PRÁCTICA 3

POR: JAIME TEJEDOR Y LUIS DÍAZ DEL RÍO

Introducción:

En esta práctica se nos pide realizar 3 ejercicios con el objetivo de familiarizarse con las operaciones de comunicación colectivas MPI_Bcast, MPI_Gather, MPI_Scatter, MPI_Reduce.

Ejercicio 1:

Implementar un programa donde el nodo 0 inicializa una variable con un valor arbitrario, después él mismo lo modifica (por ejemplo calculando el cuadrado de su valor) y finalmente lo envía al resto de nodos del comunicador.

- Hacerlo utilizando comunicación punto a punto.
- Hacerlo utilizando comunicación colectiva.

Comunicación punto a punto:

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

int main(int argc, char** argv){
    int rank;
    int size;
    int msg;
    MPI_Status status;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    if(rank == 0){
        printf("Add a number: ");
        scanf("%d", &msg);
        printf("Proceeding to make the squared of your number... \n");
        msg = msg * msg;
        printf("Sending final number %d to all processes... \n", msg);
        for(int i=1; i < size; i++){
            MPI_Send(&msg, 1, MPI_INT, i, 100, MPI_COMM_WORLD);
        }
        MPI_Barrier(MPI_COMM_WORLD);
    }else{
        int auxBufMsg;
        int numBufElemRev;
        MPI_Recv(&auxBufMsg, 1, MPI_INT, 0, 100, MPI_COMM_WORLD, &status);
        if(MPI_Get_count(&status, MPI_INT, &numBufElemRev) == MPI_SUCCESS){
            printf("Rank: %d \n Has recived value %d from rank %d \n", rank, auxBufMsg, status.MPI_SOURCE);
        }
        MPI_Barrier(MPI_COMM_WORLD);
    }

    MPI_Finalize();
    return 0;
}
```

Lo primero es declararnos las variables necesarias para nuestro programa. Declaramos rank, que servirá para saber el número de proceso, size, donde se almacenará el número de procesos, msg, que será nuestro buffer y status que nos servirá en la recepción del mensaje.

Seguidamente, y tras la función MPI_Init(), utilizamos las funciones MPI_Comm_rank() y MPI_Comm_size() para almacenar el número de proceso y el número de procesos en size y en rank respectivamente.

Como en prácticas anteriores, comprobamos mediante un if el rango.

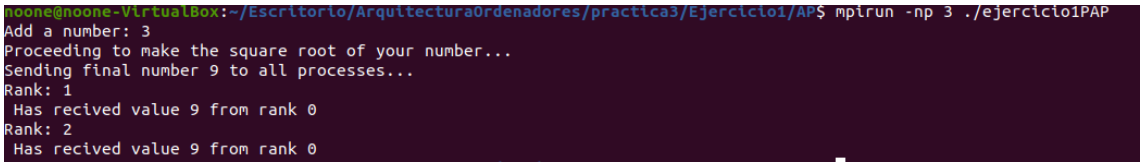
Si rank = 0 entonces pedimos al usuario que meta un número por pantalla. Además, elevamos ese número al cuadrado (msg*msg). Tras modificar el buffer msg, lo mandamos al resto de

nodos mediante un bucle for. Mandará el mensaje a todos los procesos empezando por el proceso 1.

Para este ejercicio hemos utilizado MPI_Barrier() para que el resultado se imprima en orden.

Si rank != 0, declaramos dos variables nuevas. La primera, auxBufMsg, es de tipo entero y almacenará el mensaje recibido del proceso 0. La segunda, numBufElemRev, la utilizamos únicamente para la función MPI_Get_Count(), en esta variable se almacena el número de elementos recibidos.

Tras la declaración de estas dos variables, cada proceso recibirá el mensaje enviado por el proceso 0 y comprobará que se ha recibido correctamente mediante el valor de retorno de MPI_Get_Count(). Si el mensaje se ha recibido correctamente, sacaremos por pantalla el resultado.



Estos son nuestros resultados para 3 procesos. Hay que destacar que si quisiéramos que el proceso 0 recibiera el número de si mismo, el bucle la variable “i” del bucle for debería empezar en “0” y habría que añadir la función MPI_Recv() tras el bucle donde hacemos MPI_Send().

Como se puede ver, los procesos 1 y 2 reciben correctamente el mensaje modificado. En este ejemplo hemos metido por pantalla el número 3, por lo que los procesos reciben el número 9.

Comentar que no nos hemos parado mucho a explicar el funcionamiento de las funciones MPI_Recv() y MPI_Send() ya que en anteriores prácticas se ha profundizado bastante en ellas.

Comunicación Colectiva:

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

int main(int argc, char** argv){
    int rank;
    int msg;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    if(rank == 0){
        printf("Add a number: ");
        scanf("%d", &msg);
        printf("Proceeding to make the squared of your number... \n");
        msg = msg * msg;
        printf("Sending final number %d to all processes... \n", msg);
    }

    MPI_Bcast(&msg, 1, MPI_INT, 0, MPI_COMM_WORLD);
    printf("Rank %d has recived value %d \n", rank, msg);

    MPI_Finalize();
    return 0;
}
```

