Universidad Nacional de San Antonio Abad del Cusco

Departamento Académico de Informática

ALGORITMOS PARALELOS Y DISTRIBUIDOS

Práctica Nº 6

COMUNICACIÓN COLECTIVA

1. OBJETIVO.

- Conocer las operaciones de comunicación colectiva en MPI.
- Resolver problemas utilizando comunicación colectiva en MPI.

2. CONCEPTOS TEORICOS.

2.1. INTRODUCCION.

Las operaciones colectivas son llamadas MPI diseñadas para comunicar los procesos señalados por un comunicador en una sola operación o para realizar una sincronización entre ellos. Estos se utilizan a menudo para calcular uno o más valores basados en datos aportados por otros procesos o para distribuir o recopilar datos de todos los demás procesos.
Algunas de las comunicaciones colectivas más útiles se presentan a continuación.

2.1.1. Barrera.

```
int MPI_Barrier(MPI_Comm comm)
```

Establece una barrera. Todos los procesos esperan a llegar a la barrera, para continuar la ejecución una vez han llegado. Se utiliza un comunicador que establece el grupo de procesos que se está sincronizando. Todas las funciones de comunicaciones colectivas retornan un código de error.

2.1.2. Broadast.

```
int MPI_Bcast(void *buffer, int count,
     MPI Datatype datatype, int root, MPI comm comm)
```

Lleva a cabo una operación de broadcast (comunicación uno a todos), en la que se envían count datos del tipo datatype desde el proceso raíz (root) al resto de los procesos en el comunicador. Todos los procesos que intervienen deben llamar a la función indicando el proceso que actúa como raíz. En la raíz, los datos que se envían se toman de la zona apuntada por buffer, y los que reciben almacenan en la memoria reservada en buffer. La figura 1 ilustra el funcionamiento de esta operación.

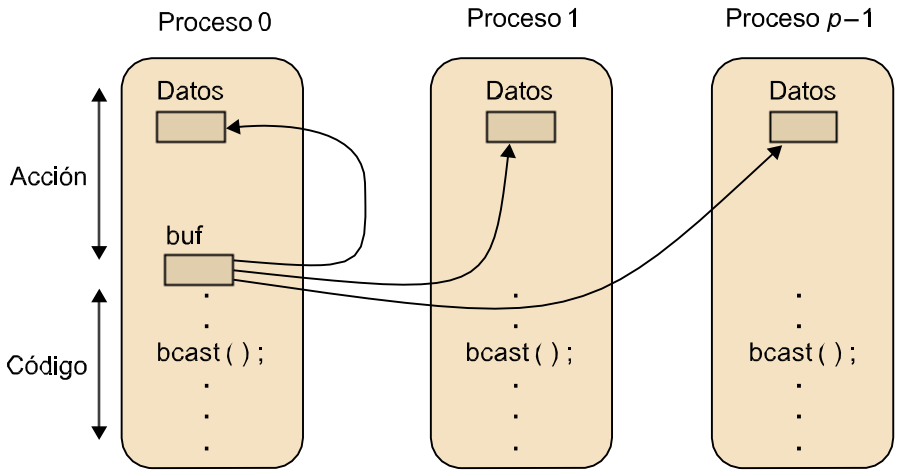


Fig. 1 Funcionamiento de Broadcast

2.1.3. Scatter

```
int MPI_Scatter (void *sendbuf, int sendcount,
          MPI_Datatype sendtype, void *recvbuf,
          int recvcount, MPI_Datatype recvtype,
          int root, MPI_Comm comm)
```

Se puede utilizar MPI_Scatter para enviar desde un proceso mensajes diferentes al resto de los procesos. En el proceso raíz, el mensaje se divide en segmentos de tamaño sendcount y el segmento i-ésimo envía el proceso i. Si se quiere enviar bloques de tamaños diferentes a los diferentes procesos, o si los bloques que hay que enviar no están seguidos en memoria, se puede utilizar la función MPI_Scatterv. La figura 2 ilustra el funcionamiento de MPI_Scatter.

