

# Procesamiento Masivo de datos: OpenMP I

**Alex Di Genova**

**12/09/2022**

# **OpenMP**

## **Introducción y funciones**

# OpenMP

## Introduction

- **Definición : OpenMP**

- OpenMP es una interfaz de programación de aplicaciones (API) para la paralelización de sistemas de memoria compartida, utilizando C, C++ o Fortran.
- La API consta de directivas de compilador para especificar y controlar la paralelización, aumentada con funciones de tiempo de ejecución y variables de entorno.
- Corresponde al usuario identificar el paralelismo e insertar las estructuras de control apropiadas en el programa (directivas).
- En C/C++, la directiva se basa en la construcción **#pragma omp**.

- **Syntax**

- **#pragma omp parallel** [clause[ clause], ...] new-line

*Structured block*

- Es responsabilidad del programador identificar qué parte(s) del código se selecciona(n) para ejecutar en paralelo y usar las diversas construcciones para garantizar resultados correctos.
- También se debe especificar la naturaleza (privada o compartida) o el "alcance" de las variables.

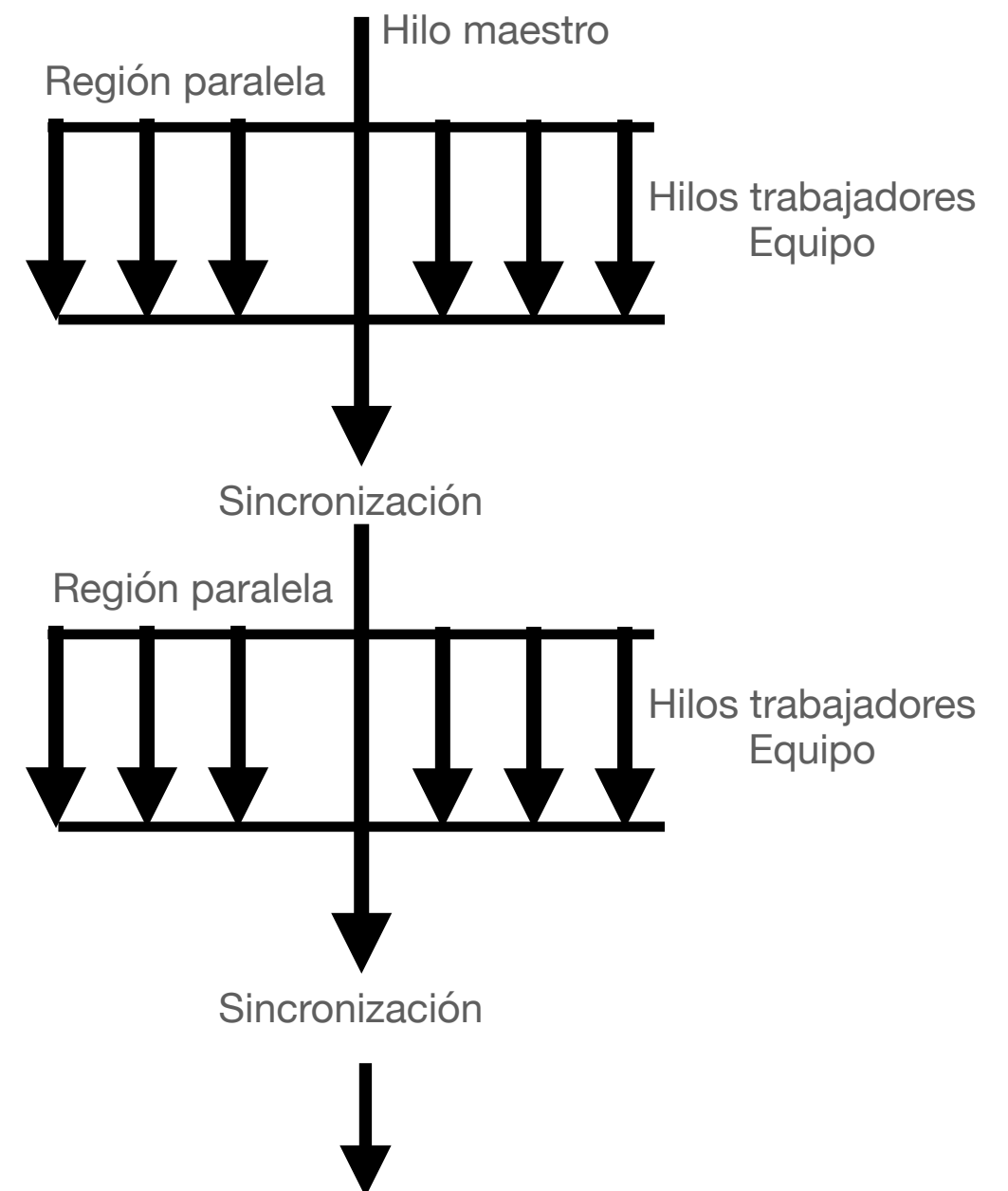
# OpenMP

## La región paralela

- **Región paralela**

- Un programa paralelo en OpenMP comienza y termina con la región paralela. Es la piedra angular de OpenMP.
- No hay límite en la cantidad de regiones paralelas, pero por razones de rendimiento es mejor mantener la cantidad de regiones al mínimo y hacerlas lo más grandes posible.
- El hilo que encuentra la región paralela se llama hilo **maestro**. Crea los hilos adicionales y está a cargo de la ejecución general.
- Los hilos que están activos dentro de una región paralela se denominan **equipos**. Varios equipos pueden estar activos simultáneamente.
  - **OMP\_NUM\_THREADS** (variable ambiente)
  - **omp\_set\_num\_threads()**
    - Función para modificar el numero de hilos
  - Clausula **num\_threads(<nt>)**
- Las instrucciones dentro de la región paralela son ejecutadas por todos los hilos.
- Fuera de las regiones paralelas, el hilo maestro ejecuta las partes del código en forma serial.
- La sincronización de hilos se produce en la barrera implícita al final de cada región paralela.

