

Procesamiento Masivo de datos: MPI II

Alex Di Genova

30/09/2024

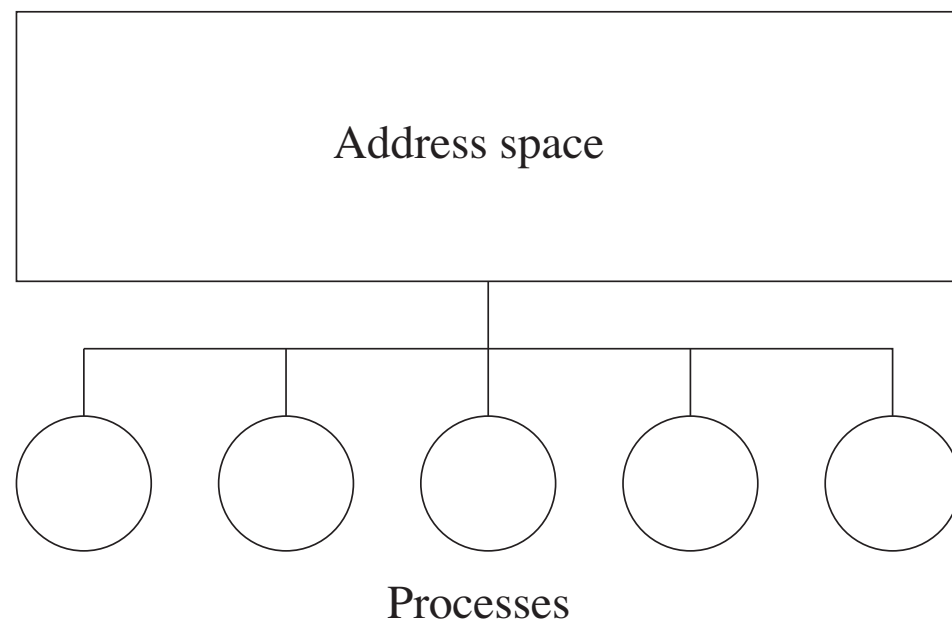
MPI

Message Passage Interface

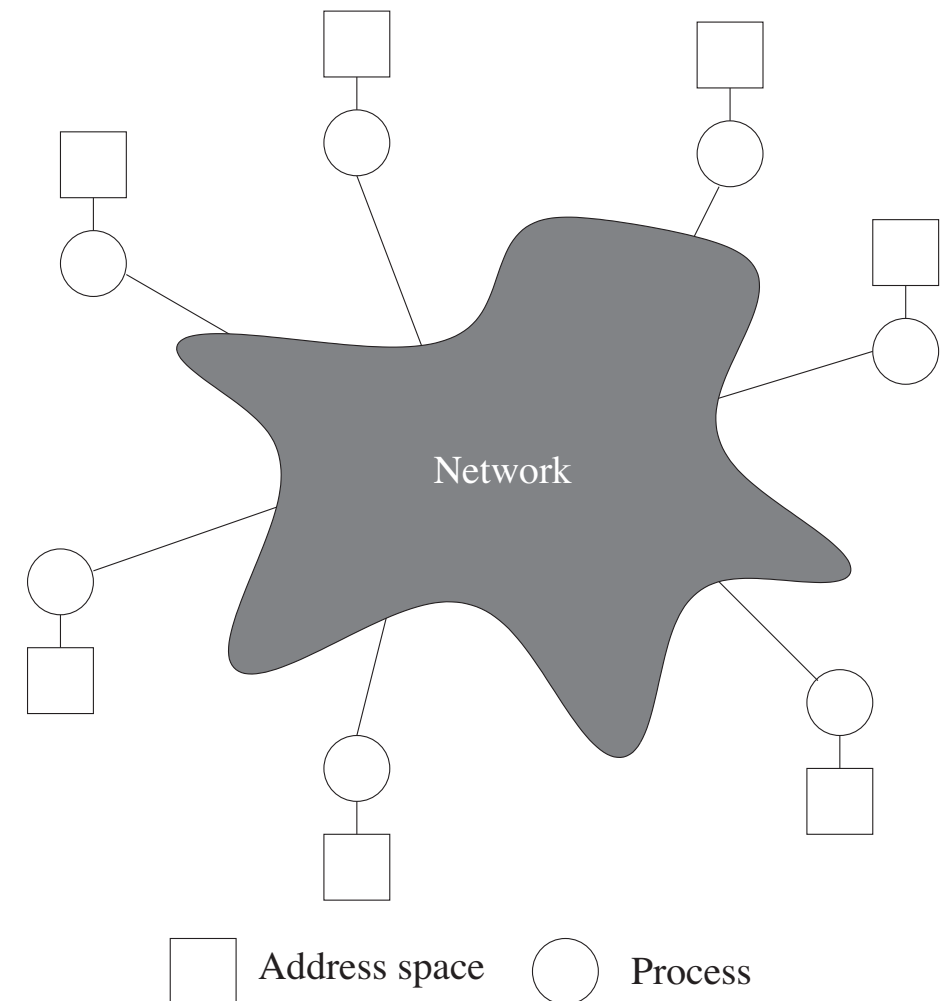
MPI

Modelo de paralelismo

Modelo de memoria compartida

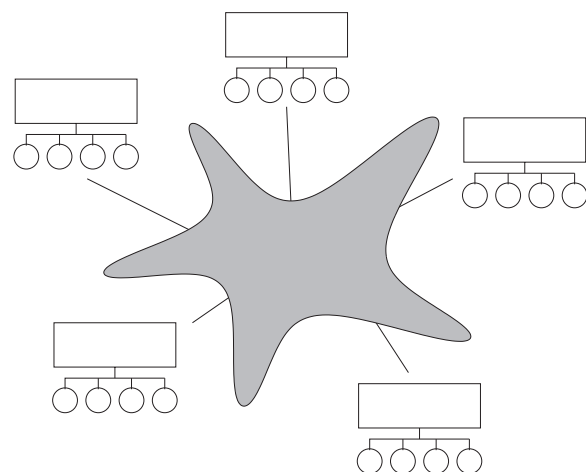


Modelo de paso de mensajes



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

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

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

MPI

Funciones colectivas basicas

- MPI broadcast (Envia un mensaje desde el proceso con ranking “root” a todos los demás procesos del comunicador.)

```
int MPI Bcast(void *buf, //variable a comunicar
              int count, // numero de elementos
              MPI Datatype datatype, // tipo de dato
              int root, //quien lo envia
              MPI Comm comm) // mundo comunicación
```

- MPI reduce (Reduce valores en todos los procesos a un único valor.)

```
int MPI Reduce(const void *sendbuf, //variable de local
               void *recvbuf, // resultado
               int count, // numero de elementos
               MPI Datatype datatype, // tipo de dato
               MPI Op op, // operacion
               int root, // proceso que colecta el resultado
               MPI Comm comm) // mundo de comunicación
```

MPI

Funciones colectivas basicas

- MPI Gather (Colecta valores de un grupo de processos)

```
int MPI_Gather(const void *sendbuf, //direccion de lo que quiero enviar
               int sendcount, //cuantos items
               MPI_Datatype sendtype, //tipo de dato
               void *recvbuf, //direccion de donde lo almaceno
               int recvcount, //cuanto voy a almacenar
