

Procesamiento Masivo de datos: OpenMP II and MPI I

Alex Di Genova

25/09/2023

OpenMP

Repaso

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**.

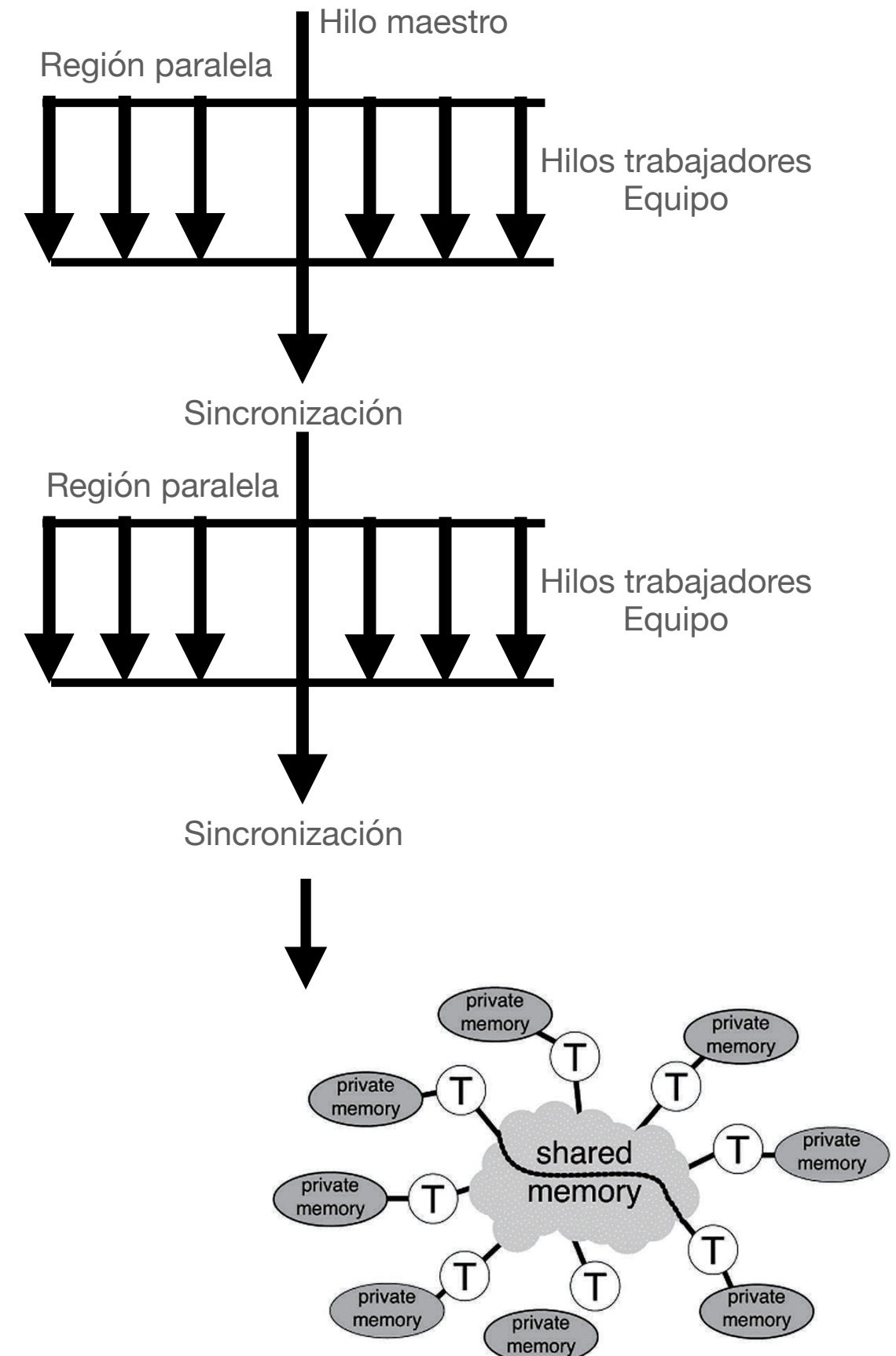
- **Syntax**

- **#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.

