Práctica 4: Comunicaciones Colectivas.

Karina Flores García y Gabriel Alejandro Herrera Gandarela

***Abstract*— In this practice we will see how collective communication between processors works, for this, we will use the MPI libraries in the language C.**

I. OBJETIVOS.

- Extender las posibilidades de comunicación entre procesos a modalidad colectiva como método de simplificación.
- Estudiar las posibles aplicaciones de las comunicaciones colectivas.

II. INTRODUCCIÓN.

A. *Comunicaciones colectivas*

En la mayoría de las aplicaciones prácticas del paso de mensajes, el escenario consiste en un proceso maestro que reparte datos a muchos procesos esclavos y que recopila los resultados que éstos le proporcionan. Es posible manejar el tráfico de mensajes que esto genera empleando las funciones de envío y recepción que se emplearon en la práctica anterior, pero si se dispone de funciones más potentes, enseguida se descartará esta posibilidad.
Las funciones de comunicación colectiva permiten que un proceso envíe datos a varios procesos y que recoja resultados de otros tantos de una sola vez. Esto simplifica la programación de la inmensa mayoría de las aplicaciones.

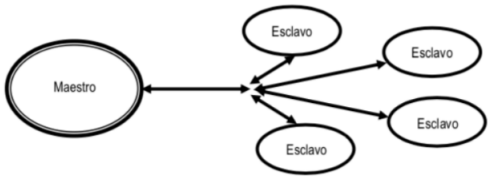


Imagen 1: Estructura Maestro-Esclavo

B. *Implementación en MPI*

El concepto de broadcast se asocia tradicionalmente a envíos programados desde un emisor a todos los posibles receptores. Se puede traducir a castellano como difusión. MPI proporciona una función para poder realizar este tipo de envíos en una única llamada. Se trata de la función:

```
int MPI_Bcast(void *buf, int count, MPI_Datatype type, int source, MPI_Comm
    comm)
```

Es un envío con fuente pero sin destino. Lo cual significa que esta función se emplea por igual en el código de todos los procesos, pero solamente aquel cuyo rango (rank) coincida con source va a enviar, el resto entiende que deben recoger lo que aquel envíe. En la imagen 2 tenemos el esquema de trabajo de la función **MPI_Bcast** donde se puede apreciar que esta función copia el contenido del buffer ***buf** del proceso source y lo copia en el resto de procesos. El concepto contrario a la difusión lo podemos denominar recolección. Consiste en recopilar la información generada en forma de resultados por los procesos esclavos. MPI proporciona la posibilidad de concentrar todos los resultados en el proceso maestro o en todos los procesos. Las funciones que realizan estas operaciones son respectivamente:

```
int MPI_Gather(void *bufsource, int count, MPI_Datatype dtype, void *
    bufdest, int count, MPI_Datatype dtype, int dest, MPI_Comm comm)

int MPI_Allgather(void  *bufsource, int count, MPI_Datatype dtype, void *
    bufdest, int count, MPI_Datatype dtype, MPI_Comm comm)
```

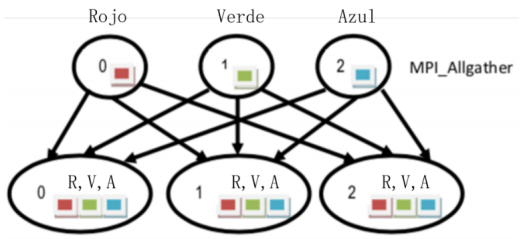


Imagen 4: Esquema de trabajo de la función MPI_Allgather

III. ANÁLISIS DEL PROBLEMA.

Cuando se habla de computo paralelo, es muy necesario tener cuidado la forma en que se van a repartir las tareas cada hilo o cada procesador (para este caso). Ésta práctica no es la excepción, hay que tener una correcta paralisación a la hora de mandar los datos, ya que, habrá procesadores que tomen la matriz para realizar su tarea cuando ésta aún no la llena el proceso maestro. Otra cuestión a tomar en cuenta es que el proceso raíz, espera a cada procesador para después imprimir el resultado.

