

Práctica 3: Introducción a MPI.

Karina Flores García y Gabriel Alejandro Herrera Gandarela

Abstract—In this practice we perform parallel programming using MPI for this. We observe the operation of the processors for performing tasks and in some cases hyper threading is observed.

I. OBJETIVOS.

- Familiarizarse con MPI.
- Envío y recepción de mensajes.

II. INTRODUCCIÓN.

A. MPI. (Message Passing Interface)

Es un estándar que define la sintaxis y la semántica de las funciones contenidas en una biblioteca en una biblioteca diseñada para explotar la existencia de programas donde requieran el uso de múltiples procesadores. MPI ofrece una serie de funciones y variables las cuales nos van a ayudar a implementar un programa paralelo. Dentro de las herramientas que nos ofrece, hay un comunicador el cual se encargará de dar un ID a cada procesador, esto para que se puedan comunicar entre sí, es importante que cada procesador tenga un ID para que lleve una correcta paralelización, de lo contrario, habrá confusión entre ellos.

En esta práctica veremos algunas de ellas y entenderemos cómo es que funciona MPI y los programas en paralelo.

III. ANÁLISIS DEL PROBLEMA.

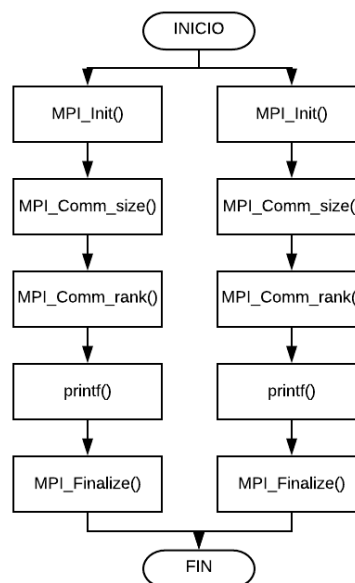
Al iniciar, nuestro código, se ejecutará de forma secuencial, para que después se ejecuten las instrucciones que nos ofrece MPI, es decir, habrá un main que dará la concurrencia a nuestro programa. En la parte secuencial, se definirán las tareas, variables y el esquema de distribución. Al ejecutar nuestro programa, debemos de ser conscientes de las características de nuestro ordenador en cuestión de procesador, ya que definiremos cuántos procesadores queremos que ejecuten dicha sección de código, si pedimos dar instrucciones a más procesadores, no se ejecutará porque no habrá suficiente unidad de procesamiento.