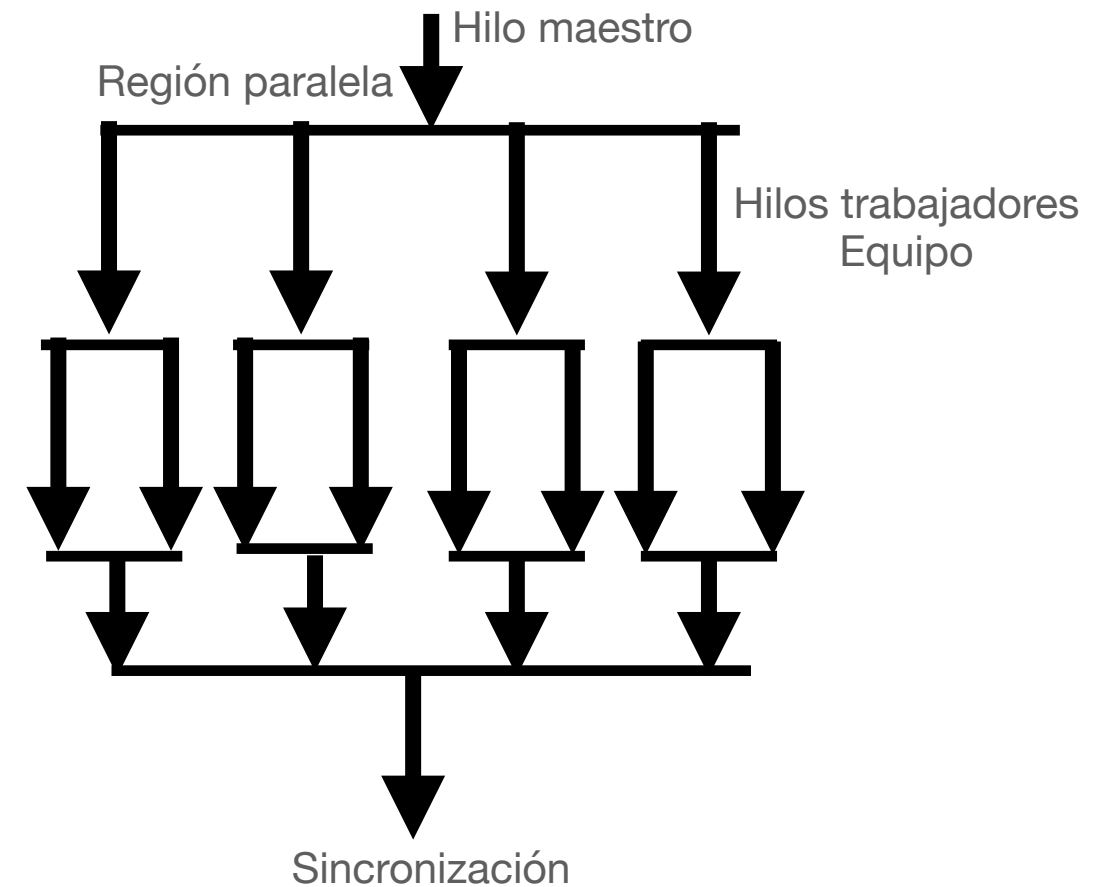
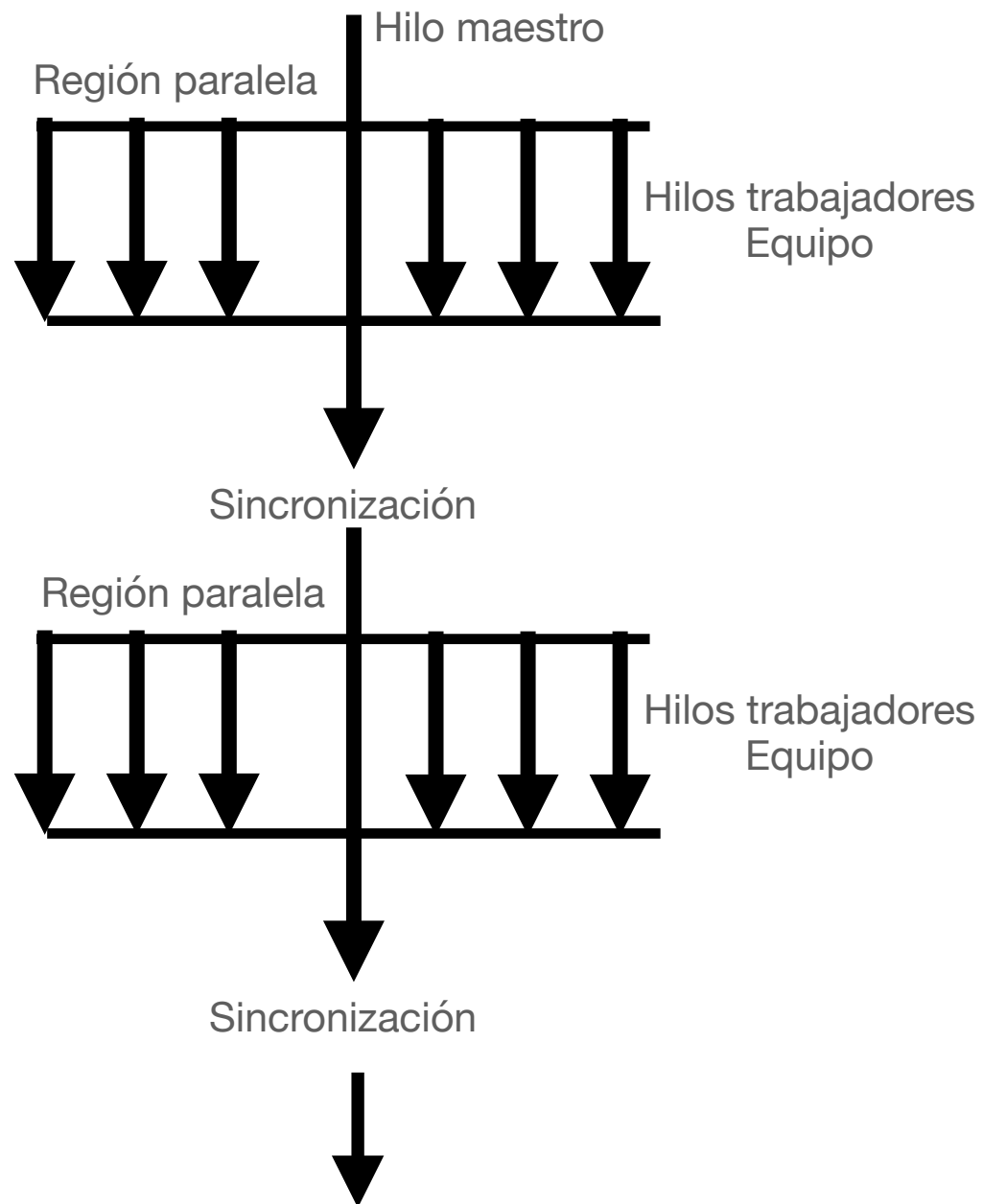
Modelo de Ejecución OpenMP (fork-join)



