

Práctica 9

Introducción a MPI (Ejemplos completos)

Jordi Blasco Lozano
Computación de alto rendimiento
Grado en Inteligencia Artificial

Índice:

Índice:	2
1. Introducción	2
2. Desarrollo	3
Ejercicio 1 - Comunicación punto a punto	3
Código para la tarea propuesta	4
Ejercicio 2 - Comunicación en cadena	6
Código para la tarea propuesta	7
Ejercicio 3 - Comunicación colectiva	8
Código para la tarea propuesta	9
Ejercicio 4 - Reducción de datos	10
Código para la tarea propuesta	11
Ejercicio 5 - Sincronización y medición de tiempos	12
Código para la tarea propuesta	13

1. Introducción

Esta práctica de MPI consta de 5 ejercicios diseñados para introducir y consolidar los conceptos básicos de la programación paralela con MPI. Se abordan temas esenciales como la comunicación punto a punto, la comunicación en cadena, la reducción de datos, y la sincronización y medición de tiempos, permitiendo experimentar con el envío y recepción de mensajes, la acumulación de resultados, y el manejo de desbalances en la carga de trabajo. Cada ejercicio proporciona ejemplos comentados, preguntas de comprensión y una tarea práctica para afianzar lo aprendido.

2. Desarrollo

Ejercicio 1 - Comunicación punto a punto

Este ejercicio tiene como objetivo afianzar el conocimiento en la comunicación punto a punto utilizando MPI (Message Passing Interface).

¿Qué hace el programa propuesto?

El programa inicia con la llamada a `MPI_Init(&argc, &argv)`, que configura el entorno de MPI y permite que los procesos se comuniquen. Cada proceso obtiene su identificador único (rank) mediante `MPI_Comm_rank(MPI_COMM_WORLD, &rank)`. Esto es fundamental para asignar roles diferentes a cada proceso según su rango.

En este ejemplo se implementa una comunicación punto a punto entre dos procesos: el proceso 0 y el proceso 1. El proceso 0 es el encargado de enviar un dato (en este caso, el número 42) al proceso 1, que se encargará de recibirlo y posteriormente imprimirlo.

Cuando el proceso 0 detecta que su rank es 0, define una variable `dato` con el valor 42 y realiza la operación de envío mediante `MPI_Send(&dato, 1, MPI_INT, 1, 0, MPI_COMM_WORLD)`. Aquí se especifica:

- La dirección del dato a enviar (`&dato`).
- El número de elementos a enviar (1).
- El tipo de dato (`MPI_INT`).
- El proceso destino (1).
- Un tag (0) que actúa como etiqueta para identificar el mensaje.
- El comunicador (`MPI_COMM_WORLD`).

En el proceso 1, al comprobar que su rank es 1, se declara la variable `recibido` para almacenar el dato entrante. Luego se llama a `MPI_Recv(&recibido, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE)`, la cual espera recibir el mensaje del proceso 0. Los parámetros indican:

- La dirección donde se almacenará el dato recibido.
- El número de elementos que se esperan.
- El tipo de dato (`MPI_INT`).
- El proceso fuente (0), es decir, de dónde se espera el mensaje.
- El tag (0), que debe coincidir con el tag usado en el envío.
- El comunicador.

Y se ignora el estado del mensaje recibido.

Una vez que el proceso 1 recibe el mensaje, se imprime el dato utilizando `printf`

¿Qué sucede si cambiamos los tags?

Los tags (etiquetas) son fundamentales para emparejar mensajes entre el envío y la recepción. En el programa, tanto el proceso 0 al enviar como el proceso 1 al recibir utilizan el tag 0. Si se cambiara el tag en uno de ellos, por ejemplo, enviando con tag 1 pero intentando recibir con tag 0, el mensaje no se emparejaría correctamente. Esto provocaría que el proceso que llama a `MPI_Recv` se quede bloqueado esperando un mensaje con el tag correcto, o se produzca un error en la comunicación.

¿Qué ocurre si el proceso 1 ejecuta `MPI_Recv` antes de que el proceso 0 envíe el mensaje?

Si el proceso 1 llama a `MPI_Recv` antes de que el proceso 0 realice el envío, este se bloqueará esperando el mensaje. Este comportamiento es normal en la comunicación bloqueante de MPI, ya que el proceso receptor se detiene hasta que el mensaje correspondiente es enviado y recibido.

¿Qué diferencia habría si el proceso 0 intentase enviar el dato a un proceso inexistente?

