Práctica 5: Función de reparto y reducción.

Karina Flores García y Gabriel Alejandro Herrera Gandarela

***Abstract*— In this practice, will see the functionality of communication collective in MPI using the functions MPI_Scatter and MPI_Reduce and how works each process in parallel with this functions.**

I. OBJETIVOS.

- Estudiar opciones más potentes dentro de las comunicaciones colectivas.

II. INTRODUCCIÓN.

A. *Comunicaciones colectivas con Scatter*

MPI proporciona ciertas funciones dentro de su biblioteca para implementar operaciones de reparto y reducción.
La función de reparto es:

```
int MPI_Scatter(void *bufsend, int count, MPI_Datatype dtype, void *bufrecv
    , int count, MPI_Datatype dtype, int source, MPI_Comm comm)
```

En este caso, parte de los datos contenidos en el buffer del proceso emisor (*bufsend) son copiados en los buffers de los procesos receptores (*bufrecv). El primer parámetro count indica cuántos datos van a cada proceso. Existe una pequeña diferencia entre las funciones MPI_Bcast y MPI_Scatter pero importante. Mientras que la función MPI_Bcast envía el mismo conjunto de datos a todos los procesos, la función MPI_Scatter envía únicamente trozos de dicho conjunto.
La función de reducción es:

```
int MPI_Reduce(void *bufsend, void *bufrecv, int count, MPI_Datatype dtype,
    MPI_op operation, int dest, MPI_Comm comm)
```

De manera parecida a la función MPI_Gather, esta función permite recolectar la información generada por los procesos esclavos. Así, ante esta función, cada proceso envía count datos almacenados en el buffer *bufsend a aquel proceso cuyo rango coincide con el parámetro dest, el cual no los almacena directamente en su buffer *bufrecv, sino que lo que ahí coloca es el resultado de realizar sobre estos datos la operación indicada en operación. Este parámetro es del tipo MPI_op, es decir, una operación MPI. Las operaciones MPI más habituales aparecen en la Tabla.

Operación	Descripción
MPI_MAX	Máximo
MPI_MIN	Mínimo
MPI_SUM	Suma
MPI_PROD	Producto
MPI_LAND	Y lógico
MPI_LOR	O lógico
MPI_LXOR	XOR lógico
MPI_BXOR	XOR a nivel de bits
MPI_MINLOC	determina el rango del proceso que contiene el valor menor.
MPI_MAXLOC	determina e rango del proceso que contiene el valor mayor.

Figura 3: Operaciones MPI

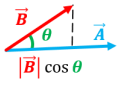
III. MODELO.

A. *Matemático*

Sean A = (Ax, Ay, Az) y B = (Bx, By, Bz); el producto escalar (denominado también producto punto o producto interno) de dos vectores se define como:

$$\mathbf{A} \cdot \mathbf{B} = \mathbf{AxBx + AyBy + AzBz}$$

El producto escalar siempre es un número real, es conmutativo y distributivo, de él surge el teorema del coseno. Además, cuando el producto escalar de dos vectores A y B es nulo (cero) significa que son perpendiculares entre sí.



Producto escalar de dos vectores

IV. ANÁLISIS DEL PROBLEMA.

Conociendo el funcionamiento, o al menos teniendo la idea, de cómo se ejecutan las funciones *MPI_Scatter* y *MPI_Reduce*, podemos analizar el orden en que queramos que se ejecuten. Para este caso, el *pid = 0*, debe de reservar memoria en los arreglos, llenarlos y después ejecutar la función *MPI_Scatter* para que distribuya cada elemento del arreglo a los distintos procesadores, además, será necesario que el número de procesadores será en número de elementos en el arreglo. Para finalizar, con la función *MPI_Scatter* podemos mandar los datos al *pid = 0* para qe imprima el resultado.