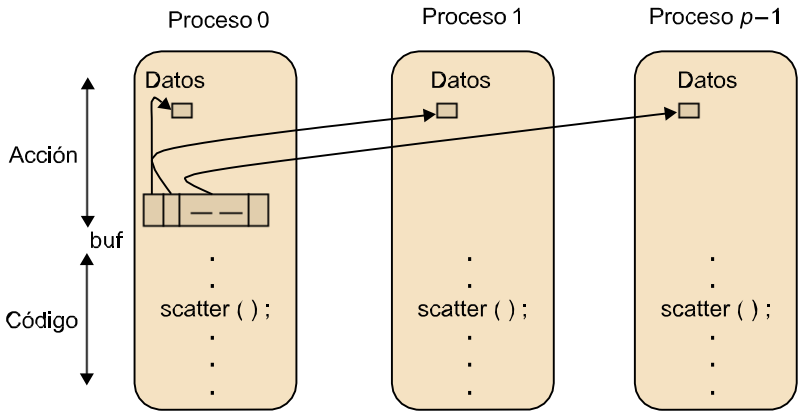


Fig.2 Funcionamiento de Scatter

2.1.4. Gather.

```
int MPI_Gather(void *sendbuf, int sendcount, MPI_Datatype
          sendtype, void *recvbuf, int recvcount,
          MPI_Datatype recvtype,int root, MPI_Comm comm)
```

Esta es la función inversa de MPI_Scatter. Todos los procesos (incluida la raíz) envían al proceso raíz sendcount datos desde sendbuf, y la raíz los almacena en recvbuf por el orden de los procesos. Si se quiere que cada proceso envíe bloques de tamaños diferentes, se utilizará MPI_Gatherv. La figura 3 ilustra el funcionamiento de MPI_Gather.

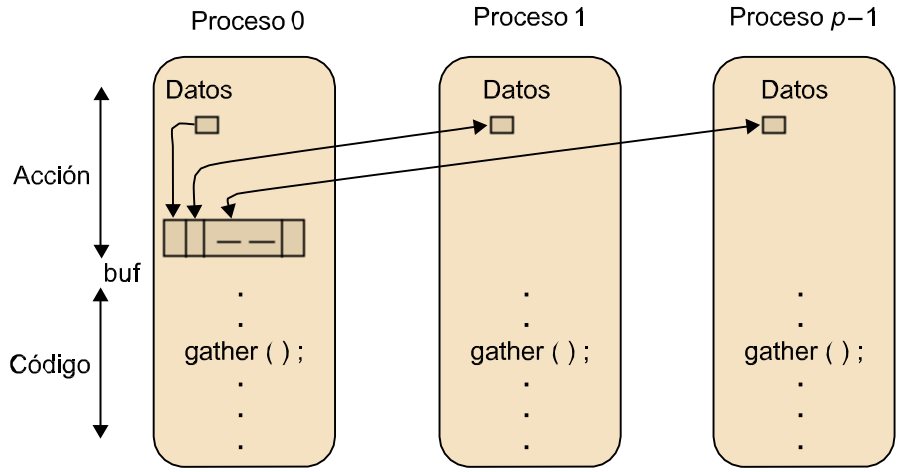


Fig. 3 Funcionamiento de Gather.

Para enviar bloques de datos de todos a todos los procesos, se utiliza:

```
int MPI_Allgather (void *sendbuf, nt sendcount, MPI_Datatype
          sendtype, void *recvbuf, int recvcount, MPI_Datatype
          recvtype,
```

Donde el bloque enviado por el i-ésimo proceso se almacena como bloque i-ésimo en recvbuf en todos los procesos. Para enviar bloques de tamaños diferentes, se utiliza MPI_Allgatherv.

Para enviar bloques de datos diferentes a los diferentes procesos, se utiliza:

```
int MPI_Alltoall(void *sendbuf, int sendcount,
          MPI_Datatype sendtype, void *recvbuf, int
          recvcount, MPI_Datatype recvtype,
```

Donde de cada proceso i, el bloque j se envía al proceso j, que lo almacena como bloque i en recvbuf. Encontramos el correspondiente MPI_Alltoallv.

2.1.5. Reduce.

```
int MPI_Reduce(void *sendbuf, void *recvbuf, int count,
          MPI_Datatype datatype, MPI_Op op,
          int root, MPI_Comm comm)
```

Lleva a cabo una reducción de todos a uno, es decir, se recogen los valores distribuidos en los diferentes procesos y se les aplica una operación a todos. Esto significa que los datos sobre los que hay que hacer la reducción se toman de la zona apuntada por sendbuf, y el resultado se deja en la apuntada por recvbuf. Si el número de datos (count) es mayor que 1, la operación se lleva a cabo elemento a elemento en los del buffer por separado. El resultado se deja en el proceso root. La operación que se aplica a los datos viene dada por op. Las operaciones que se admiten para hacer reducciones se muestran en la tabla siguiente.