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

Ejemplos

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

int main() {
    int arr[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    int n = 10;
    int sum = 0;
    //
    #pragma omp parallel for reduction(+:sum)
    for (int i = 0; i < n; i++) {
        sum += arr[i];
        printf("Thread %d: arr[%d] = %d\n",
omp_get_thread_num(), i, arr[i]);
    }

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

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

#define N 10000
double data[N];

// Función para calcular el promedio de un subconjunto del
arreglo
double calcularPromedio(int start, int end) {
    double sum = 0.0;
    for (int i = start; i < end; i++) {
        sum += data[i];
    }
    return sum / (end - start);
}

int main() {
    // Inicializar el arreglo de datos
    for (int i = 0; i < N; i++) {
        data[i] = i;
    }

    int num_threads = 4; // Número de hilos a utilizar
    double total_average = 0.0;

    #pragma omp parallel num_threads(num_threads)
    {
        int thread_id = omp_get_thread_num();
        int chunk_size = N / num_threads;
        int start = thread_id * chunk_size;
        int end = (thread_id == num_threads - 1) ? N : start +
chunk_size;

        double local_average = calcularPromedio(start, end);

        #pragma omp critical
        total_average += local_average;

        printf("Thread %d: Promedio Local = %f\n", thread_id,
local_average);
    }

    total_average /= num_threads;
    printf("Promedio Total = %f\n", total_average);

