

Procesamiento Masivo de datos: OpenMP

Alex Di Genova

09/09/2024

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.

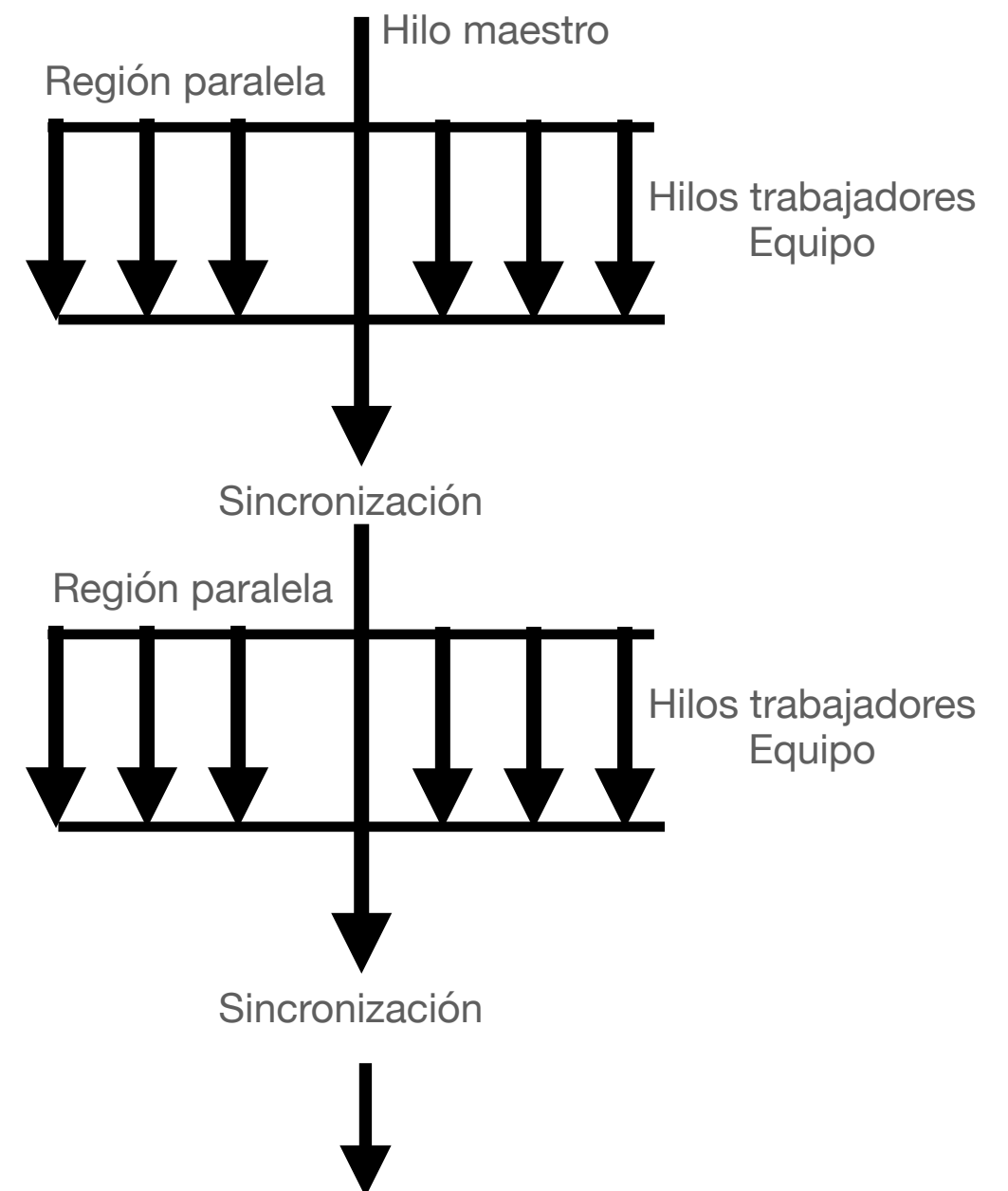
OpenMP

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)

