27 DE DICIEMBRE DE 2023

INGENIERÍA DE COMPUTADORES

PRÁCTICA 4 – OPEN MPI

Jaime Morales Vaello
Iván López Sanchís
Adrían Ríos Armero
Joan Climent Quiñones

GRUPO 07

Tarea 4.1 / 4.2: Familiarizarse con MPI y Problema paralelizable con MPI

En la tarea 4.1, se nos presenta como es un 'Hello World' mediante el uso de la librería MPI. Seguidamente, en la tarea 4.2, cogiendo de guía los diferentes métodos de este código, deberemos de pensar un problema sencillo paralelizable con MPI, que contenga llamadas de comunicación colectiva, y realizar su implementación mediante la librería.

En nuestro caso, hemos escogido como problema: ***el producto escalar entre dos vectores de forma paralela***. Para paralelizar este problema con MPI, hemos usado varias funciones de esta librería:

- **MPI_Init(&argc, &argv)**: se utiliza para inicializar el entorno MPI antes de que cualquier otro llamado a la API de MPI sea realizado en un programa. Solo se debe llamar una vez, esta vez será al principio del programa.
- **MPI_Comm_size(MPI_COMM_WORLD, &size)**: función que se utiliza para obtener el número de procesos que participan en una comunicación.
- **MPI_Comm_rank(MPI_COMM_WORLD, &rank)**: función que se utiliza para obtener el rango (o identificador) de un proceso dentro de un comunicador dado. El comunicador es esencialmente un grupo de procesos que pueden comunicarse entre sí. Esta función la utilizamos para, después, con el 'cout', saber que proceso está realizando cada procesador.
- **MPI_Get_processor_name(processor_name, &namelen)**: función que se utiliza para obtener el nombre del procesador o nodo en el que se está ejecutando un proceso MPI.
- **MPI_Scatter(&VectorA[0], tama / size, MPI_LONG, &VectorALocal[0], tama / size, MPI_LONG, 0, MPI_COMM_WORLD)**: función utilizada para distribuir datos desde el proceso maestro a los procesos trabajadores en una comunicación de tipo "scatter". Esta función la utilizamos en dos ocasiones para repartir los valores del vector A y del vector B.
- **MPI_Reduce(&producto, &total, 1, MPI_LONG, MPI_SUM, 0, MPI_COMM_WORLD)**: función que se utiliza para realizar una operación de reducción en los valores de un conjunto de procesos y producir un resultado en un solo proceso. Esta función la aplicamos junto a una suma aritmética para finalizar el producto escalar entre los dos vectores A y B.
- **MPI_Finalize()**: función que se utiliza para finalizar el entorno MPI antes de que un programa MPI termine. Cuando se ejecuta esta función, la biblioteca MPI realiza varias tareas de limpieza y termina cualquier comunicación pendiente.

A continuación, mostramos una captura del código para poder entender mejor con exactitud como se ha tratado el problema y la explicación de los argumentos de las funciones citadas anteriormente.

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

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

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Get_processor_name(processor_name, &namelen);
    cout << "Hello from process " << rank <<"on "<< processor_name <<" processor " << " \n";

    if (argc < 2) {
        if (rank == 0) {
            cout << "No se ha especificado numero de elementos, multiplo 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;
            }
        }
    }