IV. IMPLEMENTACIÓN

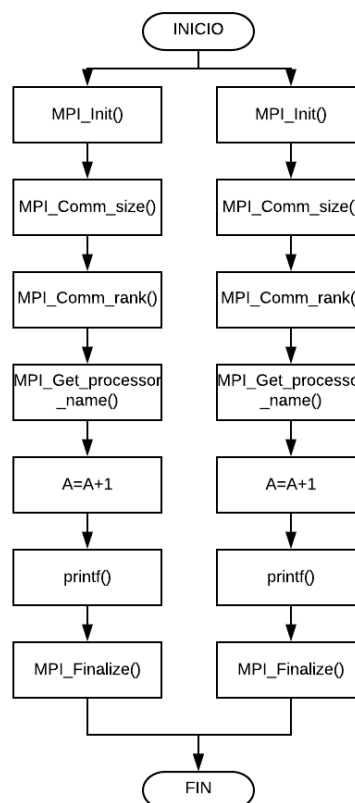
A. Desarrollo

B. Flujo del programa

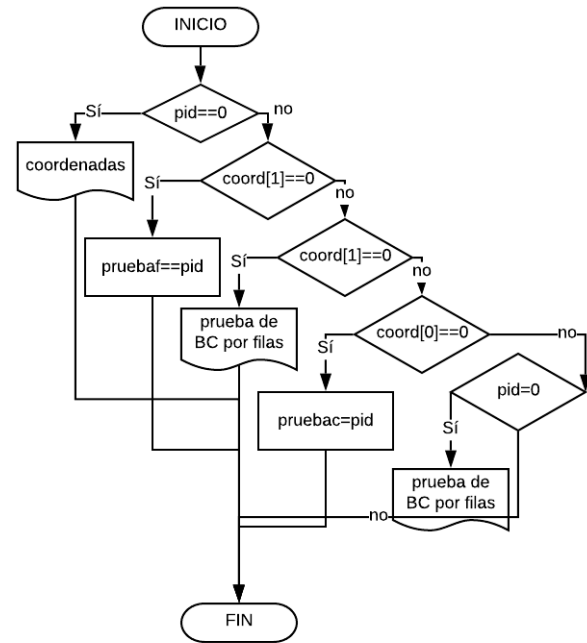
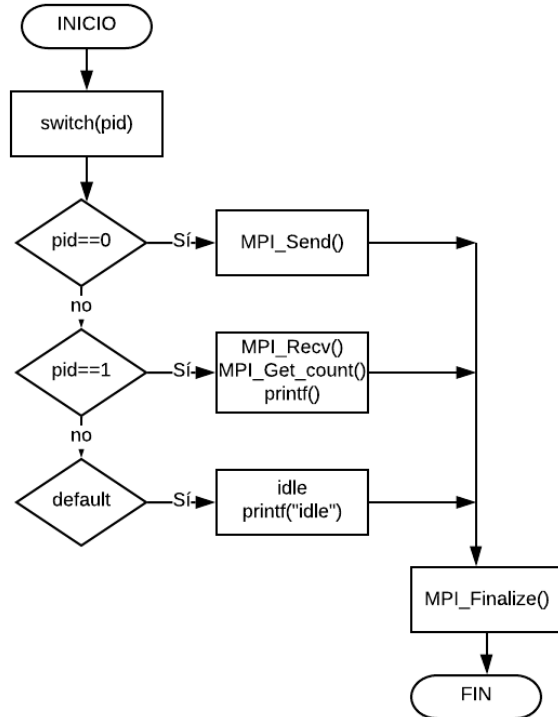
Programa 1



Programa 2

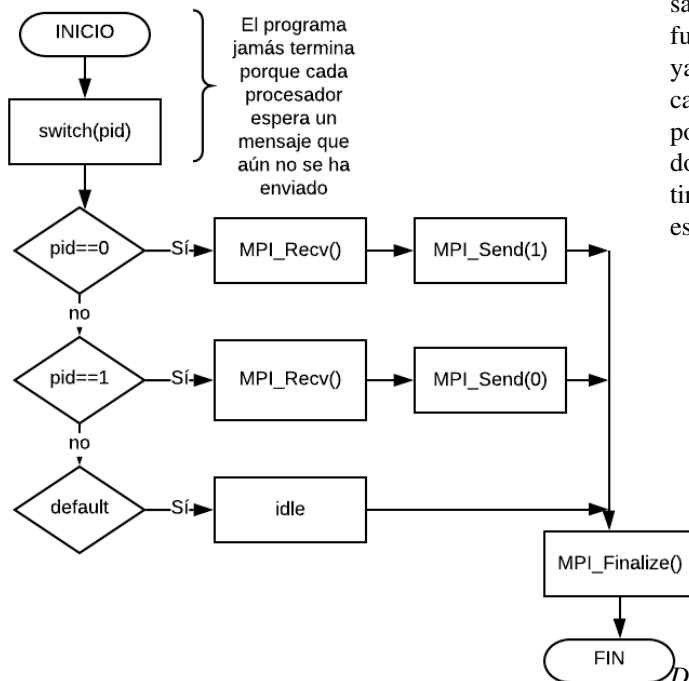


Programa 3

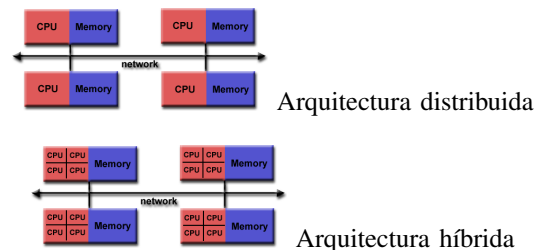


C. Diagrama de patrón arquitectónico

Programa 4



Originalmente, MPI se creó para la comunicación de ordenadores conectados en una red, ya que, en aquella época (1980s) solamente había computadoras con un solo procesador. Sin embargo, las arquitecturas de las computadoras fueron evolucionando a tal grado que, los ordenadores tenían ya varios CPU's, por lo que, MPI optó por adaptarse a estos cambios modificando sus bibliotecas para que pudieran soportar estas arquitecturas. Creando ahora, un sistema híbrido, donde, además de que se pueden intercambiar mensajes distintos ordenadores a través de la red, se pueden intercambiar estos mismos con otros procesadores del mismo ordenador.