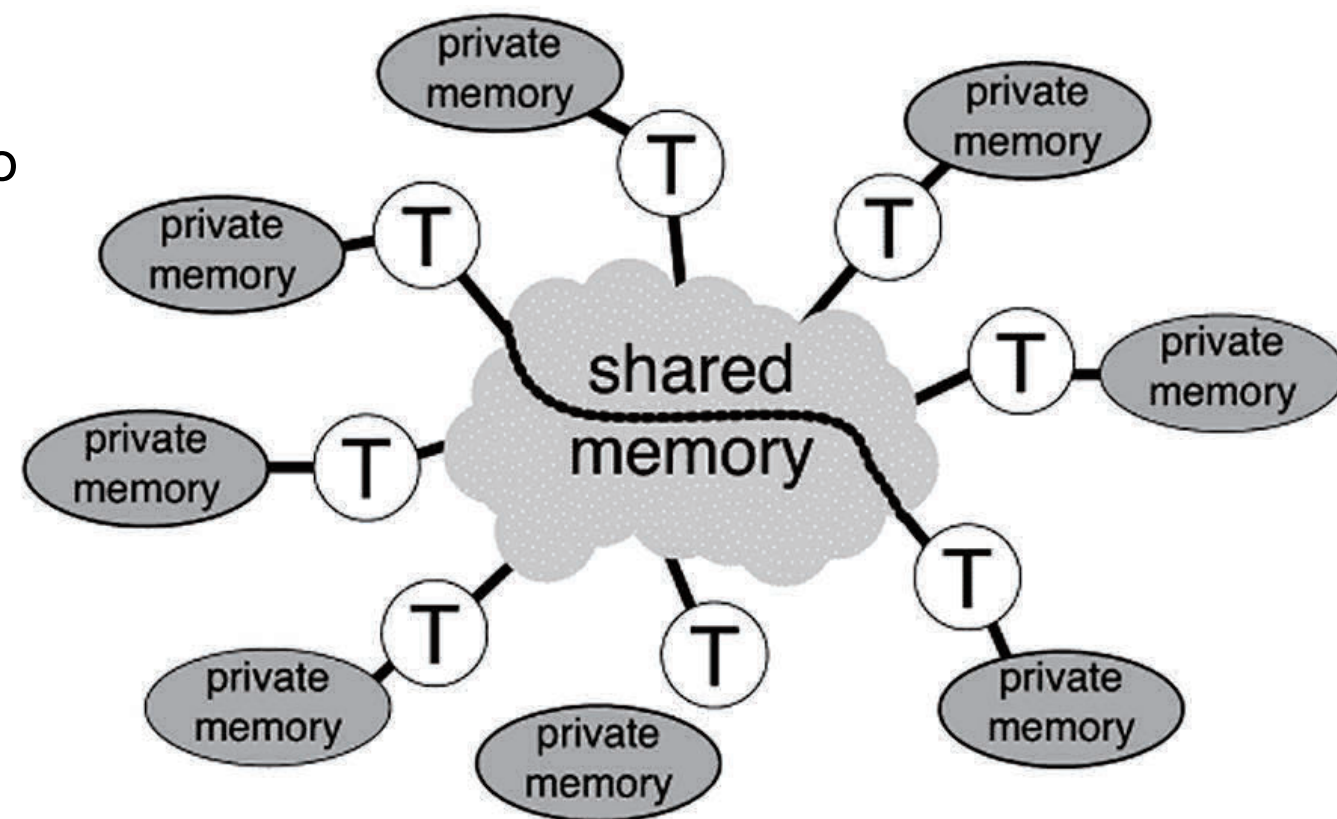


OpenMP

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.