V. IMPLEMENTACIÓN

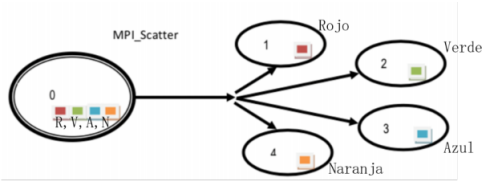
A. *Desarrollo*

Como bien sabemos, al inicio de todo programa de MPI, tiene la parte secuencial la cual iniciara el entorno, creara

variables, dara a cada procesador un pid, etc. Teniendo lo anterior, el procesador "root", reservará memoria para cada vector de acuerdo al número de procesadores en ejecución. Seguidamente, con la función *MPI_Scatter()*, mandará cada parte del vector a cada procesador, y estos se encargaran de sumar cada elemento que le corresponde. Cuando tengan el resultado, debemos colocar una barrera para que, todos los procesadores terminen su tarea. Si todos tienen su tarea terminada, la función *MPI_Scatter()* nos ayudará para mandarla al procesador padre, este se encargará sumar cada los resultados, mostrar en pantalla el resultado final e imprimir el tiempo total de ejecución.

B. *Diagrama de patrón arquitectónico*

Función Scatter:



Función Reduce:

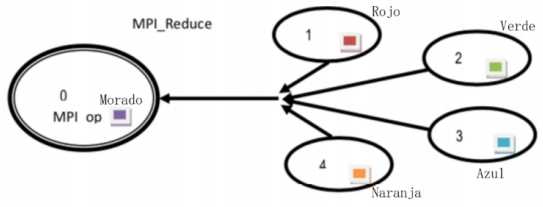
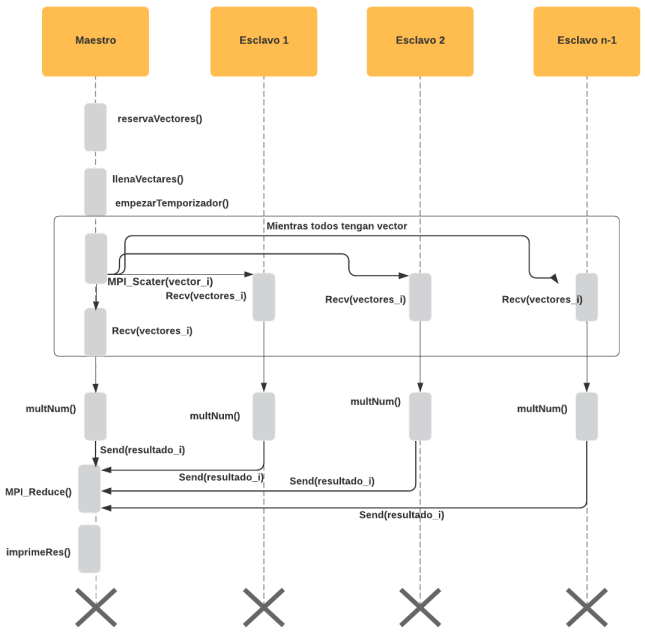
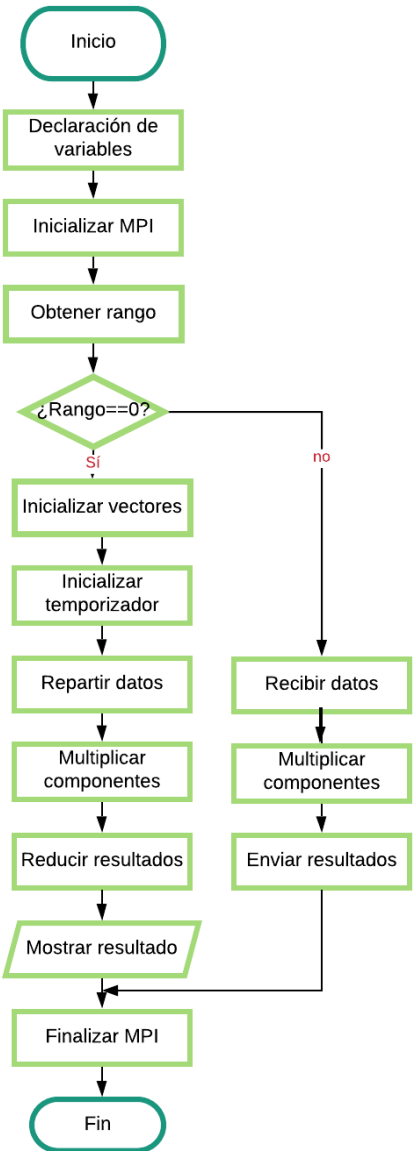


Figura 2: Operación de reducción realizada con la función MPI_Reduce.

