PRACTICA 3

Modelo de comunicación colectivo

María González Herrero

Santiago Molpeceres Díaz

UNIVERSIDAD NEBRIJA

Índice

Ejercicio 1

Se pide un programa donde el nodo raíz (0) inicialice una variable con un valor arbitrario, después este se modifica (por ejemplo, calculando el cuadrado de su valor) y finalmente lo envía al resto de los nodos.

Se pide hacerlo con comunicación punto a punto y con comunicación colectiva.

Comunicación punto a punto

Código

Primero debemos tener en cuenta que el código que ponemos es general para todos los procesos. Por eso es necesario hacer distinciones entre procesos.

Para nuestro programa primero declaramos las variables que vamos a usar.

```
int size;
int rank;
int numero;
```

- Size: El número de procesos que participan en el programa.
- Rank: El número de proceso.
- Número: Él es valor que vamos a pasar por los nodos.

La primera distinción es para el proceso 0 que es el proceso raíz.

```
13   if (rank == 0)
14   {
15      printf ("Dime de que valor quieres calcular su cuadrado, y
        pasarlo en anillo:\n");
16      scanf ("%i", &numero);
17      numero = numero * numero;
18      MPI_Send (&numero, 1, MPI_INT, (rank + 1) % size, rank,
         MPI_COMM_WORLD);
19      printf ("Proceso %i envía %d\n", rank, numero);
   }
```

Con la línea 15 pedimos el usuario que nos día el valor con el que quiere funcionar y lo guardamos en la variable “Número” con la línea 16. La línea 17 hace el cuadrado este número y lo guarda en la variable número.

La línea 18 manda el número guardado en la variable “Número” al siguiente nodo.

La siguiente distinción es con los nodos que no son el último.

```
21 else if (rank < size - 1)
22    {
23       MPI_Recv (&numero, 1, MPI_INT, rank - 1, rank - 1,
         MPI_COMM_WORLD, MPI_STATUS_IGNORE);
24       printf ("Proceso: %i recibe el valor: %i\n", rank, numero);
25       MPI_Send (&numero, 1, MPI_INT, rank + 1, rank, MPI_COMM_WORLD);
26    }
```

Estos nodos usan recepción y envío con bloqueo tal que con la línea 23, se quedan a la espera de la recepción de un mensaje del nodo que le precede y lo guarda en la variable "Número" (línea 23).

Y finalmente reenvía el mensaje al siguiente nodo (línea 25).

La última distinción es la del nodo final.

```
   else if (rank == size - 1)
   {
      MPI_Recv (&numero, 1, MPI_INT, rank - 1, rank - 1, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
      printf ("Proceso: %i recibe el valor: %i y se acaba el envío\n", rank, numero);
   }
```

Este nodo simplemente recibe el mensaje.

Recuerda:
Las líneas 19,24,30.

Son mensajes que aparecerán por consola para un mejor seguimiento del ejercicio, no son relevantes para la práctica.

Comunicación Colectiva

Código

La primera distinción es el proceso raíz 0.

```
if (rank == 0)
   {
      printf ("Dime de que valor quieres calcular su cuadrado, y pasarlo en anillo:\n");
      scanf ("%i", &numero);
      numero = numero * numero;
   }
```

En el que pedimos al usuario cual va a ser el dato que vamos a pasar por el buffer. Después usamos la función MPI_Bcast.

La función MPI_Bcast tiene este esquema.

```
1.  int MPI_Gather(const void* buffer_de_envio,
2.                 int numero_de_elementos_en_el_buffer,
3.                 MPI_Datatype tipo_de_dato_transmitido,
4.                 void* buffer_recepcion,
5.                 int numero_de_elementos_recibidos,
6.                 MPI_Datatype tipo_de_dato_recibido,
7.                 int raíz_que_envia_el_mensaje,
8.                 MPI_Comm Comunicador);
```

```
MPI_Bcast (&numero, 1, MPI_INT, 0, MPI_COMM_WORLD);
```

Por tanto, según el esquema anterior:

- Mandamos el valor guardado en la variable número
- Solo mandamos un elemento
- Es de tipo Integer
- La raíz es el nodo 0
- El comunicador es el MPI_COMM_WORLD

Esta función mandará el dato guardado en la variable número del nodo raíz a todos los procesos involucrados en el programa.

