INGENIERÍA INFORMÁTICA
Escuela Politécnica Superior
Universidad Autónoma De Madrid

Paralelismo y arquitecturas modernas

Práctica 4

David Teófilo Garitagoitia Romero
Daniel Cerrato Sánchez

Pareja 3 Grupo 1322
12/17/2021

Índice de Contenidos

1. Ejercicio 0: Información sobre la topología del sistema

Se nos pide en primer lugar obtener la información relativa a la arquitectura de la máquina sobre la que estamos trabajando.

Para ello ejecutamos los comandos:

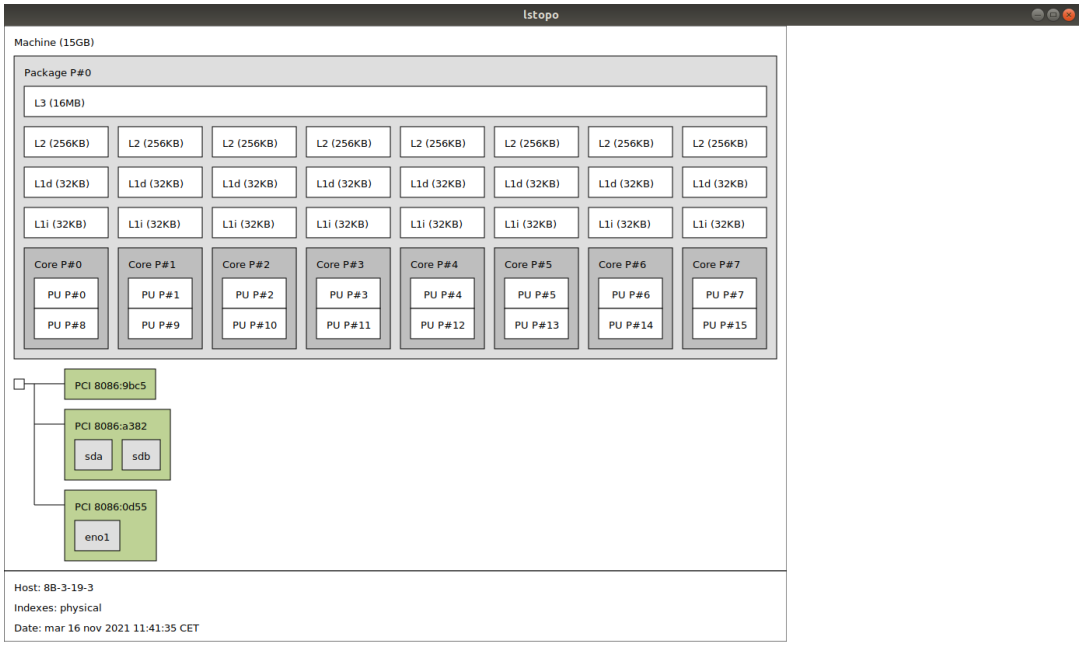
- Cat /proc/cpuinfo

Tras ejecutar el comando podemos desglosar la siguiente información:

- Dispositivo: Ordenadores de laboratorios
- Cantidad de CPUs = 8 fisicos
- Hyperthreading activo -> 16 CPUs virtuales
- Frecuencias CPUs = 3957.377 MHz

En cuanto al cluster, aparece el valor de cpu_cores a 4 mientras que siblings muestra el valor 8, por ende, el hyperthreading está activo.

También adjuntamos la imagen resultada de ejecutar lstopo



2. Ejercicio 1: Programas básicos de OpenMP

Este ejercicio servirá como toma de contacto con programas basados en OpenMP.

El primer paso será compilar el programa omp1.c para ellos empleamos el siguiente comando:

Gcc -g -Wall -D_GNU_SOURCE -o omp1 omp1.c -lgomp -lm

Tras compilar podemos pasar a dar respuesta a la primera pregunta;

¿Se pueden lanzar más threads que cores tenga el sistema y tiene sentido hacerlo?

Sí, pero esto no tiene sentido ya que cada core puede manejar únicamente un hilo a la vez y, por tanto, los hilos de superiores al número de cores (x si no se tiene activado el hyperthreading, 2x en caso contrario) deberán quedarse en espera a que los anteriores terminen para poder empezar su ejecución sin lograr por tanto el ansiado paralelismo.