               MPI_Datatype recvtype, // que tipo
               int root, //que proceso lo recibe
               MPI_Comm comm) // via de comunicaci3n
```

- MPI Scatter (Envía datos desde un proceso a todos los demás procesos en un comunicador)

```
int MPI_Scatter(const void *sendbuf, //direccion de lo que quiero enviar
                int sendcount, //cuantos items
                MPI_Datatype sendtype, //tipo de dato
                void *recvbuf, //direccion de donde lo almaceno
                int recvcount, //cuanto voy a almacenar
                MPI_Datatype recvtype, / que tipo
                int root, //que proceso lo envia
                MPI_Comm comm) // via de comunicaci3n
```

MPI

Funciones colectivas

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

int main(int argc, char** argv) {
    MPI_Init(&argc, &argv);

    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    int N = 1000; // Tamaño del arreglo
    int local_N = N / size; // Tamaño del subarreglo para cada
proceso
    int local_sum = 0;
    int global_sum = 0;

    int* data = NULL;
    int* local_data = (int*)malloc(local_N * sizeof(int));

    // Inicializar el arreglo de datos en el proceso 0
    if (rank == 0) {
        data = (int*)malloc(N * sizeof(int));
        for (int i = 0; i < N; i++) {
            data[i] = i;
        }
    }

    // Distribuir los datos entre los procesos
    MPI_Scatter(data, local_N, MPI_INT, local_data, local_N,
MPI_INT, 0, MPI_COMM_WORLD);

    // Calcular la suma local
    for (int i = 0; i < local_N; i++) {
        local_sum += local_data[i];
    }
    printf("La suma local es: %d proceso=%d\n", local_sum, rank);

