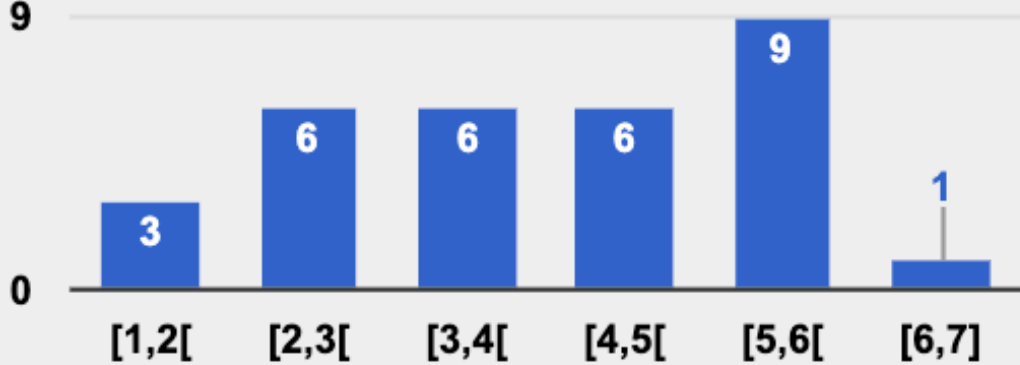


Procesamiento Masivo de datos: Modelo Híbrido MPI y OpenMP/Pthread

Alex Di Genova

17/11/2025

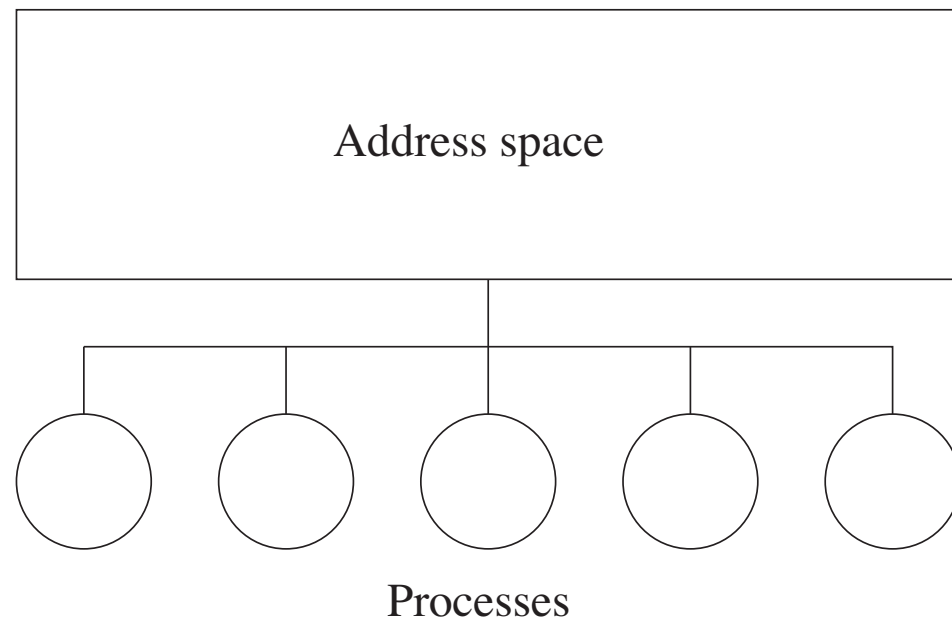
Notas Control 1

Promedio	4.0														
Mediana	4.0														
Mínimo	1.4														
Máximo	6.7														
Desviación Estándar	1.4														
Histograma	 <table><thead><tr><th>Intervalo</th><th>Frecuencia</th></tr></thead><tbody><tr><td>[1,2[</td><td>3</td></tr><tr><td>[2,3[</td><td>6</td></tr><tr><td>[3,4[</td><td>6</td></tr><tr><td>[4,5[</td><td>6</td></tr><tr><td>[5,6[</td><td>9</td></tr><tr><td>[6,7]</td><td>1</td></tr></tbody></table>	Intervalo	Frecuencia	[1,2[3	[2,3[6	[3,4[6	[4,5[6	[5,6[9	[6,7]	1
Intervalo	Frecuencia														
[1,2[3														
[2,3[6														
[3,4[6														
[4,5[6														
[5,6[9														
[6,7]	1														