De esta forma damos respuesta a la siguiente pregunta ¿Cuántos threads debería utilizar en los ordenadores del laboratorio? ¿y en el clúster? ¿y en su propio equipo?

La respuesta será tantos hilos como cores tenga el equipo, en caso de los ordenadores del laboratorio, como máximo 16 (8 cores físicos con el hyperthreading activo), esos mismo 16 para el cluster y en mi caso personal 8 al no tener el hyperthreading activo y tener 8 cores físicos.

Para la siguiente pregunta se nos pide deducir la prioridad entre las formas de indicar el número de threads que se lanzan en la región paralela, para ello se ha procedido a ejecutar múltiples veces el programa con valores contradictorios entre los threads especificados por la variable de entorno OMP_NUM_THREADS, la función omp_set_num_threads(int num_threads); y la cláusula #pragma omp parallel num_threads(numthr)

De estas ejecuciones se llega a la siguiente conclusión:

La sentencia "#pragma omp parallel private" específica de una región paralela, tiene prioridad sobre la función de OpenMP "omp_set_num_threads" que es global para todo el programa y que tiene prioridad sobre variable de entorno "OMP_NUM_THREADS" general del sistema.

Para la siguiente parte del ejercicio compilaremos omp2.c ya que es requerido para responder a las siguientes cuestiones;

1.4 ¿Cómo se comporta OpenMP cuando declaramos una variable privada?

1.5 ¿Qué ocurre con el valor de una variable privada al comenzar a ejecutarse la región paralela?

1.6 ¿Qué ocurre con el valor de una variable privada al finalizar la región paralela?

Cuando declaramos una variable como privada dentro de OpenMP se crean tantas copias como threads las cuales no estarán inicializadas al inicio de ejecutarse la región paralela ni

mantendrán valores al finalizar el hilo ya que se destruyen al finalizar la ejecución de los threads.

1.7 ¿Ocurre lo mismo con las variables públicas?

Las variables publicas son compartidas por todos los threads.

Sólo existe una copia, y todos los threads acceden y modifican dicha copia por lo que su valor final será el de la última ejecución y lo que hace un hilo sobre dicha variable afecta y repercute en los demás.

3. Ejercicio 2: Paralelizar el producto escalar

Para este ejercicio deberemos tomar la versión serie del programa del producto escalar.

En el programa podemos ver como se llama a generateVectorOne, que devuelve un vector del tamaño pasado con todos sus campos inicializados a uno.

Por tanto, el producto escalar será simplemente el tamaño del vector, pues la suma es como sum = sum + 1*1

Evidentemente, al ser serie, cuanto menor sea el tamaño, menos tiempo de ejecución necesita.

Para el siguiente apartado nos piden emplear el código paralelizado y contestar a las preguntas:

¿Es correcto el resultado? ¿Qué puede estar pasando?

Los resultados no son correctos. Los hilos están leyendo el valor de la suma de forma errónea, puesto que la paralelización hace que se lea la misma variable concurrentemente y todos los hilos trabajen con el mismo valor de suma escribiendo todos los hilos en ese valor sin tener ningún control del acceso lo que produce problemas de datarace, es decir, no se actualiza correctamente.

Para resolver el problema primero emplearemos la directiva #pragma omp critical

Esta directiva define una única sección crítica para todo el programa, la cual restring la ejecución del bloque a un único thread, de esta forma se suprime el problema del datarace al controlar que los threads solo podrán acceder de uno en uno a la variable compartida.

Por otro lado, la directiva #pragma omp atomic el cual es un caso particular de sección crítica, en el que se efectúa una operación RMW atómica sencilla, es decir asegura la actualización atómica de una localización de memoria, no múltiple por parte de los threads.

Para este caso, es preferible emplear atomic al no ser necesario definir una sección crítica de varias líneas y siendo está la opción con mejor rendimiento y óptima.

El siguiente apartado nos solicita emplear la directiva #pragma omp parallel for reduction y comparar con el punto anterior para ver cuál será la

Las operaciones de reducción utilizan variables a las que acceden todos los procesos y sobre las que se efectúa alguna operación de “acumulación” en modo atómico.

Caso típico: la suma de los elementos de un vector.