    return 0;
}
```

OpenMP

Ejemplos

- Dado un texto, contar la frecuencia de los caracteres utilizando dos hilos de OpenMP.
- Considerar que el total de caracteres ascii es 256.

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

#define MAX_TEXT_SIZE 10000

// Función para contar la frecuencia de caracteres en un texto
void countCharacterFrequency(const char* text, int* charCount) {
    int chartmp[256] = {0};
    for (int i = 0; text[i] != '\0'; i++) {
        char c = text[i];
        chartmp[c]++;
    }
    //sumamos el resultado en charCount
    #pragma omp critical
    for(int j = 0; j<256; j++){
        charCount[j]+=chartmp[j];
    }
}

int main() {
    char text[MAX_TEXT_SIZE];
    int charCount[256] = {0}; // 256 caracteres ascii
    // Texto
    const char* inputText = "Los recursos en línea ofrecen una variedad
de textos en español para mejorar
la comprensión y el aprendizaje del idioma. Estos textos pueden ser
útiles para estudiantes de diferentes niveles de habilidad.";

    // Divide the text into two segments
    char* segment1 = strncpy(text, inputText, strlen(inputText) / 2);
    char* segment2 = strncpy(text + (strlen(inputText) / 2), inputText +
(strlen(inputText) / 2), strlen(inputText) / 2);

    #pragma omp parallel
    {
        #pragma omp sections
        {
            #pragma omp section
            {
                countCharacterFrequency(segment1, charCount);
            }
            #pragma omp section
            {
                countCharacterFrequency(segment2, charCount);
            }
        }
    }

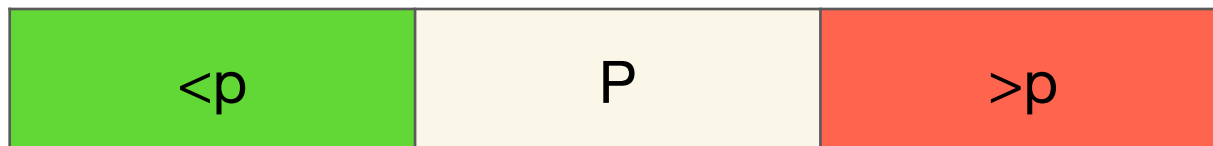
    printf("Frecuencia de caracteres:\n");
    for (int i = 0; i < 256; i++) {
        if (charCount[i] > 0) {
            printf("Caracter '%c' aparece %d veces\n", (char)i,
charCount[i]);
        }
    }

    return 0;
}
```

OpenMP

Ejemplos

- quicksort
- Ejemplo clásico de la aplicación del principio de dividir para conquistar.
- algoritmo:
 - Primero se elige un elemento al azar, que se denomina el pivote.
 - El arreglo a ordenar se reordena dejando a la izquierda a los elementos menores que el pivote, el pivote al medio, y a la derecha los elementos mayores que el pivote:



- Luego cada sub-arreglo se ordena recursivamente.
- La recursividad se detiene en principio hay 1 elemento.

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

#define ARRAY_SIZE 1000000

void quickSort(int arr[], int left, int right) {
    if (left < right) {
        int pivot = arr[right];
        int i = left - 1;

        for (int j = left; j < right; j++) {
            if (arr[j] < pivot) {
                i++;
                int temp = arr[i];
                arr[i] = arr[j];
                arr[j] = temp;
            }
        }

        int temp = arr[i + 1];
        arr[i + 1] = arr[right];
        arr[right] = temp;

        int partition = i + 1;

        #pragma omp parallel sections
        {
            #pragma omp section
            quickSort(arr, left, partition - 1);
            #pragma omp section
            quickSort(arr, partition + 1, right);
        }
    }
}

int main() {
    int arr[ARRAY_SIZE];

    // Inicializar el arreglo con números aleatorios
    for (int i = 0; i < ARRAY_SIZE; i++) {
        arr[i] = rand() % 10000;
    }