Operación	Significado	Tipos C permitidos
MPI_MAX	máximo	Enteros y punto flotante
MPI_MIN	mínimo	Enteros y punto flotante
MPI_SUM	suma	Enteros y punto flotante
MPI_PROD	producto	Enteros y punto flotante
MPI_LAND	AND lógico	Enteros
MPI_LORD	OR lógico	Enteros
MPI_LXORD	XORD lógico	Enteros
MPI_BAND	AND bit a bit	Enteros y bytes
MPI_BOR	OR bit a bit	Enteros y bytes
MPI_BXOR	XOR bit a bit	Enteros y bytes
MPI_MAXLOC	Máximo y localización	Parejas de tipos
MPI_MINLOC	Mínimo y localización	Parejas de tipos

Tabla 1. Operaciones de reducción.

La figura 4 ilustra el funcionamiento de MPI_Reduce.

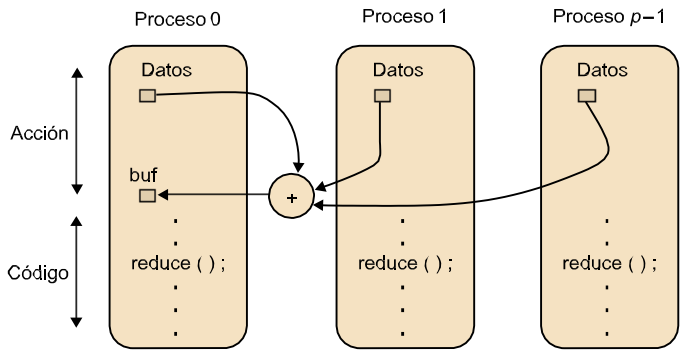


Fig. 4 Funcionamiento de reducción.

3. DESARROLLO DE LA PRÁCTICA.

3.1. Ejercicio 1. Broadcast.

```
#include <stdio.h>
#include <string.h>
#include "mpi.h"
#include <iostream>
int main(int argc, char* argv[])
{
        int my_rank;
        int n;
        MPI_Init(&argc, &argv);
        MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
        if (my_rank == 0)
        {
                printf("Hola, ingresa un número: ");
                std::cin >> n;
        }
        MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
        printf("Hola, soy el proceso %d y el número es el %d\n",
           my_rank, n);
        MPI_Finalize();

}
```

3.2. Ejercicio 2. Scatter.

```
#include <stdio.h>
#include <math.h>
#include <cstdlib>
#include "mpi.h"

int main(int argc, char **argv)
{
        int p, np;
        int nAtoms;
        int *Distribucion = NULL;
        MPI_Status status;

        MPI_Init(&argc, &argv);
        MPI_Comm_rank(MPI_COMM_WORLD, &p);
        MPI_Comm_size(MPI_COMM_WORLD, &np);

        if (p == 0)
        {
                srand(7654321);
                Distribucion = (int *)malloc(np * sizeof(int));
                for (p = 0; p < np; p++)
                {
                        Distribucion[p] = rand();
                }

        }

        MPI_Scatter(Distribucion, 1, MPI_INT, &nAtoms, 1, MPI_INT,
                  0, MPI_COMM_WORLD);

        printf("En proceso %d hay %d atomos\n", p, nAtoms);

        if (p == 0)
                delete(Distribucion);

        MPI_Finalize();

    }
```

