Actividad 5:
Programación en MPI y del paradigma map-reduce

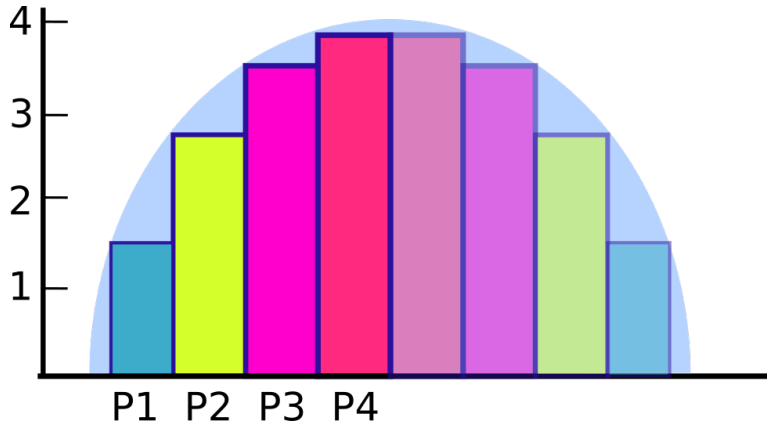
Dra. Magali Arellano Vázquez

Instrucciones generales: Los siguientes ejercicios son incrementales, es decir, cada uno de ellos tiene el objetivo de que aprendas a usar algunas funciones especificas, en cada uno se listan estas funciones. Son ejercicios clásicos y posiblemente los encuentres resueltos buscando en internet pero el objetivo es que te apropies del conocimiento y aprendas como se usa cada función con ayuda del documento con las especificaciones que se te proporcionará, es válido que veas otros códigos pero los ejercicios debes realizarlos tu. En los primeros 4 ejercicios se proporciona el código en versión secuencial (los códigos proporcionados están en C++, sin embargo, si prefieres, puedes migralo a python usando mpi4py) , el código funciona y resuelve el problema, deberás usar ese código como base y para comparar la ejecución de ese con la versión paralela que realizarás. El ejercicio 5 no es parte de los ejercicios del lenguaje MPI, en ese ejercicio, como se revisó en el material, se realiza en lenguaje python. Recuerda incluir en tu reporte:

- Código.
- Captura de pantalla de la ejecución de cada código.
- Los resultados de las métricas de desempeño que has visto en unidades anteriores

1. **Cálculo de π.** El cálculo paralelo de la aproximación del número π, con un factor de precisión. El número π es definido como:

$$\pi = \int_0^1 \frac{4}{1+x^2}$$



La aproximación de una integral usando la suma de Riemann hace posible la división del trabajo en unidades independientes, siendo un factor de precisión el número de veces que se divide.

El factor de precisión lo solicitará el proceso 0 y será distribuido entre los procesos mediante el uso de la función **MPI_Bcast**.

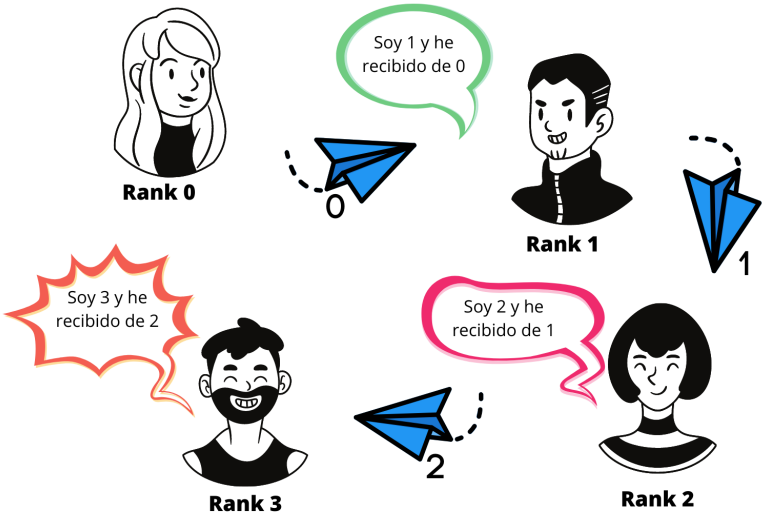
Despues de que cada proceso calcule su parte se reunirán todas las partes en el proceso 0 para mostrar el resultado con **MPI_Reduce**.

