PROGRAMACIÓN AVANZADA - PARALELISMO

Máster en Computación Gráfica, Realidad Virtual y Simulación

LUIS MIGUEL PINTO MAGUIÑA

ÍNDICE

1. Parte I

Introducción

La primera parte de la entrega supone realizar una multiplicación de matrices. Estas matrices se crean con valores 2 y 3. Posteriormente, se deberán implementar distintos algoritmos para realizar el trabajo mediante paralelismo de distintas formas. Tras esto, comparamos los resultados de la ejecución en paralelo con la ejecución en serial y hacemos un análisis de estos.

Resultado

Los siguientes resultados se corresponden a matrices cuadradas de 1000 x 1000. En caso de querer modificar estos resultados, se debe abrir el fichero main.cpp y modificar las líneas 302 y 303.

En primer lugar, la ejecución en serial nos devuelve un tiempo medio de 8,3 segundos.

Por otro lado, los diversos modos de paralelización devuelven los siguientes resultados medios:

- Paralelo con collapse: 2,565s
- Paralelo simple: 2.554s
- Data Race (critical): 43.236s (media de 3 iteraciones)
- Data Race (atomic): 2.368s
- Data Race (Reduction): 1.878s

Para el caso de critical se han utilizado 3 iteraciones para obtener la media de tiempo debido al gran tiempo que consume para ejecutarse.

Análisis

Como se puede observar, los tiempos paralelos son una mejora bastante grande en relación a la ejecución secuencial. De estos, el mejor ha resultado ser el método de reduction, ya que facilita la realización de la suma constante de los valores guardando copias locales de las variables las cuales se combinan al final. Es muy buena opción en este caso ya que disponemos de una operación conmutativa.

Pero, por otro lado, el caso de ejecución paralela usando critical, arroja un tiempo medio de 44s. Es lógico recibir estos resultados ya que, si todos los hilos intentan acceder a la misma variable para realizar cambios y, este método obliga a esperar al resto de hilos a que esperen al que esta ejecutándose en ese momento (en esa parte del código delimitada por critical), se entiende que el tiempo sea mucho mayor. Ya que, realmente en este caso, no se produciría ningún tipo de paralelismo.

2. Parte II

Introducción

En esta parte, el objetivo es rellenar una matriz con números aleatorios y obtener el número más grande dentro de esta. Se debe hacer por una parte de forma secuencial y, compararlo posteriormente con el algoritmo de paralelización que he escogido. Ambos algoritmos deben devolver el mismo resultado.

Como en el resultado anterior, el mejor de todos la obtuvo la cláusula reduction, en esta osación, he decidido paralelizar los bucles for anidados con la reduction(max). De esta forma, podemos calcular deforma conmutativa (igual que antes) todos los valores de la matriz obteniendo el máximo.

Las pruebas se han hecho con matrices de 5x5, para observar el correcto funcionamiento del algoritmo. Para cambiar esto, acceder al fichero main.cpp y cambiar las líneas 305 y 306.

Para las pruebas de redimiento, se han utilizado matrices de 10000 x 10000.

Resultados

Pruebas 5x5

- Secuencial: 0s
- Paralelo: 0s

Pruebas 10000x10000

- Secuencial: 0,2s
- Paralelo: 0,05s

Análisis

Los tiempos de las pruebas 5x5 no son nada descriptivos por lo que se van a obviar, su utilidad es simplemente para asegurar que, en miles de ejecuciones, siempre devolverían los mismos resultados.

Respecto a los tiempos de la matriz grande, se puede observar que el tiempo mejora considerablemente. Esto se debe a que, en cada hilo, se creará una variable destinada a encontrar el número máximo y, al final del todo, se combinan quedándose con el máximo total de la matriz.

Cabe resaltar que antes de este método, decidí usar una claúsula critical para que cada hilo tuviese acceso a una variable global y fuesen pisando esta para obtener el número máximo. Sin esta variable, el tiempo es similar al secuencial, pero hay momentos en los que falla puesto que el número máximo entre el secuencial y el serial no es el mismo. Y, usando critical, se obtenían tiempos de 4s. Algo mucho mayor al tiempo en secuencial.

Implementaciones adicionales

Se dispone de un bucle while, el cual se repite constantemente hasta que los resultados devueltos por ambos algoritmos sean diferentes (lo que supone que funciona mal). Para hacer uso de este, descomentar las líneas 165, 207 y 219. Además de poner las líneas 305 y 306 con valores muy pequeños (por ejemplo 5).

3. Parte III

Objetivo del programa

El objetivo principal del programa elegido es ordenar de mayor a menor los elementos aleatorios de una matriz NxN.

Partes del programa donde es posible usar paralelización

El algoritmo es muy simple. Consta de 4 bucles for anidados y luego, un condicional para conocer si el número de esa iteración es mayor a alguna de las posiciones de la matriz. De esta forma, guarda primero los números más altos en las primeras posiciones, dejando los más pequeños para las últimas.

Las partes del programa donde se puede aplicar paralelismo serán los bucles anidados.

¿Cómo se convertirá a paralelo?

Como he explicado en el punto anterior, la forma de paralelizar este algoritmo, será paralelizando los 4 bucles for anidados y utilizando la clausula "collapse(4)". De forma que se utilice un único bucle for de NxNxNxN. Los posibles problemas podrían ser el continuo acceso por todos los hilos. Pero en este caso no se da esta situación, por lo que no es necesario controlar el acceso por separado de los hilos a las variables.

Medir tiempos y comparar resultados

Utilizando una matriz de 1000x1000, obtenemos los siguientes tiempos:

- Ordenado(serial) en: 3.762s
- Ordenado(paralelo) en: 1.053s

Se puede observar que el tiempo de ejecución disminuye considerablemente gracias al algoritmo paralelizado.

Justificación

Como se ha explicado, el "punto débil" de este algoritmo está en la parte de hacer 4 bucles anidados de 1000 iteraciones cada uno. Por lo que, al paralelizar estos 4 bucles, se consigue una mejora en el rendimiento del algoritmo.