    // Creacion 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;
}
```

Finalmente, siguiendo los siguientes pasos, pasamos a la conexión de las 3 máquinas:

1. Todas las máquinas deben tener el mismo usuario. Todos los ejecutables deben tener el mismo nombre en cada máquina y estar en la misma carpeta. En nuestro caso, hemos entrado mediante el usuario del integrante del grupo Adrián Ríos y el nombre del ejecutable es ‘producto_escalar’.

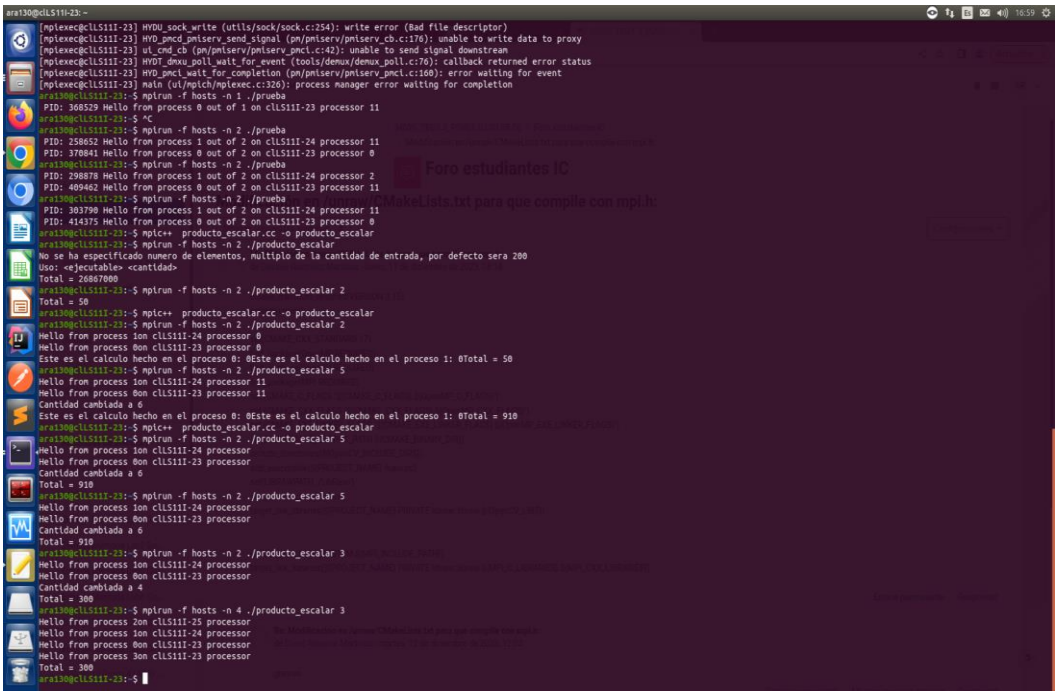
2. Compilar empleando mpicc o mpic++: *mpic++ producto_escalar.cc -o producto_escalar*.

3. Crear el **archivo de hosts** con las IPs o nombres de cada nodo de la red y copiar al resto de máquinas en el mismo directorio de trabajo donde se encuentran los ejecutables. El archivo de hosts es un fichero de texto plano que especifica el nombre o la IP de las máquinas que forman el supercomputador. **Cada nombre o IP en una línea**. Introducimos en el fichero en primer lugar la IP de la máquina ‘host’ (valga la redundancia) seguida de las IP de las otras 2 máquinas.

4. Crear un certificado de clave pública ssh en uno de los nodos: *ssh-keygen –t rsa* (esto crea la carpeta oculta ~/.ssh).

5. Copiar el archivo *~/.ssh/id_rsa.pub* en la carpeta *~/.ssh* del resto de máquinas y cambiar su nombre a *authorized_keys*. Si no existe la carpeta oculta *~/.ssh* en una máquina, crearla previamente ejecutando el comando *ssh-keygen –t rsa*.

6. Ejecutar el programa en la máquina donde se creó el certificado: mpirun -f hosts -n <num_procesos> ./producto_escalar <argumento>



Tarea 4.3: Paralelización del código de la práctica anterior

Main

Se inicia el entorno MPI mediante la función MPI_Init, con la obtención del rango y la cantidad de procesos MPI usando MPI_Comm_rank y MPI_Comm_size, respectivamente. Posteriormente, se realiza una verificación de la disponibilidad de los argumentos de entrada del programa. En caso de que no se proporcionen los argumentos suficientes, se despliega un mensaje de error y se finalizan todos los procesos MPI utilizando MPI_Finalize().