Modelo Híbrido

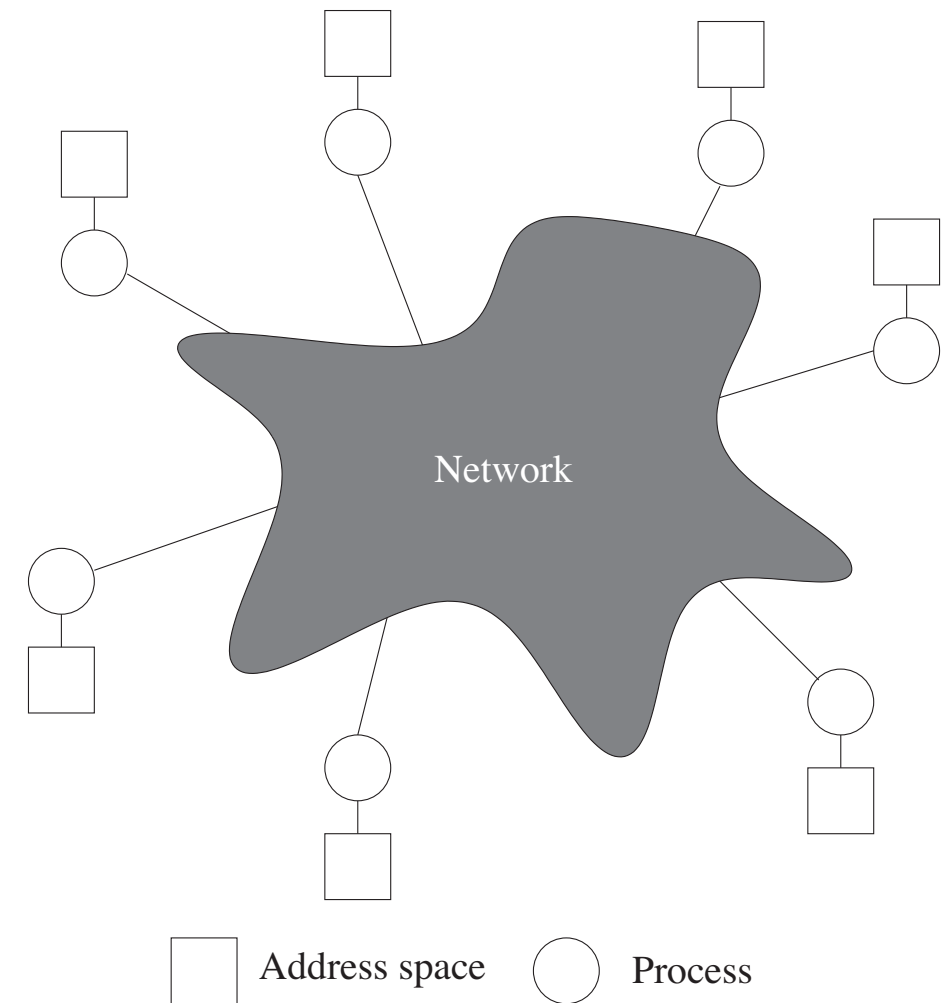
Modelo de paralelismo

Modelo de memoria compartida

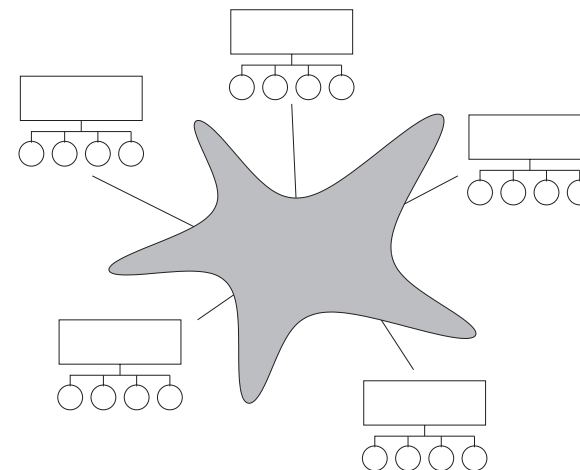


PThread/ OpenMP / Fork

Modelo de paso de mensajes



Modelo Híbrido



Procesos y hilos

MPI y OpenMP/Pthreads

- MPI = Process, OpenMP = Thread
- El programa comienza con un solo proceso.
- Los procesos tienen su propio espacio de memoria (privado)
- Un proceso puede crear uno o más hilos
- Los hilos creados por un proceso comparten su espacio de memoria
 - Leer y escribir en las mismas direcciones de memoria
 - Comparten los mismos identificadores de proceso y descriptores de archivo
- Cada hilo tiene un contador de instrucciones único y un puntero de pila
- Un hilo puede tener almacenamiento privado en la pila

Modelo híbrido

MPI & OpenMP/Phtreads

- MPI = Process, OpenMP = Thread
- Paralelización de dos niveles
 - Ideal para el hardware de un clúster.
 - MPI entre nodos
 - OpenMP dentro de los nodos de memoria compartida

Modelo Híbrido

Ventajas

- **Escalabilidad:** MPI permite escalar una aplicación en múltiples servidores. OpenMP y Pthreads se utilizan para paralelizar el cómputo local en un servidor (nodo). Escalabilidad tanto a nivel de nodos como a nivel de núcleos de CPU.
- **Mejor Rendimiento/Flexibilidad:** MPI se centra en la comunicación entre procesos, mientras que OpenMP y Pthreads se centran en el paralelismo a nivel de hilos.