El control de la operación se deja en manos de OpenMP, si se declaran la variable de tipo reduction; es preferible el uso de reducciones ya que el uso de atomics tiene el precio de sincronización para evitar condiciones de carrera

El siguiente apartado, nos pide analizar el tiempo de ejecución ya que para entradas pequeñas no compensa el uso de paralelización debido al coste añadido u overhead de lanzar los hilos.

Para que el programa sea eficiente OpenMP nos permite en los pragma parallel for o parallel incluir una directiva para paralelizar solo bajo una condición como se ejemplifica en el siguiente fragmento de código. #pragma omp parallel if (M>threshold)

Para este apartado se nos pide estimar la cantidad de threshold o umbral haciendo un recorrido iterativo para encontrar el punto a partir del cual compensa paralelizar para ellos;

Se partirá de un valor inicial a criterio de los alumnos y siempre suficientemente grande para que los resultados de tiempos sean significativos y se irá subiendo o bajando hasta alcanzar el punto deseado. Ya que puede haber cierta inestabilidad numérica, diremos que un threshold T es válido si se cumplen las dos siguientes condiciones: 1. El tiempo medio de ejecución del programa serie es menor que el del programa paralelo para vectores de tamaño aproximado ceil(0.8T). 2. El tiempo medio de ejecución del programa serie es mayor que el del programa paralelo para vectores de tamaño aproximado ceil(1.2T) .

Para buscar ese valor se ha creado el programa calculate_Thresholds.c

Este programa comprueba desde el valor 1, incrementando de 1 en 1 el valor de T hasta encontrar el primer valor que cumpla la premisa establecida para la búsqueda de Threshold

En este primero se hace 20 veces el cálculo en serie para T (para luego sacar la media de esas 20 ejecuciones que es con la que comprobaremos las ejecuciones), lo mismo para el paralelo para 0.8T y 1.2T después comprobamos que se cumplan los valores y retornamos el valor de thresholds;

La ejecución del programa en los PCs del laboratorio retorna como resultado que el valor de Thresholds es T=911

Por ejemplo, para mi PC se tiene como resultado que el Thresholds es 390.

Para valores inferiores como por ejemplo el 66, los resultados muestran (primero tiempo serie luego en paralelo hasta ceil(0.8*66) y luego paralelo hasta ceil(1.2*66)

Probando con 66

0. 000000 0.000625 0.000641

Vemos como en serie es muy superior a las otras dos versiones.

Para el valor del thresholds se tiene, sin embargo:

Probando con 390

0.000005 0.000031 0.000002

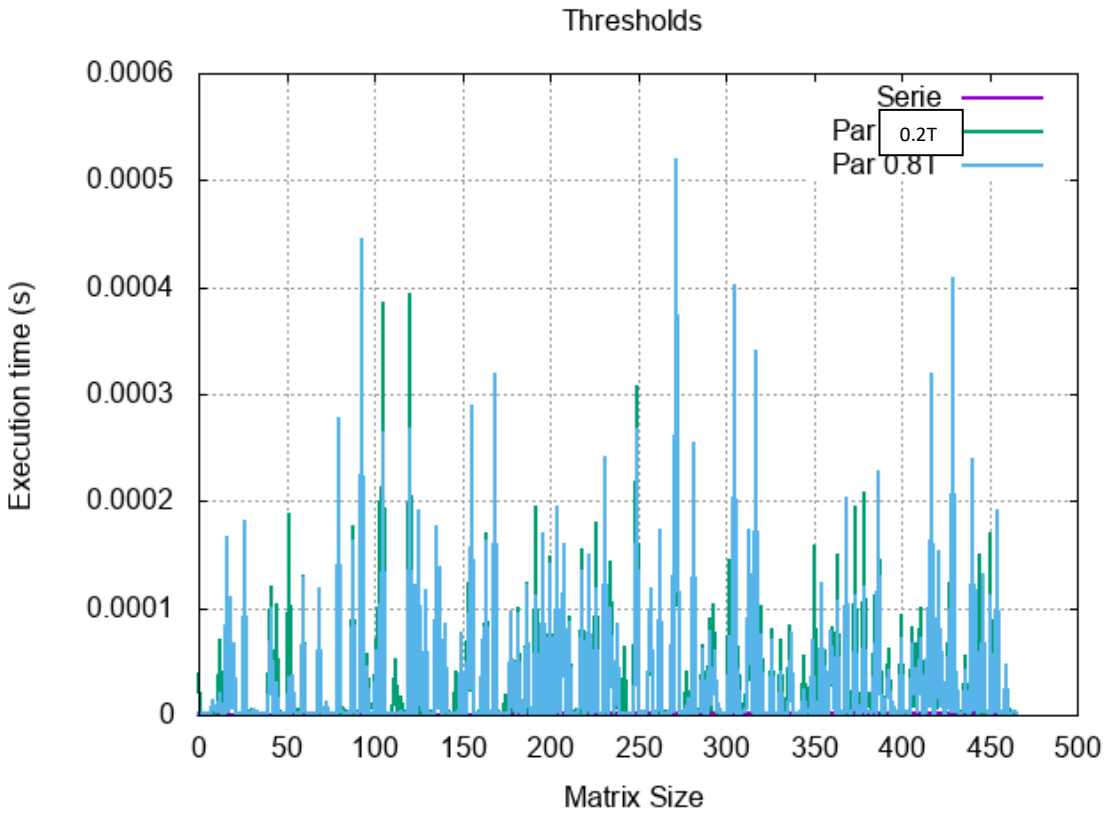
Por último, para valores superiores como el 10000 tenemos los siguientes resultados:

Probando con 10000

0.000036 0.000025 0.000022

Como vemos tanto para 0.8*10000 como para 1.2*10000 los resultados muestran como la versión paralela es superior en cuanto a rendimiento en comparación con la versión serie.

La gráfica obtenida no permite sacar conclusiones claras.



4. Ejercicio 3: Paralelizar la multiplicación de matrices

En este ejercicio se nos piden recuperar el código de la anterior práctica

Se pide desarrollar versiones paralelas y rellenar las tablas.

Para ello codificamos las tres versiones y declaramos que pueda recibir el número de hilos por argumento, de esta forma mediante un .sh, ejecutamos reiteradas veces cada versión para cada número de hilos que se nos pide rellenar en la tabla para así extraer la media y poder realizar la tabla y responder a los distintos apartados de este ejercicio.

Los resultados de la ejecución son los siguientes:

Tabla de tiempos y tabla de aceleración

N=1000

Version\#hilos	1	2	3	4
Serie	10.426	10.426	10.426	10.426
P-b1	10.3622	8.103533	6.1574	5.25168893
P-b2	10.795	9.231005	12.7342	11.765132
P-b3	11.752	10.69227	8.824	9.1

Version\#hilos	1	2	3	4
Serie	1	1	1	1
P-b1	1.00615	1,2865	1.69324	1,985
P-b2	0.96	1.15	0.8	0.88
P-b3	0.88716	0.975	1.1815	1.14

N=1400

Version\#hilos	1	2	3	4
Serie	67.49051	67.49051	67.49051	67.49051
P-b1	70.585	17.244248	17.098	14.995
P-b2	66.029784	40.18	43.1228	36.055
P-b3	66.54735	36.62613	35.60862	33.63478

Version\#hilos	1	2	3	4
Serie	1	1	1	1
P-b1	0,9561521	3,9137688	3,94678367	4,50083
P-b2	1,022111166	1,67969	1,56506	1,8718
P-b3	1,01416510196	1,84267	1,895327	2,0065

3.1 ¿Cuál de las tres versiones obtiene peor rendimiento? ¿A qué se debe? ¿Cuál de las tres versiones obtiene mejor rendimiento? ¿A qué se debe?