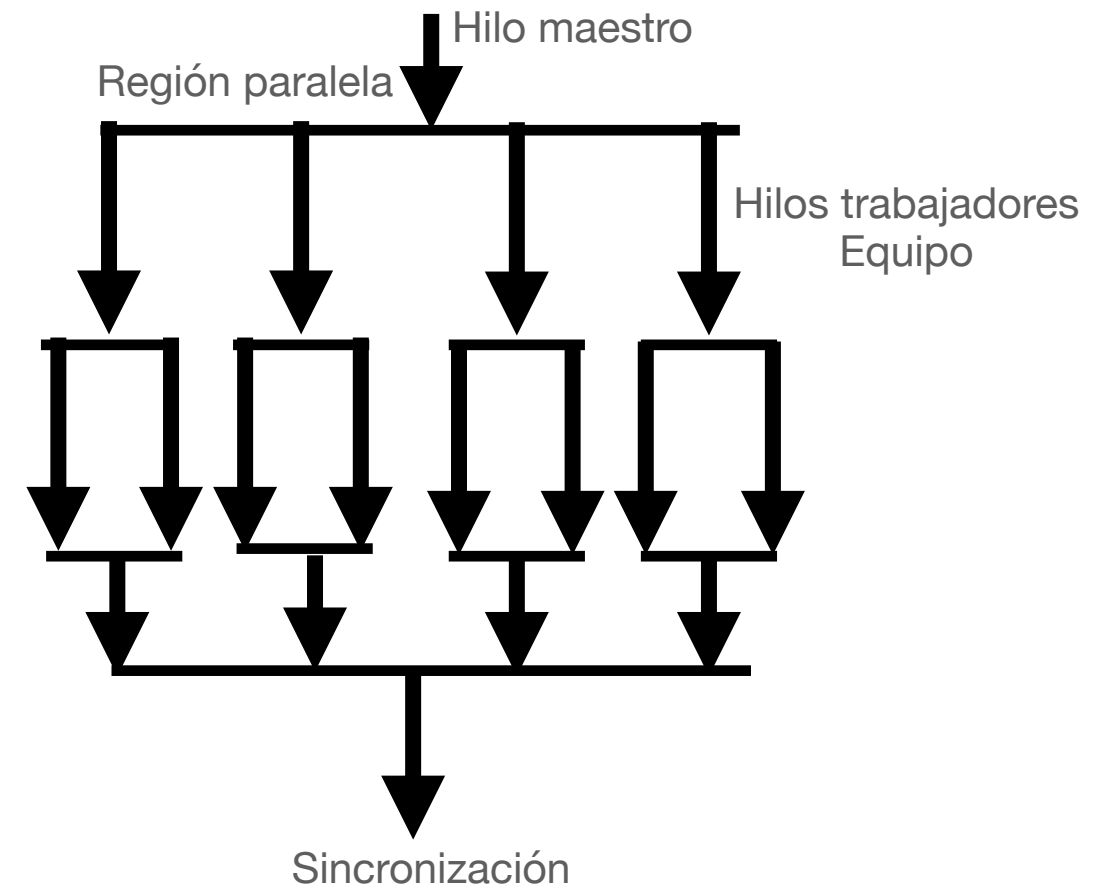
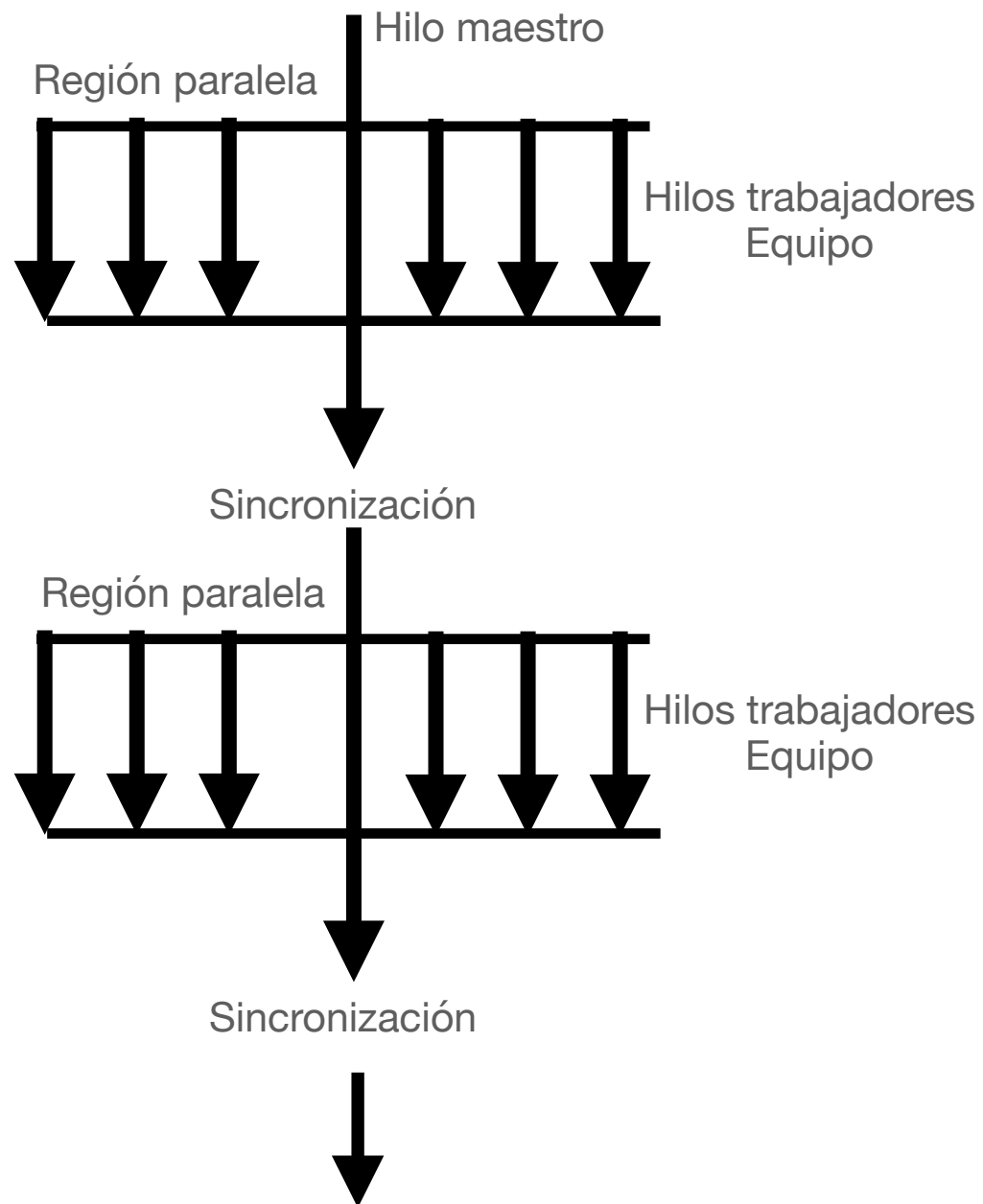
```
#pragma omp single
```

```
    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.

OpenMP

Funciones de tiempo de ejecución

- Estas funciones se pueden usar para consultar la configuración y también sobrescribir los valores iniciales, ya sea establecidos de forma predeterminada o especificados a través de variables de ambiente definidas antes del inicio del programa.
- Un ejemplo es el número de hilos utilizados para ejecutar una región paralela. El valor inicial depende de la implementación, pero a través de la variable de entorno OMP_NUM_THREADS, este número puede establecerse explícitamente antes de que se inicie el programa. Durante la ejecución del programa, la función **omp_set_num_threads()** se puede usar para aumentar o disminuir el número de hilos que se usarán en las siguientes regiones paralelas.

Función	Descripción
omp_set_num_threads	Establece el número de hilos.
omp_get_num_threads	Número de hilos en el equipo actual.