    // Sumar las sumas locales para obtener la suma global
    MPI_Reduce(&local_sum, &global_sum, 1, MPI_INT, MPI_SUM, 0,
MPI_COMM_WORLD);

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

    MPI_Finalize();

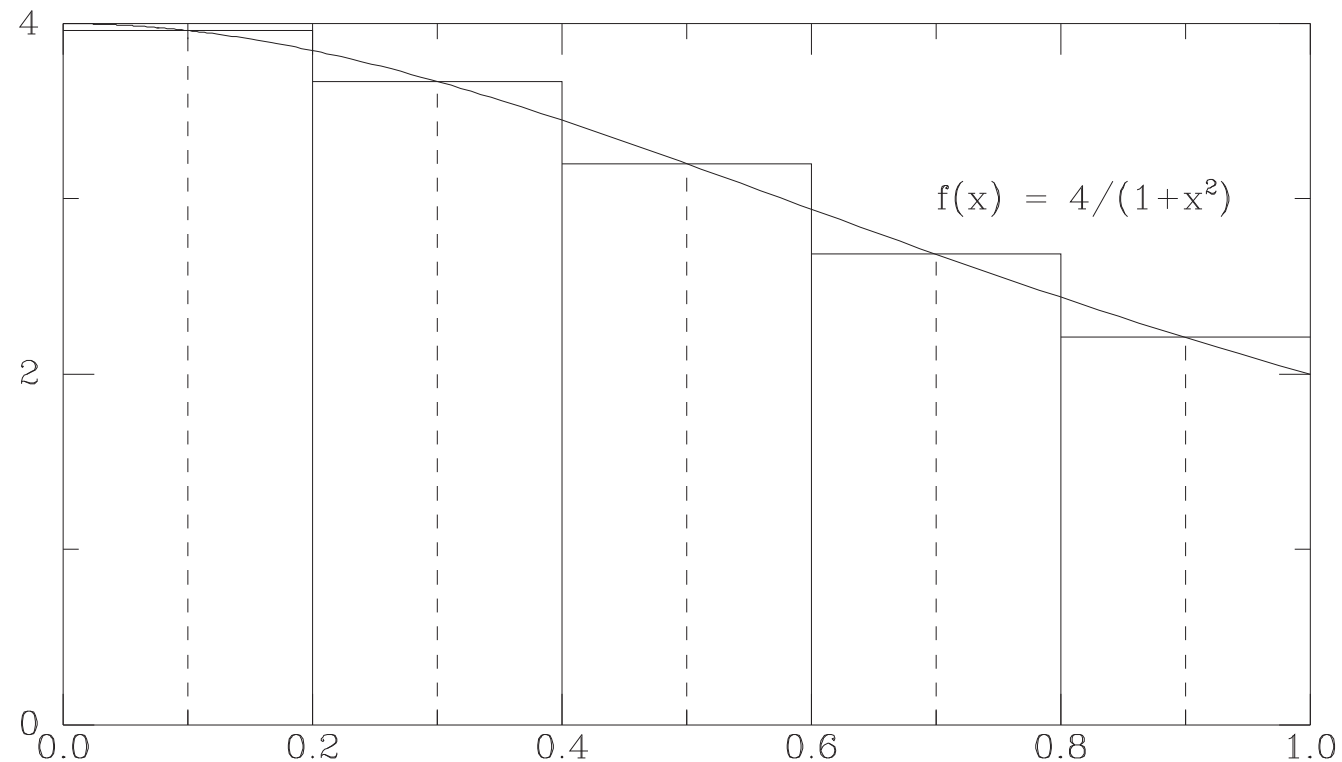
    return 0;
}
```


MPI

Funciones colectivas basicas (ejemplo)

Calculando el valor de pi

$$\int_0^1 \frac{1}{1+x^2} dx = \arctan(x)|_0^1 = \arctan(1) - \arctan(0) = \arctan(1) = \frac{\pi}{4},$$



MPI

Aproximando el valor de pi

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

int main(int argc, char *argv[])
{
    int n=5000, myid, numprocs, i;
    double PI25DT = 3.141592653589793238462643;
    double mypi, pi=0, h, sum, x;
    MPI_Init(NULL, NULL);
    MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myid);
    //comunicamos el numero de intervalos a cada proceso del mundo
    //calculamos los intervalos y repetimos si es necesario
    MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
    h = 1.0 / (double) n;
    sum = 0.0;
    for (i = myid + 1; i <= n; i += numprocs) {
        x = h * ((double)i - 0.5);
        sum += (4.0 / (1.0 + x*x));
    }
    mypi = h * sum;
    printf("my id : %d  mypi : %.16f\n", myid, mypi);

    //colectamos los calculos de cada trabajador usando MPI_Reduce

    MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0,
               MPI_COMM_WORLD);
    //imprimimos el valor luego de sumar
    if (myid == 0)
        printf("pi es aproximadamente %.16f, El error es %.16f\n",
               pi, fabs(pi - PI25DT));

    MPI_Finalize();
    return 0;
}
```