Si el proceso 0 intenta enviar el dato a un proceso que no existe (por ejemplo, especificando un destino con un rank que no se haya iniciado), MPI detectaría un error. Este error puede causar que el programa termine de forma inesperada o que se comporte de manera indefinida, ya que se estaría intentando comunicar con un proceso que no forma parte del comunicador. Es esencial que el rango del proceso destino exista y esté incluido en el comunicador utilizado.

¿Cómo compilar y ejecutar con `mpicc`?

- **Compilación**

Para compilar con nuestro compilador `mpicc` lo hacemos igual que cualquier otro archivo `.c`, `mpicc -o ejercicio1 ejercicio1.c`.

Si queremos compilar con un compilador de `c++` podemos usar el de `mpic++`, `mpic++ -o ejercicio1 practica1.cpp`

- **Ejecución**

Para la ejecución de los programas lo haremos de la siguiente forma, tanto si es de `c` como si es de `c++` `mpirun -np 3 ./ejercicio1`, el número que sigue a `-np` es el número de procesos que se ejecuta en paralelo

Código para la tarea propuesta

La tarea consiste en modificar el ejemplo para que el proceso 0 envíe un número al proceso 2, y este último lo multiplique por 3 antes de imprimirlo.

¿Cómo funciona?

Proceso 0: Envía un número (en este ejemplo, 42) al proceso 2 utilizando MPI_Send.

Proceso 2: Recibe el mensaje con MPI_Recv, multiplica el valor recibido por 3 y muestra el resultado por pantalla.

MPI_Send y MPI_Recv

Estas funciones permiten la comunicación bloqueante entre procesos. El proceso emisor envía datos a un proceso receptor especificando, entre otros parámetros, el identificador del proceso destino y un tag para identificar el mensaje. El receptor, a su vez, llama a MPI_Recv esperando recibir el mensaje con el tag correspondiente.

Manejo de Procesos

La estructura condicional (if y else if) en el código permite asignar tareas específicas según el rango del proceso. Esto garantiza que solo el proceso 0 envíe datos y solo el proceso 2 los reciba y procese.

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

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

    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    if (rank == 0) {
        int dato = 42; // Valor a enviar
        MPI_Send(&dato, 1, MPI_INT, 2, 0, MPI_COMM_WORLD);
    } else if (rank == 2) {
        int recibido;
        // Recibimos el dato enviado por el proceso 0
        MPI_Recv(&recibido, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        // Multiplicamos el dato por 3
        int resultado = recibido * 3;
        printf("Proceso 2 recibió %d y al multiplicarlo por 3 da: %d\n", recibido, resultado);
    }
    MPI_Finalize();
    return 0;
}
```

Salida



practica8 - .shell.txt

```
1 root@07fae025b509:/workdir# mpirun -np 4 ./ejercicio1_mod
2 Proceso 2 recibió 42 y al multiplicarlo por 3 da: 126
```

Ejercicio 2 - Comunicación en cadena

El programa propuesto utiliza la comunicación en cadena para que el dato viaje de un proceso al siguiente hasta alcanzar el último.

¿Cómo funciona?

- **Proceso 0:**
Inicia con un dato = 0.
Envía este dato al proceso 1 usando `MPI_Send`.
- **Procesos intermedios (1 hasta size-2):**
Cada proceso recibe el dato del proceso anterior mediante `MPI_Recv`.
Se incrementa el valor recibido (con `recibido++`).
Envía el nuevo valor al siguiente proceso con `MPI_Send`.
- **Proceso final (rank == size - 1):**
Recibe el dato, lo incrementa y, al no tener a quién enviarlo (ya que es el último), imprime el valor final.

```
practica8 - .shell.txt
1 root@07fae025b509:/workdir# mpirun -np 10 ./ejercicio2
2 Valor final: 9
3 root@07fae025b509:/workdir# mpirun -np 2 ./ejercicio2
4 Valor final: 1
```

¿Qué sucede si un proceso omite el envío?

Si algún proceso no realiza el envío (por ejemplo, si no llama a `MPI_Send` después de incrementar el dato), la cadena se rompe. Los procesos posteriores se quedarán bloqueados en `MPI_Recv`, ya que nunca recibirán el mensaje esperado.

¿Qué ocurre si ejecutamos el programa con solo un proceso?

Si se ejecuta el programa con un solo proceso, éste tendrá rank 0 y se ejecutará únicamente el bloque de código correspondiente al proceso 0, que intenta enviar el dato al proceso 1. Como no existe el proceso 1, se generará un error o un comportamiento indefinido. Por ello, se requiere que el programa se ejecute con al menos dos procesos para que la comunicación en cadena tenga sentido.

¿Cómo podríamos hacer que el proceso 0 también imprima el dato inicial?

Para que el proceso 0 imprima el dato inicial, se puede incluir una instrucción de impresión (printf) antes de enviar el dato. Esto permitiría ver el valor original que se está transmitiendo en la cadena.

Código para la tarea propuesta

La tarea consiste en modificar el programa para que:

- El dato inicial sea 10 (en lugar de 0).
- Cada proceso incremente el dato en 2 (en lugar de 1) antes de enviarlo al siguiente.