Funciones necesarias:

- MPI_Init
- MPI_Finalize
- MPI_Comm_size
- MPI_Comm_rank
- MPI_Bcast
- MPI_Reduce

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

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

        // Calculo de PI
        int n;
        cout<<"introduce la precision del calculo (n > 0): ";
        cin>>n;
        double PI25DT = 3.14159265358979323846264 3;
        double h = 1.0 / (double) n;
        double sum = 0.0;
        for (int i = 1; i <= n; i++) {
                double x = h * ((double)i - 0.5);
                sum += (4.0 / (1.0 + x*x));
        }
        double pi = sum * h;
        cout << "La aproximacion del valor de  PI es: " << pi << ", con un error de " << fabs(pi -PI25DT) << endl;
        return 0;
}
```

2. Paralelizar el siguiente programa, este encadena el envío y la recepción de un mensaje, se considera que el identificador del procesador (id) es el rango del proceso que envía. Los mensajes deben enviarse en forma encadenada, es decir, el primer procesador enviará un mensaje al segundo, el segundo los recibirá del primero y enviará un mensaje al tercero, así sucesivamente hasta terminar con los procesos lanzados. Cada procesador que reciba un mensaje debe escribir en pantalla "Soy el proceso X y he recibido M", siendo X el rango del proceso y M el mensaje recibido.



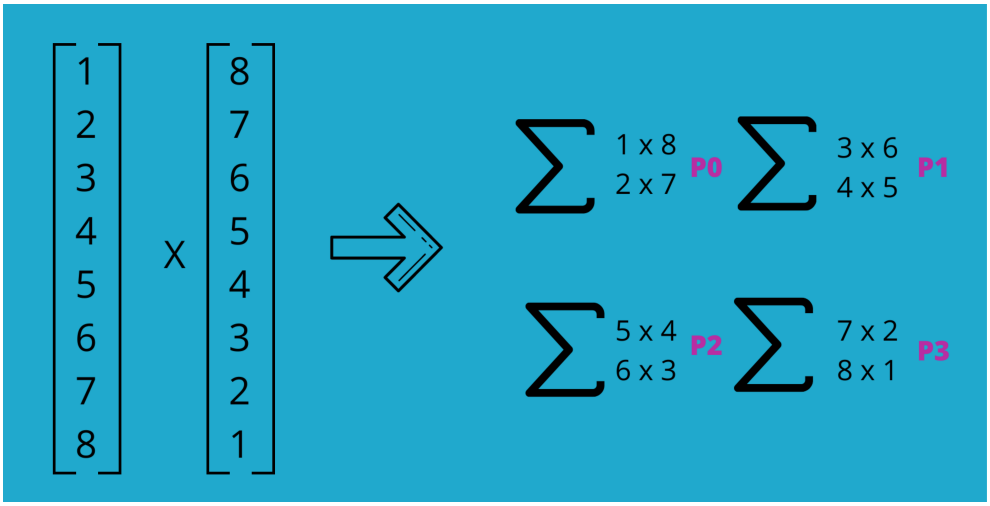
No olvides inicializar y finalizar las estructuras de paralelismo (**MPI_Init** y **MPI_Finalize**). Recuerda obtener el rango y el total de procesos participantes. Recuerda que el último proceso no envía ningún mensaje. Si observas **MPI_Send** y **MPI_Recv** se puede especificar a quien se envía o de quien se recibe un mensaje. Estas funciones reciben el mensaje siempre por referencia (MPI_Send(&rank, ...), MPI_Recv(&buzon, ...)).

Funciones necesarias:

- MPI_Init
- MPI_Finalize
- MPI_Comm_size
- MPI_Comm_rank
- MPI_Send
- MPI_Recv

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

int main(int argc, char *argv[])
{
    int rank, contador;
    MPI_Status estado;

    MPI_Init(&argc, &argv); // Inicializamos la comunicacion de los procesos
    MPI_Comm_rank(MPI_COMM_WORLD, &rank); // Obtenemos el valor de nuestro identificador

    //Envia y recibe mensajes
    MPI_Send(&rank //referencia al vector de elementos a enviar
            ,1 // tamaño del vector a enviar
            ,MPI_INT // Tipo de dato que envias
            ,rank // pid del proceso destino
            ,0 //etiqueta
            ,MPI_COMM_WORLD); //Comunicador por el que se manda

    MPI_Recv(&contador // Referencia al vector donde se almacenara lo recibido
            ,1 // tamaño del vector a recibir
            ,MPI_INT // Tipo de dato que recibe
            ,rank // pid del proceso origen de la que se recibe
            ,0 // etiqueta
            ,MPI_COMM_WORLD // Comunicador por el que se recibe
            ,&estado); // estructura informativa del estado

        cout<< "Soy el proceso "<<rank<<" y he recibido "<<contador<<endl;

    MPI_Finalize();
    return 0;
}
```

3. Realiza el producto escalar de dos vectores en forma paralela. El producto escalar de dos vectores se define como la sumatoria del producto entre los elementos del vector.



Cada proceso solo necesita conocer una parte de los vectores, usando la función MPI_Scatter te ayudará a repartir las tareas.

