15 |
| sum( 7 8 9 )    | → 24 |
| sum( 10 11 12 ) | → 33 |

The number of cores depends on the computer hardware, while the threads are established by you.



# EXERCISE 2.1:

Last session, you solved the following problems:

1. You want to get the mean value of each column of a data frame.
2. Write a program that generates 30 databases and stores them in three different folders depending on the number of the database. The first 10 databases go in folder “first”, the databases between 11 and 20 go to “second”, and the last ones go into folder “third”. Hint: The databases can be a modification of an existing R-database.



# EXERCISE 2.1:

**How would you parallelize them? Write the pseudocode:**

1. You want to get the mean value of each column of a data frame.
2. Write a program that generates 30 databases and stores them in three different folders depending on the number of the database. The first 10 databases go in folder “first”, the databases between 11 and 20 go to “second”, and the lasts ones go into folder “third”. Hint: The databases can be a modification of an existing R-database.



# ADVANTAGES OF PARALLELIZING

- **Speed-up computation:** If you're old enough to remember the days of single-core CPUs, you know that upgrading to a multi-core one made all the difference. Simply put, more cores equals more speed, that is if your R scripts take advantage of parallel processing (they don't by default).
- **Efficient use of resources:** Modern CPUs have multiple cores, and your R scripts only utilize one by default. Now, do you think that one core doing all the work while the rest sit idle is the best way to utilize compute resources? Likely no, you're far better off by distributing tasks across more cores.
- **Working on larger problems:** When you can distribute the load of processing data, you can handle larger datasets and more complex analyses that were previously out of reach. This is especially the case if you're working in data science since modern datasets are typically huge in size.



# THINGS TO CONSIDER

1. Parallelizing only works when the processes do not need to share data. This means that **each process must be independent from the others**.
2. The Operating System matters, i.e., the speed at which it process goes depends on whether you are using Mac, Windows, or Linux. From all of them, **Linux is the fastest** and it allows to fork (we will cover this later).
3. Before parallelizing **you always need to:**
  - Think about the number of processes you will run independently, i.e. think about the number of threads and cores.
  - Check the size of the data each process will handle.
  - Add up all the sizes.
  - Is this less than the amount of RAM your computer has?
  - Are you leaving enough RAM for your computer to run the processes without overflow?



**5 MINS BREAK**





# 3. PARALLELIZING IN R

---



# NOW LET'S PARALLELIZE IN R!

There are several packages to parallelize in R. In this session, we will use “doParallel”.

The package doParallel is very flexible and it allows to pass data, functions, and packages to each of the cores. This is very useful when we want to do more than simple math on vectors.

The **doParallel package** is a “**parallel backend**” for the foreach package. It provides a mechanism needed to execute foreach loops in parallel. The foreach package must be used in conjunction with a package such as doParallel in order to execute code in parallel.



# STARTING THE TASK

To use `doParallel`, we need **register a parallel backend to use**, otherwise `foreach` will execute tasks sequentially, even when the `%dopar%` operator is used.

- To register `doParallel` to be used with `foreach`, **you must call the `registerDoParallel` function**.
- If you call this with no arguments, **on Windows you will get three workers and on Unix-like systems you will get a number of workers equal to approximately half the number of cores on your system**.
- You can also specify a cluster (as created by the `makeCluster` function) or a number of cores. The `cores` argument specifies the number of worker processes that `doParallel` will use to execute tasks, which will by default be equal to one-half the total number of cores on the machine. You don't need to specify a value for it, however.



# GETTING INFORMATION ABOUT THE PARALLEL BACKEND

- `detectCores()`: This attempts to detect the number of available CPU cores.
- `getDoParWorkers()`: To find out how many workers foreach is going to use.

In this jargon, a worker refers to a core. So, “`getDoParWorkers`” tells you the number of cores that you will be using.

- `getDoParRegistered()`: Checks if the `doPar` backend has been registered. `TRUE` means the work will be executed in parallel.



