Práctica 2

Por Jaime Tejedor y Luis Díaz de Río

Se pide lo siguiente:

Ejercicio 1:

Implementar un código donde utilizando comunicación punto a punto dos procesos rebotan continuamente los mensajes entre sí, hasta que deciden detenerse una vez alcanzado límite autoimpuesto.

```c
#include "mpi.h"
#include <stdio.h>
#include <string.h>

int main(int argc, char** argv){
    int rank;
    char msg[20];
    MPI_Status status;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    strcpy(msg, "Hello World!");
    for(int i=0; i < 3; i++){
        if(rank == 0){
            MPI_Send(&msg, 13, MPI_CHAR, 1, 100, MPI_COMM_WORLD);
            printf("Sending to child message: %s \n", msg);
            MPI_Recv(&msg, 13, MPI_CHAR, 1, 100, MPI_COMM_WORLD, &status);
        }else{
            MPI_Recv(&msg, 13, MPI_CHAR, 0, 100, MPI_COMM_WORLD, &status);
            printf("Sending to master message: %s \n", msg);
            MPI_Send(&msg, 13, MPI_CHAR, 0, 100, MPI_COMM_WORLD);
        }
    }
    MPI_Finalize();
    return 0;
}
```

Para este ejercicio hemos utilizado como base parte del código dado en el enunciado de la práctica.

Al no necesitar el tamaño, ya que el enunciado pide que el mensaje rebote entre dos procesos, no necesitamos el número de procesos. Creamos la variable rank, donde se almacenará el número del proceso en el que estamos. El mensaje, como en el ejemplo que hay en el enunciado, es un string que hemos llamado msg. Además creamos una variable de tipo MPI_Status (En este ejercicio no lo utilizamos por lo que podríamos dejarlo como MPI_STATUS_IGNORE).

Tras llamar a la función MPI_Init(), utilizamos la función MPI_Comm_rank para almacenar en la variable rank el proceso en el que estamos. También utilizamos la función strcpy() para almacenar en el string msg “Hello World!”.

Para iterar utilizamos un bucle for, en nuestro caso hemos decidido que los procesos reboten 3 veces.

En el interior del for tenemos un if, donde comprobamos el proceso en el que estamos. Hay que destacar que el mensaje “Hello World!” está compuesto por 12 caracteres, aunque cuando enviamos y recibimos el mensaje decimos que el tamaño va a ser de 13. El elemento 13 es un carácter “NULL” con el que sirve para diferenciar entre distintas cadenas dentro de un mismo string.

En cuanto a las funciones de MPI_Send() y MPI_Recv(), tanto para rank=0 y rank=1 son iguales, cambiando únicamente a donde se envía y de donde recibe el mensaje. Además, para diferenciar entre rank=0 y rank=1 utilizamos distintos printf().

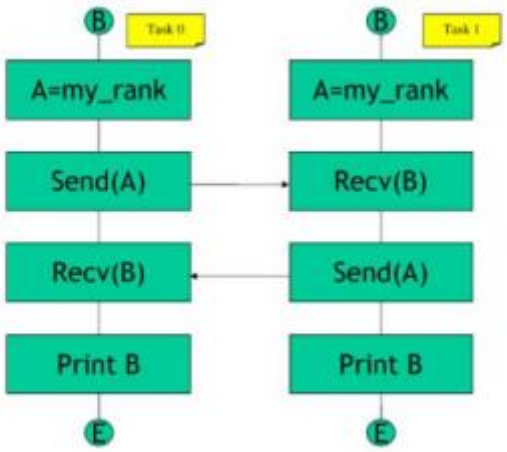
```
noone@noone-VirtualBox:~/Escritorio/ArquitecturaOrdenadores/practica2/Ejercicio1$ mpirun -np 2 ./ejercicio1
Sending to child message: Hello World!
Sending to master message: Hello World!
Sending to child message: Hello World!
Sending to master message: Hello World!
Sending to child message: Hello World!
Sending to master message: Hello World!
```

Como podemos ver, el mensaje se envía 3 veces (el número de veces que hemos indicado) y funciona correctamente.

Ejercicio 2:

Implementar un código, que comunica dos procesos usando comunicación punto a punto. Cada proceso debe enviar un array de datos al otro. Para ello, cada proceso declara dos arrays de floats, A y B, de tamaño fijo (10000). Todas las posiciones del array A se inicializan con el rango respectivos de cada proceso. Los arrays A y B serán los buffers para las operaciones de SEND y RECEIVE, respectivamente.