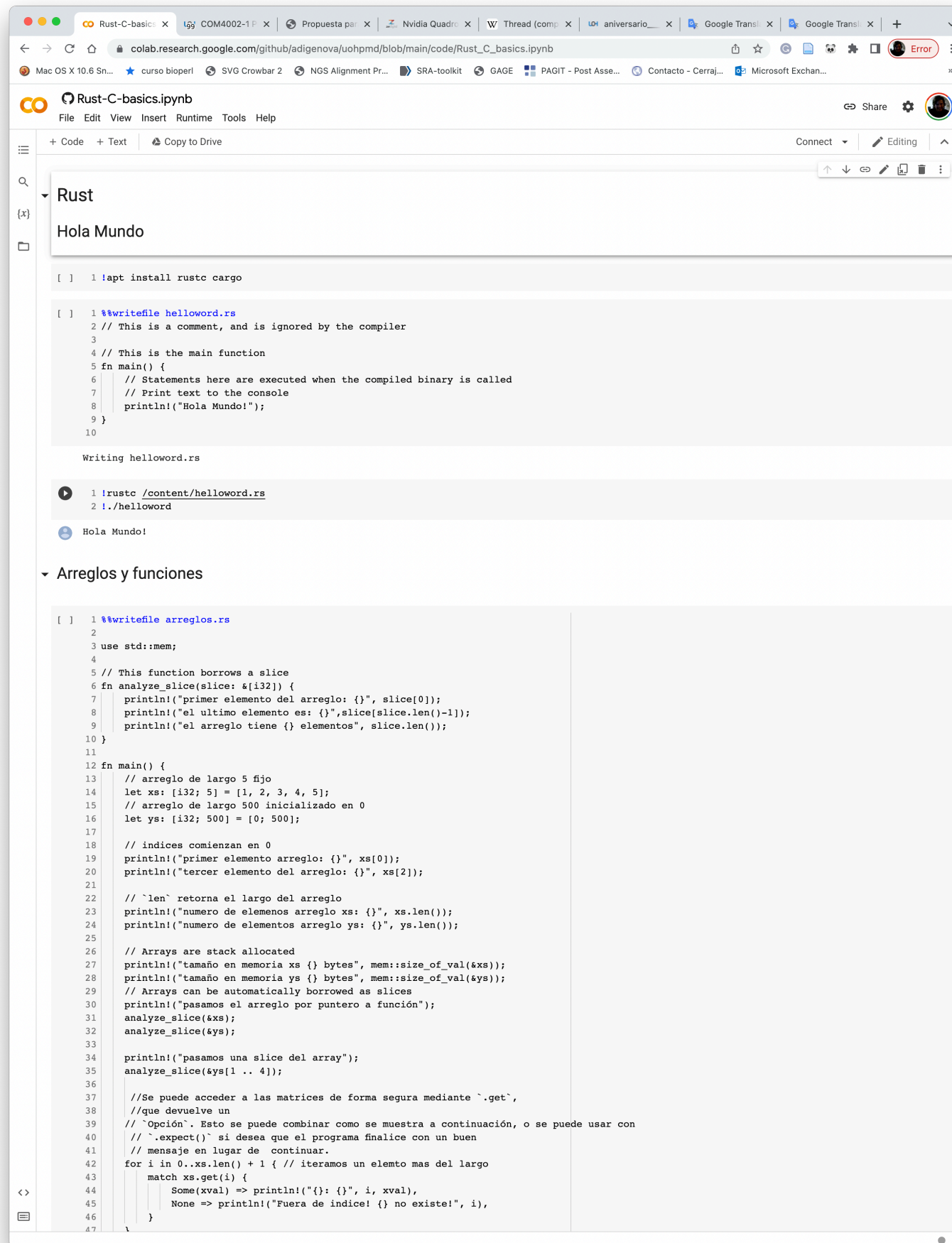
    printf("Arreglo no ordenado:\n");
    for (int i = 0; i < 100; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");

    // Ordenar el arreglo usando Quick Sort paralelizado
    #pragma omp parallel
    {
        #pragma omp single
        quickSort(arr, 0, ARRAY_SIZE - 1);
    }

    printf("Arreglo ordenado:\n");
    for (int i = 0; i < 100; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");

    return 0;
}
```

Ejemplos en Google Colab



The screenshot shows a Google Colab notebook with the following content:

```
Rust
Hola Mundo

[ ] 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

Writing helloworld.rs

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

Hola Mundo!
```

Arreglos y funciones

```
[ ] 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 elemento 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/OMP2.ipynb>