MPI

Multiplicación de matrices

```
int main(void)
{
    int      size, row, column;

    size = ARRAY_SIZE;

    //puntero a la matriz resultante
    int *final_matrix;
    int num_worker, rank;
    MPI_Init(NULL, NULL);
    MPI_Comm_size(MPI_COMM_WORLD, &num_worker);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    if(rank == 0){
        // inicializamos los valores de la MA
        inicializamos_matriz(size, MA);
        //inicializamos los valores de la MB
        inicializamos_matriz(size, MB);
        //imprimimos
        printf("La matriz A es;\n");
        imprimir_matriz(size,MA);
        printf("La matriz B es;\n");
        imprimir_matriz(size,MB);
        //reservamos la memoria para la matriz final
        final_matrix = (int *) malloc(sizeof(int*) * size*size);
    }

    MPI_Bcast(MA, size*size , MPI_INT, 0, MPI_COMM_WORLD);
    MPI_Bcast(MB, size*size , MPI_INT, 0, MPI_COMM_WORLD);

    //chequeamos si proceso 1 recibio la información
    if(rank == 1){
        printf("id:%d La matriz A es;\n",rank);
        imprimir_matriz(size,MA);
        printf("id:%d La matriz B es;\n",rank);
        imprimir_matriz(size,MB);
    }
}
```

```
// determinamos la fila de inicio y fin para la proceso
trabajador
int startrow = rank * ( size / num_worker);
int endrow = ((rank + 1) * ( size / num_worker)) -1;
//calculamos las sub-matrices
int number_of_rows = size / num_worker;
int *result_matrix = (int *) malloc(sizeof(int*) *
number_of_rows * size);
    //multiplicamos
    int rowl=0;
    for(row = startrow; row <= endrow; row++) {
        for (column = 0; column < size; column++) {
            mult(size, row, column,rowl, MA, MB, result_matrix);
            rowl++;
        }
    }
    // log information
    if(rank==1){
        printf("id:%d startrow=%d, endrow=%d,work=%d;
\n",rank,startrow,endrow,number_of_rows*size);
        imprimir_matriz2(number_of_rows,size,result_matrix);
    }

    //recolectamos los resultados de la matriz
    MPI_Gather(result_matrix, number_of_rows*size, MPI_INT,
        final_matrix, number_of_rows*size, MPI_INT, 0,
MPI_COMM_WORLD);

    //imprimimos la matriz luego de recolectar los resultados
    if(rank == 0){
        printf("La matriz resultante C es (MPI);\n");
        imprimir_matriz2(size,size,final_matrix);
    }

    MPI_Finalize();

    return 0;
}
```

MPI

Tipos de datos

MPI Datatype	C Datatype
MPI_CHAR	signed char
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_WCHAR	wchar_t
MPI_SHORT	short
MPI_INT	int
MPI_LONG	long
MPI_LONG_LONG_INT	long long
MPI_SIGNED_CHAR	signed char
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long
MPI_UNSIGNED_LONG_LONG	unsigned long long
MPI_C_COMPLEX	float _Complex
MPI_C_DOUBLE_COMPLEX	double _Complex
MPI_C_LONG_DOUBLE_COMPLEX	long double _Complex
MPI_PACKED	
MPI_BYTE	

