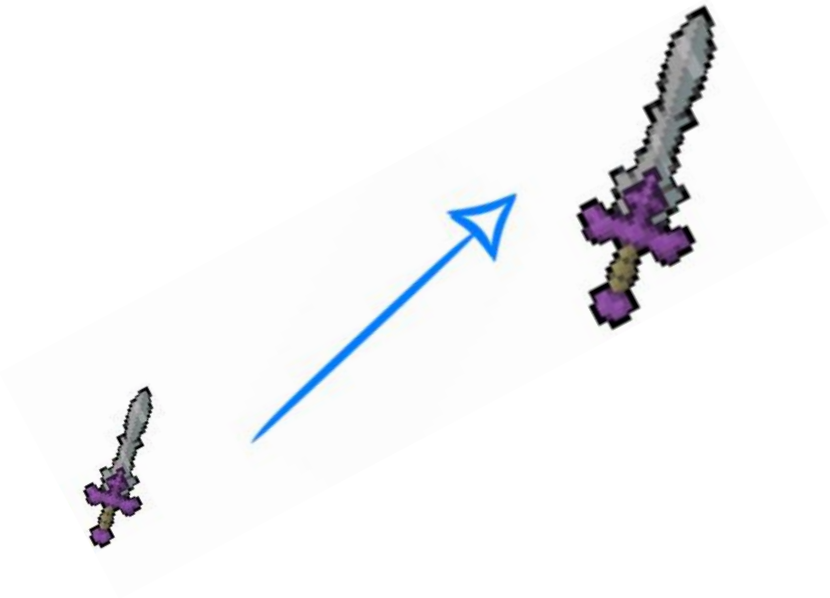
Ingeniería de los computadores

Práctica 3

Paralelismo en sistemas de memoria distribuida



Autores:

Juan López Quiles
Óscar Falcó Herrera
Manuel García Cremades
Alejandro Jesús Aliaga Hyder

Grupo 3 de prácticas
Martes 13:00 - 15:00
Laboratorio 22

Índice

1. Introducción

La práctica 3 nos propone la paralelización de una aplicación en un sistema de memoria distribuida utilizando técnicas de paso de mensaje, se ha considerado por parte de la práctica que sea realizado mediante el uso cuatro equipos. Para ello se realizará la paralelización a nivel de proceso de la aplicación realizada en la práctica anterior. Con este objetivo hemos realizado un análisis del problema a paralelizar y hemos obtenido las partes de mayor peso computacional, partes que serán el objetivo a paralelizar. Una vez obtenidas las partes a paralelizar haremos uso de la herramienta **MPI** para adaptar el algoritmo.

Como es evidente, los cambios introducidos en la aplicación han producido cambios en el tipo de comunicación y el estilo de programación paralela usada en la práctica, en este caso, hemos usado un tipo de comunicación todos-a-uno (uno-a-todos), razonando que uno de los equipo será el que se comunicara con el resto y posteriormente recibirá el resultado individual de cada uno y un estilo de programación por paso de mensajes como explicaremos más adelante con el uso de la herramienta **MPI**. También trataremos posteriormente el cambio más importante: el uso de un tipo de estructura Dueño-esclavo que, tras el análisis realizado, hemos visto que era el que mejor se adaptaba para la resolución del problema.

Para finalizar, una vez obtenido el algoritmo, se han realizado una serie de pruebas para poder obtener una serie de resultados con los cuales poder comparar con los resultados obtenidos en prácticas anteriores (algoritmo sin paralelizar y paralelizado a nivel de hilo) y así razonar las conclusiones obtenidas en el ejercicio.

2. MPI

MPI o “**interfaz de paso de mensajes**” es una librería que combina recursos de otros muchos proyectos como FT-MPI, LA-MPI, LAM/MPI y PACX-MPI. Dado que MPI ha sido implementado para casi toda arquitectura de memoria distribuida se convierte en el más rápido (por su optimización) y portable entre otras bibliotecas.
MPI está diseñada para ser usada en programas que exploten múltiples procesadores y su principal ventaja reside en que no requiere del uso de memoria compartida, lo que la convierte en la herramienta ideal en la programación de sistemas distribuidos como es nuestro caso. **MPI** también es usada para la aportación de sincronización entre procesos, dentro de los tipos de paso de mensajes podemos hablar por tanto de mensajes síncronos o asíncronos según si se requiere o no espera entre mensajes. La espera de mensajes por tanto puede conllevar el bloqueo del programa. Un ejemplo de uso de mensajes síncronos es la típica estructura cliente/servidor.