Se debe implementar un contador que se incremente cada vez que el primer proceso envía el mensaje. El programa termina cuando el contador llegue a un límite autoimpuesto. El programa debe guiarse por el siguiente esquema:



Comentar, antes de nada, que el programa no termina de funcionar al 100% bien. El mensaje se manda y se escribe correctamente en los buffers correspondientes. El problema es el contador, no hemos conseguido que funcione perfectamente, aunque funciona a medias.

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

float * initArray(float rank){
    float static aux[10000];
    for(int i=0; i < 10000; i++){
        aux[i] = rank;
    }
    return aux;
}

int main(int argc, char** argv){
    int rank;
    int counter = 1;
    int aux;
    MPI_Status status;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    do{
        if(rank == 0){
            float *A = initArray(rank);
            float B[10000];
            MPI_Send(A, 10000, MPI_FLOAT, 1, 200, MPI_COMM_WORLD);
            counter++;
            MPI_Recv(B, 10000, MPI_FLOAT, 1, 200, MPI_COMM_WORLD, &status);
            if(counter == 3){
                if(MPI_Get_count(&status, MPI_FLOAT, &aux) == MPI_SUCCESS){
                    printf("Sources: %d\n", status.MPI_SOURCE);
                }
                printf("Rank %d has %lf from array B\n\n", rank, B[0]);
            }
        }else{
            float *A = initArray(rank);
            float B[10000];
            MPI_Send(A, 10000, MPI_FLOAT, 0, 200, MPI_COMM_WORLD);
            MPI_Recv(B, 10000, MPI_FLOAT, 0, 200, MPI_COMM_WORLD, &status);
            if(MPI_Get_count(&status, MPI_FLOAT, &aux) == MPI_SUCCESS){
                if(counter == 2){
                    printf("Sources: %d\n", status.MPI_SOURCE);
                    printf("Rank %d has %lf from array B\n\n", rank, B[0]);
                }
                counter++;
            }
        }
    }while(counter != 3);
    MPI_Finalize();
    return 0;
}
```

En este ejercicio nos creamos una función llamada “InitArray()” con la que vamos a inicializar cada uno de los arrays “A”. Le entra una variable de tipo float llamada rank, que es el valor con el que se inicializa cada uno de los elementos del array. Devuelve un array de floats. Dentro de la función se declara un array auxiliar que hemos llamada aux, le decimos que es static para evitar problemas a la hora de devolver el array aux. Mediante un for recorremos cada posición del array y lo igualamos a la variable rank. Finalmente devolvemos el array.

Las variables rank y status tienen el mismo objetivo que en el ejercicio anterior. Además creamos una variable counter (el contador que se pide en el enunciado) y una variable aux para cuando hagamos MPI_Get_Count.

Como en el ejercicio anterior llamamos a las funciones MPI_Init() y MPI_Comm_rank().

Tras eso, y dentro de un bucle do-while que parará una vez en contador haya alcanzado el valor elegido, en nuestro caso el 3.

Dentro del bucle tenemos un if para diferenciar los dos procesos que tenemos. Tanto si rank=0 como si rank=1, declaramos los arrays de la misma manera. Los array "A" se inicializan mediante la función initArray() explicada previamente y los array "B" son declarados pero no son inicializados.

Si rank=0, mandamos el mensaje al proceso 1, aumentamos el contador y recibimos el mensaje que nos envía el proceso 1. Si el contador llega a 3, comprobamos que efectivamente el mensaje ha sido recibido correctamente para sacar por pantalla de donde procede el mensaje. Además sacamos el mensaje definitivo.

En el caso de que rank=1. Hacemos algo similar, pero el mensaje se envía y es recibido al proceso 0. Tras la función de MPI_Recv() comprobamos mediante la función MPI_Get_count() si el mensaje ha llegado correctamente.

Este es el único problema con el funcionamiento del programa, nunca llegaba a sacar el mensaje de rank=1 al llegar counter a 3, por lo que decidimos sacar por pantalla el mensaje una iteración antes. Salvo por eso, sacamos por pantalla el mensaje una vez recibido y counter=2. También aumentamos el contador una vez recibido el mensaje.

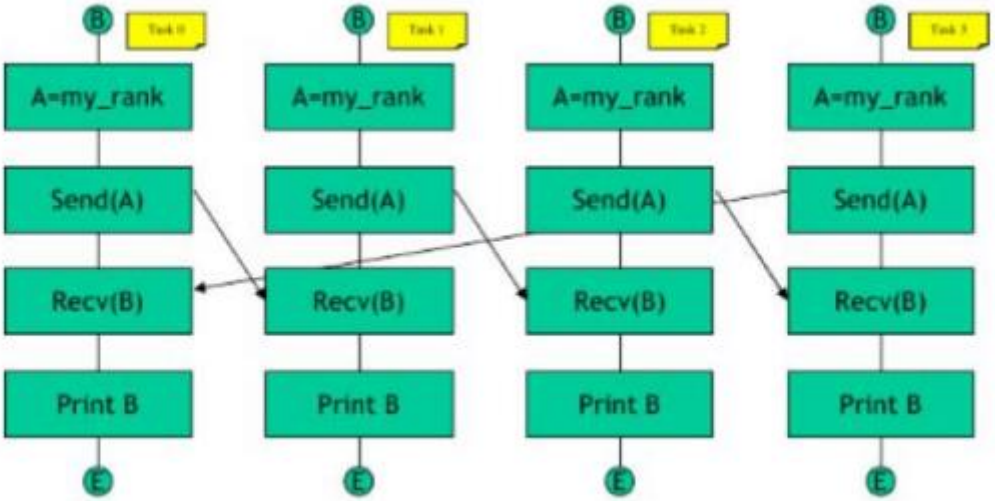
```
noone@noone-VirtualBox:~/Escritorio/ArquitecturaOrdenadores/practica2/Ejercicio2$ mpirun -np 2 ./ejercicio2
Sources: 0
Rank 1 has 0.000000 from array B