Como se puede ver, el código se simplifica mucho. Para empezar solo tenemos dos variables, rank y msg donde se almacena el número de proceso y el mensaje respectivamente.

La mayor diferencia es que esta vez solo necesitamos comprobar si rank = 0 para definir el valor del mensaje.

Por cómo funciona MPI_Bcast(), no es necesario estar mandado y recibiendo mensajes mediante MPI_Recv() y MPI_Send(). Por cada proceso recibirá el mensaje del proceso especificado tras el datatype. En nuestro caso el proceso raíz es el proceso 0, por lo que el mensaje se transmitirá al resto de procesos desde ahí. El buffer es tanto de entrada como de salida.

Una vez el proceso recibe el mensaje, lo sacamos por pantalla.

```
noone@noone-VirtualBox:~/Escritorio/ArquitecturaOrdenadores/practica3/Ejercicio1/BC$ mpirun -np 3 ./ejercicio1Col
Add a number: 5
Proceeding to make the squared of your number...
Sending final number 25 to all processes...
Rank 0 has recived value 25
Rank 2 has recived value 25
Rank 1 has recived value 25
```

Esta vez el número añadido por consola es el 5. Como se puede ver en el resultado el mensaje se envía correctamente a todos los procesos. Los mensajes están desordenados ya que no hemos utilizado MPI_Barrier().

Ejercicio 2:

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).

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

int main(int argc, char** argv){
    int rank;
    int bufSource[9];
    int auxData[3];
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    if(rank == 0){
        for(int i=0; i < 9; i++){
            bufSource[i] = i;
        }
    }

    MPI_Scatter(&bufSource, 3, MPI_INT, &auxData, 3, MPI_INT, 0, MPI_COMM_WORLD);
    for(int i=0; i < 3; i++){
        auxData[i] = auxData[i] + rank;
    }
    MPI_Gather(&auxData, 3, MPI_INT, &bufSource, 3, MPI_INT, 0, MPI_COMM_WORLD);
    if(rank == 0){
        printf("Result: \n");
        for(int i=0; i < 9; i++){
            if(i+1 != 9){
                printf("%d-", bufSource[i]);
            }else{
                printf("%d", bufSource[i]);
            }
        }
        printf("\n");
    }

    MPI_Finalize();
    return 0;
}
```

Como variables tenemos rank, donde se almacena el número de proceso en el que estamos, bufSource, que es un array de nueve números enteros que utilizaremos como array inicial y que más tarde dividiremos entre cada uno de los procesos y auxData, un array de tres números enteros donde guardaremos la porción de información correspondiente a cada proceso.

Tras las funciones iniciales de MPI, comprobamos si rank = 0 para inicializar el array bufSource tal y como se pide en el enunciado del ejercicio. El array se inicializa mediante un for.

Mediante MPI_Scatter() dividimos el array bufSource. Hemos decidido utilizar tres procesos, por lo que dividiremos el array en porciones de tres elementos. Las porciones se almacenan temporalmente en el array auxData.

Tras dividir el mensaje, y mediante un bucle for, modificamos cada elemento del array tal y como se pide el enunciado (cada index + rank).

Con la función MPI_Gather() escribimos en el array bufSource el resultado almacenado en auxData.

Finalmente volvemos a comprobar en rank en el que estamos mediante un if. Una vez se pasa por todos los procesos imprimimos el resultado almacenado el bufSource.

La impresión se hace mediante un for, el if en el interior es meramente estético.