La Interfaz de Paso de Mensajes, por tanto, es un protocolo usado para la comunicación entre computadoras y un estándar para la comunicación entre diversos equipos que usan un mismo programa.
Los lenguajes de programación que usa son C, C++, Fortran y Ada.

A continuación vamos a comentar algunas de las llamadas usadas por nuestro programa de la biblioteca MPI (**#include <mpi.h>**) con su función general.

MPI_Init: Esta función inicializa todas las estructuras de datos necesarias para permitir la comunicación entre procesos basados en el envío de mensajes MPI.

MPI_Comm_size: Esta función devuelve el tamaño del comunicador seleccionado.

MPI_Comm_rank: Esta función devuelve el rango del proceso que lo llama.

MPI_Finalize: Finaliza la comunicación paralela entre los procesos.

MPI_Send: Realiza el envío de un mensaje de un proceso fuente a otro destino.

MPI_Recv: Esta función bloquea el proceso hasta que se reciba un mensaje con las características especificadas.

MPI_Abort: Usado para finalizar cuando se produce un proceso fallido.

También se han requerido el uso de algunas constantes propias de **MPI**:

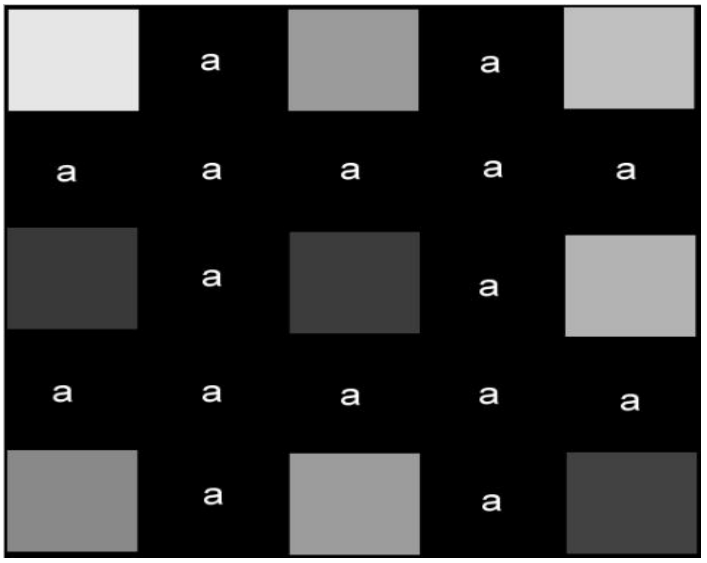
MPI_Status, **MPI_COMM_WORLD** (Identificador del comunicador al que pertenecen todos los procesos de una ejecución MPI)**, MPI_INT** (tipo int) y **MPI_DOUBLE** (tipo double)**,**

3. Análisis del problema a paralelizar

El problema propuesto para su paralelización, en este caso a nivel de proceso, es la interpolación de imágenes del mismo modo que fue en la práctica anterior y con la cual se comparan los resultados.

Como ya explicamos en prácticas anteriores la interpolación bilineal consiste en la interpolación del valor de los colores de los cuatro pixeles adyacentes a los nuevos pixeles a generar en el aumento del tamaño una imagen, para poder conocer el color de estos. Según la posición y disposición de los pixeles conocidos, los nuevos pixeles pueden usar tanto los ejes diagonales como los ejes-xy. Dado su funcionamiento podemos observar que es una aplicación paralelizable al consistir básicamente en el cálculo del valor del color de todos los nuevos píxeles generados.