omp_get_max_threads	número de hilos en la siguiente región paralela.
omp_get_num_procs	Número de procesadores disponibles para el programa.
omp_get_thread_num	Número de hilo dentro de la región paralela.
omp_in_parallel	Comprueba si está dentro de una región paralela.
omp_get_dynamic	Comprueba si el ajuste del hilo está habilitado.
omp_set_dynamic	Habilita o deshabilita el ajuste del hilos.
omp_get_nested	Comprueba si el paralelismo anidado está habilitado.
omp_set_nested	Habilita o deshabilita el paralelismo anidado.

Ejemplos OpenMP

Hello world

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>

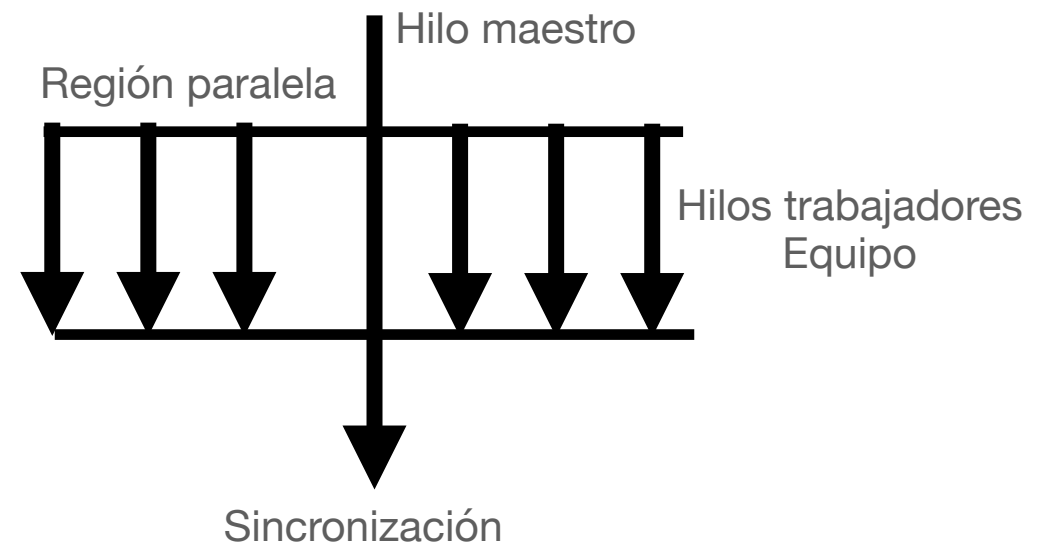
int main(int argc, char* argv[])
{
    // Comienzo de la region paralela
    #pragma omp parallel
    {
        printf("Hola Mundo... desde hilo = %d \n",
omp_get_thread_num());
    }
    // Fin de la region paralela
}

#definimos los hilos a utilizar
%env OMP_NUM_THREADS=3

!gcc -o holamundo_omp -fopenmp holamundo_omp.c

!./holamundo_omp
```