 - Control fino sobre cómo se divide y gestiona el trabajo.
- **Mejor uso de Recursos:** Al usar hilos en el nivel local de cada nodo, se pueden aprovechar al máximo los recursos de la CPU, especialmente en sistemas multiprocesador o con múltiples núcleos.
- **Reducción de la Latencia:** Al paralelizar tareas a nivel de nodo mediante hilos, se puede reducir la latencia en las comunicaciones entre los procesos MPI.
- **Facilita la Programación Concurrente:** MPI y OpenMP/Pthreads se centran en aspectos diferentes de la programación paralela, lo que facilita la programación concurrente y puede reducir la complejidad de la implementación.

Modelo híbrido

MPI & OpenMP

- En la programación híbrida, cada proceso puede tener varios hilos ejecutando simultáneamente
 - Todos los hilos dentro de un proceso comparten todos los objetos MPI (Comunicadores, mensajes, etc).
- MPI define 4 niveles de seguridad de hilos
 1. MPI_THREAD_SINGLE (solo 1 hilo)
 2. MPI_THREAD_FUNNELED (>1, pero solo el hilo maestro puede utilizar funciones MPI)
 3. MPI_THREAD_SERIALIZED (> 1, pero solo un hilo puede realizar llamadas MPI a la vez)
 4. MPI_THREAD_MULTIPLE (> 1, cualquier hilo puede realizar llamadas MPI en cualquier momento)
- MPI_Init_thread (no MPI_Init) si más de un hilo es necesario.
 - MPI_Init_thread(int required, int *provided)

