#include <iostream>

#include <cstdlib>

#include <ctime>

#include "mpi.h"

using namespace std;

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

int numeroProcesadores,idProceso,psize;

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 ejecucion

tFin; //Tiempo en el que acaba la ejecucion

MPI\_Init(&argc, &argv);

MPI\_Comm\_size(MPI\_COMM\_WORLD, &numeroProcesadores);

//MPI\_Comm\_size(MPI\_COMM\_WORLD, &psize); // Obtenemos el valor de nuestro identificador

cout << "Hay " << numeroProcesadores << " procesos en el comunicador global";

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &idProceso);

A = new long \*[numeroProcesadores]; // Reservamos tantas filas como procesos haya

x = new long [numeroProcesadores]; // El vector sera del mismo tamanio que el numero de procesadores

// Solo el proceso 0 ejecuta el siguiente bloque

if (idProceso == 0) {

A[0] = new long [numeroProcesadores \* numeroProcesadores];

for (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 (int i = 0; i < numeroProcesadores; i++) {

for (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 comprobacion

comprueba = new long [numeroProcesadores];

// Lo calculamos de forma secuencial

for (int i = 0; i < numeroProcesadores; i++) {

comprueba[i] = 0;

for (int j = 0; j < numeroProcesadores; j++) {

comprueba[i] += A[i][j] \* x[j];

}

}

} // Termina el trozo de codigo 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 reparticion de esta

// manera ya que la matriz esta creada como un unico 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, // Numero de columnas a compartir

MPI\_LONG, // Tipo de dato a recibir

0, // Proceso raiz que envia 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 compartira

0, // Proceso raiz que envia los datos

MPI\_COMM\_WORLD); // Comunicador utilizado (En este caso, el global)

// Hacemos una barrera para asegurar que todas los procesos comiencen la ejecucion

// a la vez, para tener mejor control del tiempo empleado

MPI\_Barrier(MPI\_COMM\_WORLD);

// Inicio de medicion de tiempo

tInicio = MPI\_Wtime();

long subFinal = 0;

for (int i = 0; i < numeroProcesadores; i++) {

subFinal += miFila[i] \* x[i];

}

// Otra barrera para asegurar que todas ejecuten el siguiente trozo de codigo lo

// mas proximamente posible

MPI\_Barrier(MPI\_COMM\_WORLD);

// fin de medicion de tiempo

tFin = MPI\_Wtime();

// Recogemos los datos de la multiplicacion, por cada proceso sera un escalar

// y se recoge en un vector, Gather se asegura de que la recoleccion se haga

// en el mismo orden en el que se hace el Scatter, con lo que cada escalar

// acaba en su posicion 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 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();

if (idProceso == 0) {

int errores = 0;

cout << "El resultado obtenido y el esperado son:" << endl;

for (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;

}