Hola Mundo... desde hilo = 1
Hola Mundo... desde hilo = 0
Hola Mundo... desde hilo = 2



Ejemplos OpenMP

For

```
#include <omp.h>
#include <stdio.h>

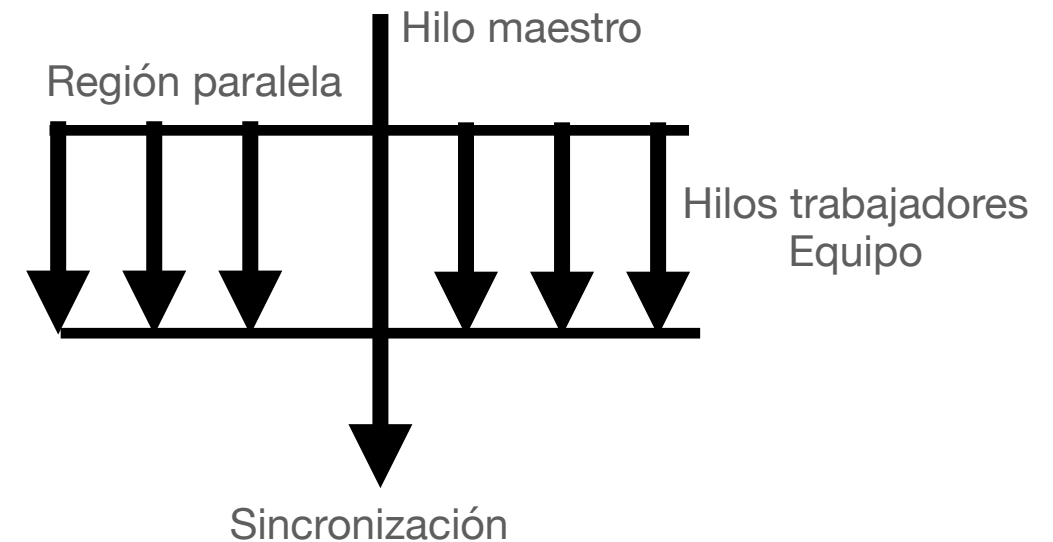
int main() {
    int k;

    #pragma omp parallel
    {
        for (k = 0; k < 10; k++)
            printf("Itr: %d tid=%d\n", k, omp_get_thread_num());
    }
    return 0;
}
```

```
!gcc -o for_openmp1 -fopenmp for_openmp1.c
```

```
%env OMP_NUM_THREADS=3
!./for_openmp1
```

```
env: OMP_NUM_THREADS=3
Itr: 0 tid=0
Itr: 1 tid=0
Itr: 2 tid=0
Itr: 3 tid=0
Itr: 4 tid=0
Itr: 0 tid=1
Itr: 1 tid=1
Itr: 2 tid=1
Itr: 3 tid=1
Itr: 4 tid=1
```



```
#include <omp.h>
#include <stdio.h>
```

```
int main() {
    int k;

    #pragma omp parallel
    {
        #pragma omp for
        for (k = 0; k < 10; k++)
            printf("Itr: %d tid=%d\n", k,
omp_get_thread_num());
    }
    return 0;
}
```

```
env: OMP_NUM_THREADS=2
Itr: 5 tid=1
Itr: 6 tid=1
Itr: 7 tid=1
Itr: 8 tid=1
Itr: 9 tid=1
Itr: 0 tid=0
Itr: 1 tid=0
Itr: 2 tid=0
Itr: 3 tid=0
Itr: 4 tid=0
```