```
noone@noone-VirtualBox:~/Escritorio/ArquitecturaOrdenadores/practica3/Ejercicio2$ mpirun -np 3 ./ejercicio2
Result:
0-1-2-4-5-6-8-9-10
```

El resultado es el esperado, si dividimos el array final en porciones de tres y hacemos los cálculos podemos comprobarlo.

Porción 1:

Rank = 0

Index = [0,1,2]

Elem0 = 0 + 0 = 0

Elem1 = 0 + 1 = 1

Elem2 = 0 + 2 = 2

0-1-2

Porción 2:

Rank = 1

Index = [3, 4, 5]

Elem3 = 1 + 3 = 4

Elem4 = 1 + 4 = 5

Elem5 = 1 + 5 = 6

4-5-6

Porción 3:

Rank = 2

Index = [6,7,8]

Elem6 = 2 + 6 = 8

Elem7 = 2 + 7 = 9

Elem8 = 2 + 8 = 10

8-9-10

Si juntamos todas las tramas: **0-1-2-4-5-6-8-9-10**

El resultado es correcto.

Ejercicio 3:

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.

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

void printArray(int * toPrint, int arrSize){
    for(int i=0; i < arrSize; i++){
        if(i+1 != arrSize){
            printf("%d-", toPrint[i]);
        }else{
            printf("%d", toPrint[i]);
        }
    }
    printf("\n");
}

int main(int argc, char** argv){
    int rank;
    int finalBufSum[3];
    int finalBufMul[3];
    int auxData[3];
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    for(int i=0; i < 3; i++){
        auxData[i] = rank + 1;
    }

    MPI_Reduce(&auxData, &finalBufSum, 3, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
    MPI_Reduce(&auxData, &finalBufMul, 3, MPI_INT, MPI_PROD, 0, MPI_COMM_WORLD);

    if(rank == 0){
        printf("Sum result: ");
        printArray(finalBufSum, 3);
        printf("Product result: ");
        printArray(finalBufMul, 3);
    }

    MPI_Finalize();

    return 0;
}
```

Para este ejercicio declaramos como siempre la variable rank, que tiene la misma función que en el resto de ejercicios, dos arrays, finalBufSum y finalBufMul, que almacenan tres números enteros y donde se guardarán los resultados de MPI_Reduce(), la suma en el primero y el producto en el segundo. También declaramos un array auxiliar llamado auxData que también almacena tres números enteros y que inicializaremos en cada proceso.

Tras las dos funciones iniciales, utilizamos un bucle for para inicializar cada elemento del array con rank + 1 tal y como se pide en el enunciado.

Una vez tenemos inicializado el array auxData utilizamos MPI_Reduce() dos veces. La primera se encarga de la suma (lo especificamos con MPI_SUM), guardamos el resultado dentro de finalBufSum. La segunda se encarga de la multiplicación, funciona igual que con la suma, salvo

que esta vez le especificamos que haga el producto con MPI_PROD y guardamos el resultado en el array finalBufMul.

Finalmente comprobamos si rank = 0 para sacar por pantalla los resultados almacenados. Esta vez hemos utilizado una función, llamada printArray, que recibe un puntero a un entero y un tamaño. La razón por la que añadimos el tamaño es que hemos hecho distintas pruebas y era más sencillo de esta forma.

El interior de la función es un for idéntico al utilizado en el ejercicio anterior para imprimir.

```
noone@noone-VirtualBox:~/Escritorio/ArquitecturaOrdenadores/practica3/Ejercicio3$ mpirun -np 3 ./ejercicio3
Sum result: 6-6-6
Product result: 6-6-6
```

Hemos utilizado tres procesos, los resultados al principio resultan extraños pero son correctos.

Demostración:

Elem0 = [1,1,1]

Elem1 = [2,2,2]

Elem2 = [3,3,3]

Suma:

1 + 2 + 3 = 6.

Multiplicación:

1 * 2 * 3 = 6.

Esta operación se repite para cada elemento por lo que es lógico, al ser idénticos, que se repitan.

Conclusión:

Estamos contentos con los resultados obtenidos, no solo hemos aprendido el funcionamiento de estas nuevas operaciones sino que hemos ampliado nuestros conocimientos sobre el funcionamiento de MPI.

Bibliografía:

MPI_Bcast:

- https://lsi2.ugr.es/jmantas/ppr/ayuda/mpi_ayuda.php?ayuda=MPI_Bcast
- https://www.mpich.org/static/docs/latest/www3/MPI_Bcast.html

MPI_Scatter:

- https://lsi2.ugr.es/jmantas/ppr/ayuda/mpi_ayuda.php?ayuda=MPI_Scatter
- https://www.mpich.org/static/docs/latest/www3/MPI_Scatter.html

MPI_Gather:

- https://lsi2.ugr.es/jmantas/ppr/ayuda/mpi_ayuda.php?ayuda=MPI_Gather
- https://www.mpich.org/static/docs/latest/www3/MPI_Gather.html

MPI_Allgather:

- https://www.mpich.org/static/docs/latest/www3/MPI_Allgather.html

MPI_Barrier: (Respuesta elegida. Utilizado ejercicio 1)

- https://stackoverflow.com/questions/29019744/output-isnt-ordered-parallel-programing-with-mpi