La siguiente imagen muestra los nuevos pixeles a calcular como **a** y nos sirve para mostrar el proceso:

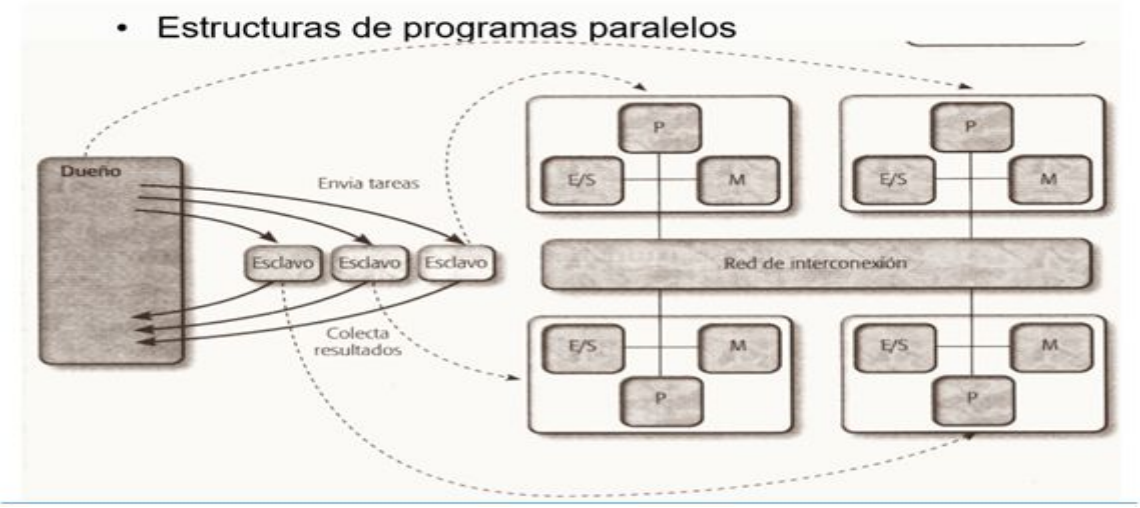


En esta práctica realizaremos su paralelización a nivel de proceso para lo que necesitaremos distribuir las partes de mayor peso computacional entre los equipos usados. Después de observar nuestra aplicación, se concluye que la parte con mayor peso computacional, y por tanto la que se realizará entre los equipos usados, es el cálculo de los nuevos píxeles generados al reescalar la imagen.

Para el distribuir la carga de trabajo entre los equipos y que estos solo realicen una porción del cálculo de los pixeles de la imagen a paralelizar se ha procedido al uso de una estructura del tipo Dueño-esclavo.

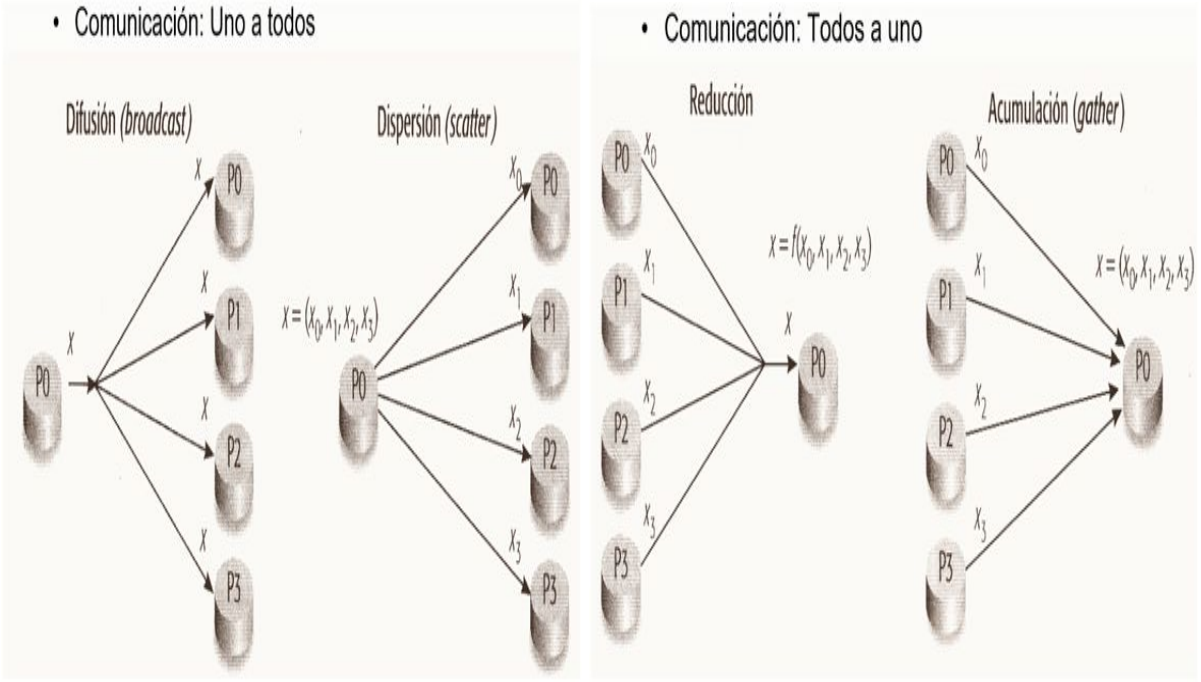
En él, el proceso dueño distribuye las tareas a los procesos esclavos y toma de ellos los resultados con los que el proceso dueño dará el resultado final. El reparto de estas tareas se puede producir de forma dinámica (en ejecución) o estática (realizada por el programador).