Ejemplos OpenMP

Secciones paralela

```
#include <omp.h>
#include <stdio.h>
#include <unistd.h>

void Work1(){
    printf("executing work 1 hilo:%d\n", omp_get_thread_num());
    sleep(1);
}

void Work2(){
    printf("executing work 2 hilo:%d\n", omp_get_thread_num());
    sleep(1);
}

void Work3(){
    printf("executing work 3 hilo:%d\n", omp_get_thread_num());
}

void Work4(){
    printf("executing work 4 hilo:%d\n", omp_get_thread_num());
    sleep(1);
}

int main() {

    #pragma omp parallel sections
    {
        { Work1(); }
        #pragma omp section
        { Work2();
          Work3();}
        #pragma omp section
        { Work4(); }
    }

    return 0;
}
```

```
%env OMP_NUM_THREADS=5
!./omp_sections
```

```
env: OMP_NUM_THREADS=5
executing work 1 hilo:1
executing work 2 hilo:2
executing work 4 hilo:3
executing work 3 hilo:2
```

```
%env OMP_NUM_THREADS=10
!./omp_single
```

```
int main() {
```

```
    #pragma omp parallel
    {
        Work1();
        #pragma omp single
        { Work2();
          Work3();
        }
        Work4();
    }
}
```

```
    return 0;
}
```

```
env: OMP_NUM_THREADS=10
executing work 1 hilo:0
executing work 1 hilo:1
executing work 1 hilo:3
executing work 1 hilo:2
executing work 1 hilo:4
executing work 1 hilo:5
executing work 1 hilo:6
executing work 1 hilo:7
executing work 1 hilo:8
executing work 1 hilo:9
executing work 2 hilo:3
executing work 3 hilo:3
executing work 4 hilo:3
executing work 4 hilo:4
executing work 4 hilo:1
executing work 4 hilo:8
executing work 4 hilo:0
executing work 4 hilo:7
executing work 4 hilo:2
executing work 4 hilo:9
executing work 4 hilo:5
executing work 4 hilo:6
```

Ejemplos OpenMP

For anidados y sección crítica

```
int main() {
```

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

```
#pragma omp parallel num_threads(5)
{ //1 x 5 = 5 threads
    Work2();
}
return 0;
}
```

```
%env OMP_NUM_THREADS=3
!./omp_nested
```

executing work 1 hilo:0
executing work 2 hilo:2
executing work 2 hilo:4
executing work 2 hilo:0
executing work 2 hilo:1
executing work 2 hilo:3