IV. IMPLEMENTACIÓN

A. *Desarrollo*

Al inicio del programa, se inicilizarán las variables que se van a utilizar, entre ellas el rango, el número de procesadores que estarán activados y las matrices. Seguidamente, de forma secuencial inicializaremos el entorno de MPI, teniendo esto, podemos comprarar entre el procesador maestro y el esclavo. Para el caso del maestro, llenaremos las matrices con valores aleatorios, levantar el temporizador y distribuir las matrices a cada uno de los escalvos con la función **MPI_Bcast()** que repartirá las matrices a cada uno de los procesadores. Cada procesador, sumará la columna de cada una de las matrices que le corresponde para después regresar un arreglo como resultado, para ello se empleará la función MPI_Gather(). Por último, el procesador maestro, deberá de obtener el valor de

la suma de cada esclavo e imprimir en pantalla la matriz resultado.

B. Diagrama de patrón arquitectónico

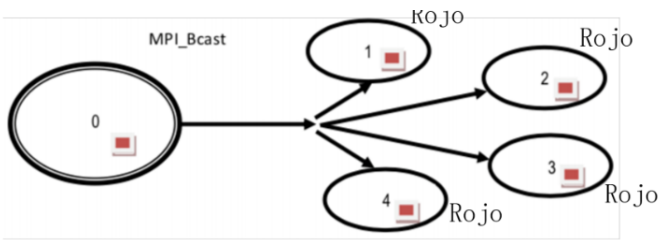


Imagen 2: Esquema de trabajo de la función MPI_Bcast

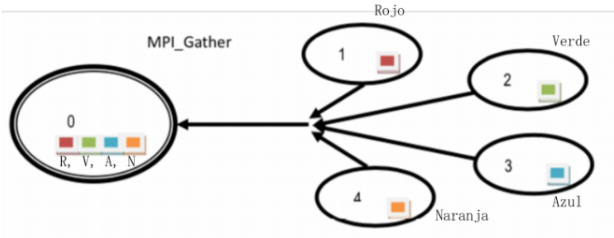
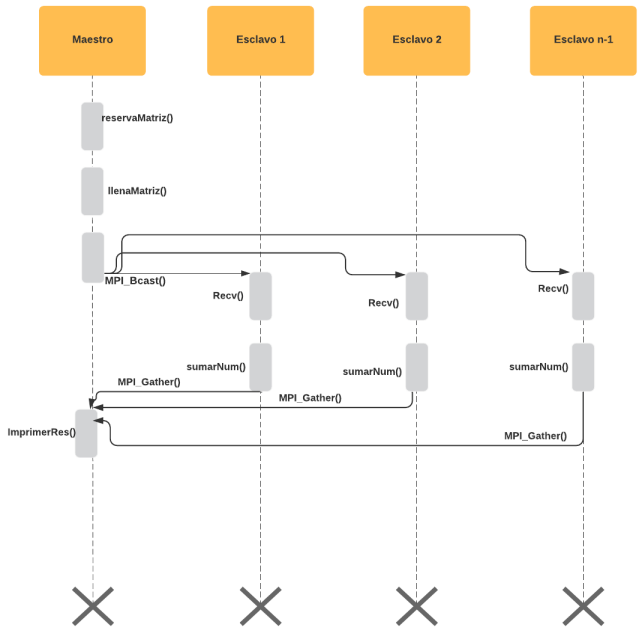
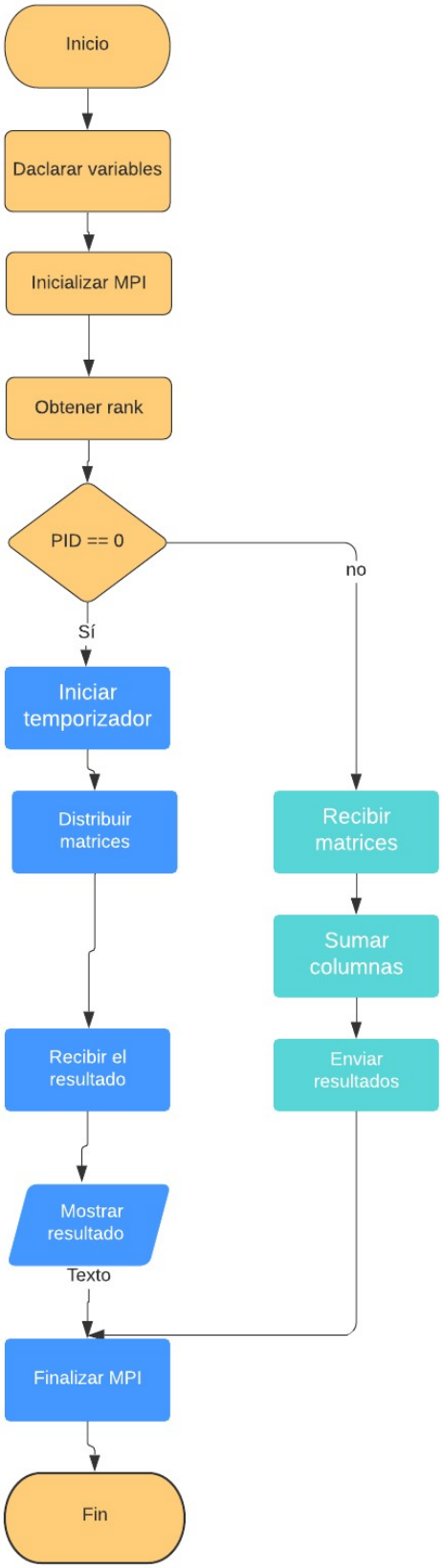