En nuestro caso somos nosotros los que hemos distribuido la carga y lo que distribuimos es el fragmento de los nuevos pixeles de la imagen a calcular y una vez el dueño haya recibido los resultados el mismo procederá a realizar el resultado final, es decir, la nueva imagen reescalada.



También podemos añadir que mediante el uso de **MPI** hemos modificado el estilo de programación paralela a una de **paso de mensajes,** caracterizada en que cada procesador tiene un espacio de direcciones propio.

En cuanto al tipo de comunicación podemos decir que hemos usado un tipo de comunicación **todos-a-uno** (**uno-a-todos**) de dispersión. Podemos observar esto en la manera en la que el dueño envia a cada equipo un mensaje diferente (cada equipo calcula un fragmento de la imagen distinto) y después de que cada esclavo acabe sus cálculos todos ellos envían el resultado al dueño el cual se encarga de darnos el resultado final.



4.Implementación

En esta práctica, la idea la teníamos clara, dado que era una modificación de la práctica anterior, pero pasar de la paralelización a nivel de datos, a la paralelización a nivel de proceso. Cambiar la forma de hacerlo, pero con la misma premisa.
Después de debatir, el grupo decidió que la estructura para realizarlo sería: Master-Slave.

Master-Slave

La estructura Master-Slave se define como una estructura en la cual el ordenador “Master” va a dividir los datos y enviarlos a los ordenadores “esclavos” o “slave” en inglés. En ellos, El Máster recibirá una imagen y pasará parte de esa imagen a los esclavos. Los esclavos realizan el trabajo de interpolación de la imagen y envían el resultado final al Master. Todos los esclavos realizarán el mismo trabajo pero con diferentes datos. Finalmente, el master recibe la información de todos los slaves y la recopila y construye el resultado final, que en nuestro caso, será la imagen anteriormente pasada al master, pero de mayor tamaño.

4.1 Cambios realizados para paralelizar el algoritmo

Los cambios más importantes en esta práctica ha sido el cambio de estructura, dado que dió lugar a debate y una modificación en la implementación de nuestro programa. Con lo cual, la implantación del Master controlando el vio y la recepción de la información se podría señalar como uno de los grandes cambios realizados. Dado que tuvimos que definir cómo implementamos su comportamiento, aunque la idea de que haría la teníamos clara.
Más entrando en el código realizado, uno de los cambios realizados fue el cambio del paso de información entre las máquinas. Para ello utilizamos las funciones Recv y Sendv.

```
MPI_Recv(&offset, 1, MPI_INT, source, mtype, MPI_COMM_WORLD, &status);
MPI_Send(&offset, 1, MPI_INT, MASTER, mtype, MPI_COMM_WORLD);
```

Por lo demás, la verdad, después de deducir como hariamos el master, el código del slave es bastante parecido al codigo main de la práctica anterior, con lo cual no tuvimos demasiados problemas en su implementación.

5. Análisis de rendimiento

A continuación, vamos a estudiar de nuevo la ganancia del problema defendido tras ser paralelizar a nivel de proceso. Una vez expuesta la mejora de rendimiento, esta será evaluada aplicando los parámetros de compilación dados por el compilador g++. Para que la comparación sea lo más significativa posible, se usarán los mismos flags empleados en la anterior entrega.