```
#include <mpi.h>
#include <stdio.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);

    if (rank == 0) {
        int dato = 10; // Dato inicial modificado a 10
        // Imprimir el dato inicial en el proceso 0
        printf("Proceso 0 inicia con: %d\n", dato);
        MPI_Send(&dato, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
    } else {
        int recibido;
        // Cada proceso recibe el dato del proceso anterior
        MPI_Recv(&recibido, 1, MPI_INT, rank - 1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        // Incrementa el dato en 2
        recibido += 2;
        // Si no es el último proceso, se envía al siguiente; de lo contrario, se imprime el
        resultado final
        if (rank < size - 1) {
            MPI_Send(&recibido, 1, MPI_INT, rank + 1, 0, MPI_COMM_WORLD);
        } else {
            printf("Valor final en el proceso %d: %d\n", rank, recibido);
        }
    }

    MPI_Finalize();
    return 0;
}
```

Salida

```
practica8 - .shell.txt

1 root@07fae025b509:/workdir# mpirun -np 4 ./ejercicio2_mod
2 Proceso 0 inicia con: 10
3 Valor final en el proceso 3: 16
4 root@07fae025b509:/workdir# mpirun -np 7 ./ejercicio2_mod
5 Proceso 0 inicia con: 10
6 Valor final en el proceso 6: 22
```

Ejercicio 3 - Comunicación colectiva

En el ejercicio se usa `MPI_Bcast` para difundir un dato desde un proceso raíz a todos los procesos que participan en el comunicador.

¿Cómo funciona?

- El proceso con rank 0 inicializa la variable dato con el valor 99.
- Se llama a `MPI_Bcast(&dato, 1, MPI_INT, 0, MPI_COMM_WORLD)`, lo que permite que todos los procesos, sin importar su rank, reciban el valor 99.

Cada proceso imprime el dato recibido.

```
practica8 - .shell.txt
1 root@07fae025b509:/workdir# mpirun -np 4 ./ejercicio3
2 Proceso 0 recibió: 99
3 Proceso 2 recibió: 99
4 Proceso 1 recibió: 99
5 Proceso 3 recibió: 99
6 root@07fae025b509:/workdir# mpirun -np 3 ./ejercicio3
7 Proceso 0 recibió: 99
8 Proceso 1 recibió: 99
9 Proceso 2 recibió: 99
```

¿Qué pasa si el root es diferente?

Si se cambia el proceso root (el que inicializa el dato) por otro (por ejemplo, no el 0), el proceso designado es el que debe inicializar el dato. Todos los procesos recibirán el valor desde ese proceso. Es importante que el proceso root tenga el dato inicial correcto, ya que es el que se transmite a los demás.

¿Qué problema evitaríamos usando `MPI_Bcast` en lugar de `MPI_Send`?

Usar `MPI_Bcast` evita tener que realizar múltiples llamadas a `MPI_Send` para enviar el dato a cada proceso individualmente. Con `MPI_Bcast` se garantiza que todos los procesos reciben el mismo dato de manera sincronizada y se simplifica el código, además de reducir el riesgo de errores en la comunicación.

¿Qué ocurre si un proceso no participa en la operación colectiva?

En una operación colectiva, todos los procesos que pertenecen al comunicador deben participar. Si un proceso no participa en la llamada a `MPI_Bcast`, se pueden generar bloqueos o comportamientos indefinidos, ya que la operación espera la participación de todos los procesos.

Código para la tarea propuesta

La tarea consiste en modificar el programa para que:

- El proceso 2 sea el root de la difusión (inicializa el dato).
- El proceso 2 envíe un número a todos los procesos mediante `MPI_Bcast`.
- Una vez recibido el dato, cada proceso lo incrementa en 1 antes de imprimirlo.

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

int main(int argc, char** argv) {
    MPI_Init(&argc, &argv);
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    int dato;
    if (rank == 2) dato = 99; // El root inicializa el dato
    MPI_Bcast(&dato, 1, MPI_INT, 2, MPI_COMM_WORLD); // Difusión a todos
    dato += 1;
    printf("Proceso %d recibió: %d\n", rank, dato);
    MPI_Finalize();
    return 0;
}
```