Las diversas funciones de procesamiento de imágenes se ejecutan simultáneamente entre los distintos procesos MPI. Cada uno de estos procesos trabaja de manera independiente en una sección específica de la imagen.

Tras la realización de las operaciones de procesamiento de la imagen, se emplea MPI_Gather para reunir las partes procesadas de la imagen en el proceso 0, que es la raíz. Este enfoque permite una distribución eficiente del trabajo de procesamiento de imágenes entre múltiples procesos MPI, aprovechando la capacidad de procesamiento paralelo para acelerar este proceso. Como resultado, se obtiene una imagen final mejorada y procesada.

Gamma Curve

La función gammaCurve ha sido paralelizada utilizando la biblioteca MPI (Message Passing Interface). MPI es una biblioteca estándar para la programación paralela en sistemas distribuidos. Aquí está lo que se ha hecho:
Se obtienen el rango (ID) y el tamaño total (número de procesos) del comunicador global MPI_COMM_WORLD usando MPI_Comm_rank y MPI_Comm_size. Cada proceso obtiene un rango único, que va desde 0 hasta size-1.
Se divide el trabajo entre los procesos. Cada proceso calcula un "chunk" del array curve. El tamaño del chunk es imax / size, y el rango del proceso se utiliza para determinar el índice de inicio. Si hay un residuo de la división imax / size, se añade al último proceso.
Cada proceso realiza su cálculo en su propio rango de índices en el array curve.
Una vez que todos los procesos han terminado sus cálculos, se utiliza MPI_Allgather para recoger los resultados de todos los procesos. MPI_Allgather recoge los datos de todos los procesos y los distribuye a todos los procesos. En este caso, se utiliza MPI_IN_PLACE como el búfer de envío, lo que significa que los datos enviados son los mismos que los datos recibidos, y MPI_DATATYPE_NULL como el tipo de datos, lo que significa que no se envían datos.