5.1. Teórica

Antes de comenzar a paralelizar el problema hemos estimado la ganancia al paralelizar a nivel de proceso para saber si es rentable la implementación. Al paralelizar a nivel de proceso hemos usado 4 ordenadores, como los datos a operar son una imagen que usa el formato RGBA, podemos enviar a cada ordenador una de las componentes. Por tanto en vez de interpolar de manera simultánea varios píxeles como hicimos en la versión a nivel de thread, se opera de manera simultánea las componentes de cada pixel.

La forma en la que calculamos la ganancia es la siguiente:

$$Ganancia = \frac{Tiempo\ sin\ mejora}{Tiempo\ con\ mejora}$$

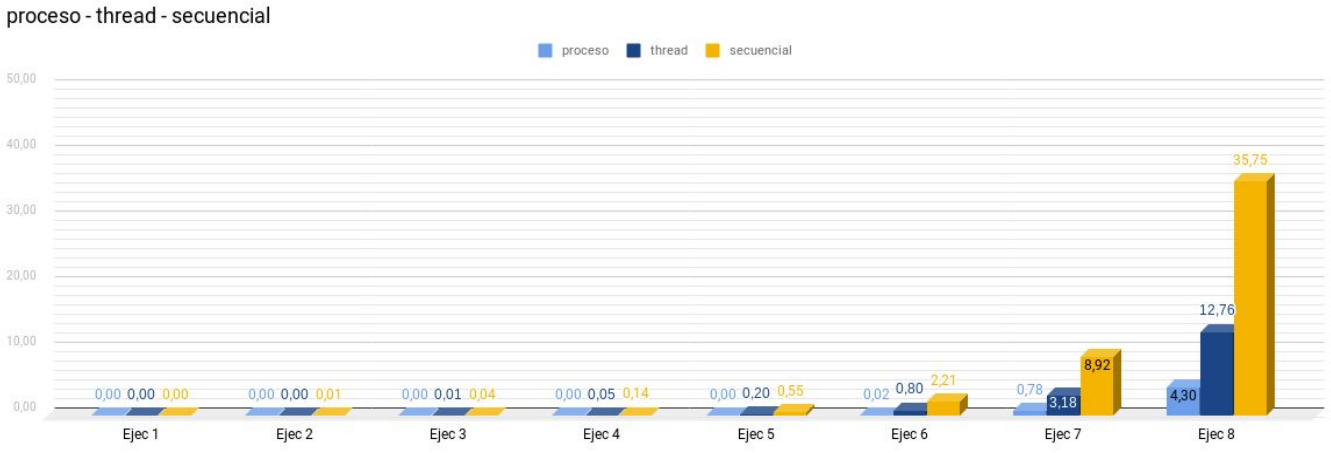
Como vamos a usar 4 ordenadores en vez de 1, teóricamente debemos tener una ganancia de 4. Siendo el cálculo el siguiente:

Tiempo con mejora = ¼ Tiempo sin mejora

Ganancia = Tiempo sin mejora / (¼ Tiempo sin mejora) = 4 veces

5.2. Resultados obtenidos tras paralelizar a nivel de proceso y thread

A continuación se van a comparar los datos obtenidos durante las tres entregas, respecto a la versiones sin optimizar de la secuencial, thread y proceso.



Como podemos ver en la gráfica, los tiempos de la versión de proceso no es posible estimarlos hasta la sexta ejecución (doblado 6 veces el tamaño de la imagen original) debido a la rapidez en que se ejecutan.

En la última ejecución podemos ver una aumento considerable en la versión por proceso. Aunque no se produce una mejora tan grande como la calculada en la ganancia teórica esta no se queda lejos.