3.3. Ejercicio 3. Gather.

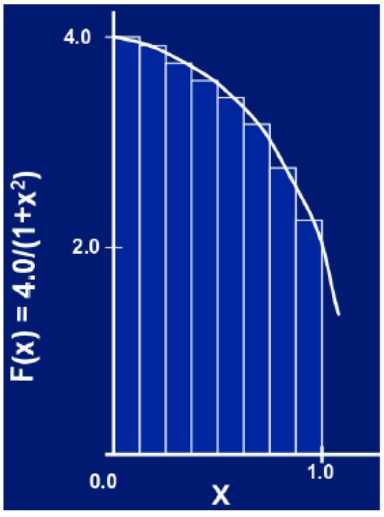
```
#include <stdio.h>
#include <string.h>
#include "mpi.h"
int main(int argc, char* argv[])
{
        int i, j, k, l;
```

```
        int iarray1[16];
        int iarray2[4];
        int myid, numprocs, ierr;
        MPI_Status status;

        MPI_Init(&argc, &argv);
        MPI_Comm_rank(MPI_COMM_WORLD, &myid);
        MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
        printf("soy el nodo %d de %d procesos\n", myid, numprocs);

        for (i = 0; i<16; i++)
                iarray1[i] = 0;
        for (j = 0; j<4; j++)
                iarray2[j] = (myid * 4) + j;

        MPI_Gather(iarray2, 4, MPI_INTEGER, iarray1, 4,
                 MPI_INTEGER, 0, MPI_COMM_WORLD);
        if (myid == 0)
                for (i = 0; i<16; i++)
                        printf("%d\n", iarray1[i]);
        MPI_Finalize();

}
```

3.4. Ejercicio 4. Reduce.

```
#include <stdio.h>
#include <string.h>
#include "mpi.h"
int main(int argc, char* argv[])
{
        int my_rank;
        int num_procs;
        int num, prod;
        MPI_Init(&argc, &argv);
        MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
        MPI_Comm_size(MPI_COMM_WORLD, &num_procs);
        num = my_rank + 1;
        MPI_Reduce(&num, &prod, 1, MPI_INT, MPI_PROD, 0,
                 MPI_COMM_WORLD);
        if (my_rank == 0)
         printf("Hola, soy el proceso %d y %d nodos
       calculamos el producto:%d\n", my_rank, num_procs, prod);
        MPI_Finalize();

}
```

3.5. Ejercicio 5. Cálculo de Pi (versión 1)



```c
#include <stdio.h>
#include <math.h>
#include <sys/types.h>
#include <stdlib.h>
#include <time.h>
// #include <sys/time.h>
#include "mpi.h"

time_t  *ti, *tf;
double  t0, t1, tt;

double calcular_pi(int a, int b, int intervalos) {
        double width, localsum;
        int i;

        /* Inicializa ancho de intervalos */
        width = 1.0 / intervalos;

        /* Realizar calculos locales */
        localsum = 0;
        for (i = a; i <= b; i++)
        {
                register double x = (i + 0.5) * width;
                localsum += 4.0 / (1.0 + x * x);
        }
        localsum *= width;

        return localsum;
 }

void main(int argc, char **argv) {
  int           np, ip, p, resto, cota_inf, cota_sup;
  int         particiones, sub_particion;
  double        suma, parcial;
  MPI_Status    status;

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &np);
  MPI_Comm_rank(MPI_COMM_WORLD, &ip);

  if (ip == 0) /* maestro */
  {
     printf("Ingrese cantidad de particiones: ");
     scanf_s("%d", &particiones);
     printf("\n");
     sub_particion = particiones / np;
     resto = particiones % np;
     cota_sup = -1;
     /* Se reparte carga en los np-1 procesadores restantes */
     for (p = 1; p<np; p++) {
        cota_inf = cota_sup + 1;
        /* Envia cota inferior de subintervalo*/
        MPI_Send(&cota_inf, 1, MPI_INT, p, 0, MPI_COMM_WORLD);
        cota_sup = cota_inf + sub_particion - 1;
        /* Envia cota superior de subintervalo*/
        MPI_Send(&cota_sup, 1, MPI_INT, p, 0, MPI_COMM_WORLD);
        /* Envia numero de particiones*/
        MPI_Send(&particiones, 1, MPI_INT, p, 0, MPI_COMM_WORLD);
     }
     t0 = (double)time(ti);
     if (ti != NULL)
        t0 = (double)*ti;
     cota_inf = cota_sup + 1;
     cota_sup = cota_inf + sub_particion - 1 + resto;

     /* El Procesador maestro no Flojea para nada, tambien realiza sus calculos*/
     suma = calcular_pi(cota_inf, cota_sup, particiones);

    /* Maestro espera, recibe y acumula totales parciales de cada procesador */
    for (p = 1; p<np; p++) {
        MPI_Recv(&parcial, 1, MPI_DOUBLE, p, 0, MPI_COMM_WORLD, &status);
        suma += parcial;
```