Sources: 1
Rank 0 has 1.000000 from array B
```

Como hemos comentado antes, el mensaje del proceso 1 sale antes pero podemos ver el origen de cada mensaje (Sources). El intercambio ha funcionado correctamente, el array B del proceso 0 tiene inicializada cada posición con el rango del proceso 1 y viceversa (El array B del proceso 1 tiene 10000 ceros).

Ejercicio 3:

Implementar un código que usando comunicación punto a punto, lleva a cabo una operación de send/recieve circular, como muestra el siguiente esquema:



Al igual que en el ejercicio anterior, cada proceso genera sus dos buffers A y B con un tamaño de 1000 posiciones inicializadas con sus respectivos rangos en A. Los arrays de tipo B se usan para recibir los mensajes que lleguen del nodo origen. Como se observa, cada proceso envía al que tiene a su derecha y recibe solo del que tiene a su izquierda. El programa termina cuando todos los procesos hayan enviado y recibido un mensaje. Asegurarse de que el programa funcione para un número de procesos arbitrario.

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

float * initArray(float rank){
    float static aux[1000];
    for(int i=0; i < 1000; i++){
        aux[i] = rank;
    }
    return aux;
}

int main(int argc, char** argv){
    int rank;
    int size;
    int numSElements;
    MPI_Status status;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    if(rank == 0){
        float *A = initArray(rank);
        float B[1000];
        MPI_Send(A, 1000, MPI_FLOAT, rank+1, 100, MPI_COMM_WORLD);
        MPI_Recv(B, 1000, MPI_FLOAT, size-1, 100, MPI_COMM_WORLD, &status);
        if(MPI_Get_count(&status, MPI_FLOAT, &numSElements) == MPI_SUCCESS){
            printf("Rank: %d. \n Recived %d elements from process %d. \n Value recived: %lf \n", rank, numSElements, status.MPI_SOURCE, B[100]);
        }
    }else{
        float *A = initArray(rank);
        float B[1000];
        MPI_Recv(B, 1000, MPI_FLOAT, rank-1, 100, MPI_COMM_WORLD, &status);
        if(MPI_Get_count(&status, MPI_FLOAT, &numSElements) == MPI_SUCCESS){
            printf("Rank: %d. \n Recived %d elements from process %d. \n Value recived: %lf \n", rank, numSElements, status.MPI_SOURCE, B[100]);
        }
        if(rank != size-1){
            MPI_Send(A, 1000, MPI_FLOAT, rank+1, 100, MPI_COMM_WORLD);
        }else{
            MPI_Send(A, 1000, MPI_FLOAT, 0, 100, MPI_COMM_WORLD);
        }
    }
    MPI_Finalize();
    return 0;
}
```

Para este ejercicio hemos hecho modificaciones del ejercicio 2.

La función “initArray()” ahora inicializa arrays de 1000 elementos en vez de 10000. La variable counter desaparece (ya no necesitamos ningún contador). Las variables rank y status se mantienen igual. Se añade una nueva variable llamada size, donde se almacena el número de procesos que tenemos y la variable aux del ejercicio 2 ahora se llama numSElements.