Ejemplos

MPI/OpenMP

```
#include <stdio.h>
#include <stdlib.h>
//we load omp/mpi
#include <omp.h>
#include <mpi.h>
// defines the MPI_THREADS_MODE
#define MPI_THREAD_STRING(level) \
    ( level==MPI_THREAD_SERIALIZED ? "THREAD_SERIALIZED" : \
      ( level==MPI_THREAD_MULTIPLE ? "THREAD_MULTIPLE" : \
        ( level==MPI_THREAD_FUNNELED ? "THREAD_FUNNELED" : \
          \
            ( level==MPI_THREAD_SINGLE ? \
              "THREAD_SINGLE" : "THIS_IS_IMPOSSIBLE" ) ) ) ) )

int main(int argc, char ** argv)
{
    /* Estos son los soportes de hilos deseados y disponibles.
       Se puede utilizar un código híbrido en el que todas las llamadas
       MPI se realizan desde el hilo principal (FUNNELED).
       Si los hilos realizan llamadas MPI, MULTIPLE es el apropiado. */
    int requested=MPI_THREAD_FUNNELED, provided;

    /* Intentamos activar los hilos MPI usando el modo requerido:
       MPI_THREAD_FUNNELED*/
    MPI_Init_thread(&argc, &argv, requested, &provided);
    if (provided<requested)
    {
        printf("MPI_Init_thread provee %s cuando %s fue solicitado.
Terminando el programa. \n",
               MPI_THREAD_STRING(provided), MPI_THREAD_STRING(requested) );
        exit(1);
    }

    int world_size, world_rank;

    MPI_Comm_size(MPI_COMM_WORLD,&world_size);
    MPI_Comm_rank(MPI_COMM_WORLD,&world_rank);

    printf("Hola desde %d de total :%d  procesos\n", world_rank,
world_size);

    //ocupamos openMP para crear una seccion paralela
    #pragma omp parallel
    {
        int omp_id  =omp_get_thread_num();
        int omp_num =omp_get_num_threads();
        printf("MPI rank # %2d OpenMP thread # %2d of %2d \n", world_rank,
omp_id, omp_num);
        fflush(stdout);
    }

    MPI_Finalize();
    return 0;
}
```

```
! mpicc -o mpi_openmp_e1 mpi_openmp_e1.c -fopenmp
```

```
%env OMP_NUM_THREADS=3
! mpirun --oversubscribe --allow-run-as-root -np 4
./mpi_openmp_e1
```

```
env: OMP_NUM_THREADS=3
Hola desde 0 de total :4  procesos
Hola desde 1 de total :4  procesos
Hola desde 3 de total :4  procesos
Hola desde 2 de total :4  procesos
MPI rank # 1 OpenMP thread # 1 of 3
MPI rank # 1 OpenMP thread # 2 of 3
MPI rank # 2 OpenMP thread # 1 of 3
MPI rank # 1 OpenMP thread # 0 of 3
MPI rank # 2 OpenMP thread # 2 of 3
MPI rank # 2 OpenMP thread # 0 of 3
MPI rank # 0 OpenMP thread # 0 of 3
MPI rank # 0 OpenMP thread # 1 of 3
MPI rank # 0 OpenMP thread # 2 of 3
MPI rank # 3 OpenMP thread # 0 of 3
MPI rank # 3 OpenMP thread # 1 of 3
MPI rank # 3 OpenMP thread # 2 of 3
```


Ejemplos

MPI/OpenMP

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

#define VECTOR_SIZE 1000

int main(int argc, char* argv[]) {
    int rank, size;
    int local_size, local_start, local_end;
    double* vectorA, * vectorB;
    double local_sum = 0.0, global_sum = 0.0;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    // Calcular el tamaño local de los vectores
    local_size = VECTOR_SIZE / size;
    local_start = rank * local_size;
    local_end = local_start + local_size;

    // Asignar memoria para los vectores locales
    vectorA = (double*)malloc(local_size * sizeof(double));
    vectorB = (double*)malloc(local_size * sizeof(double));

    // Inicializar vectores locales
    #pragma omp parallel for
    for (int i = local_start; i < local_end; i++) {
        vectorA[i - local_start] = 1.0; // Inicializar vectorA con 1.0
        vectorB[i - local_start] = 2.0; // Inicializar vectorB con 2.0
    }

    // Calcular el producto escalar local
    #pragma omp parallel for reduction(+:local_sum)
    for (int i = 0; i < local_size; i++) {
        local_sum += vectorA[i] * vectorB[i];
    }

    printf("El producto escalar local es: %lf %d\n", local_sum, rank);
    // Reducir los resultados locales en un resultado global
    MPI_Reduce(&local_sum, &global_sum, 1, MPI_DOUBLE, MPI_SUM, 0,
MPI_COMM_WORLD);