### Modelo de Ejecución OpenMP (fork-join)

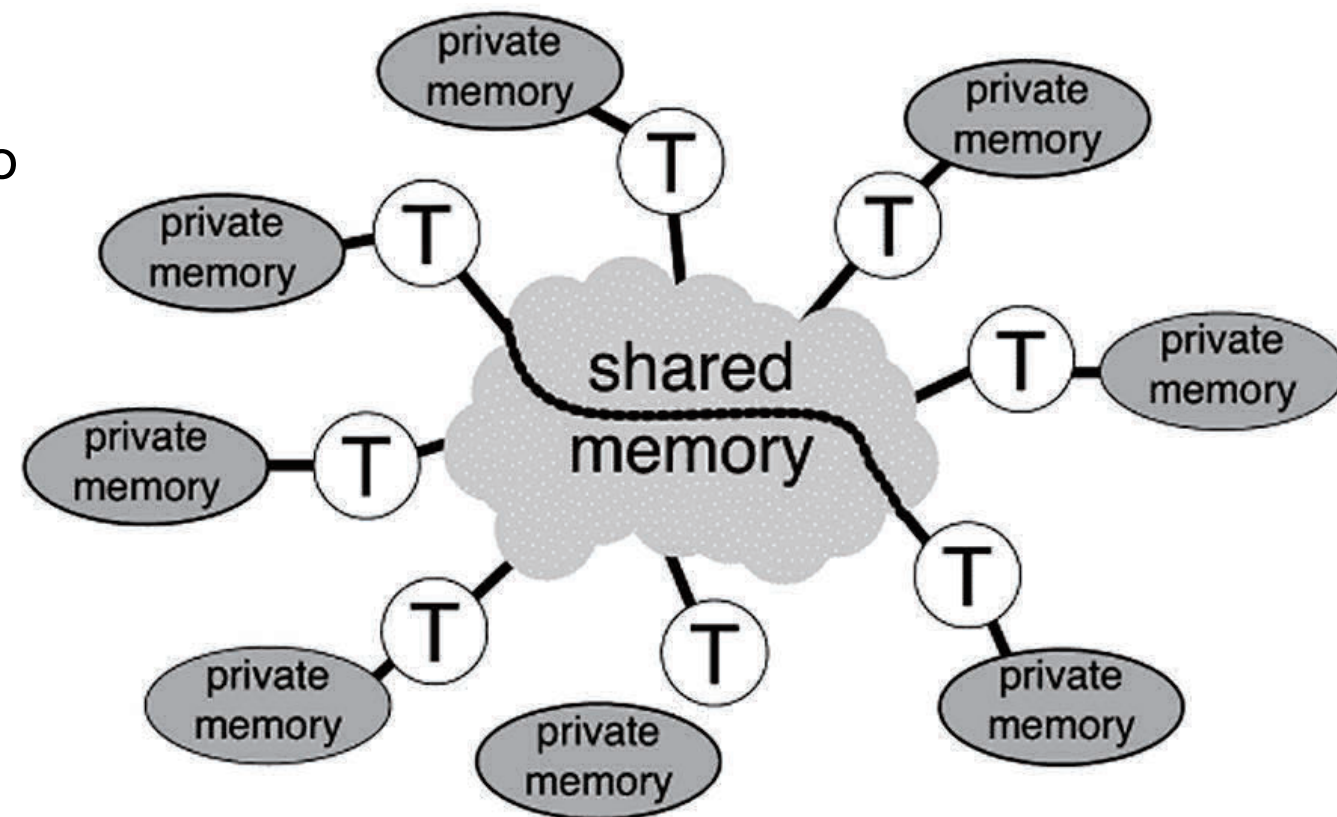


# OpenMP

## Modelo de Memoria

- **Modelo de Memoria**

- Un programa OpenMP tiene dos tipos elementales diferentes de memoria: privada y compartida.
- **Variables privadas.** Cada hilo tiene acceso único a su memoria privada y ningún otro hilo puede interferir. Nunca existe el riesgo de un conflicto de acceso con otro hilo. Aunque varios hilos pueden usar el mismo nombre para una variable privada, estas variables se almacenan en diferentes ubicaciones de memoria.
- **Variables compartidas.** Cada hilo puede leer, así como escribir, cualquier variable compartida. Es responsabilidad del programador manejar correctamente esta situación.
  - variables globales se comparten por defecto.



# OpenMP

## Construcciones de trabajo compartido

- Una construcción de trabajo compartido debe colocarse dentro de una región paralela. Al encontrar una construcción de trabajo compartido, el OpenMP distribuye el trabajo a realizar entre los hilos activos en la región paralela.
- **Construcción de ciclos**, proporciona una forma sencilla de asignar el trabajo asociado con las iteraciones de ciclos a los hilos.

- **#pragma omp for [clase,[], ...clause]**

- Las iteraciones del ciclo se distribuyen en los hilos y se ejecutan en paralelo.

- **Construcción de secciones**, son ideales para llamar a diferentes funciones en paralelo.

```
#pragma omp sections [clause[[],],clause]...
```

```
{
```

```
    #pragma omp section
```

```
        bloque de código
```

```
    #pragma omp section
```

```
        bloque de código
```

```
}
```

- **Construcciones únicas**, especifica que el bloque dado es ejecutado por un solo hilo. No se especifica qué hilo. Otros hilos omiten el bloque y esperan (barrera) a que finalice la construcción.