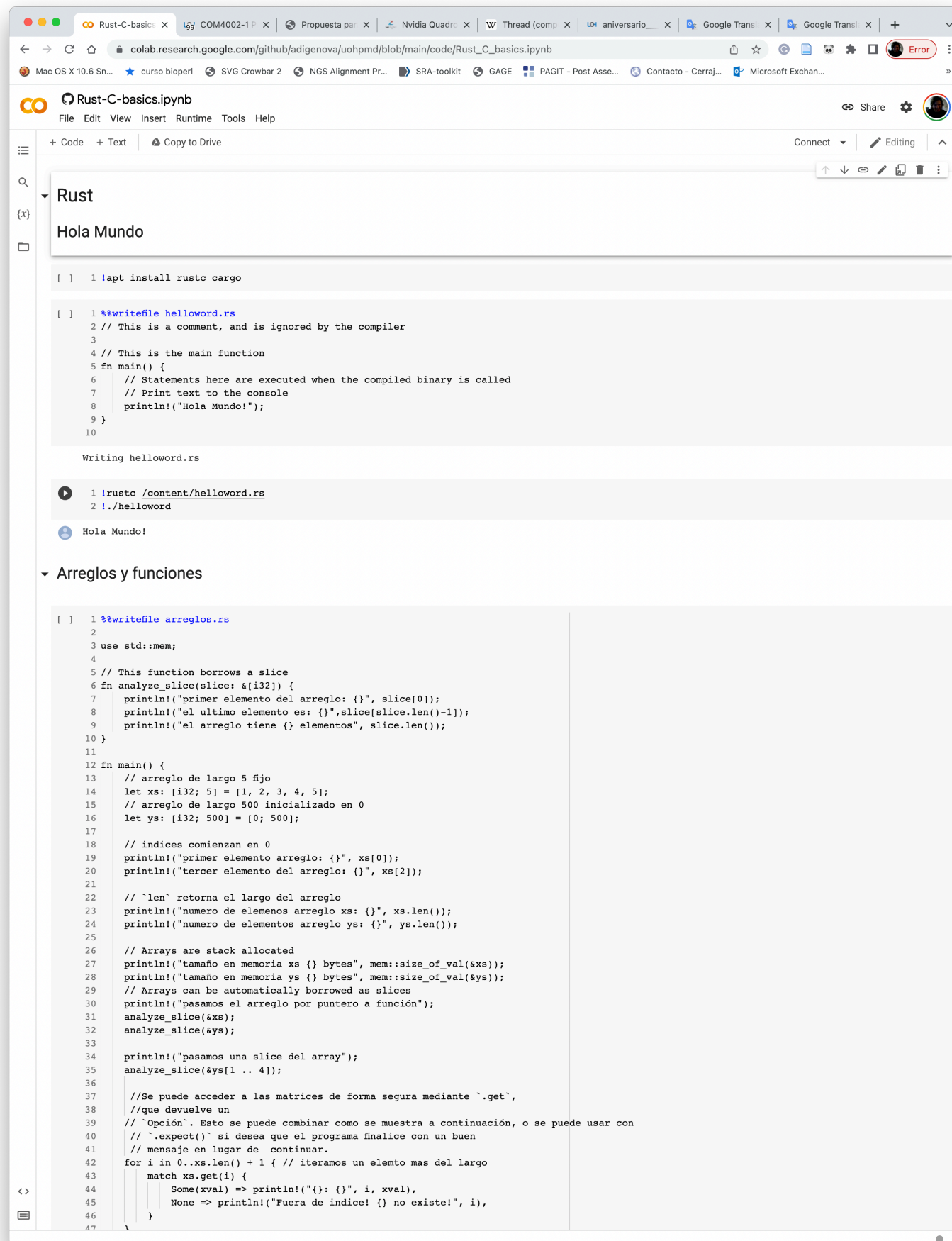
MPI

Tipos de datos

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

int main(int argc, char** argv) {
    int ProcRank, ProcNum, mTag = 0;
    struct { int x;
            int y;
            int z;
        } point;
    MPI_Datatype ptype;
    MPI_Status status;
    MPI_Init(NULL, NULL);
    MPI_Comm_rank(MPI_COMM_WORLD, &ProcRank);
    MPI_Type_contiguous(3, MPI_INT, &ptype);
    MPI_Type_commit(&ptype);
    if (ProcRank == 1) {
        point.x = 45; point.y = 36; point.z = 0;
        MPI_Send(&point, 1, ptype, 0, mTag, MPI_COMM_WORLD);
    }
    if (ProcRank == 0) {
        MPI_Recv(&point, 1, ptype, 1, mTag, MPI_COMM_WORLD, &status);
        printf("Proceso: %d recibio punto con coordenadas (%d; %d; %d)\n", ProcRank, point.x, point.y, point.z);
    }
    MPI_Finalize();
}
```

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 editor area contains two code cells. The first cell has a terminal output showing 'Hola Mundo!'. The second cell contains Rust code for arrays and functions.

```
[ ] 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/MPI_II.ipynb

Consultas?

Consultas o comentarios?

Muchas gracias