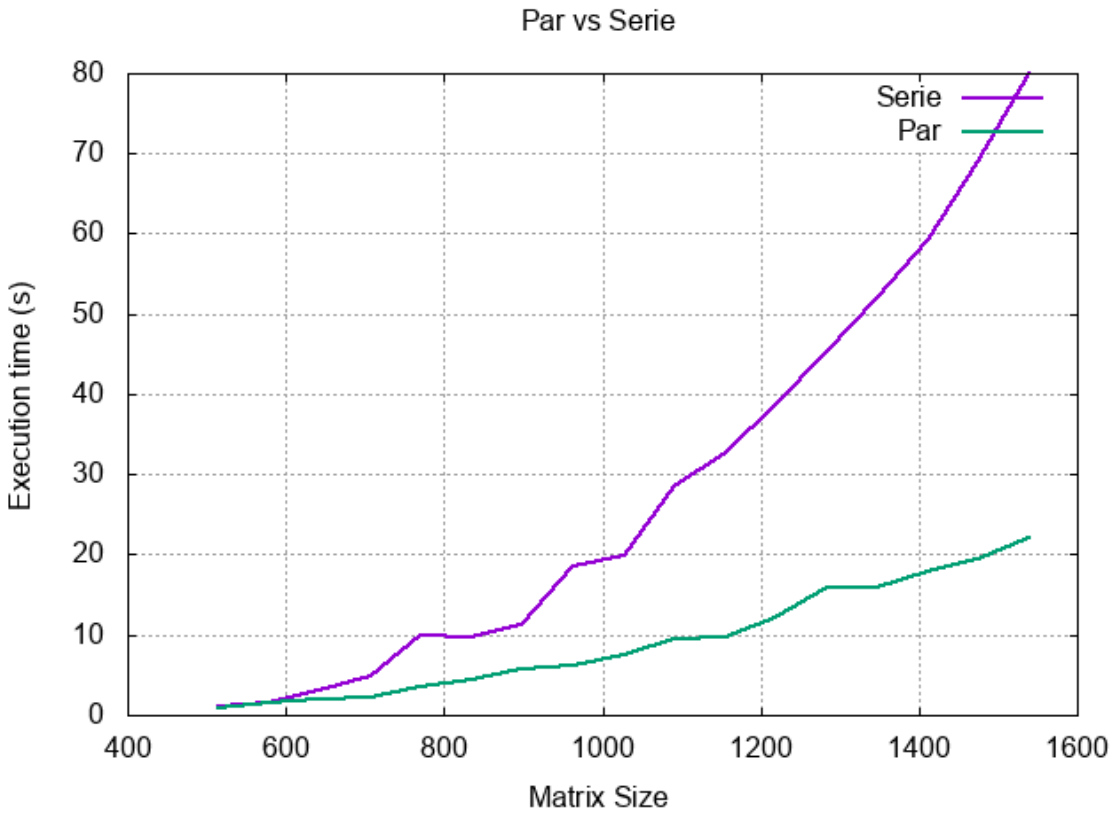
El peor rendimiento lo tiene el bucle con la paralelización en el bucle intermedio; Esto se debe a que el bucle se mueve por las columnas de las matrices 2 y 3, lo que hace que tarde más al tener que cambiar más de bloque tanto para lectura como, sobre todo, para escritura.

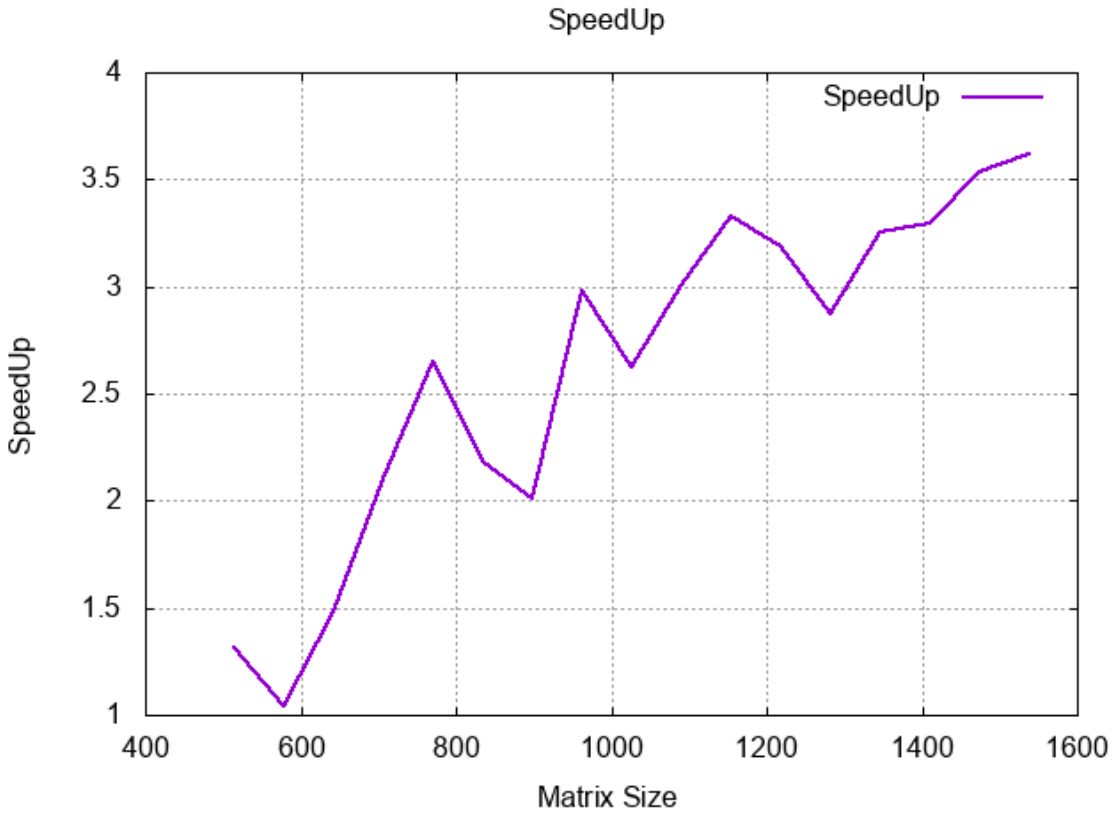
El mejor rendimiento lo tiene el bucle interno, pues los datos que va leyendo son las filas de la matriz 1, columnas de la matriz 2, pero la escritura la hace siempre sobre el mismo bloque, sin tener que moverse, esto la hace mucho más eficiente cuanto más grande sea la matriz.