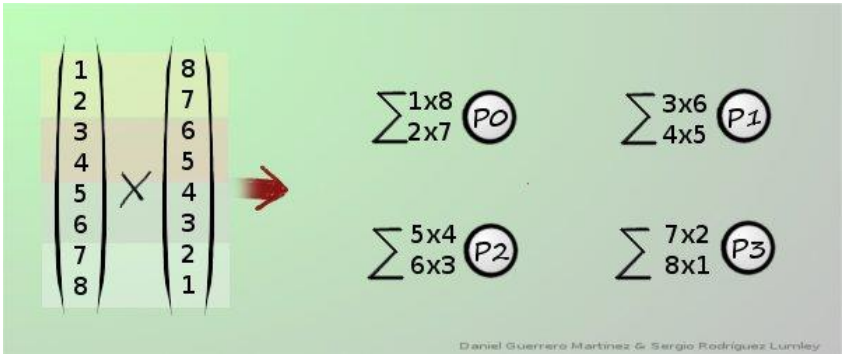
```
        }
        t1 = (double)time(tf);
        if (tf != NULL)
            t1 = (double)*tf;
        /* Imprime el resultado */
        printf("Resultado final, pi es : %g\n", suma);
        printf("Tiempo de inicio %g\n", t0);
        printf("Tiempo de final  %g\n", t1);

        tt = t1 - t0;
        printf("Tiempo de ejecucion total %g [seg.]\n", tt);
        printf("Tiempo de ejecucion total %g [min.]\n", tt / 60);
        printf("Tiempo de ejecucion total %g [hrs.]\n", tt / 3600);
     }
     else { /* Esclavos */

    /* Procesador esclavo p recibe cota inferior */
    MPI_Recv(&cota_inf, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
    /* Procesador esclavo p recibe cota superior */
    MPI_Recv(&cota_sup, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
    /* Procesador esclavo p recibe numero de particiones */
    MPI_Recv(&particiones, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
    /* Procesador esclavo p realiza ahora sus calculos*/
    suma = calcular_pi(cota_inf, cota_sup, particiones);
    /* Procesador esclavo envia resultado parcial a procesador 0 o maestro*/
    MPI_Send(&suma, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);

  }
  MPI_Finalize();
  exit(0);
}
```

3.6. Ejercicio 6. Cálculo de Pi (versión 2).

```
#include <math.h>
#include "mpi.h"   // Biblioteca de MPI
#include <cstdlib> // Incluido para el uso de atoi
#include <iostream>
using namespace std;

int main(int argc, char *argv[])
{
        int n, // Numero de iteraciones
            rank, // Identificador de proceso
            size; // Numero de procesos
        double PI25DT = 3.14159265358979323846264643;
        double mypi, // Valor local de PI
               pi,   // Valor globarl de PI
               h,   // Aproximacion del area para el calculo de PI
               sum;  // Acumulador para la suma del area de PI
        bool valor_por_parametros = true; // Comprueba si hay valores por parametros

        MPI_Init(&argc, &argv); // Inicializamos los procesos
        MPI_Comm_size(MPI_COMM_WORLD, &size); // Obtenemos el numero total de procesos
        MPI_Comm_rank(MPI_COMM_WORLD, &rank); // Obtenemos el valor de nuestro proc.
        // Solo el proceso 0 va a conocer el numero de iteraciones que vamos a
        // ejecutar para la aproximacion de PI
        if (rank == 0) {
           cout << "introduce la precision del calculo (n > 0): ";
           cin >> n;
        }

        // El proceso 0 reparte al resto de procesos el numero de iteraciones
        // que calcularemos para la aproximacion de PI
        MPI_Bcast(&n, // Puntero al dato que vamos a enviar
                1,  // Numero de datos a los que apunta el puntero
                MPI_INT, // Tipo del dato a enviar
                0, // Identificacion del proceso que envia el dato
                MPI_COMM_WORLD);
        if (n <= 0) {
                MPI_Finalize();
                exit(0);
```

```
        }
        else {
                // Calculo de Pi
                h = 1.0 / (double)n;
                sum = 0.0;
                for (int i = rank + 1; i <= n; i += size) {
                        double x = h * ((double)i - 0.5);
                        sum += (4.0 / (1.0 + x*x));
                }
                mypi = h * sum;

                // Todos los procesos ahora comparten su valor local de PI,
                // lo hacen reduciendo su valor local a un proceso
                // seleccionada a traves de una operacion aritmetica.
                MPI_Reduce(&mypi, // Valor local de PI
                        &pi,  // Dato sobre el que vamos a reducir el resto
                        1,        // Numero de datos que vamos a reducir
                        MPI_DOUBLE,  // Tipo de dato que vamos a reducir
                        MPI_SUM,  // Operacion que aplicaremos
                        0, // proceso que va a recibir el dato reducido
                        MPI_COMM_WORLD);

                // Solo el proceso 0 imprime el mensaje, ya que es la unica que
                // conoce el valor de PI aproximado.
                if (rank == 0)
                        cout << "El valor aproximado de PI es: " << pi
                        << ", con un error de " << fabs(pi - PI25DT)
                        << endl;
        }

        // Terminamos la ejecucion de los procesos, despues de esto solo existira
        // el proceso 0
        // ¡Ojo! Esto no significa que los demas procesos no ejecuten el resto
        // de codigo despues de "Finalize", es conveniente asegurarnos con una
        // condicion si vamos a ejecutar mas codigo (Por ejemplo, con "if(rank==0)".
        MPI_Finalize();
        return 0;
}
```