D. Diagrama UML de secuencia

Programa 5

Diagrama UML del programa #1 con 1 procesador

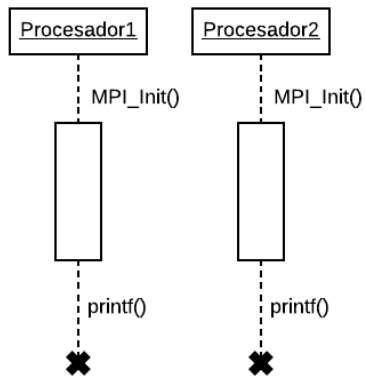


Diagrama UML del programa #1 con 2 procesadores

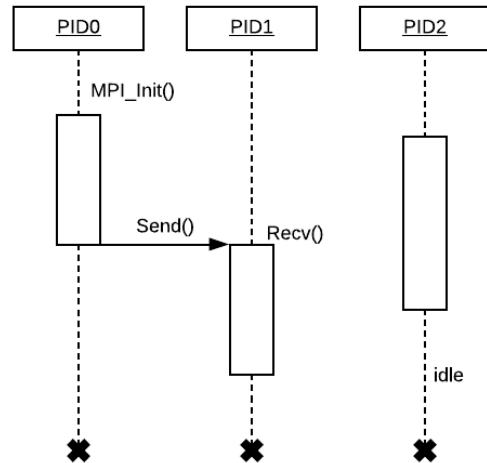


Diagrama UML del programa #4 con 2 procesadores

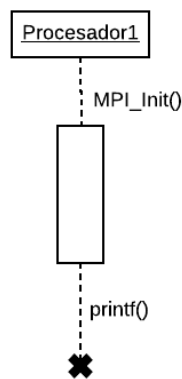


Diagrama UML del programa #2 con 1 procesador

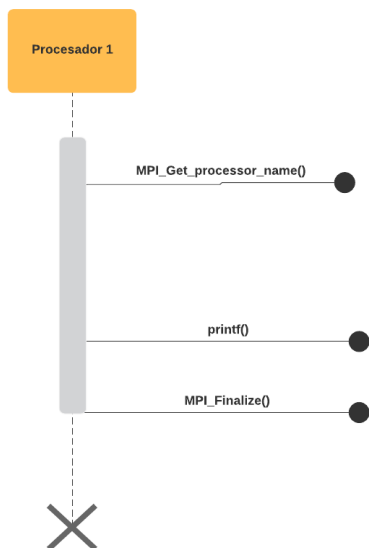
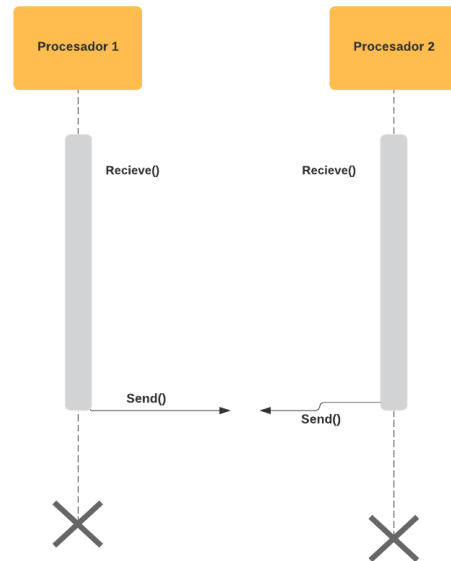


Diagrama UML del programa #3 con 3 procesadores

V. CÓDIGO FUENTE

A. Programa 1: hola.c

```

1  int rank, size;
2  MPI_Init(&argc, &argv);
3  MPI_Comm_size(MPI_COMM_WORLD, &size);
4  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
5  printf("Hello world, i am %d of %d \n", rank,
        size );
6  MPI_Finalize();

```

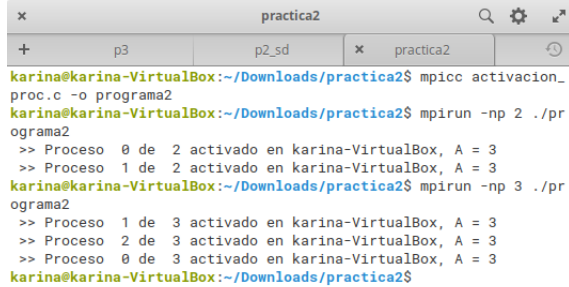
```

x      p3
+  x    p3      p2_sd      practica2
karina@karina-VirtualBox:~/Documents/SD/p3$ mpicc hola.c -o hola
karina@karina-VirtualBox:~/Documents/SD/p3$ mpirun -np 3 ./hola
Hello world, i am 1 of 3
Hello world, i am 2 of 3
Hello world, i am 0 of 3
karina@karina-VirtualBox:~/Documents/SD/p3$