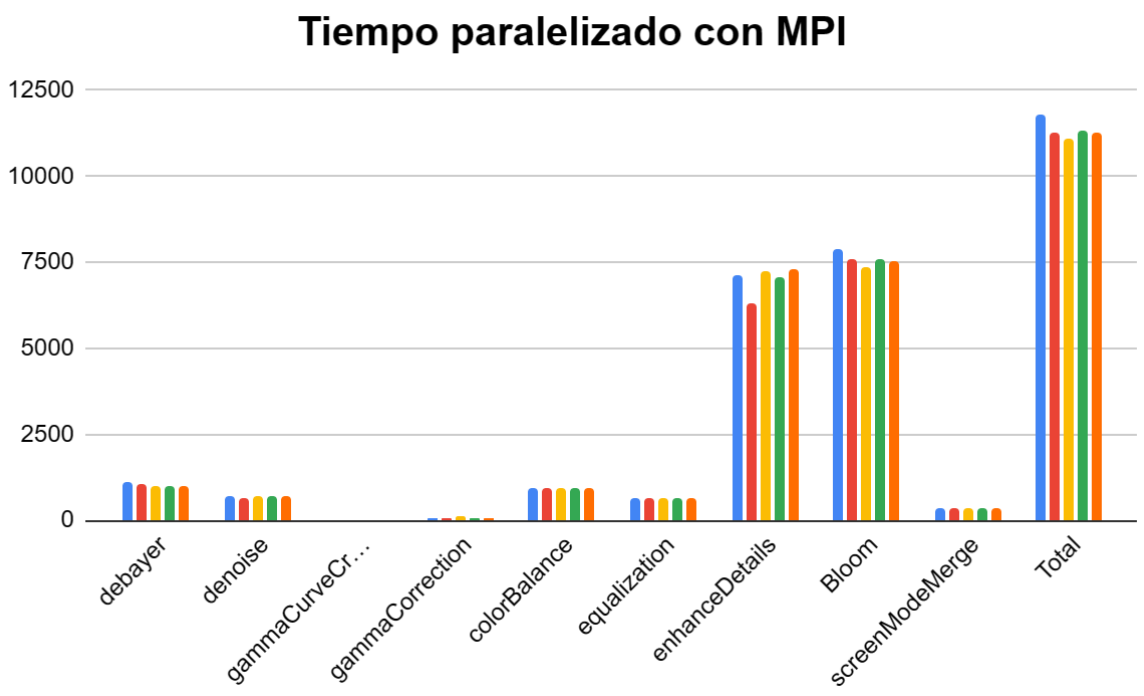
La elección de estas funciones MPI permite dividir el trabajo entre varios procesos y luego recoger los resultados. Esto puede acelerar el cálculo de la curva gamma, especialmente si imax es grande y hay muchos procesos disponibles.

Diferencias del tiempo

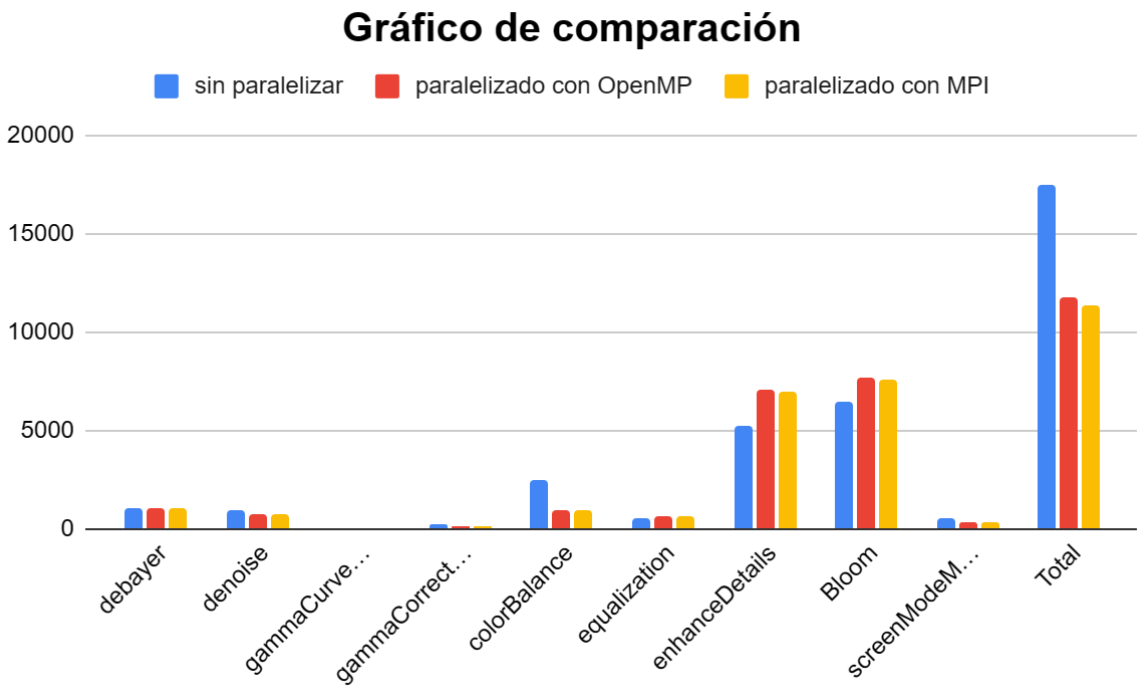
	sin paralelizacion					paralelizado con MPI				
debayer	1538	1052	948	954	940	1103	1042	980	991	977
denoise	1557	821	830	829	847	727	643	712	682	701
gammaCurveCreation	2	2	2	2	3	2	0	1	1	1
gammaCorrection	202	201	203	201	203	89	83	95	78	80
colorBalance	2463	2450	2472	2474	2484	956	943	921	938	930
equalization	544	543	587	550	559	667	654	665	658	647
enhanceDetails	5378	5152	5353	5132	5198	7113	6291	7243	7049	7291
Bloom	6884	6638	6103	6488	6141	7873	7571	7367	7620	7548
screenModeMerge	486	490	477	479	484	361	349	370	352	367
Total	19054	17349	16975	17109	16859	11778	11285	11111	11320	11251

	paralelizado con OpenMP					paralelizado con MPI				
debayer	1088	1038	965	1045	999	1103	1042	980	991	977
denoise	724	645	704	697	712	727	643	712	682	701
gammaCurveCreation	1	0	0	1	1	2	0	1	1	1
gammaCorrection	97	103	107	99	99	89	83	95	78	80
colorBalance	933	923	936	952	930	956	943	921	938	930
equalization	676	666	669	642	652	667	654	665	658	647
enhanceDetails	7134	6380	7191	7077	7380	7113	6291	7243	7049	7291
Bloom	7983	7481	7494	7641	7713	7873	7571	7367	7620	7548
screenModeMerge	372	362	377	348	355	361	349	370	352	367
Total	12216	11558	11600	11775	11818	11778	11285	11111	11320	11251

Como vemos en la imagen, hemos conseguido reducir los tiempos respecto al programa original, hemos conseguido que sean similares a la paralelización que llevamos a cabo en la práctica anterior.



Como hemos paralelizado las funciones que hay en el main podemos ver que el tiempo total es significativamente más bajo que la suma de todas las funciones.



Aquí vemos la comparativa de los tiempos entre la paralelización con MP, MPI y sin paralelizar.