Recuerda:
Las líneas 20-27

Son mensajes que aparecerán por consola para un mejor seguimiento del ejercicio, no son relevantes para la práctica.

Salida por pantalla

```
Dime de que valor quieres calcular su cuadrado, y pasarlo en anillo:
2
Proceso 0 envia 4
Proceso: 1 recibe el valor: 4
Proceso: 2 recibe el valor: 4 y se acaba el envio
```

Ejercicio 2

Se pide: implementar un programa donde el nodo 0 inicializa un array unidimensional asignando a cada valor su índice. Este array es dividido en partes, donde cada una de ellas será mandada a un proceso/nodo diferente. Después de que cada nodo haya recibido su porción de datos, los actualiza sumando a cada valor su rank. Por último, cada proceso envía su porción modificada al proceso root. (Hacerlo para que el número de datos total (N) sea múltiplo del número de procesos)

Primero debemos tener en cuenta que el código que ponemos es general para todos los procesos. Por eso es necesario hacer distinciones entre procesos.

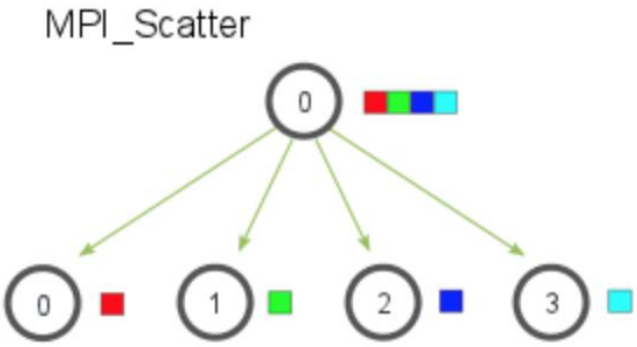
Para nuestro programa primero declaramos las variables que vamos a usar.

```
int size;
int rank;
int numero;
int my_value; // donde lo guardo
int deVuelta;
```

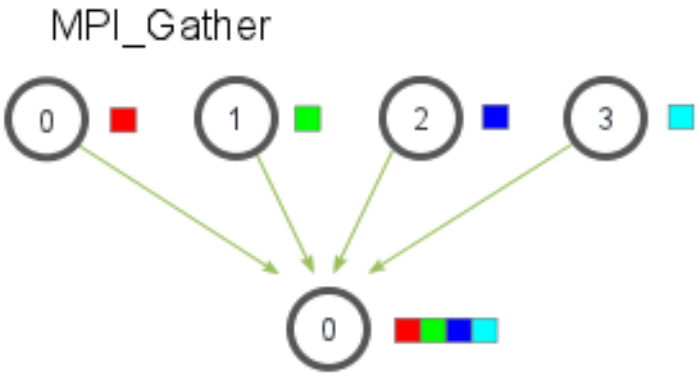
- size: El número de procesos que participan en el programa
- rank: El valor del proceso
- numero: El valor que vamos a pasar
- my_value: El valor que el proceso recibe
- deVuelta: El valor que los procesos devuelven al nodo raíz.

Antes de meternos al código veamos las dos funciones que entran en juego para este ejercicio.

La primera es la función MPI_Scatter. Esta función hace una difusión del mensaje que quiere pasar el nodo raíz al resto de los procesos.



Y después tenemos la función MPI_Gather. Es la función inversa a MPI_Scatter. Los nodos del buffer envían al nodo raíz un mensaje.



Una vez interiorizadas estas funciones vemos el código.