# USING FOREACH


## Pseudocode:

**Input:** n


**Output:** The addition

## Pseudocode:

- Start the variables i and sum
- i=1
- sum=0
- while(i<=n)
  - sum=sum + i
  - i=i+1
- return sum



```
SUM<-foreach(i=1:12,.combine = '+')%do%{
 sum(i,na.rm = TRUE)
}
```





# USING %DOPAR%

## Pseudocode:

**Input:** n

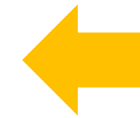
**Output:** The addition

## Pseudocode:

- Start the variables i and sum
- i=1
- sum=0
- while(i<=n)
  - sum=sum + i
  - i=i+1
- return sum

registerDoParallel()

SUM<-foreach(i=1:12,.combine = '+' )%dopar%{  
 sum(i,na.rm = TRUE)  
}





# ENDING THE TASK

If you are using snow-like functionality, **you will want to stop your cluster when you are done using it.** The `doParallel` package's `.onUnload` function will do this automatically if the cluster was created automatically by `registerDoParallel`, but **if you created the cluster manually you should stop it using the “`stopCluster`” function.**

You can also **force the cluster to stop using: `stopImplicitCluster()`**



# USING %DOPAR%

## Pseudocode:

**Input:** n

**Output:** The addition

## Pseudocode:

- Start the variables i and sum
- i=1
- sum=0
- while(i<=n)
  - sum=sum + i
  - i=i+1
- return sum

```
registerDoParallel()
```

```
SUM<-foreach(i=1:12,.combine = '+')%dopar%{
 sum(i,na.rm = TRUE)
}
```

```
stopImplicitCluster()
```







# EXERCISE 3.1:

**Translate your pseudocode from exercise 2.1 using foreach and %dopar%:**

1. You want to get the mean value of each column of a data frame.
2. Write a program that generates 30 databases and stores them in three different folders depending on the number of the database. The first 10 databases go in folder “first”, the databases between 11 and 20 go to “second”, and the lasts ones go into folder “third”. Hint: The databases can be a modification of an existing R-database.

## 4. SERVERS: JUPYTERHUB

---



# WHY TO USE A COMPUTING SERVER

1. Computing servers tend to use Linux. Which, as mentioned earlier, is the fastest in terms of parallelization.
2. Computing servers tend to have more RAM and cores (threads) than standard laptops.
3. Computing servers tend to be on 24/7, so you can leave the process running and continue working on other stuff in the meantime. However, you should always check when the admin reboots the server, as that will kill your processes.



# VU - JUPYTERHUB

VU gives access to two computing servers:

## 1. High Performance Computer (HPC)

- To access the HPC, you need to know how to operate a computer terminal. To use it, you need to know how to schedule your jobs using a .sh file.

## 2. JupyterHub

- To access JupyterHub, you just need to open your favorite browser (perhaps delete the cookies) and open the JupyterHub webpage. This webpage offers a Graphic User Interface where you can choose your favorite programming software and work from there.

In this course, we will use JupyterHub, as the main point of this course is to teach you to parallelize and for that JupyterHub does a very good job. Furthermore, learning to use the terminal and to schedule jobs are courses on their own, which you can follow at VU-IT department:

<https://vu.nl/en/research/portal/research-impact-support-portal/high-performance-research-computing>



# EXERCISE 4.1:

1. Access JupyterHub and open Rstudio.: <https://hub.compute.vu.nl/>
  - You can follow this guide if you need to upload and download files: [https://societal-analytics.nl/blogs/20250201\\_computing-power/#jupyterhub](https://societal-analytics.nl/blogs/20250201_computing-power/#jupyterhub)
2. Write your code again in a new script in JupyterHub:
  - a) You want to get the mean value of each column of a data frame.
  - b) Write a program that generates 30 databases and stores them in three different folders depending on the number of the database. The first 10 databases go in folder “first”, the databases between 11 and 20 go to “second”, and the last ones go into folder “third”. Hint: The databases can be a modification of an existing R-database.



## EXERCISE 4.2:

Now, let us test which code is faster: the one in your computer or the one in JupyterHub? To measure the time, we will use the function “system.time()”.

Here is an example:

```
system.time({
 SUM<-foreach(i=1:12,.combine = '+')%dopar%{
 sum(i,na.rm = TRUE)}
 })['elapsed']
,
```



# WHY IS LINUX FASTER?

Linux generally tends to be faster than Windows for parallel processing due to its optimized process management and resource allocation, particularly its use of the fork system call for creating new processes.

This allows Linux to efficiently duplicate a process and its environment, which is advantageous in parallel computing.

Windows, on the other hand, relies on a more resource-intensive process creation method, which can impact performance, especially in scenarios requiring numerous parallel tasks.



# WHEN TO PARALLELIZE

**With small tasks, the overhead of scheduling the tasks and returning the results can be greater than the time to execute the task itself,** resulting in poor performance. This means that the problems we have solved so far don't exploit the parallel capabilities. They were just pedagogical examples, not a benchmark.

**Parallelizing must be used for big problems.** So, as a final task today, we will analyze the processing time for a big data problem.





Join us!