```

B. Programa 2: activacion_proc.c

```
1 MPI_Init(&argc, &argv);
2 MPI_Comm_size(MPI_COMM_WORLD, &npr);
3 MPI_Comm_rank(MPI_COMM_WORLD, &pid);
4 MPI_Get_processor_name(nombrepr, &dnom);
5 A = A + 1;
6 printf(" >> Proceso %2d de %2d activado en %s, A = %d\n", pid, npr, nombrepr, A);
```



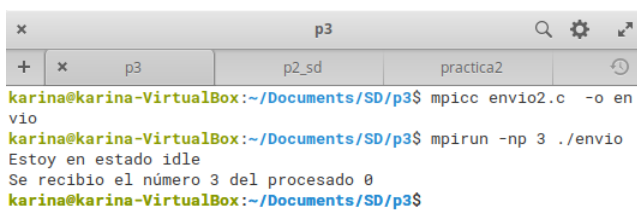
C. Programa 3: envio.c

Programa con if-else

```
1 if (pid == 0)
2 {
3     destino = 1;
4     tag = 0;
5     temp = 3;
6     MPI_Send(&temp, 1, MPI_INT, destino, tag,
7             MPI_COMM_WORLD);
8 } else if (pid == 1) {
9     origen = 0; tag = 0;
10    MPI_Recv(&temp, 1, MPI_INT, origen, tag,
11            MPI_COMM_WORLD, &info);
12    MPI_Get_count(&info, MPI_INT, &ndat);
13    printf("Se recibio el n mero %d del procesado %d\n", temp, info.MPI_SOURCE);
14 }
15 MPI_Finalize();
```

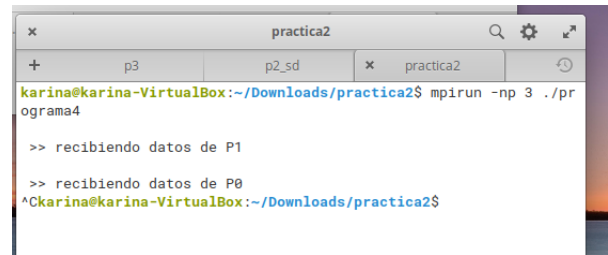
Programa con switch-case

```
1 switch(pid) {
2     case 0:
3         destino = 1;
4         tag = 0;
5         temp = 3;
6         MPI_Send(&temp, 1, MPI_INT, destino, tag,
7                 MPI_COMM_WORLD);
8         break;
9     case 1:
10        origen = 0; tag = 0;
11        MPI_Recv(&temp, 1, MPI_INT, origen, tag,
12                MPI_COMM_WORLD, &info);
13        MPI_Get_count(&info, MPI_INT, &ndat);
14        printf("Se recibio el n mero %d del procesado %d\n", temp, info.MPI_SOURCE);
15        break;
16     default:
17        printf("Estoy en estado idle\n");
18        break;
19 }
```



D. Programa 4: deadlock.c

```
1 if(pid == 0) {
2     A = 5;
3     printf("\n >> recibiendo datos de P1 \n");
4     origen = 1; tag = 1;
5     MPI_Recv(&B, 1, MPI_INT, origen, tag,
6             MPI_COMM_WORLD, &info);
7     printf("\n >> enviando datos a P1\n");
8     destino = 1; tag = 0;
9     MPI_Send(&A, 1, MPI_INT, destino, tag,
10            MPI_COMM_WORLD);
11    C = A + B;
12    printf("\n C es %d en proc0 \n\n", C);
13 } else if (pid == 1) {
14    B = 6;
15    printf("\n >> recibiendo datos de P0 \n");
16    origen = 0; tag = 1;
17    MPI_Recv(&A, 1, MPI_INT, origen, tag,
18            MPI_COMM_WORLD, &info);
19    printf("\n >> enviando datos a P0\n");
20    destino = 0; tag = 0;
21    MPI_Send(&B, 1, MPI_INT, destino, tag,
22            MPI_COMM_WORLD);
23    C = A + B;
24    printf("\n Ce es %d en prol\n\n", C);
25 }
```