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
```

```
int main() {
    int k;
    int sum=0;
```

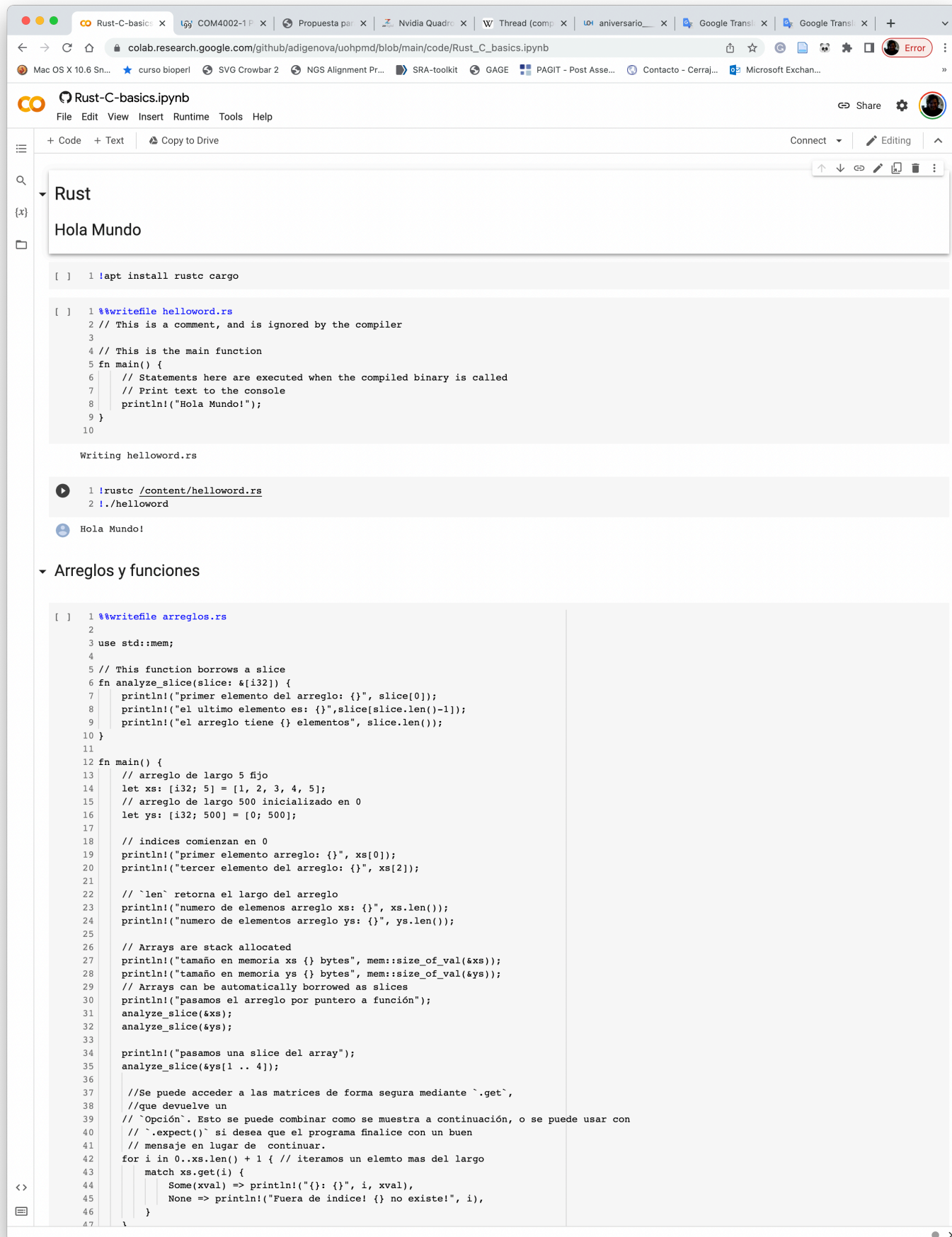
```
#pragma omp parallel for shared(sum)
for (k = 0; k < 10; k++) {
    int c=rand()%50;
    printf("Itr: %d tid=%d, my_contri=%d\n", k,
omp_get_thread_num(),c);
    #pragma omp critical
    sum+=c;
}
```

```
printf("Sum=%d",sum);
return 0;
}
```

```
%env OMP_NUM_THREADS=5
!./sync_openmp
```

```
env: OMP_NUM_THREADS=5
Itr: 2 tid=1, my_contri=33
Itr: 3 tid=1, my_contri=35
Itr: 6 tid=3, my_contri=27
Itr: 7 tid=3, my_contri=36
Itr: 8 tid=4, my_contri=15
Itr: 9 tid=4, my_contri=42
Itr: 4 tid=2, my_contri=36
Itr: 5 tid=2, my_contri=49
Itr: 0 tid=0, my_contri=43
Itr: 1 tid=0, my_contri=21
Sum=337
```

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 compile and 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