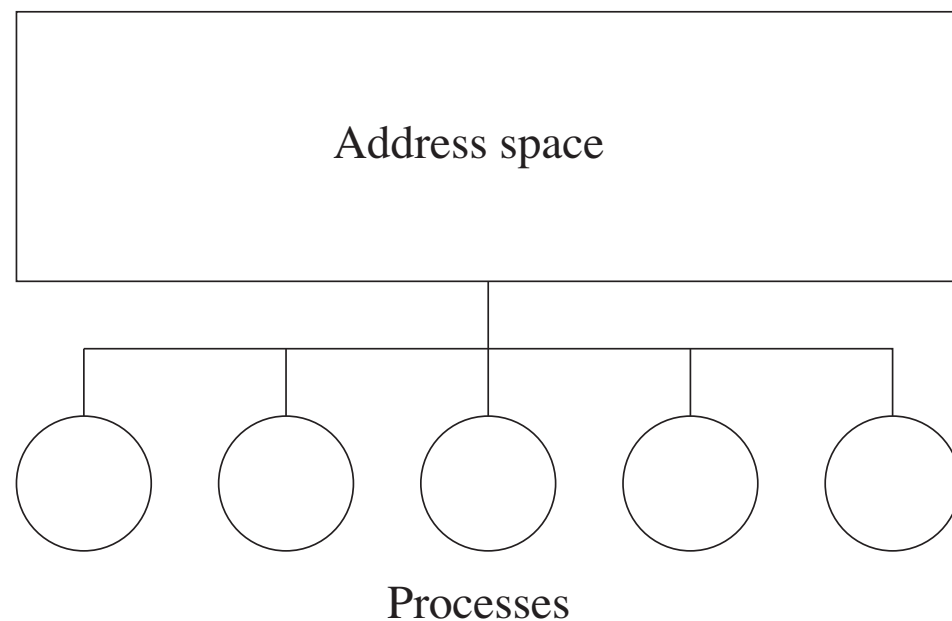
MPI

Message Passage Interface

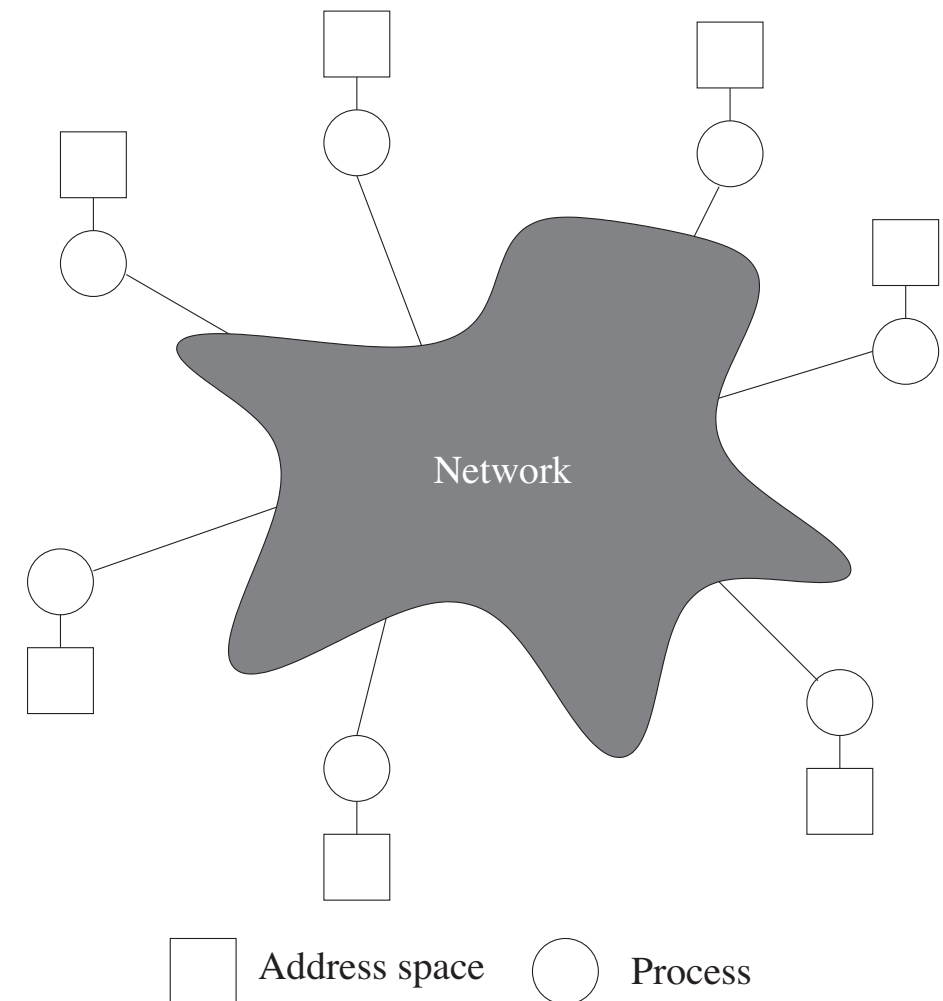
MPI

Modelo de paralelismo

Modelo de memoria compartida

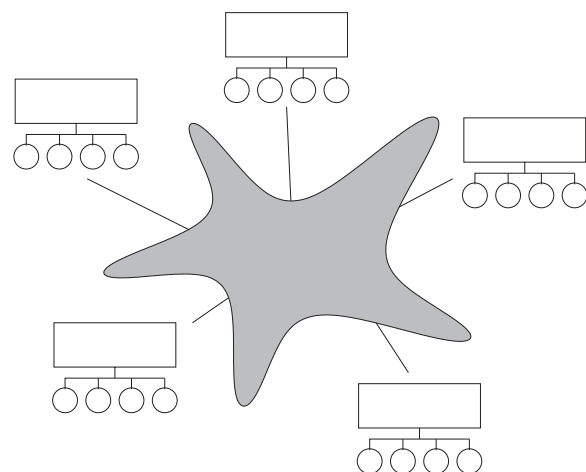


Modelo de paso de mensajes



Modelo Híbrido

PThread/ OpenMP / Fork



- El modelo de paso de mensajes postula un conjunto de procesos que solo tienen memoria local pero que pueden comunicarse con otros procesos enviando y recibiendo mensajes.
- Es una característica definitoria del modelo de paso de mensajes que la transferencia de datos desde la memoria local de un proceso a la memoria local de otro requiere que ambos procesos realicen operaciones.
- MPI es una implementación específica del modelo de paso de mensajes