3.7. Ejercicio 7. Producto Escalar

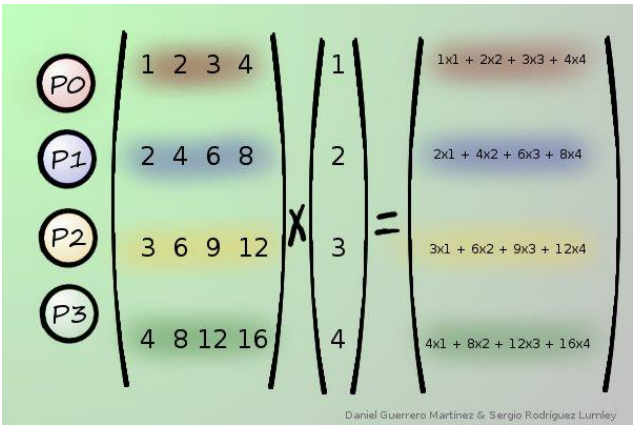



```
#include "mpi.h"
#include <vector>
#include <cstdlib>
#include <iostream>
using namespace std;

int main(int argc, char *argv[]) {
    int tama, rank, size;

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

    if (argc < 2)
    {
      if (rank == 0)
      {
       cout << "No se ha especificado número de elementos,múltiplo de la cantidad
              de entrada, por defecto sera " << size * 100;
```

```
   cout << "\nUso: <ejecutable> <cantidad>" << endl;
  }
  tama = size * 100;
}
else
{
  tama = atoi(argv[1]);
  if (tama < size)
     tama = size;
  else
  {
    int i = 1, num = size;
    while (tama > num)
    {
      ++i;
      num = size*i;
    }
    if (tama != num)
     {
       if (rank == 0)
         cout << "Cantidad cambiada a " << num << endl;
       tama = num;
     }
   }
}

// Creación y relleno de los vectores
vector<long> VectorA, VectorB, VectorALocal, VectorBLocal;
VectorA.resize(tama, 0);
VectorB.resize(tama, 0);
VectorALocal.resize(tama/size, 0);
VectorBLocal.resize(tama/size, 0);
if (rank == 0) {
 for (long i = 0; i < tama; ++i) {
  VectorA[i] = i + 1; // Vector A recibe valores 1, 2, 3, ..., tama
  VectorB[i] = (i + 1)*10; // Vector B recibe valores 10, 20, 30,..,tama*10
 }
}

// Repartimos los valores del vector A
MPI_Scatter(&VectorA[0], // Valores a compartir
        tama / size, // Cantidad que se envia a cada proceso
        MPI_LONG, // Tipo del dato que se enviara
        &VectorALocal[0], // Variable donde recibir los datos
        tama / size, // Cantidad que recibe cada proceso
        MPI_LONG, // Tipo del dato que se recibira
        0,  // proceso principal que reparte los datos
        MPI_COMM_WORLD); // Comunicador (En este caso, el global)
// Repartimos los valores del vector B
MPI_Scatter(&VectorB[0],
        tama / size,
        MPI_LONG,
        &VectorBLocal[0],
        tama / size,
        MPI_LONG,
        0,
        MPI_COMM_WORLD);

// Calculo de la multiplicacion escalar entre vectores
long producto = 0;
for (long i = 0; i < tama / size; ++i) {
    producto += VectorALocal[i] * VectorBLocal[i];
}
long total;

// Reunimos los datos en un solo proceso, aplicando una operacion
// aritmetica, en este caso, la suma.
MPI_Reduce(&producto, // Elemento a enviar
        &total, // Variable donde se almacena la reunion de los datos
        1, // Cantidad de datos a reunir
        MPI_LONG, // Tipo del dato que se reunira
        MPI_SUM, // Operacion aritmetica a aplicar
        0, // Proceso que recibira los datos
        MPI_COMM_WORLD); // Comunicador

if (rank == 0)
    cout << "Total = " << total << endl;
```

```
        // Terminamos la ejecucion de los procesos, despues de esto solo existira
        // el proceso 0
        // ¡Ojo! Esto no significa que los demas procesos no ejecuten el resto
        // de codigo despues de "Finalize", es conveniente asegurarnos con una
        // condicion si vamos a ejecutar mas codigo (Por ejemplo, con "if(rank==0)".
          MPI_Finalize();
          return 0;

    }
```