Al igual que en los ejercicios 1 y 2 llamamos a las funciones MPI_Init() y MPI_Comm_rank() pero esta vez también llamamos a la función MPI_Comm_size() con la que almacenamos en la variable size el número de procesos que tenemos.

El interior del if no cambia mucho. Se definen los arrays A y B en cada proceso al igual que en el ejercicio anterior. Esta vez tenemos un número genérico de procesos, por lo que en rank=0 mandamos al siguiente proceso el mensaje almacenado en A y recibimos el mensaje del size-1 y el mensaje recibido lo almacenamos en el array B. Recibimos de size-1 porque el primer proceso empieza en 0, si tenemos 2 procesos, el último proceso será el proceso 1.

Utilizamos la función MPI_Get_Count() igual que en el ejercicio anterior para comprobar si el mensaje ha llegado correctamente y además sacar información que vamos a imprimir como el tamaño del mensaje y de que proceso viene el mensaje.

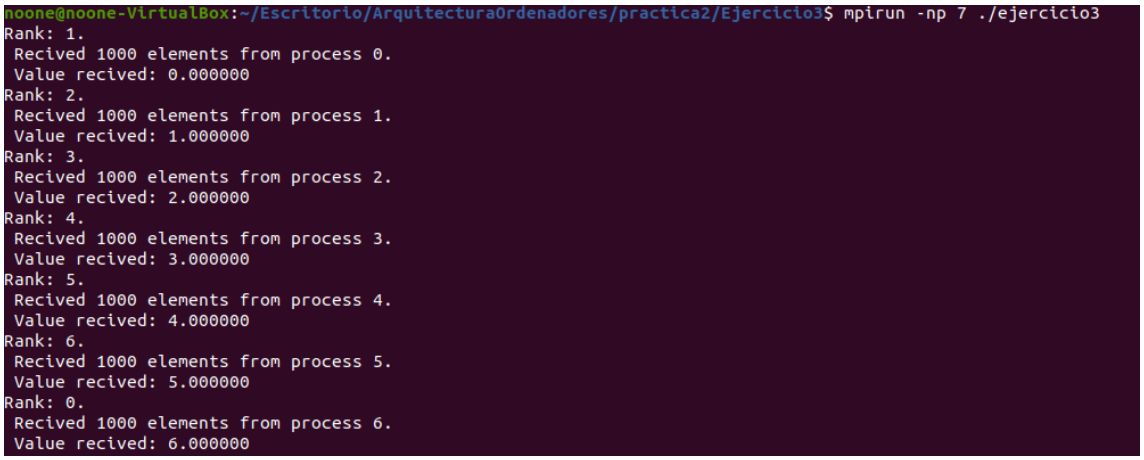
La parte del código que más cambia es cuando rank != 0, es decir, cualquier proceso que no sea el proceso 0. Cada proceso siempre recibe el mensaje de rank-1 (proceso anterior). Si el mensaje que recibe correctamente, hacemos lo mismo que en rank=0. Para evitar mandar un mensaje a un proceso que no existe, y para que el proceso 0 puede recibir su mensaje, hacemos un if en el que comprobamos si estamos o no en el último proceso.

Si no estamos en el último proceso, mandamos a rank+1 (siguiente proceso). Si el proceso en el que estamos es el último, mandamos el mensaje al proceso 0. De esta forma cerramos en anillo.

Resultados para 4 procesos:

El proceso 0 es el último en ser impreso por pantalla ya que también es el último en recibir su mensaje. El primero en recibir el mensaje es el proceso 1. Como podemos ver, cada proceso recibe el mensaje del proceso anterior y lo envía al siguiente. Los números de “Rank:” son los correspondientes al proceso en cuestión. Se muestra además el número de elementos recibimos y de que proceso lo hemos recibido. Por último recibimos el valor de uno de los elementos del array B de cada proceso.

Como demostración de que funciona para cualquier número de procesos este es el resultado que obtendríamos con 7 procesos:



Conclusión:

Estamos contentos con los resultados obtenidos, aún con el pequeño fallo en el contador del segundo ejercicio. Hemos adquirido conocimientos útiles, como el valor de retorno de las funciones de MPI (En esta práctica MPI_SUCCESS que significa que no hay errores) que pueden ser muy útiles para detectar errores inesperados.

Bibliografía:

MPI_Get_Count:

- https://www.mpich.org/static/docs/v3.2.x/www3/MPI_Get_count.html
- https://lsi2.ugr.es/jmantas/ppr/ayuda/mpi_ayuda.php?ayuda=MPI_Get_count

MPI_SUCCESS:

- https://www.rookiehpc.com/mpi/docs/mpi_success.php