

# Procesamiento Masivo de datos: Modelo Hibrido MPI/OpenMP

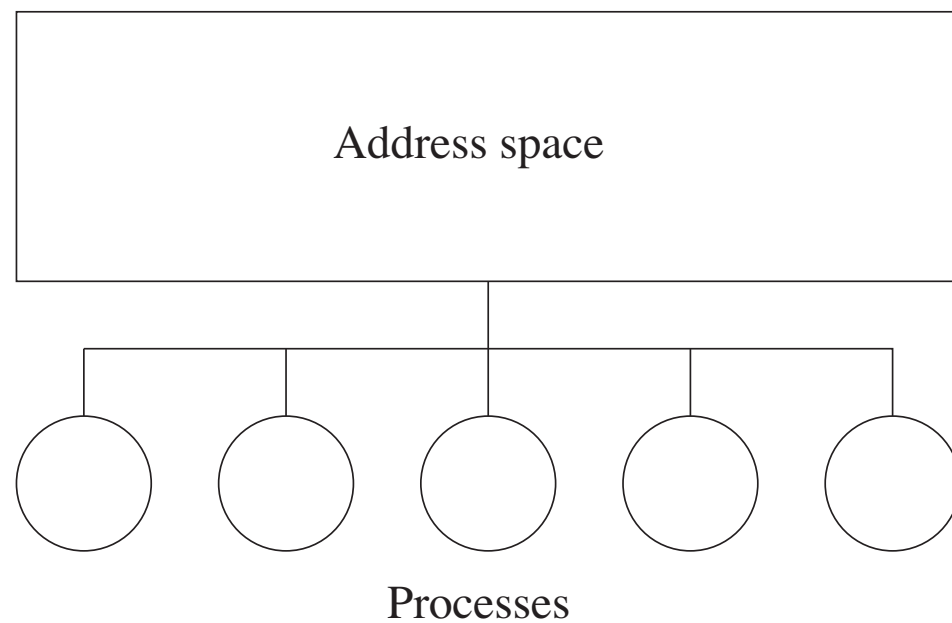
**Alex Di Genova**

**29/09/2022**

# Modelo Híbrido

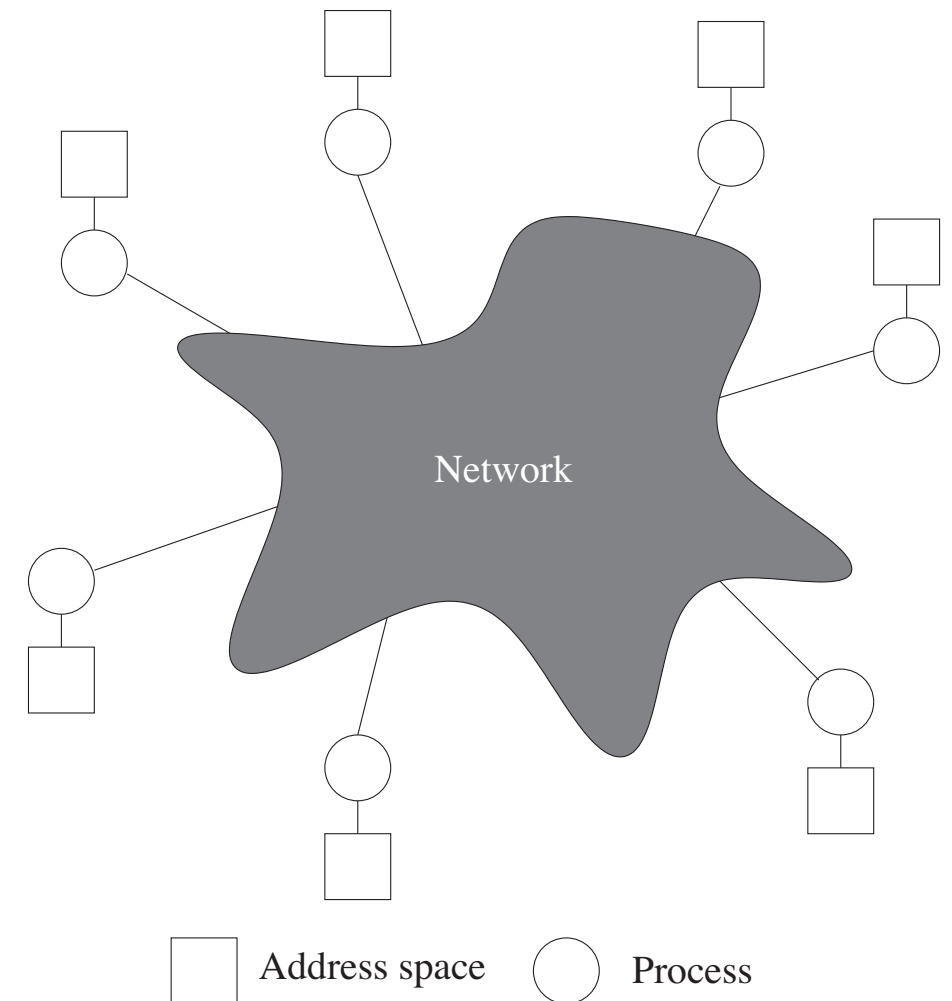
## Modelo de paralelismo

Modelo de memoria compartida

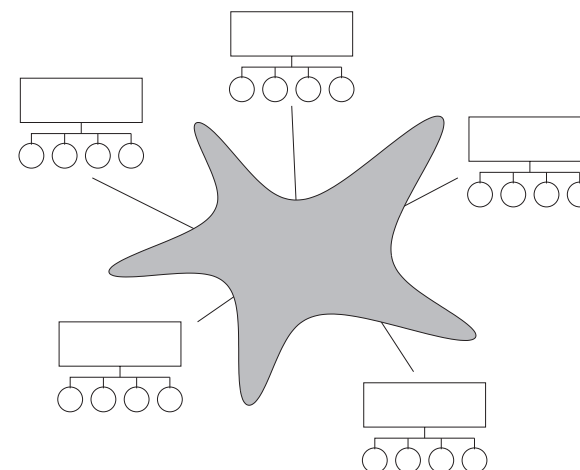


PThread/ OpenMP / Fork

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

- 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
- ¿Por qué?
  - Ahorra memoria al no duplicar datos
  - Minimiza la comunicación de MPI al tener solo 1 proceso por nodo

# 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)

# MPI/OpenMP time

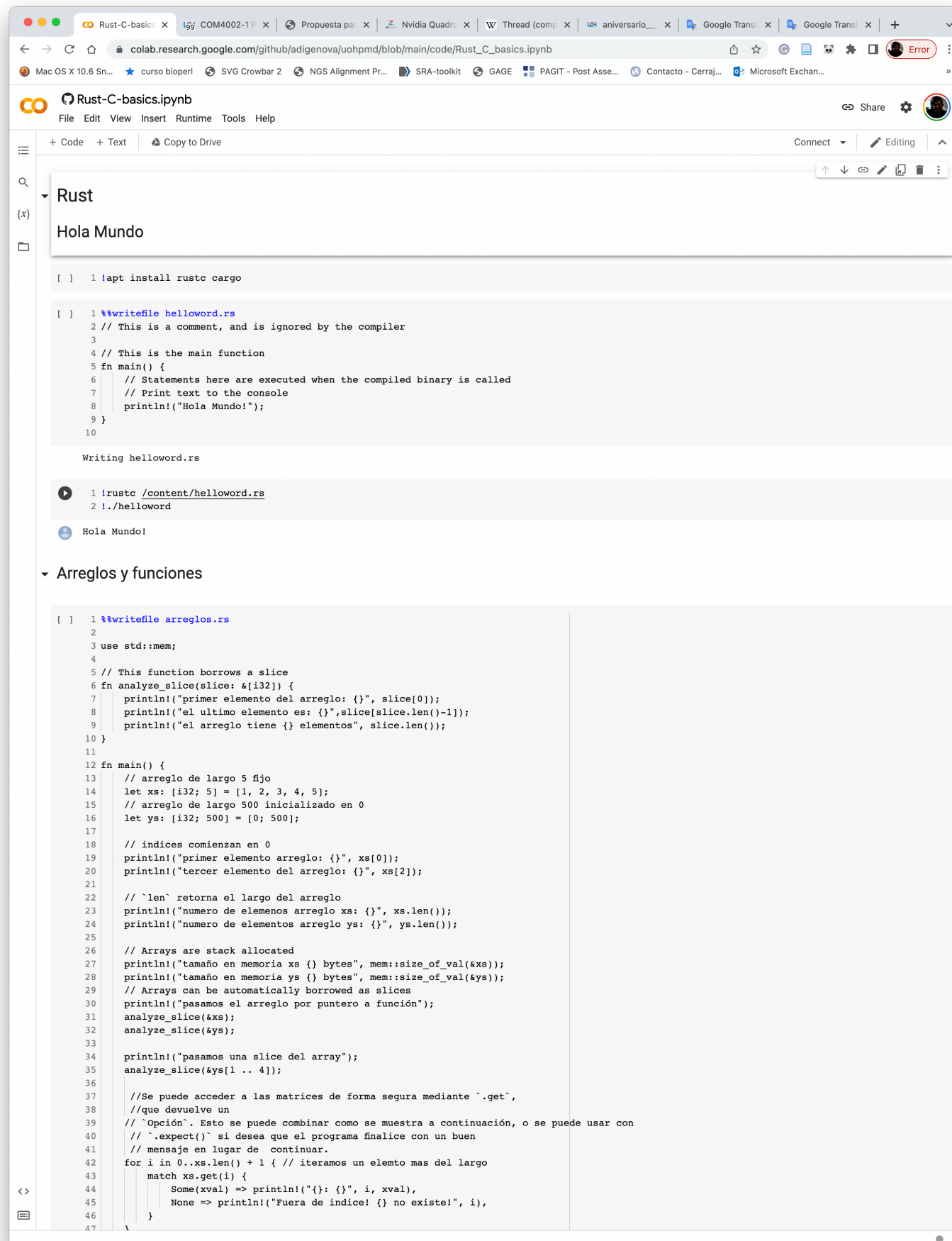
- OpenMP time:

```
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 vertical ellipsis. The left sidebar shows a file explorer with a folder named 'Rust' containing a file 'Hola Mundo'. The main editor area displays Rust code. The first cell contains a terminal command to install Rust and Cargo. The second cell contains Rust code for a 'helloworld.rs' file. The third cell shows the output 'Hola Mundo!'. The fourth cell is a section header 'Arreglos y funciones' followed by a code cell for 'arreglos.rs'.

```
[ ] 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](https://github.com/adigenova/uohpmd/blob/main/code/MPI_II.ipynb)

# Consultas?

Consultas o comentarios?

Muchas gracias