Código

Primero vamos a ver el funcionamiento del nodo raíz.

```
16  if (rank == 0)
17  {
18    int numeros_enviar[size*2];
19    int numeros_recibidos[size*2];
20    for (int i = 0; i < size*2; i++)
21    {
22       numeros_enviar[i] = i;
23    }
24    MPI_Scatter (&numeros_enviar, ((size*2) /size), MPI_INT,
        &numeros_recibidos, ((size*2)/size), MPI_INT, 0, MPI_COMM_WORLD);
25    printf ("El proceso %d ha recibido el valor de:\n", rank);
26    int k=0;
27    while (k<((size*2) /size))
28    {
29       printf ("Recibido: %d\n", numeros_recibidos[k]);
30       k=k+1;
31    }
32    MPI_Gather (&numeros_recibidos, ((size*2) /size), MPI_INT,
        numeros_recibidos, ((size*2) /size), MPI_INT, 0, MPI_COMM_WORLD);
33
34    while (! numeros_recibidos[(size*2) - 1]) {};
35
36    printf ("El proceso 0 ha recibido:\n");
37    int j = 0;
38    while (j < size*2)
39    {
40       printf ("Iteracion %i - %i\n", j, numeros_recibidos[j]);
41       j = j + 1;
42    };
43  }
```

Primero declaramos los arrays de envío y recepción (números_enviar y números_recividos). Como el ejercicio nos pide que el número total de elementos sea múltiplo del número de procesos, multiplicamos el número de procesos por 2;

```
    int numeros_enviar[size*2];
    int numeros_recibidos[size*2];
```

Rellenamos el array de envío cuyo valor de cada celda es su posición.

```
for (int i = 0; i < size*2; i++)
{
   numeros_enviar[i] = i;
}
```

Y enviamos el mensaje con la función MPI_Scatter, que tiene este esquema:

```
1.  int MPI_Scatter(const void* buffer_de_envio,
2.                  int número_de_elemenos_a_enviar_a_cada_proceso,
3.                  MPI_Datatype Tipo_de_dato_a_enviar,
4.                  void* buffer_de_recepcion,
5.                  int número_de_elemenos_que_recibira_de_cada_proceso,
6.                  MPI_DatatypeTipo_de_dato_de_recepcion,
7.                  int raiz,
8.                 MPI_Comm Comunicador);
```

```
24 MPI_Scatter (&numeros_enviar, ((size*2) /size), MPI_INT,
        &numeros_recibidos, ((size*2) /size), MPI_INT, 0, MPI_COMM_WORLD);
```

Como indica el esquema, mandamos los elementos guardados en "números_enviar", y enviaremos una partición equitativa para cada proceso son ((size*2) /size), de tipo Integer, y guardaremos los datos a recibir en "números_recibidos" que voy a recibir ((size*2) /size) elementos de cada proceso, de tipo Integer; asignamos la raíz del buffer al nodo 0, y para finalizar indicamos el comunicador (MPI_COMM_WORLD).

Para recibir la respuesta del resto de nodos usamos la función MPI_Gather, con este esquema:

```
1.  int MPI_Gather(const void* buffer_de_envio,
2.              int número_de_elementos_a_enviar_en_el_buffer,
3.              MPI_Datatype tipo_de_dato_de_envio,
4.              void* buffer_de_recepcion,
5.              int número_de_elementos_por_mensaje,
6.              MPI_Datatype tipo_de_datos_de_recepcion,
7.              int raiz,
8.              MPI_Comm Comunicador);
```

```
MPI_Gather (&numeros_recibidos, ((size*2) /size), MPI_INT,
        &numeros_recibidos, ((size*2) /size), MPI_INT, 0, MPI_COMM_WORLD);
```

Por tanto, enviaremos los datos guardados en "números_recibidos", enviaremos ((size*2) /2) elementos, de tipo Integer, y los guardaremos en "números_recibidos"; recibiremos un número total de ((size*2) /2) por cada proceso; declaramos el nodo 0 como raíz y asignamos MPI_COMM_WORLD como comunicador.

Para el resto de los nodos se les aplica el siguiente código

```
    int numeros_enviar[((size*2) /size)];
    int numeros_recibidos[((size*2) /size)];
    int deVuelta[((size*2) /size)];
```

La dimensión de estos arrays es de ((size*2) /size) ya que solo van a trabajar con los datos que van a recibir del nodo raíz, y no son tantos como los que transmite, por tanto, los limitamos a su porción de datos.