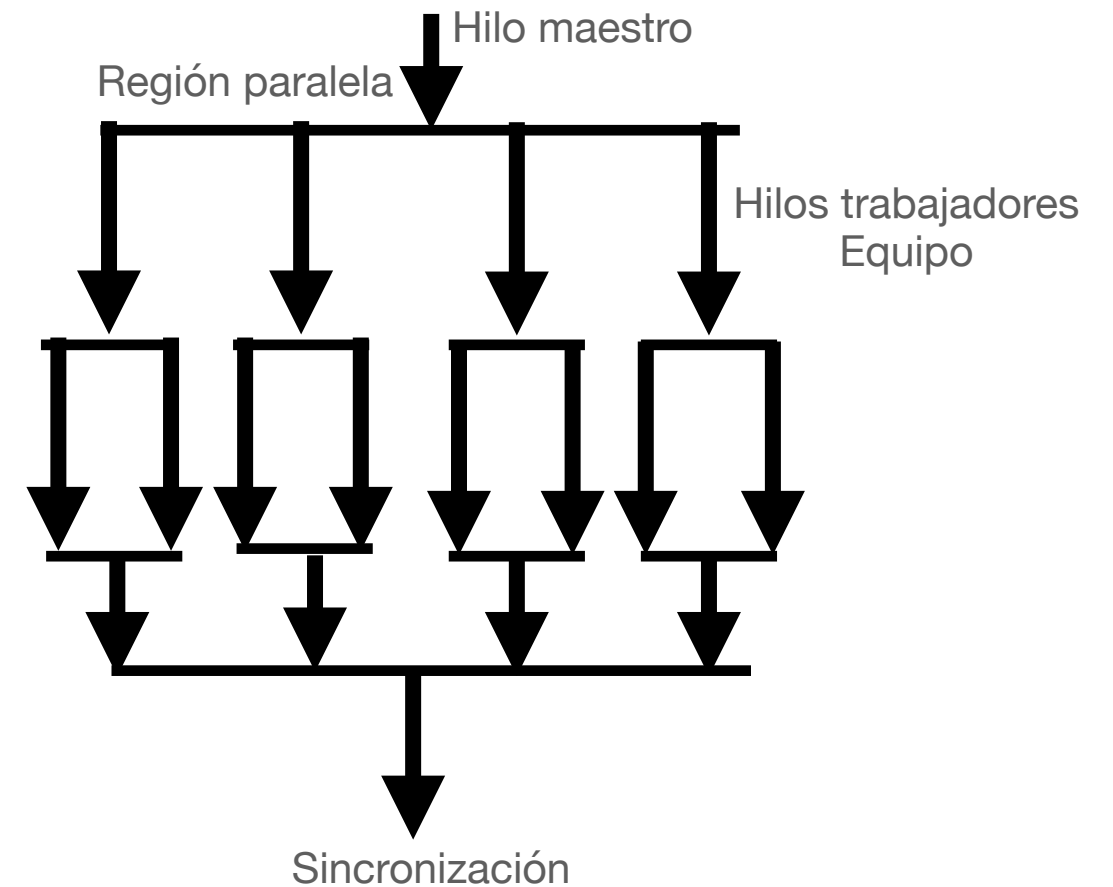
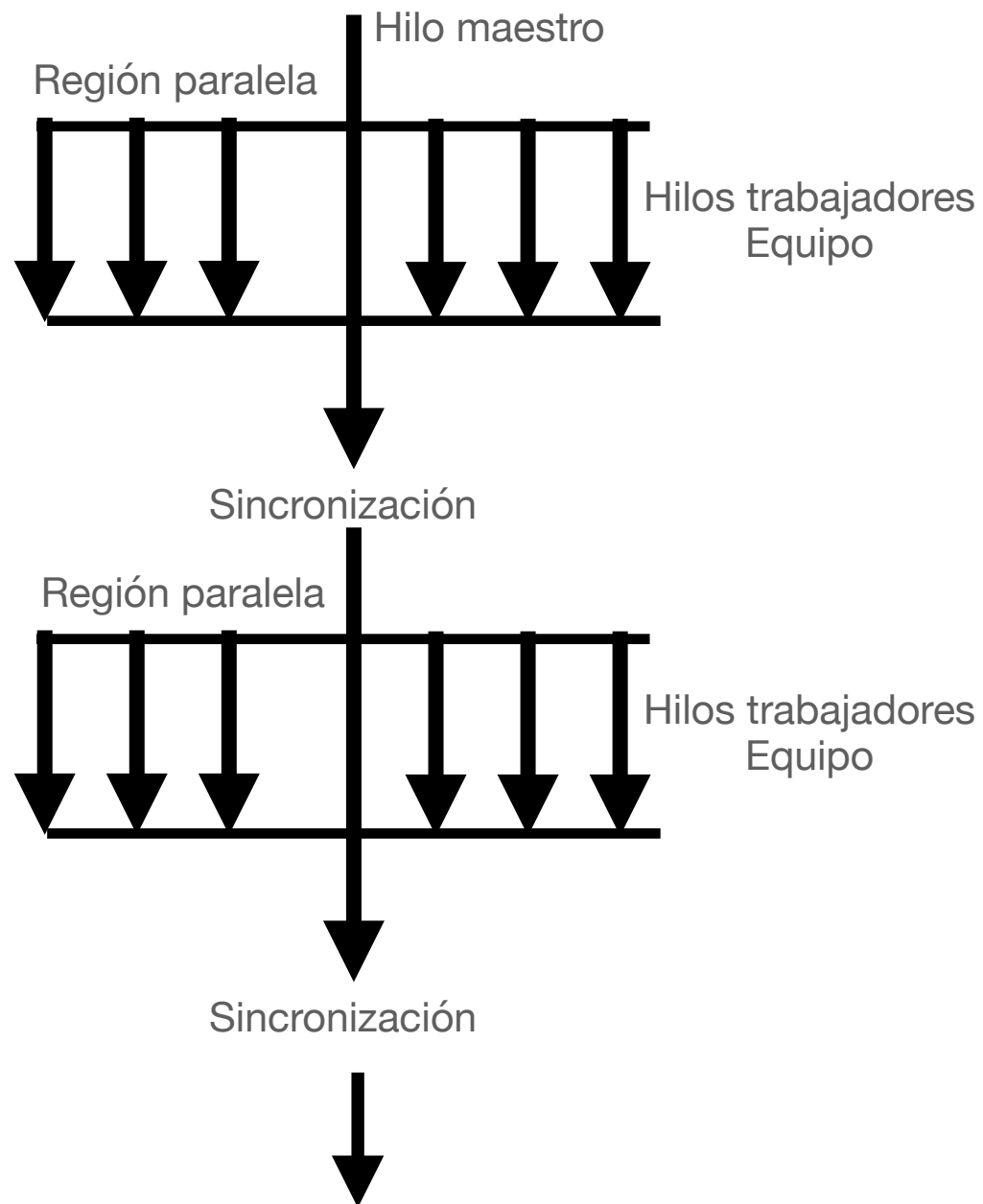
```
#pragma omp single
```

```
    Bloque de código
```