C. *Diagrama UML de secuencia*



D. *Flujo del programa*



VI. CÓDIGO FUENTE

```
if(pid==0){
        //Llenando vectores A y B
        tiempo_inicio = MPI_Wtime();
        for(i=0;i<val;i++){
          vectorA[i] = rand() % 9;
          vectorB[i] = rand() % 9;
        }

        //Imprimiendo vectorA
        for(i=0;i<val;i++){
          printf("%d|",vectorA[i]);
        }
        printf("\n");


        //Imprimiendo vectorB
        for(i=0;i<val;i++){
          printf("%d|",vectorB[i]);
        }
        printf("\n");

        //Producto escalar calculado secuencialmente
```

```
      for(i=0;i<val;i++){
        productoSec += vectorA[i] * vectorB[i];
      }

      //Impresion del producto calculado secuencialmente
      printf("----------------------------------------");
      printf("\nProducto calculado secuencialmente: %d", productoSec);
    }
```

```
 //Se coloca una barrera para que todos se sincronizen
    MPI_Barrier(MPI_COMM_WORLD);
    //Se hace el envio del vectorA y vectorB
    MPI_Scatter(&vectorA,1,MPI_INT,&datoA,1,MPI_INT,0,MPI_COMM_WORLD);
    MPI_Scatter(&vectorB,1,MPI_INT,&datoB,1,MPI_INT,0,MPI_COMM_WORLD);
```

```
//Se hace el calculo del producto escalar

    for(i=0;i<val;i++){
      printf("datoA: %d\n",datoA );
      printf("datoB: %d\n",datoB );
      productoParal = datoA * datoB;
      printf("Multiplicacion: %d\n",productoParal );
    }
    //Se espera a que terminen
    MPI_Barrier(MPI_COMM_WORLD);
    //Se envian los datos del resultdo de la suma
    MPI_Reduce(&productoParal,&total,1,MPI_INT,MPI_SUM,0,MPI_COMM_WORLD);
```

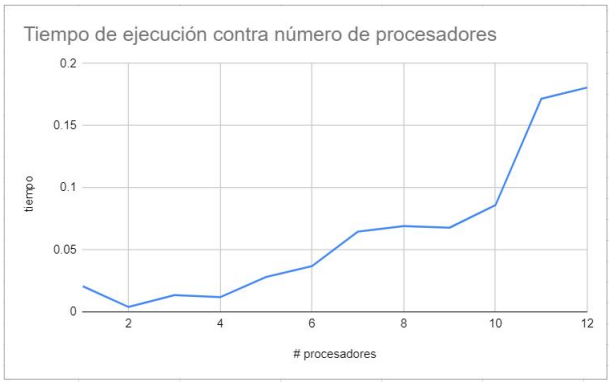
```
if(pid==0){
      printf("\n------------------------------------\n");
      printf("Recibiendo datos en pid=0\n");
      printf("Producto calculado paralelamente: %d", total);
      //Calculo del tiempo total
      tiempo_fin = MPI_Wtime();
      tiempo_total = tiempo_fin - tiempo_inicio;
      printf("\nTiempo total: %f", tiempo_total);

    }
```

VII. COMPARACIÓN

TABLE I
COMPARACIÓN ENTRE NÚMERO DE PROCESADORES Y TIEMPO DE EJECUCIÓN

Número de procesadores	Tiempo de ejecución (s)
1	0.0207
2	0.0039
3	0.0134
4	0.0118
5	0.0281
6	0.0368
7	0.0646
8	0.0691
9	0.0678
10	0.0859
11	0.1716
12	0.1806