Una vez declaradas esperamos la recepción del MPI_Scatter del nodo raíz, entonces usamos:

```
MPI_Scatter (NULL, 0, MPI_INT, &numeros_recibidos,
((size*2) /size), MPI_INT, 0, MPI_COMM_WORLD);
```

Si nos fijamos esta función tiene el valor null en el buffer de envío y es porque no buscamos enviar nada. Solo nos centramos en recibir lo que manda el nodo raíz.

La traducción sería: No enviamos ningún elemento, por tanto, enviamos 0 elementos, pero aun así debemos indicar el tipo que serían así que de tipo Integer, y los datos recibidos los guardo en "números_recibidos" y vamos a recibir ((size*2) /2) elementos de tipo Integer de un buffer con la raíz en el nodo 0 y el comunicador MPI_COMM_WORLD.

Como dice el ejercicio, como respuesta debemos devolver el valor recibido sumado al rank.

```
    for (int i=0; i<((size*2) /size); i++) {
       deVuelta[i]=numeros_recividos[i]+rank;
    }
```

Y finalmente lo enviamos.

```
MPI_Gather (&deVuelta, ((size*2) /size), MPI_INT, NULL, 0, MPI_INT, 0,
MPI_COMM_WORLD);
```

La traducción sería: Enviaremos los datos que están guardados en "deVuelta", enviando ((size*2) /2) elementos, de tipo Integer, pero no vamos a guardar ninguna recepción y el buffer de recepción va a esperar 0 elementos, pero tenemos que declarar el tipo de dato que sería, que en este caso es Integer; señalamos que la raíz del buffer es 0 y declaramos el comunicador MPI_COMM_WORLD.

Salida por pantalla

```
El proceso 0 ha recibido el valor de :
Recibido:0
Recibido:1
El proceso 1 ha recibido el valor de :
Recibido:2
Recibido:3
El proceso 2 ha recibido el valor de :
Recibido:4
Recibido:5
El proceso 0 ha recibido:
Iteracion 0 - 0
Iteracion 1 - 1
Iteracion 2 - 3
Iteracion 3 - 4
Iteracion 4 - 6
Iteracion 5 - 7
```

Recuerda:
Las líneas 27-31,34-42,50-56.

Son mensajes que aparecerán por consola para un mejor seguimiento del ejercicio, no son relevantes para la práctica.

Ejercicio 3

Se pide:

Implementar un programa donde cada proceso inicializa un array de una dimensión, asignando a todos los elementos el valor de su rank+1. Después el proceso 0 (root) ejecuta dos operaciones de reducción (suma y después producto) sobre los arrays de todos los procesos.

Código

Para este ejercicio debemos declarar las siguientes variables:

```
int size, rank;
int arr[size], recep1[size], envio[size], recep2[size];
```

- size: El número de procesos que participan en el programa
- rank: El valor que tiene cada proceso
- arr: Array del primer envío
- recep1: Array de la primera recepción
- envio: Array del segundo envío
- recep2: Segunda recepción.

Se crean tantos arrays debido a que no podemos usar dos veces el mismo buffer.

Primero declaramos el array del primero envío, que es arr; tal que

```
for (int i = 0; i < size; i++)
{
   arr[i] = rank + 1;
}
```

Y hacemos la primera operación, con la función MPI_Reduce, que tiene el siguiente esquema:

```
1.  int MPI_Reduce(const void* buffer_de_envio,
2.                 void* buffer_de_recepcion,
3.                 int numero_de_elementos_en_el_buffer_de_envio,
4.                 MPI_Datatype tipo_de_datos_en_el_buffer,
5.                 MPI_Op operacion,
6.                 int raiz,
7.                 MPI_Comm comunicador);
```

Por tanto, declaramos la función tal que:

```
MPI_Reduce (&arr, &recep1, size, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
```

La traducción sería:

Enviamos los datos guardados en “arr” y los recibidos en “recep1”. Van a ser distribuidos en un N=size y van a ser de tipo Integer. Indicamos que queremos hacer la operación suma y que el nodo raíz es el 0. Finalmente declaramos el comunicador MPI_COMM_WORLD

Para la segunda parte debemos usar otros buffers, y eso nos lleva a tener que mover datos, tal que el array “envió” debe tener los mismos valores que arr, pero no podemos igualar arrays, ya que una de ellas la estamos usando como buffer.