E. Programa 5: topologia_maya.c

```
1 if(pid==0) printf("\n --- coordenadas en la
2 malla\n");
3 printf("Proceso %d > pid_malla = %d, coords =
4 (%d, %d), pid_m = %d\n", pid, pid_malla, coord
5 [0], coord[1], pid_m);
6 //creacion de comunicadores para las filas de
7 la malla
8 coord_libre[0] = 0;
9 coord_libre[1] = 1;
10 MPI_Cart_sub(malla_com, coord_libre, &fila_com);
11 //prueba de comunicacion: el primer nodo de
12 cada fila envia
13 //su pid a resto
14 if(coord[1] == 0) pruebaf = pid;
15 MPI_Bcast(&pruebaf, 1, MPI_INT, 0, fila_com);
16 MPI_Comm_rank(fila_com, &pid_fila);
17 if(pid==0) printf("\n --- prueba de BC por
18 filas \n");
19 printf("Proceso %d > coords = (%d, %d),
20 pid_primer_fila = %d, pid_fila: %d\n", pid,
21 coord[0], coord[1], pruebaf, pid_fila);
22 //creacion de comunicadores para las columnas
23 de la malla
24 coord_libre[0] = 1;
25 coord_libre[1] = 0;
26 MPI_Cart_sub(malla_com, coord_libre, &col_com);
27 //prueba de comunicacion: el primer nodo de
28 cada fila, envia su pid al resto
29 if(coord[0]==0) pruebac = pid;
30 MPI_Bcast(&pruebac, 1, MPI_INT, 0, col_com);
31 MPI_Comm_rank(col_com, &pid_col);
32 if(pid==0) printf("\n --- prueba de BC por
33 columnas \n");
```

```
printf("Proceso %d > coords = (%d, %d),
pid_primer_col = %d ,pid_col: %d \n", pid,
coord[0], coord[1], pruebac, pid_col);
```

VI. COMPARACIÓN

Programa 1: En este programa el paralelismo se da sin tener comunicación entre los procesos, todos ejecutan todo el código. En nuestro caso solamente pudimos usar 3 procesadores para la ejecución por lo que se imprime tres veces el mensaje.

Programa 2: En este programa no hay comunicación entre procesos, solamente trata de obtener el nombre del procesador que está ejecutando e imprimirlo en pantalla.

Programa 3: En este caso si hay comunicación entre los procesos paralelos, el proceso con pid=0 le envía un mensaje al pid=1, ese mismo imprime el mensaje y los demás quedan en estado idle.

Programa 4: Hay que tener cuidado a la hora de llevar a cabo la lógica de nuestro programa, en este caso en particular, el programa nunca va terminar ya que, el procesador con Pid = 0, está esperando a que le llegue un mensaje del Pid = 1; y el Pid = 1, espera un mensaje del Pid = 0, a esto se le conoce como: deadlock o abrazo de la muerte.

Programa 5: En este caso observamos que la forma de ejecutar los procesos simula una malla, es decir que la organización se encuentra en una forma similar a una matriz. Suponemos que se trata de la virtualización de los procesadores aunque en nuestra computadora no notamos algo significativo debido a que solo nos permitió ejecutar con 4 procesadores, así que no hubo posibilidad de ver la virtualización y corroborar su funcionamiento.

VII. CONCLUSIONES

En esta práctica, se pudo observar la estandarización que maneja MPI con sus múltiples funciones y variables. Empezamos observando dos programas que solamente se encargaban de levantar el comunicador de MPI, esto para que nuestro programa sepa qué estará trabajando con este entorno. Seguidamente, notamos la transmisión de mensajes entre procesadores, utilizando la función Send(), hemos capaces de transmitir un mensaje al procesador que estaba a la escucha de algún mensaje, es decir, el procesador que estaba a la escucha, se mantenía en estado perezoso hasta que le llegara un mensaje.

Unos de los errores, en cuestión de lógica, se mostro en el programa 4, donde ambos procesadores se instanciaban (por decirlo de alguna forma), sin embargo, siempre estaban a la escucha debido a que ambos esperaban un mensaje de algún procesador para poder continuar con su ejecución, a este error se le llama deadlock o abrazo de la muerte.

Para el último se observó, cómo funciona la topología malla, esta se encarga que todos los procesadores estén interconectados entre sí, listos para recibir o mandar algún mensaje.

REFERENCES

- [1] <https://computing.llnl.gov/tutorials/mpi/>
- [2] Ayala, J.A. (04 marzo 2020) Práctica 3: Introducción a MPI. Sistemas Distribuidos, 3.