MPI

Ventajas del modelo de paso de mensajes

- **Universalidad.** El modelo de paso de mensajes encaja bien en procesadores separados conectados por una red de comunicación (rápida o lenta). Por lo tanto, coincide con el nivel más alto del hardware de la mayoría de las supercomputadoras paralelas actuales.
- **Expresividad.** Se ha encontrado que el paso de mensajes es un modelo útil y completo para expresar algoritmos paralelos.
- **Facilidad de depuración.** La depuración de programas paralelos sigue siendo un área de investigación desafiante. La razón es que una de las causas más comunes de error es la sobrescritura inesperada de la memoria. El modelo de paso de mensajes, al controlar las referencias a la memoria de manera más explícita que cualquiera de los otros modelos, facilita la localización de lecturas y escrituras de memoria erróneas.
- **Rendimiento.** La razón más convincente por la que el paso de mensajes seguirá siendo una parte permanente del entorno informático paralelo es el rendimiento. A medida que las CPU modernas se han vuelto más rápidas, la gestión de sus cachés y la jerarquía de la memoria en general se ha convertido en la clave para aprovechar al máximo estas máquinas.

MPI

Que es?

- **MPI es una libreria, no un lenguaje.**
 - Especifica los nombres, las secuencias de llamada y los resultados de las funciones que se llamarán desde los programas C.
 - Los programas que los usuarios escriben en C se compilan con compiladores ordinarios y se vinculan con la libreria MPI.
- **MPI es una especificación, no una implementación particular.**
 - Un programa MPI correcto debería poder ejecutarse en todas las implementaciones de MP sin cambios.
- **MPI aborda el modelo de paso de mensajes.**
 - Enfoque en el movimiento de datos entre espacios de direcciones separados.

MPI

Conceptos basicos

- La comunicación se produce cuando una parte del espacio de direcciones de un proceso se copia en el espacio de direcciones de otro proceso.
- Esta operación es cooperativa y ocurre solo cuando el primer proceso ejecuta una operación de **envío (send)** y el segundo proceso ejecuta una operación de recepción (**receive**).
 - **MPI_Send**(address, count, datatype, destination, tag, comm)
 - **MPI_Recv**(address, maxcount, datatype, source, tag, comm, status)
 - **tag** es un número entero que se utiliza para la coincidencia de mensajes.
 - **comm** identifica un grupo de procesos y un contexto de comunicación.
 - **destination/source** es el ranking del destino/fuente en el grupo asociado con el comunicador **comm**.

MPI

comunicaciones colectivas

- Operaciones realizada por todos los procesos en un cómputo.
- Las operaciones colectivas son de dos tipos:
 - Las operaciones de **movimiento de datos** se utilizan para reorganizar los datos entre los procesos. La más simple de ellas es **broadcast**, pero se pueden definir muchas operaciones elaboradas de dispersión (**scattering**) y recopilación (**gathering**).
 - Operaciones de **cálculo colectivo** (mínimo, máximo, suma, OR lógico, etc., así como operaciones definidas por el usuario).

MPI

Funciones basicas

Función	Descripción
MPI_Init	Inicializar MPI
MPI_Comm_size	Determina cuántos procesos hay
MPI_Comm_rank	qué proceso soy
MPI_Send	Envio un mensaje
MPI_Recv	Recibo un mensaje
MPI_Finalize	Termina MPI

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

```
int main(int argc, char** argv) {
    //Iniciamos el ambiente MPI
    MPI_Init(NULL, NULL);

    // obtenemos el numero de procesadores
    int world_size;
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);
```