Por tanto, hay que hacerlo de la siguiente manera.

```
if (rank == 0)
{
    for (int i = 0; i < size; i++)
    {
        envio[i] = rank+1;
    }
}
else
{
    for (int i = 0; i < size; i++)
    {
        envio[i] = rank +1;
    }
}
```

Y finalmente hacer la siguiente operación.

```
MPI_Reduce (&envio, &recep2, size, MPI_INT, MPI_PROD, 0,
MPI_COMM_WORLD);
```

Su traducción sería:

Enviamos los datos guardados en “envío” y los recibidos los guardamos en “recep2”. Van a ser distribuidos en un N=size y van a ser de tipo Integer. Indicamos que queremos hacer la operación suma y que el nodo raíz es el 0. Finalmente declaramos el comunicador MPI_COMM_WORLD

Esquema Conceptual

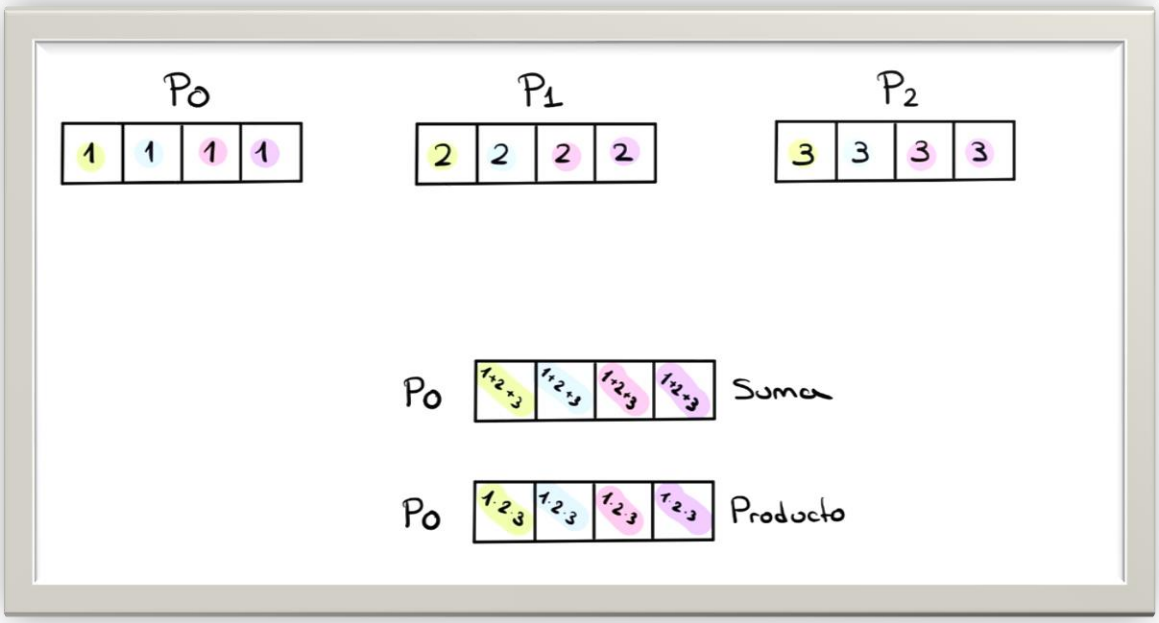


Imagen de Laura Rodríguez Soriano

Salida por consola

Para un ejemplo de size =4

```
La suma en la posicion 0 es: 10
El producto en la posicion 0 es: 24
La suma en la posicion 1 es: 10
El producto en la posicion 1 es: 24
La suma en la posicion 2 es: 10
El producto en la posicion 2 es: 24
La suma en la posicion 3 es: 10
El producto en la posicion 3 es: 24
```

Recuerda:

Las líneas 33-42

Son mensajes que aparecerán por consola para un mejor seguimiento del ejercicio, no son relevantes para la práctica.

Webgrafía

https://www.rookiehpc.com/mpi/docs/mpi_scatter.php

https://www.rookiehpc.com/mpi/docs/mpi_gather.php

https://www.rookiehpc.com/mpi/docs/mpi_reduce.php