Salida

```
practica8 - .shell.txt

1 root@07fae025b509:/workdir# mpirun -np 3 ./ejercicio3_mod
2 Proceso 0 recibió: 100
3 Proceso 1 recibió: 100
4 Proceso 2 recibió: 100
5 root@07fae025b509:/workdir# mpirun -np 4 ./ejercicio3_mod
6 Proceso 2 recibió: 100
7 Proceso 3 recibió: 100
8 Proceso 0 recibió: 100
9 Proceso 1 recibió: 100
```

Ejercicio 4 - Reducción de datos

El código propuesto suma los rangos (identificadores) de todos los procesos. Para ello, cada proceso conoce su propio rank y se utiliza la función `MPI_Reduce` con el operador `MPI_SUM` para acumular estos valores en el proceso root (en este caso, el proceso 0).

```
practica8 - .shell.txt
1 root@07fae025b509:/workdir# mpirun -np 4 ./ejercicio4
2 Suma de ranks: 6
3 root@07fae025b509:/workdir# mpirun -np 7 ./ejercicio4
4 Suma de ranks: 21
```

Corroboración de los datos:

- procesos (0,1,2,3) suma = 6
- procesos (0,1,2,3,4,5,6,) = 21

¿Qué diferencia hay entre `MPI_Gather` y `MPI_Reduce`?

Mientras que `MPI_Gather` recopila datos de todos los procesos y los almacena en un único arreglo en el proceso root (sin aplicar operaciones aritméticas o lógicas), `MPI_Reduce` combina los datos de todos los procesos aplicando una operación (por ejemplo, suma, máximo, mínimo, etc.) para obtener un único resultado en el proceso root. En nuestro ejemplo, se usa `MPI_Reduce` para sumar los ranks.

¿Qué pasaría si omitimos el proceso root en `MPI_Reduce`?

Si se omite el proceso root, o se especifica un valor incorrecto, el resultado combinado no se almacenará correctamente en un proceso en particular, lo que puede ocasionar que el proceso no tenga acceso al resultado final. Además, la llamada a `MPI_Reduce` requiere que se especifique un proceso root válido para que la operación se realice de forma correcta.

¿Qué otras operaciones podrías realizar en lugar de suma?

Además de la suma (`MPI_SUM`), `MPI_Reduce` permite realizar otras operaciones, tales como el máximo (`MPI_MAX`), el mínimo (`MPI_MIN`), el producto (`MPI_PROD`), la operación lógica AND (`MPI_LAND`), entre otras. En el ejercicio que nos ocupa, vamos a utilizar `MPI_MAX` para calcular el máximo de los ranks.

Código para la tarea propuesta

La tarea consiste en modificar el programa para que utilizando `MPI_Reduce` con el operador `MPI_MAX`, se calcule el máximo de los ranks. Usaré el proceso 0 para imprimir el resultado que nos da `MPI_MAX`.

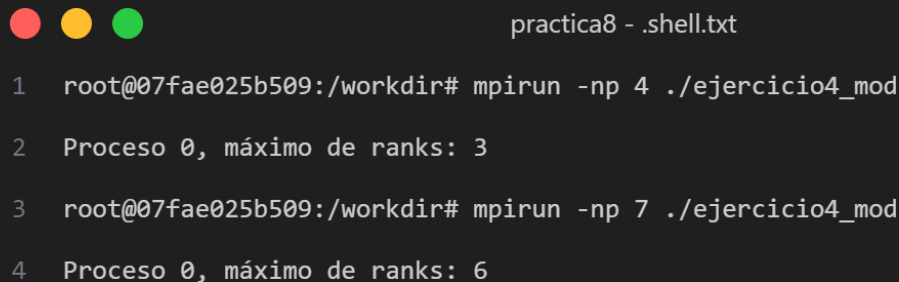
```
#include <mpi.h>
#include <stdio.h>

int main(int argc, char** argv) {
    MPI_Init(&argc, &argv);
    int rank, maxRank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Reduce(&rank, &maxRank, 1, MPI_INT, MPI_MAX, 0, MPI_COMM_WORLD);

    if (rank == 0) {
        printf("Proceso %d, máximo de ranks: %d\n", rank, maxRank);
    }

    MPI_Finalize();
    return 0;
}
```

Salida



```
practica8 - .shell.txt

1 root@07fae025b509:/workdir# mpirun -np 4 ./ejercicio4_mod
2 Proceso 0, máximo de ranks: 3
3 root@07fae025b509:/workdir# mpirun -np 7 ./ejercicio4_mod
4 Proceso 0, máximo de ranks: 6
```

Ejercicio 5 - Sincronización y medición de tiempos

El programa original mide el tiempo que tarda cada proceso en llegar a la barrera de sincronización. Cada proceso registra un tiempo inicial (con `MPI_Wtime()`), se sincroniza con `MPI_Barrier` y luego registra el tiempo final, mostrando la diferencia. Esto nos permite ver cuánto tarda cada proceso en sincronizarse.