3.8. Ejercicio 8. Producto Matriz-Vector.




```
#include <iostream>
#include <cstdlib>
#include <ctime>
#include <mpi.h>

using namespace std;

int main(int argc, char * argv[]) {

    int numeroProcesadores,
            idProceso;
    long **A, // Matriz a multiplicar
          *x, // Vector que vamos a multiplicar
          *y, // Vector donde almacenamos el resultado
          *miFila, // La fila que almacena localmente un proceso
          *comprueba; // Guarda el resultado final (calculado secuencialmente), su valor
                      // debe ser igual al de 'y'

    double tInicio, // Tiempo en el que comienza la ejecución
           tFin; // Tiempo en el que acaba la ejecución

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &numeroProcesadores);
    MPI_Comm_rank(MPI_COMM_WORLD, &idProceso);

    A = new long *[numeroProcesadores]; // Reservamos tantas filas como procesos haya
    x = new long [numeroProcesadores]; // El vector será del mismo tamaño que el número
    // de procesadores

    // Solo el proceso 0 ejecuta el siguiente bloque
    if (idProceso == 0) {
        A[0] = new long [numeroProcesadores * numeroProcesadores];
        for (unsigned int i = 1; i < numeroProcesadores; i++) {
            A[i] = A[i - 1] + numeroProcesadores;
        }
        // Reservamos especio para el resultado
        y = new long [numeroProcesadores];

        // Rellenamos 'A' y 'x' con valores aleatorios
        srand(time(0));
        cout << "La matriz y el vector generados son " << endl;
        for (unsigned int i = 0; i < numeroProcesadores; i++) {
            for (unsigned int j = 0; j < numeroProcesadores; j++) {
                if (j == 0) cout << "[";
                A[i][j] = rand() % 1000;
                cout << A[i][j];
```

```
            if (j == numeroProcesadores - 1) cout << "]";
            else cout << "  ";
        }
        x[i] = rand() % 100;
        cout << "\t  [" << x[i] << "]" << endl;
    }
    cout << "\n";

    // Reservamos espacio para la comprobación
    comprueba = new long [numeroProcesadores];
    // Lo calculamos de forma secuencial
    for (unsigned int i = 0; i < numeroProcesadores; i++) {
        comprueba[i] = 0;
        for (unsigned int j = 0; j < numeroProcesadores; j++) {
            comprueba[i] += A[i][j] * x[j];
        }
    }
} // Termina el trozo de código que ejecuta solo 0

// Reservamos espacio para la fila local de cada proceso
miFila = new long [numeroProcesadores];

// Repartimos una fila por cada proceso, es posible hacer la repartición de esta
// manera ya que la matriz esta creada como un único vector.
MPI_Scatter(A[0], // Matriz que vamos a compartir
        numeroProcesadores, // Numero de columnas a compartir
        MPI_LONG, // Tipo de dato a enviar
        miFila, // Vector en el que almacenar los datos
        numeroProcesadores, // Número de columnas a compartir
        MPI_LONG, // Tipo de dato a recibir
        0, // Proceso raíz que envía los datos
        MPI_COMM_WORLD); // Comunicador utilizado (En este caso, el global)

// Compartimos el vector entre todas los procesos
MPI_Bcast(x, // Dato a compartir
        numeroProcesadores, // Numero de elementos que se van a enviar y recibir
        MPI_LONG, // Tipo de dato que se compartirá
        0, // Proceso raíz que envía los datos
        MPI_COMM_WORLD); // Comunicador utilizado (En este caso, el global)


// Hacemos una barrera para asegurar que todas los procesos comiencen la ejecución
// a la vez, para tener mejor control del tiempo empleado
MPI_Barrier(MPI_COMM_WORLD);
// Inicio de medición de tiempo
tInicio = MPI_Wtime();

long subFinal = 0;
for (unsigned int i = 0; i < numeroProcesadores; i++) {
    subFinal += miFila[i] * x[i];
}

// Otra barrera para asegurar que todas ejecuten el siguiente trozo de código lo
// más próximamente posible
MPI_Barrier(MPI_COMM_WORLD);
// fin de medicion de tiempo
tFin = MPI_Wtime();

// Recogemos los datos de la multiplicación, por cada proceso será un escalar
// y se recoge en un vector, Gather se asegura de que la recolección se haga
// en el mismo orden en el que se hace el Scatter, con lo que cada escalar
// acaba en su posición correspondiente del vector.
MPI_Gather(&subFinal, // Dato que envia cada proceso
        1, // Numero de elementos que se envian
        MPI_LONG, // Tipo del dato que se envia
        y, // Vector en el que se recolectan los datos
        1, // Numero de datos que se esperan recibir por cada proceso
        MPI_LONG, // Tipo del dato que se recibira
        0, // proceso que va a recibir los datos
        MPI_COMM_WORLD); // Canal de comunicacion (Comunicador Global)

// Terminamos la ejecución de los procesos, después de esto solo existirá
// el proceso 0
// Ojo! Esto no significa que los demás procesos no ejecuten el resto
// de código después de "Finalize", es conveniente asegurarnos con una
// condición si vamos a ejecutar mas código (Por ejemplo, con "if(rank==0)".
MPI_Finalize();
```

```
    if (idProceso == 0) {

        unsigned int errores = 0;

        cout << "El resultado obtenido y el esperado son:" << endl;
        for (unsigned int i = 0; i < numeroProcesadores; i++) {
            cout << "\t" << y[i] << "\t|\t" << comprueba[i] << endl;
            if (comprueba[i] != y[i])
                errores++;
        }

        delete [] y;
        delete [] comprueba;
        delete [] A[0];

        if (errores) {
            cout << "Hubo " << errores << " errores." << endl;
        } else {
          cout << "No hubo errores" << endl;
          cout << "El tiempo tardado ha sido " << tFin - tInicio << "segundos." << endl;
        }

    }

    delete [] x;
    delete [] A;
    delete [] miFila;

}
```

