



UA

**Universidad
de Alicante**

ESCUELA POLITÉCNICA SUPERIOR

Introducción a MPI 2

Curso académico 2024 / 2025

Semestre 2

Grado en Ingeniería en Inteligencia Artificial

Grupo 1

Profesora: Ricardo Moreno Rodríguez

Alumno: Santiago Álvarez Geanta

ÍNDICE

Introducción	2
Ejercicios	2
Ejercicio 1: Comunicación Punto a Punto	2
Ejercicio 2: Comunicación en Cadena	3
Ejercicio 3: Comunicación Colectiva	4
Ejercicio 4: Reducción de Datos	5
Ejercicio 5: Sincronización y Medición de Tiempos	6

Introducción

Esta práctica tiene como objetivo consolidar los conocimientos fundamentales sobre la utilización de MPI (Message Passing Interface) para la programación paralela. A través de ejercicios progresivos, se explorará la comunicación punto a punto, la comunicación en cadena, la comunicación colectiva, la reducción de datos y la sincronización de procesos. Se espera que el estudiante desarrolle habilidades prácticas en la implementación de aplicaciones paralelas en sistemas multicomputador.

Ejercicios

Ejercicio 1: Comunicación Punto a Punto

La comunicación entre procesos en MPI se logra mediante **MPI_Send** y **MPI_Recv**. En este ejercicio, el proceso 0 envía un número entero al proceso 2, que lo recibe, lo multiplica por 3 y lo imprime. Si se modifican los tags en **MPI_Send** y **MPI_Recv**, la comunicación podría verse afectada si los valores no coinciden, ya que MPI utiliza estos identificadores para asociar los mensajes con sus respectivos receptores. Además, si el proceso receptor ejecuta **MPI_Recv** antes de que el proceso emisor envíe el dato, este quedará bloqueado hasta que el mensaje sea enviado. En el caso de que un proceso intente enviar un mensaje a un proceso inexistente, el programa fallaría debido a que MPI requiere que todos los procesos involucrados en la comunicación existan dentro del comunicador definido.

Modificación requerida: El proceso 0 debe enviar un número al proceso 2, quien lo multiplicará por 3 antes de imprimirlo.

Código modificado:

```
1  #include <mpi.h>
2  #include <stdio.h>
3
4  int main(int argc, char** argv) {
5      MPI_Init(&argc, &argv);
6      int rank;
7      MPI_Comm_rank(MPI_COMM_WORLD, &rank);
8
9      if (rank == 0) {
10         int dato = 7;
11         MPI_Send(&dato, 1, MPI_INT, 2, 0, MPI_COMM_WORLD);
12     } else if (rank == 2) {
13         int recibido;
14         MPI_Recv(&recibido, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
15         recibido *= 3;
16         printf("Proceso 2 recibió y multiplicó: %d\n", recibido);
17     }
18
19     MPI_Finalize();
20     return 0;
21 }
```

Ejercicio 2: Comunicación en Cadena

En este ejercicio, el proceso 0 inicia el envío de un dato que es modificado y reenviado sucesivamente por cada proceso hasta alcanzar el último. Si un proceso omite el envío, la comunicación se interrumpe, provocando que los procesos posteriores no reciban el dato. En el caso de ejecutar el programa con un solo proceso, la comunicación no ocurriría y solo se imprimiría el valor inicial sin modificaciones. Para hacer que el proceso 0 también imprima el dato original antes de enviarlo, bastaría con incluir una línea de impresión antes de ejecutar `MPI_Send`.