Cada proceso puede registrar tiempos distintos porque llegan a la barrera en momentos diferentes debido a variaciones en la ejecución o carga de trabajo.

```
practica8 - .shell.txt
1 root@ce6bfd6aadd1:/workdir# mpirun -np 4 ./ejercicio5
2 Proceso 0: Tiempo = 0.000488
3 Proceso 1: Tiempo = 0.000488
4 Proceso 2: Tiempo = 0.000490
5 Proceso 3: Tiempo = 0.000494
```

¿Por qué cada proceso puede registrar un tiempo diferente?

Aunque todos inician la medición con `MPI_Wtime()`, cada proceso puede llegar a la barrera en momentos distintos debido a diferencias en la carga de trabajo, la ejecución en diferentes nodos, o variaciones en el sistema operativo. Esto hace que el tiempo registrado ($t_2 - t_1$) varíe entre procesos.

¿Qué pasaría si un proceso hace un `sleep(3)` antes de llegar a la barrera?

Si un proceso se retrasa (por ejemplo, haciendo `sleep(3)`) antes de llegar a la barrera, todos los demás procesos quedarán esperando en la barrera hasta que ese proceso se una. Como consecuencia, el tiempo medido por cada proceso incluirá este retraso, lo que incrementará la diferencia ($t_2 - t_1$).

¿Qué utilidad tiene medir estos tiempos en aplicaciones reales?

Medir los tiempos es fundamental para identificar cuellos de botella y desbalances en la carga de trabajo. En aplicaciones reales, esta información permite optimizar la distribución de tareas y mejorar el rendimiento general del programa, ya que se pueden detectar procesos que están retrasando la sincronización.

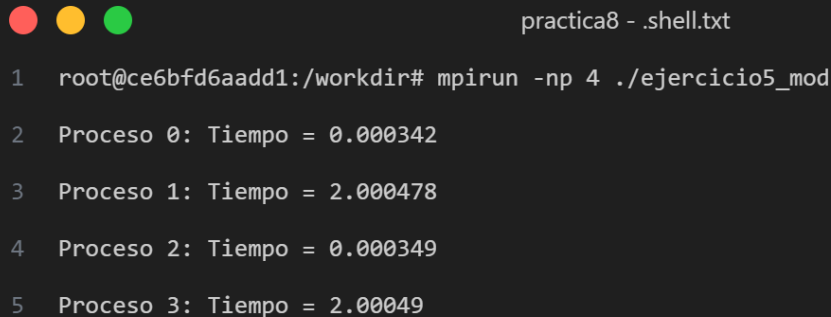
Código para la tarea propuesta

La tarea consiste en simular un desbalance de carga haciendo que los procesos con rank par esperen 2 segundos antes de la barrera y, de esta forma, medir el tiempo total que tarda cada proceso en sincronizarse. Para lograrlo, se introduce un retraso condicional en los procesos pares usando la función `sleep(2)`.

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

int main(int argc, char** argv) {
    MPI_Init(&argc, &argv);
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    if (rank % 2 == 0) { // Esperamos 2 segundos en los rangos pares
        sleep(2);
    }
    double t1 = MPI_Wtime();
    MPI_Barrier(MPI_COMM_WORLD); // Sincronización
    double t2 = MPI_Wtime();
    printf("Proceso %d: Tiempo = %f\n", rank, t2 - t1);
    MPI_Finalize();
    return 0;
}
```

Salida



```
practica8 - .shell.txt

1 root@ce6bfd6aadd1:/workdir# mpirun -np 4 ./ejercicio5_mod
2 Proceso 0: Tiempo = 0.000342
3 Proceso 1: Tiempo = 2.000478
4 Proceso 2: Tiempo = 0.000349
5 Proceso 3: Tiempo = 2.00049
```

Salida esperada ya que hemos hecho el `sleep` antes de que se registre el tiempo, por esto mismo los procesos 0 y 2 empiezan a medir el tiempo después de haber esperado los 2 segundos