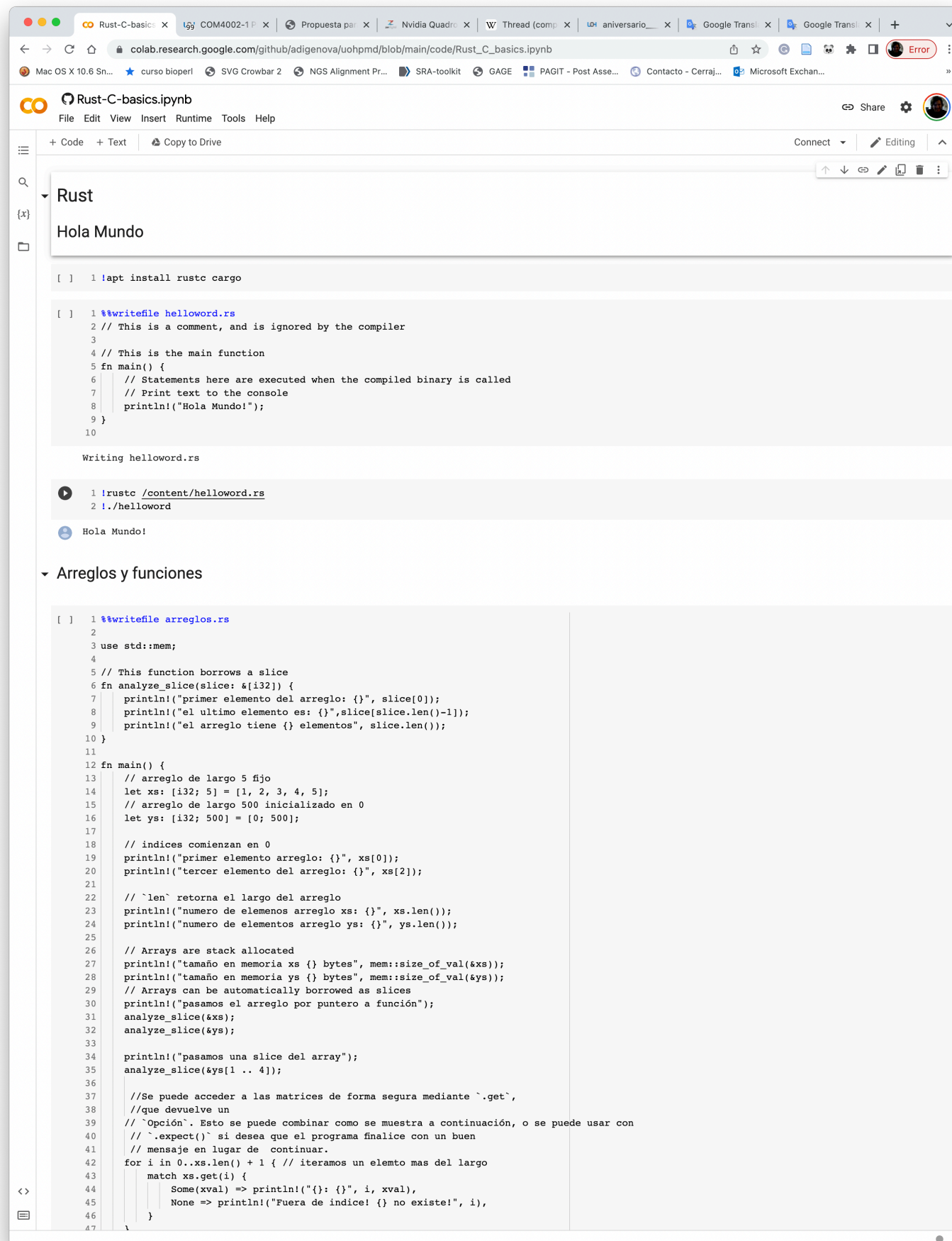
```
    // obtenemos el ranking para cada proceso
    int world_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
```

```
    // obtenemos el nombre del procesador
    char processor_name[MPI_MAX_PROCESSOR_NAME];
    int name_len;
    MPI_Get_processor_name(processor_name, &name_len);
    if (world_rank != 3){
        printf("Hola mundo desde procesador %s, ranking %d de %d procesadores\n",
            processor_name, world_rank, world_size);
    }else{
        printf("castigado\n");
    }
    // terminamos el ambiente MPI
    MPI_Finalize();
}
```

```
! mpicc -o mpi_holamundo mpi_holamundo.c
! mpirun --allow-run-as-root -np 4 ./mpi_holamundo
```

Hola mundo desde procesador d4092f21c366, ranking 1 de 4 procesadores
Hola mundo desde procesador d4092f21c366, ranking 2 de 4 procesadores
castigado
Hola mundo desde procesador d4092f21c366, ranking 0 de 4 procesadores

Google Colab



The screenshot shows a Google Colab notebook interface. The browser tabs at the top include 'Rust-C-basics', 'COM4002-1 F...', 'Propuesta pa...', 'Nvidia Quad...', 'W Thread (comp...', 'aniversario...', 'Google Transl...', and another 'Google Transl...'. The address bar shows 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 notebook has a menu bar with 'File', 'Edit', 'View', 'Insert', 'Runtime', 'Tools', and 'Help'. Below the menu bar are tabs for '+ Code', '+ Text', and 'Copy to Drive'. On the right side, there are buttons for 'Connect', 'Editing', and a small error icon.

The notebook content is organized into sections. The first section is titled 'Rust' and contains the following 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, there is a status bar that says 'Writing helloworld.rs'. Then, there is a code cell with the following commands:

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

The output of the second cell is 'Hola Mundo!'.

The next section is titled 'Arreglos y funciones' and contains the following 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 elemento 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/MPI_I.ipynb

Consultas?

Consultas o comentarios?

Muchas gracias