    // El proceso 0 imprime el resultado
    if (rank == 0) {
        printf("El producto escalar global es: %lf\n", global_sum);
    }

    // Liberar memoria y finalizar MPI
    free(vectorA);
    free(vectorB);
    MPI_Finalize();

    return 0;
}
```

```
! mpicc -o ppunto ppunto.c -fopenmp
```

```
%env OMP_NUM_THREADS=4
! mpirun --oversubscribe
--allow-run-as-root -np 4
./ppunto
```

```
env: OMP_NUM_THREADS=4
```

```
El producto escalar local es: 500.000000
```

```
El producto escalar local es: 500.000000
```

```
El producto escalar local es: 500.000000
```

```
El producto escalar local es: 500.000000
```

```
El producto escalar global es: 2000.000000
```

```

#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
#include <pthread.h>

#define ARRAY_SIZE 1000000

// Estructura que contiene los datos que se pasan a cada
hilo
struct ThreadData {
    int* array;
    int local_size;
    int local_sum;
};

// Función que se ejecuta en cada hilo para calcular la
suma local
void* calculateSum(void* arg) {
    struct ThreadData* data = (struct ThreadData*)arg;
    for (int i = 0; i < data->local_size; i++) {
        data->local_sum += data->array[i];
    }
    return NULL;
}

```

```
! mpicc -o ppunto_pt ppunto_pt.c -lpthread
```

```
! mpirun --oversubscribe
--allow-run-as-root -np 4 ./ppunto_pt
```

```

int main(int argc, char* argv[]) {
    int rank, size;
    int* data;
    int local_size, local_start, local_end;
    int local_sum = 0;
    pthread_t* threads;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    // Calcular el tamaño local del array
    local_size = ARRAY_SIZE / size;
    local_start = rank * local_size;
    local_end = local_start + local_size;

    // Asignar memoria para el array local
    data = (int*)malloc(local_size * sizeof(int));
    threads = (pthread_t*)malloc(size * sizeof(pthread_t));

    // Inicializar el array local con valores aleatorios
    for (int i = 0; i < local_size; i++) {
        data[i] = rand() % 10; // Valores aleatorios entre 0 y 9
    }

    // Crear hilos para calcular la suma local en paralelo
    struct ThreadData threadData;
    threadData.array = data;
    threadData.local_size = local_size;
    threadData.local_sum = 0;

    pthread_create(&threads[rank], NULL, calculateSum, &threadData);

    // Esperar a que todos los hilos terminen
    pthread_join(threads[rank], NULL);

    // Realizar una operación de reducción para obtener la suma global
    MPI_Reduce(&threadData.local_sum, &local_sum, 1, MPI_INT, MPI_SUM,
0, MPI_COMM_WORLD);

    // El proceso 0 imprime la suma global
    if (rank == 0) {
        printf("Suma global: %d\n", local_sum);
    }

    // Liberar memoria y finalizar MPI
    free(data);
    free(threads);
    MPI_Finalize();