Modificación requerida: Haz que el dato inicial sea igual a 10 y cada proceso lo incremente en 2 antes de pasarlo al siguiente.

```
1  #include <mpi.h>
2  #include <stdio.h>
3
4  int main(int argc, char** argv) {
5      MPI_Init(&argc, &argv);
6      int rank, size;
7      MPI_Comm_rank(MPI_COMM_WORLD, &rank);
8      MPI_Comm_size(MPI_COMM_WORLD, &size);
9
10     int dato;
11     if (rank == 0) {
12         dato = 10;
13         MPI_Send(&dato, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
14     } else {
15         MPI_Recv(&dato, 1, MPI_INT, rank - 1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
16         dato += 2;
17         if (rank < size - 1) {
18             MPI_Send(&dato, 1, MPI_INT, rank + 1, 0, MPI_COMM_WORLD);
19         } else {
20             printf("Valor final: %d\n", dato);
21         }
22     }
23
24     MPI_Finalize();
25     return 0;
26 }
```

Ejercicio 3: Comunicación Colectiva

La función `MPI_Bcast` se usa para difundir datos de un proceso (root) a todos los demás procesos. En este ejercicio, el proceso 0 envía un dato a todos los procesos. Si el root cambia, el valor que se difunde será diferente. Usar `MPI_Bcast` evita enviar el mensaje manualmente a cada proceso, lo cual sería más complicado con `MPI_Send`. Si un proceso no participa, no recibirá el dato.

Modificación requerida: El proceso 2 debe enviar un número a todos los procesos con `MPI_Bcast`, y cada uno debe incrementarlo en 1.

```
1  #include <mpi.h>
2  #include <stdio.h>
3
4  int main(int argc, char** argv) {
5      MPI_Init(&argc, &argv);
6
7      int rank;
8      MPI_Comm_rank(MPI_COMM_WORLD, &rank);
9      int dato;
10
11     if (rank == 2) {
12         dato = 100;
13     }
14     int MPI_Bcast()
15     MPI_Bcast(&dato, 1, MPI_INT, 2, MPI_COMM_WORLD);
16
17     dato++; |
18     printf("Proceso %d recibió: %d\n", rank, dato);
19
20     MPI_Finalize();
21     return 0;
22 }
23
```

Ejercicio 4: Reducción de Datos

Con `MPI_Reduce`, se pueden realizar operaciones en los datos distribuidos entre los procesos y obtener un resultado global. En este ejercicio, los procesos suman sus ranks y el proceso 0 imprime la suma total. Si se usa `MPI_Gather`, se recolectan los datos sin hacer una operación de reducción. Existen otras operaciones como la multiplicación o el máximo.

Modificación requerida: Todos los procesos deben calcular y mostrar el máximo de los ranks utilizando `MPI_Reduce`.

```
1  #include <mpi.h>
2  #include <stdio.h>
3
4  int main(int argc, char** argv) {
5      MPI_Init(&argc, &argv);
6
7      int rank, max_rank;
8      MPI_Comm_rank(MPI_COMM_WORLD, &rank);
9
10     MPI_Reduce(&rank, &max_rank, 1, MPI_INT, MPI_MAX, 0, MPI_COMM_WORLD);
11
12     if (rank == 0) {
13         printf("El máximo de los ranks es: %d\n", max_rank);
14     }
15
16     MPI_Finalize();
17     return 0;
18 }
19
```

Ejercicio 5: Sincronización y Medición de Tiempos

La función `MPI_Barrier` se usa para sincronizar a todos los procesos. El tiempo de ejecución puede medirse con `MPI_Wtime`. Si un proceso realiza una espera antes de la barrera, el tiempo de sincronización se afectará. Medir estos tiempos ayuda a entender los cuellos de botella y optimizar la ejecución en aplicaciones paralelas.

Modificación requerida: Simula un desbalance de carga haciendo que los procesos con rank par esperen 2 segundos antes de la barrera y mide el tiempo total.

```
1  #include <mpi.h>
2  #include <stdio.h>
3  #include <unistd.h>
4
5  int main(int argc, char** argv) {
6      MPI_Init(&argc, &argv);
7
8      int rank;
9      MPI_Comm_rank(MPI_COMM_WORLD, &rank);
10
11     double t1 = MPI_Wtime();
12
13     if (rank % 2 == 0) {
14         sleep(2);
15     }
16
17     MPI_Barrier(MPI_COMM_WORLD);
18
19     double t2 = MPI_Wtime();
20     printf("Proceso %d: Tiempo = %f segundos\n", rank, t2 - t1);
21
22     MPI_Finalize();
23     return 0;
24 }
```