UNIVERSIDAD DE GRANADA

Arquitectura y computación de altas prestaciones (2022-2023)
Grado en Ingeniería Informática

Práctica 2

Realizado por:
Hlib Bukhtieiev

Ejercicio 1

Este ejercicio presenta una estructura típica de maestro-esclavo. El enfoque de paralelización es sencillo: el recoge una petición y la manda a trabajar a uno de los esclavos, espera su respuesta e imprime el resultado por pantalla.

El maestro tiene 4 funcionalidades – descritas detalladamente en el pdf de la práctica. Éstas no presentan un mayor interés excepto la de cerrar la aplicación, puesto que de alguna forma habría que crear un mensaje para todos los hijos que les hiciera salir del bucle y pasar al estado finalizado. Una de las opciones sería crear un mensaje del estilo “-1111”, y al recibir los hijos esta cadena acabarían con sus tareas, sin embargo, este método tiene una desventaja y es que priva al usuario de la opción de escribir dicha cadena lo cual le resta flexibilidad a mi programa. Teniendo en cuenta eso, he creado una etiqueta especial que les haría a los hijos salir del bucle de trabajo. El funcionamiento es el siguiente: el maestro recibe un número por pantalla, en el caso de ser 0, se envía un mensaje basura a todos los esclavos con una etiqueta distinta a la que se usa para la comunicación normal, los esclavos comprueban la etiqueta del mensaje antes de empezar con el funcionamiento normal, en el caso de ser una etiqueta de salida, los esclavos se salen del bucle de ejecución y pasan al estado finalizado.

Las funcionalidades 1-3 no están enfocadas a una mejora de prestaciones del programa, sin embargo la funcionalidad “4”, la que trata de hacer las funcionalidades 1-3 a la vez, se podría decir que podemos ejecutarla de forma concurrente. Pues primero desde el master envio petición de trabajo al proceso 2, ya que éste no necesita una lectura previa de datos por la entrada estándar, luego leo los datos para las funcionalidades “1” y “3” y las mando a trabajar.

Algo a comentar es que la escritura de mensajes por la salida estándar no está serializada, ya que no ha sido el objetivo de este ejercicio, y por tanto los mensajes pueden aparecer en el orden distinto al esperado.

Ejercicio 2

Este ejercicio ha consistido en paralelizar el procesamiento de la matriz que representa una imagen. Una de las condiciones iniciales ha sido que solamente el proceso 0 tenga acceso a la matriz entera. El enfoque que he decidido tomar es paralelización por columnas, ya que reserva de memoría dinámica para la matriz ha sido realizada por columnas teniendo en cuenta la necesidad de tener los datos contiguos en memoria.

El desarrollo se puede subdividir en 5 etapas principales:

1. **Lectura de la imagen**

Paso trivial, solamente dentro del proceso 0, abrimos el fichero con la imagen, leemos los datos en la matriz de entrada, reservamos memoria necesaria para la matriz con los datos procesados.

2. **Reparto de datos**

```
MPI_Bcast(&Alto, 1, MPI_INT, MASTER, MPI_COMM_WORLD);

// Calculating number of columns that is gonna be sent to each process.
// Index from which data is sent is calculated given the contiguos memory allocation of a matrix.
// Good luck lol
for (int i = 1; i < size; i++){
    complement_size = (i < rest) + (i < size-1);
    complement_id = (i - 1 < rest) + (i - 1 < size-1);
    rel_size = Alto*(cols_per_process + complement_size + 1);
    id = (rest) ? ((i - 1 < rest) ? cols_per_process + (i-1)*(cols_per_process + complement_id-1)
                  : rest*(cols_per_process + 1) + cols_per_process-1 + (i-1-rest)*cols_per_process)
               : cols_per_process-1 + (i-1)*cols_per_process;

    MPI_Send(Original[id], rel_size, MPI_UNSIGNED_CHAR, i, TAG, MPI_COMM_WORLD);
}
```

En este paso uso la función MPI_Bcast para comunicar el número de filas de la matriz, de esta forma al enviar una cantidad de bytes a un proceso éste podrá computar el número de columnas que componen esos bytes.

Lo siguiente que vemos en la imagen es el propio reparto de columnas según el número de procesos que se han creado. Hay muchos modificadores y operadores ternarios porque había que contemplar varios escenarios a la hora de repartir los datos:

- Existencia o no de un resto al dividir el número de columnas entre el número de procesos implicados. Si hay resto, se reparte una columna de más a cada proceso hasta agotar el resto.
- Al saber la naturaleza del proceso de convolución – para procesar columna “i” necesitamos los datos de las “i-1” e “i+1” – tenemos que entender que si el proceso Z procesa los datos de la matriz desde la columna X hasta la Y hay que enviarle los datos de la matriz desde la columna X-1 hasta Y+1, sin olvidarnos de controlar los índices para el primer y último proceso.

3. Procesamiento de la imagen

Este paso no supone mayor complejidad, todos los procesos llaman a la función de convolución con la parte de la matriz que hayan recibido. La función de convolución no ha sido modificada.

4. Recolección de los resultados

```
// Same thing as above, calculating index and size, however this time it is done in order to
// write into result matrix
for (int i = 1; i < size; i++){
    complement_size = (i < rest);
    complement_id = (i-1 < rest);
    rel_size = Alto*(cols_per_process + complement_size);
    id = (rest) ? ((i-1 < rest) ? i*(cols_per_process + complement_id) : rest*(cols_per_process + 1) + (i-rest)*cols_per_process)
                : i*cols_per_process;
    MPI_Recv(Salida[id], rel_size, MPI_UNSIGNED_CHAR, i, TAG, MPI_COMM_WORLD, &status);
}
```

Al recibir los datos procesados los escribimos directamente en la matriz de salida, otra vez tenemos que tener en cuenta la existencia del resto. Una cosa que se me ha ocurrido para mejorar las prestaciones es usar la recepción asíncrona ya que no existe la condición de carrera a la hora de

escribir los datos en la matriz. Sin embargo, la mejora sería prácticamente despreciable, así que no he optado por implementar el programa de esa forma.

5. **Escritura en el fichero de la imagen**

Tras finalizar la recepción de los datos de todos los procesos, abrimos el fichero indicado y escribimos en éste.

Comparación de tiempos

Procesadores	Tiempo
1	0.014321
2	0.007083
4	0.006558
6	0.006231
8	0.006095

Como podemos ver, para muchos procesadores la sobrecarga hace que la mejora sea prácticamente despreciable. Eso se debe a que la resolución de la imagen proporcionada es demasiado pequeña, para apreciar las ventajas del paralelismo necesitamos de una imagen de 4K por lo menos.