Imagen 3: Esquema de trabajo de la función MPI_Gather

C. Diagrama UML de secuencia



D. Flujo del programa



V. CÓDIGO FUENTE

```
//El proceso 0 genera el contenido de las matrices y lo distribuye a los demas
if(pid==0){
//Calculo del tiempo en que inicio el programa
  tiempo_inicio = MPI_Wtime();
//Llenando matrices A y B
  for(i=0;i<j;++i){
    for(j=0;j<n;++j)
    {
        matrizA[i][j] = rand() % 6;
        matrizB[i][j] = rand() % 6;
     }
  }
  //Imprimiendo matriz A
  printf("Matriz A en pid=0\n");
  for(int i = 0; i<size; i++){
    for(int j= 0;j<size;j++){
      printf("%d|", matrizA[i][j]);

    }
    printf("\n");
  }

  //Imprimiendo matriz B
  printf("Matriz B en pid=0\n");
  for(int i = 0; i<size; i++){
    for(int j= 0;j<size;j++){
      printf("%d|", matrizB[i][j]);
    }
    printf("\n");
  }
  //Calcula la suma de las matrices de forma secuencial
for(int i=0;i<n;i++){
    for(int j=0;j<n;j++){
      matrizC[i][j] = matrizA[i][j]+matrizB[i][j];
    }
  }
  //Imprime la matriz que calculo
  printf("Matriz C en pid=0\n");
  for(int i = 0; i<size; i++){
    for(int j= 0;j<size;j++){
      printf("%d|", matrizC[i][j]);
    }
    printf("\n");
  }
}

//Se coloca una barrera para que todos se sincronizen
MPI_Barrier(MPI_COMM_WORLD);
//Se hace el envio de la matrizA y matrizB a todos los procesos
MPI_Bcast(&matrizA,val,MPI_INT,0,MPI_COMM_WORLD);
MPI_Bcast(&matrizB,val,MPI_INT,0,MPI_COMM_WORLD);
//Se hace la suma de matrices
for(i=0;i<j;++i){
  for(j=0;j<n;++j)
  {
      arreglo1[k]= matrizA[i][j]+matrizB[i][j];
      k++;
  }
}
//Se envian los datos del resultdo de la suma
MPI_Gather(arreglo1,val,MPI_INT,arreglo2,val,MPI_INT,0,MPI_COMM_WORLD);

if(pid==0){
  printf("\n
---------------------------------------\n");
  printf("Recibiendo datos despues del gather\n");
  printf("Matriz recibida de los procesos\n");
  //El arreglo que se envia se le da formato de matriz
  for(int l=0; l<val;l++){
    if((l)%n==0){
      printf("\n");
    }
    printf("%d|",arreglo2[l]);
  }
  printf("\n");
  //Se calcula el tiempo total de la ejecucion
  tiempo_fin = MPI_Wtime();
  tiempo_total = tiempo_fin - tiempo_inicio;
  printf("Tiempo total de ejecucion: %f\n", tiempo_total);
}

MPI_Finalize();
return 0;
}
```