    return 0;
}

```

MPI/OpenMP time

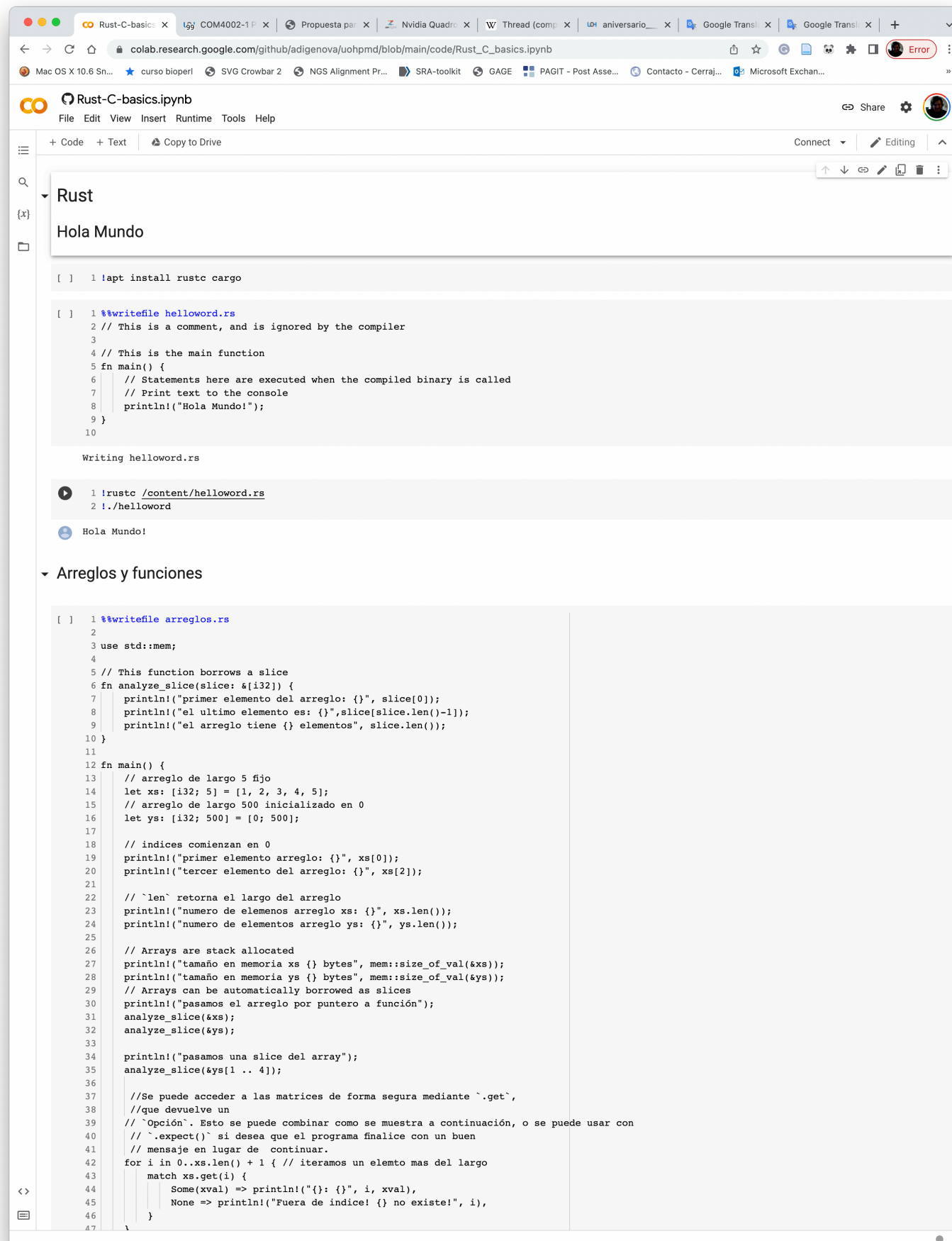
- OpenMP time:

```
double t1, t2;  
t1=omp_get_wtime();  
//do something expensive...  
t2=omp_get_wtime();  
printf("Total Runtime = %g\n", t2-t1);
```

- MPI time:

```
double t1 = MPI_Wtime();  
//do something expensive...  
double t2 = MPI_Wtime();  
if(my_rank == final_rank) {  
printf("Total runtime = %g s\n", (t2-t1));  
}
```

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' with a share icon and a settings icon. The menu bar includes 'File', 'Edit', 'View', 'Insert', 'Runtime', 'Tools', and 'Help'. The toolbar has '+ Code', '+ Text', 'Copy to Drive', 'Connect', 'Editing', and a scroll bar. The left sidebar shows a file explorer with 'Rust' and 'Hola Mundo'. The main code area contains 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

Writing helloworld.rs

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

Hola Mundo!
```

Below the code, there is a section titled 'Arreglos y funciones' (Arrays and functions) with more 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 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_OPenMP.ipynb

Consultas?

Consultas o comentarios?

Muchas gracias