Funciones necesarias:

- MPI_Init
- MPI_Finalize
- MPI_Comm_size
- MPI_Comm_rank
- MPI_Scatter
- MPI_Reduce

```cpp
#include <vector>
#include <cstdlib>
#include <iostream>
using namespace std;

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

    if (argc < 2) {
    cout << "No se ha especificado numero de elementos, por defecto sera " << 100;
    cout << "\n Uso: <ejecutable> <cantidad>" << endl;
        tama = 100;
    } else {
        tama = atoi(argv[1]);
    }

    // Creacion y relleno de los vectores
    vector<long> VectorA, VectorB;
    VectorA.resize(tama, 0);
    VectorB.resize(tama, 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
    }

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

    cout << "Total = " << total << endl;
    return 0;
}
```

4. Realizar la multiplicación paralela de una matriz $N \times N$ con un vector $N \times 1$.

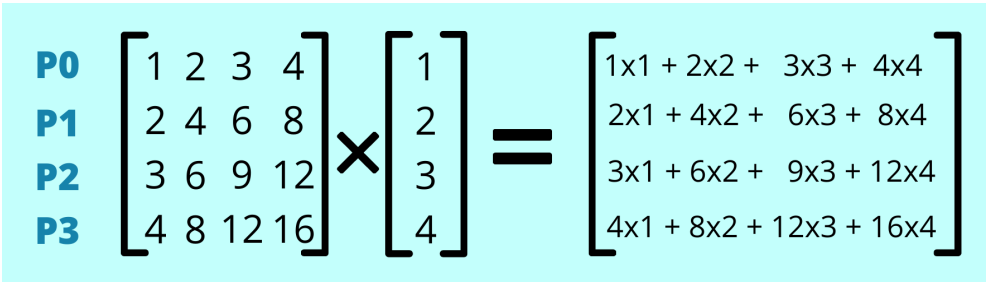
   En este problema la matriz A se descompondrá en filas, una por cada proceso. Por lo tanto, cada proceso tendrá una fila de la matriz y el vector por el que se multiplicará.

   a) El proceso 0 generará a la matriz A y el vector x.

b) La matriz A será distribuida (usando **MPI_Scatter**), y el vector se difundirá (usando **MPI_Bcast**).

c) Cada proceso opera con los datos que tiene.

d) Se reconstruye el vector solución en el proceso 0 mediante **MPI_Gather**.



Recuerda que el número de filas de la matriz (que es igual al número de columnas, se asume que la matriz es cuadrada) debe ser igual al número de procesos disponibles. Cada proceso obtendrá como resultado un número, que al unirlos (**MPI_Gather**) en el proceso 0 forman el vector solución. Utilizando **MPI_Wtime** podemos tomar tiempos, pero hay que tener en cuenta que los tiempos son locales a un proceso, es conveniente medir el tiempo asegurándose que los procesos tardan lo mismo, con **MPI_Barrier** se logra la sincronización de todos los procesos del comunicador.

Funciones necesarias:

- MPI_Init
- MPI_Finalize
- MPI_Comm_size
- MPI_Comm_rank
- MPI_Barrier
- MPI_Wtime
- MPI_Gather
- MPI_Scatter
- MPI_Bcast

```cpp
#include <iostream>
#include <cstdlib>
#include <ctime>

using namespace std;
int main(int argc, char * argv[]) {

    long **A, // Matriz a multiplicar
         *x, // Vector que vamos a multiplicar
         *comprueba; // Guarda el resultado final (calculado secuencialmente)
    int n;
    if (argc <= 1) {// si no se pasa por parametro el tamaño de la matriz,
                    //se coge por defecto el numero de procesadores
        cout << "Falta el tamanio de la matriz, por defecto cogemos 10"<< endl;
        n = 10;
    } else
        n = atoi(argv[3]);

        A = new long *[n];//reservamos espacio para las n filas de la matriz.
        x = new long [n];//reservamos espacio para el vector.
    //Rellena la matriz
    A[0] = new long [n * n];
    for (unsigned int i = 1; i < n; i++) {
        A[i] = A[i - 1] + n;
    }

    // Rellena A y x con valores aleatorios
    srand(time(0));
    cout << "La matriz y el vector generados son " << endl;
    for (unsigned int i = 0; i < n; i++) {
        for (unsigned int j = 0; j < n; j++) {
            if (j == 0) cout << "[";
            A[i][j] = rand() % 1000;
            cout << A[i][j];
            if (j == n - 1) cout << "]";
            else cout << "  ";
        }
        x[i] = rand() % 100;
        cout << "\t  [" << x[i] << "]" << endl;
    }
    cout << "\n";

    comprueba = new long [n];
    //Calculamos la multiplicacion secuencial para
    //despues comprobar que es correcta la solucion.
    for (unsigned int i = 0; i < n; i++) {
        comprueba[i] = 0;
        for (unsigned int j = 0; j < n; j++) {
            comprueba[i] += A[i][j] * x[j];
        }
    }
    cout << "El resultado obtenido y el esperado son:" << endl;
    for (unsigned int i = 0; i < n; i++) {
        cout << comprueba[i] << endl;
    }

    delete [] comprueba;
    delete [] A[0];
    delete [] x;
    delete [] A;
    return 0;

}
```

5. Utilizando el código de map-reduce, visto en las presentaciones, contar todas la palabras de los tres archivos de texto que les proporcionarán en la carpeta ejercicio_5

   ```
   /ejercicio_5
   ```

   Tengan cuidado al considerar que es posible que el código también cuente los espacios, los caracteres especiales y otras cosas que no se desean contar, por lo que deberán filtrar todo eso. El código debe poder ejecutarse de la siguiente forma:

   ```
   ls *.txt | python map.py | python reduce.py > res.txt
   ```