VI. COMPARACIÓN

1) **Las comunicaciones colectivas facilitan la programación y simplifican el código. ¿Se puede pensar que acortan el tiempo de ejecución de los programas?**
   Si aunque esto se visualiza más en programas de cálculos muy grandes, en matrices de tamaños pequeños no se nota una gran diferencia con un programa secuencial pero sin duda las comunicación colectiva y distribución de tareas en paralelo acortan los tiempo de ejecución. Aun que en algunas ocasiones utilizan más recursos y tardan el mismo tiempo que una comunicación secuencial por lo que se debe valorar cuando hacer uso de las comunicaciones colectivas.
2) **Explicar qué refleja la medida de tiempo realizada.**
   El tiempo que tardo en enviarse la información a todos los procesos, hacer el cálculo secuencialmente, hacer el cálculo de forma paralela y regresar el resultado al proceso con pid=0.
3) **Plantear otras posibilidades de medida de tiempos de ejecución que permitan distinguir los tiempos invertidos en comunicación entre procesos y los tiempos dedicados al cálculo.**

```
Matriz A p
1|3|
5|4|
Matriz B p
4|1|
1|0|
Lo recibe el mero mero

5|4|
6|4|
Tiempo total de ejecución: 0.000092
```

Ejecución del programa con dos procesadores

```
Matriz A p
1|3|5|
4|3|2|
2|5|0|
Matriz B p
4|1|1|
0|1|1|
1|4|0|
Lo recibe el mero mero

5|4|6|
4|4|3|
3|9|0|
Tiempo total de ejecución: 0.000296
```

Ejecución del programa con tres procesadores

```
Matriz A p
1|3|5|4|
3|2|2|5|
0|4|5|3|
2|2|1|5|
Matriz B p
4|1|1|0|
1|1|1|4|
0|4|2|3|
2|1|1|0|
Lo recibe el mero mero

5|4|6|4|
4|3|3|9|
0|8|7|6|
4|3|2|5|
Tiempo total de ejecución: 0.000247
```

Ejecución del programa con cuatro procesadores

VII. CONCLUSIONES

Se cumplieron los objetivos de la prácticas gracias a las funciones que nos proporciona MPI. Éstas son de gran utilidad, sin embargo, se observó que no son tan fácil de manejarlas, ya que, al estar trabajando en paralelo con los procesadores, en nuestro caso, de la máquina virtual, hay unos que se adelantaban a realizar su parte del código cuando era necesario que se esperaran a que terminara el proceso maestro para que pudiera repartir la matriz completa a los esclavos. Para mejorar esto, pusimos barreras las cuales se encargaban de poner en estado "perezoso" a los demás procesadores, esto pasaba hasta que el proceso root, terminaba parte de su tarea en el código.

REFERENCES

[1] https://computing.llnl.gov/tutorials/mpi/
[2] https://www.cs.buap.mx/ mtovar/doc/PCPO15/ComunicacionPP_2.pdf