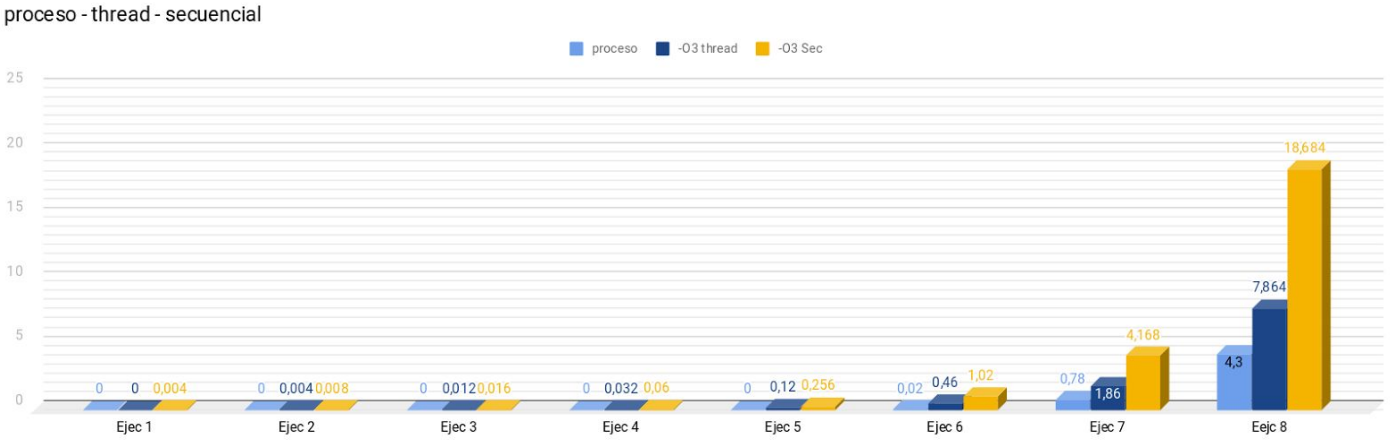
El cálculo de la ganancia respecto a las versiones secuencial y thread es la siguiente:

- Respecto a la secuencial:
Ganancia = 35.75 / 4.30 = 26,68

- Respecto a la thread:
Ganancia = 12,76 / 4.30 = 2,967

5.3. Comparación con la versión secuencial y thread optimizadas

Los resultados obtenidos tras comparar la versión de proceso con la secuencial y thread optimizadas son lo siguientes:



El cálculo de la ganancia respecto a las versiones secuencial y thread optimizadas es la siguiente:

- Respecto a la secuencial:
Ganancia = 18.684 / 4.30 = 4.34

- Respecto a la thread:
Ganancia = 7.864 / 4.30 = 1.828

6. Problemas encontrados durante la paralelización a nivel de proceso

Durante esta práctica, encontramos problemas de diferente índole, que resumimos a continuación:

-Problemas a la hora de estructurar esta práctica: Tuvimos que debatir largo y tendido sobre cómo implementamos este problema que se nos presenta. Probamos con varias estructuras pero la estructura que más nos convenció fue la que finalmente elegimos: Master-Slave.

-Problema al deducir el funcionamiento de MPI: Tuvimos que buscar ejemplos de funcionamiento de MPI en los cuales probariamos e iríamos aprendiendo sus funcionalidades y facilidades.

- Problemas al realizar el Master: No fue un gran desafío, pero dió para discutir el como realizaremos el Master, que sería quien controlaría la división de la información y su posterior unión.

- Problemas para implementar totalmente la práctica: En esta práctica hemos tenido dificultades, dado que con MPI no hemos podido utilizar imágenes de gran tamaño, dado que se saturaba la red y no conseguimos que funcionase nuestro programa. En nuestras pruebas, al comenzar con imágenes pequeñas no tuvimos problemas pero llegado a un tamaño algo grande, ya se satura la red. Igualmente, hemos conseguido comprobar la mejora sin problema

7. Conclusión

Durante la realización de las prácticas, hemos buscado un problema de alta carga computacional que hemos abordado con distintas técnicas de programación para aprovechar al máximo los recursos de las computadoras donde se ejecute.

En la primera entrega tras realizar la implementación secuencial del problema, nos hemos dado cuenta de lo ineficiente que es no paralelizar problemas con una gran número de operaciones, ya que estos provocan esperar minutos para ejecutar nuestro programa

Esto nos ha llevado a la necesidad de comprender conceptos como el hyper threading, thread, proceso, la librería mpi…

Una vez paralelizado el programa tanto a nivel de thread como proceso, como hemos comprobado una gran mejora del rendimiento al ejecutar el programa.

Por tanto tras finalizar las prácticas, hemos comprendido la importancia de aprender a paralelizar nuestros algoritmos cuando sea necesario.

8. Bibliografía

https://lsi.ugr.es/jmantas/ppr/ayuda/mpi_ayuda.php

http://silveiraneto.net/estudos/mpi-client-server/