La comparación nos da como conclusión que entre mayor numero de procesadores, mayor es el tiempo de procesamiento, esto debido a que la cantidad de procesadores es proporcional al aumento de elementos en los vectores para el cálculo del producto vectorial.

```
alejandro@alejandro-VirtualBox:~/Documents/SistemasDistribuidos/practica5$ mpirun -np 2 ./arreglos
1|0|
7|7|
----------------------------------------
datoA: 0
datoB: 7
Producto calculado secuencialmente: 1253032362datoA: 1
datoB: 7
Multiplicacion: 7
datoA: 1
datoB: 7
Multiplicacion: 7
Multiplicacion: 0
datoA: 0
datoB: 7
Multiplicacion: 0

------------------------------------
Recibiendo datos en pid=0
Producto calculado paralelamente: 7
Tiempo total: 0.007994alejandro@alejandro-VirtualBox:~/Documents/practica5$ stribuidos/pr
```

Arreglo de 2 elementos

```
alejandro@alejandro-VirtualBox:~/Documents/SistemasDistribuidos/practica5$ mpicc arreglosMPI.c -o arreglos
alejandro@alejandro-VirtualBox:~/Documents/SistemasDistribuidos/practica5$ mpirun -np 3 ./arreglos
1|0|5|
7|7|7|
----------------------------------------
Multiplicacion: 0
Multiplicacion: 0
Multiplicacion: 0
Multiplicacion: 35
Multiplicacion: 35
Multiplicacion: 35
Producto calculado secuencialmente: -828632416Multiplicacion: 7
Multiplicacion: 7
Multiplicacion: 7

------------------------------------
Recibiendo datos en pid=0
Producto calculado paralelamente: 42
Tiempo total: 0.026532alejandro@alejandro-VirtualBox:~/Documents/practica5$ stribuidos/pr
```

Arreglo de 3 elementos

```
alejandro@alejandro-VirtualBox:~/Documents/SistemasDistribuidos/p
ractica5$ mpicc arreglosMPI.c -o arreglos
alejandro@alejandro-VirtualBox:~/Documents/SistemasDistribuidos/p
ractica5$ mpirun -np 12 ./arreglos
1|0|5|1|6|5|5|5|6|7|8|6|
7|7|7|3|1|4|7|4|0|1|8|6|
----------------------------------------
Producto calculado secuencialmente: 2100755857
----------------------------------------
Recibiendo datos en pid=0
Producto calculado paralelamente: 233
Tiempo total: 0.234097alejandro@alejandro-VirtualBox:~/Documents/
ractica5$ stribuidos/pr
```

Arreglo de 12 elementos

Nota: Los tiempos mostrados en las capturas de pantalla pueden variar respecto a la tabla de comparación.

VIII. CONCLUSIONES

Se cumplieron los objetivos gracias a la correcta implementacion que nos ofrece MPI, pudimos analizar, entender y ejecutar las herramientas de comunicación colectiva. La parte complicada fue la coordinación del programa ya que, en ocasiones, algunos procesadores al terminar más rápido que otros, mandaban a llamar la siguiente tarea cuando era necesario que todos terminaran su tarea para ahora sí continuar con el flujo del programa, esto se resolvió con barreras que se ejecutaban antes de cada instrucción y esperaba a que todos terminaran. También se pudo observar que a medida que incrementabamos en número de procesadores de ejecución, tardaba más nuestro programa, si bien, en ocasiones si ejecutábamos el programa con el mismo número de procesadores varias veces, el tiempo variaba, sin embargo, no era tan notorio.

Una de las partes más importantes de esta práctica, fue que notamos y aplicamos la diferencia entre MPI_Broadcast y MPI_Scatter, en esta ocasión a cada procesador desde un principio le asignamos solamente un elemento de cada vector para que realizara la multiplicación de los elementos y posteriormente con la función MPI_SUM recolectamos cada elemento multiplicado por cada uno de los procesadores y los sumamos, con ello se logró el calculo del producto vectorial utilizando MPI de una forma sencilla y optima.

Es de notarse que MPI nos proporciona operaciones muy útiles para el manejo de datos colectivamente.

REFERENCES

[1] Ayala,J.A.(29 marzo 2020) Práctica 5:Función de reparto y reducción. Sistemas Distribuidos, 2.

[2] https://lsi.ugr.es/jmantas/ppr/tutoriales/tutorial_mpi.php?tuto=04_producto_escalar

[3] https://miprofe.com/producto-escalar/