# OpenMP

## Paralelismo anidado

Modelo de Ejecución OpenMP (fork-join)



```
#pragma omp parallel num_threads(1)
{
    Work1();

    #pragma omp parallel num_threads(5)
    { //1 x 5 = 5 threads
        Work2();
    }
}
```

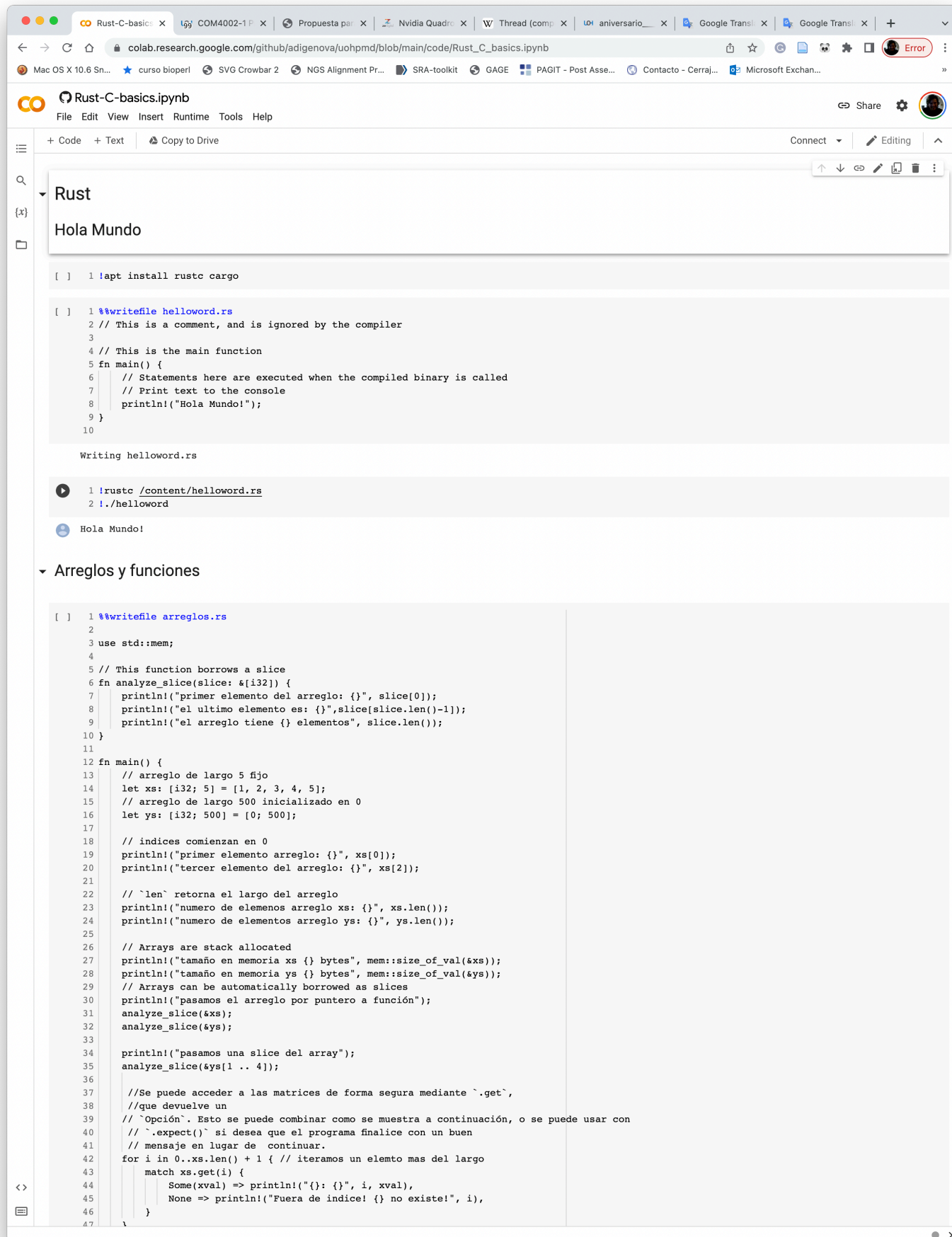
# OpenMP

## Sincronización

- **Contrucción barrera**, obliga a todos los hilos a esperar hasta que todos hayan alcanzado la región de barrera en el programa (**default**).
- **Construccion critica**, Una región crítica es un bloque de código ejecutado por todos los hilos, pero se garantiza que solo un hilo a la vez puede estar activo en la región.
- **La construcción atómica**, se utiliza para garantizar el acceso mutuamente excluyente a una ubicación de memoria específica, representada a través de una variable. Se garantiza que el acceso a esta ubicación será atómico.



# Google Colab



The screenshot shows a Google Colab notebook interface. The browser address bar displays the URL: `colab.research.google.com/github/adigenova/uohpmd/blob/main/code/Rust_C_basics.ipynb`. The notebook title is "Rust-C-basics.ipynb". The interface includes a menu bar (File, Edit, View, Insert, Runtime, Tools, Help) and a toolbar with options like "+ Code", "+ Text", and "Copy to Drive".

The notebook content is organized into sections. The first section, titled "Rust", contains a code cell with the following Rust code:

```
[ ] 1 !apt install rustc cargo

[ ] 1 %%writefile helloworld.rs
2 // This is a comment, and is ignored by the compiler
3
4 // This is the main function
5 fn main() {
6     // Statements here are executed when the compiled binary is called
7     // Print text to the console
8     println!("Hola Mundo!");
9 }
10
```

Below the code cell, the output shows "Writing helloworld.rs". A subsequent cell contains the commands to run the program:

```
1 !rustc /content/helloworld.rs
2 !./helloworld
```

The output of this cell is "Hola Mundo!".

The second section, titled "Arreglos y funciones", contains a code cell with the following Rust code:

```
[ ] 1 %%writefile arreglos.rs
2
3 use std::mem;
4
5 // This function borrows a slice
6 fn analyze_slice(slice: &[i32]) {
7     println!("primer elemento del arreglo: {}", slice[0]);
8     println!("el ultimo elemento es: {}", slice[slice.len()-1]);
9     println!("el arreglo tiene {} elementos", slice.len());
10 }
11
12 fn main() {
13     // arreglo de largo 5 fijo
14     let xs: [i32; 5] = [1, 2, 3, 4, 5];
15     // arreglo de largo 500 inicializado en 0
16     let ys: [i32; 500] = [0; 500];
17
18     // indices comienzan en 0
19     println!("primer elemento arreglo: {}", xs[0]);
20     println!("tercer elemento del arreglo: {}", xs[2]);
21
22     // `len` retorna el largo del arreglo
23     println!("numero de elemenos arreglo xs: {}", xs.len());
24     println!("numero de elementos arreglo ys: {}", ys.len());
25
26     // Arrays are stack allocated
27     println!("tamaño en memoria xs {} bytes", mem::size_of_val(&xs));
28     println!("tamaño en memoria ys {} bytes", mem::size_of_val(&ys));
29     // Arrays can be automatically borrowed as slices
30     println!("pasamos el arreglo por puntero a función");
31     analyze_slice(&xs);
32     analyze_slice(&ys);
33
34     println!("pasamos una slice del array");
35     analyze_slice(&ys[1 .. 4]);
36
37     //Se puede acceder a las matrices de forma segura mediante `.get`,
38     //que devuelve un
39     // `Opción`. Esto se puede combinar como se muestra a continuación, o se puede usar con
40     // `.expect()` si desea que el programa finalice con un buen
41     // mensaje en lugar de continuar.
42     for i in 0..xs.len() + 1 { // iteramos un elemto mas del largo
43         match xs.get(i) {
44             Some(xval) => println!("{}", i, xval),
45             None => println!("Fuera de indice! {} no existe!", i),
46         }
47     }
```

<https://github.com/adigenova/uohpmd/blob/main/code/OpenMP.ipynb>

# Consultas?

Consultas o comentarios?

Muchas gracias