4. TRABAJO

4.1. Escribir un programa paralelo con MPI para calcular la integral de la función:

$$f = 1/(x+1) + 1/(x^2+1)$$

Utilice el método de suma de trapecios bajo la curva.

4.2. Escribir un programa paralelo para que en una ejecución con cuatro procesos, P0 reparte datos del vector B (de 16 enteros) de la siguiente manera: a P2: B[3], B[4], B[5]; a P1: B[7], B[8]; a P0: B[10]; y a P3: B[12], B[13], B[14], B[15]. Tras ello, cada proceso suma 100 a los elementos recibidos, y, finalmente, se recopilan los datos finales en P0, en las mismas posiciones iniciales del vector B.

5. BIBLIOGRAFIA

- Castro Rodero Ivan, G. B. F. Programación y Computación Paralelas.: Universidad Oberta de Catalunya.
- Gropp William, L. E., Skjellum Anthony. (2014). Using MPI Portable Parallel Programming with the
- J., Q. M. (2003). Parallel Programming in C with MPI and OpenMP. Singapore: Mc Graw Hill.
- Peter, P. (1997). Parallel Programming with MPI. San Francisco California: Morgan Kaufmann Publishers.
- Peter, P. (2011). An Introduction to Parallel Programming. Burlington USA: Elsevier Inc.
- Wilkinson Barry, A. M. (2004). Parallel Programming (Second Edition): Prentice Hall.