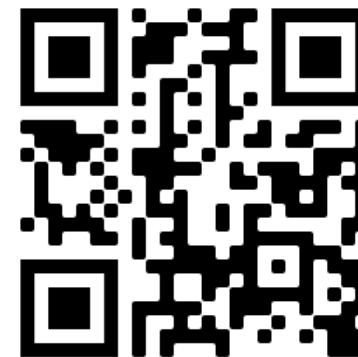
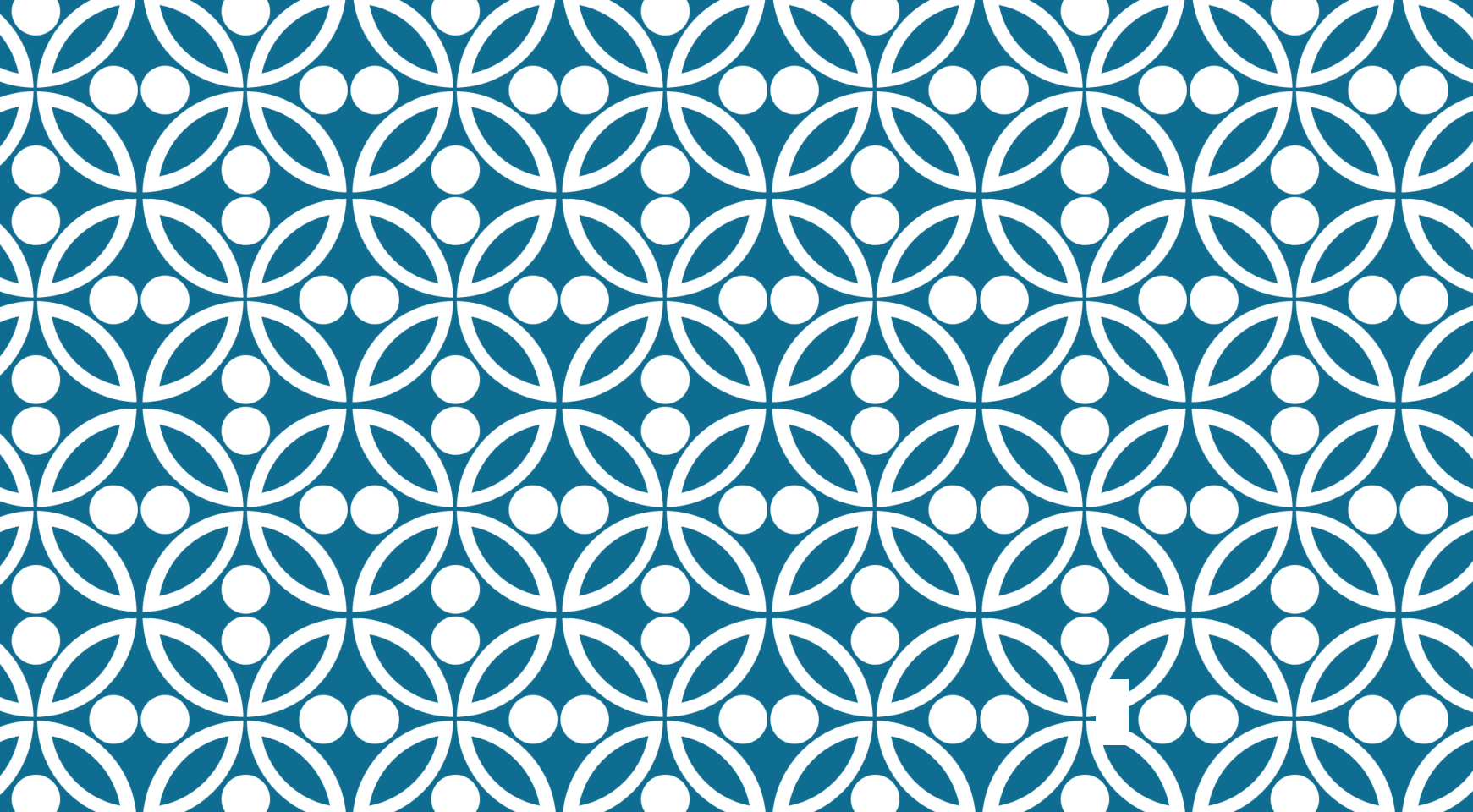
<https://forms.office.com/e/8Bgd2YsasJ>

All about the lab:

<https://societal-analytics.nl/>

Contact us at:

[analytics-lab.fsw@vu.nl](mailto:analytics-lab.fsw@vu.nl)



<https://sofiag1l.github.io/>

# THANKS!

Dr. Sofia Gil-Clavel