3.2 En base a los resultados, ¿cree que es preferible la paralelización de grano fino (bucle más interno) o de grano grueso (bucle más externo) en otros algoritmos?

En base a los resultados, es preferible la paralelización de grano fino.

Gráficos obtenidos de los resultados (los resultados numéricos se guardan en los .dat):





Hay picos pero poco a poco se va quedando más estable alrededor del 3.5 de aceleración sobre todo pasados los 1400.

5. Ejercicio 4: Ejemplo de integración numérica

Para este ejercicio nos piden estudiar el false sharing en programas de integración para estimar el valor de pi.

Viendo el programa se observa cómo se emplean 100000000 rectángulos por lo que h será h=1/100000000=1e-08

Todos los programas retornan un resultado correcto a la operación de aproximar el valor de pi como se puede observar en la siguiente tabla donde se guardan el average time y pi de cada una de las versiones.

Con respecto a la pregunta de si en la versión pi_par2, ¿Tene sentido declarar sum como una variable privada y qué sucede cuando se declara una variable de tipo puntero como privada?

Si esta se declara como first ptivate el efecto es similar al de shared, es decir cada hilo tiene su propio puntero sum ya que se generan copias para cada thread pero esas copias apuntan a la mismo posición de memoria, por lo que en la práctica es lo mismo que emplear el shared.

En cambio, si declaras una variable de tipo puntero como privada, esta al ser privada no estará inicializada por lo que no apuntará a una posición valida de memoria originando segmentation fault.

Ejercicio 4.4: ¿Cuál es la diferencia entre las versiones pi_par5, pi_par3 y la versión pi_par1? Explique lo que es False sharing e indique qué versiones se ven afectados por ello (y en qué medida). ¿Por qué en pi_par3 se obtiene el tamaño de línea de caché?

Para responder a esta pregunta empezaremos con una breve introducción, los subprocesos no leen datos directamente de memoria principal, sino que se almacenan en caché, la unidad más pequeña es la línea de cache, por tanto, si múltiples hilos si requieren de modificar la misma línea de cache, las variables afectan inadvertidamente el rendimiento del otro.

Como hemos estudiado en la parte teórica de la asignatura, el fenómeno de false sharing es una situación que tiene lugar cuando los threads, aunque no estén accediendo a las mismas variables, comparten un bloque cache que contiene las diferentes variables.

Debido a los protocolos de coherencia cache, cuando un thread actualiza una variable en el bloque cache y otro thread quiere acceder a una variable diferente pero que está en el mismo bloque, este bloque antes de utilizarse se escribe en memoria.

Sabiendo esto, en pi_par1 también existe false sharing ya que todos acceden a una variable global (sum)

Por otro lado, en pi_par2 sí que se produce false sharing ya que al no calcularse el tamaño de línea de cache, puede darse la situación de que dos hilos tengan posiciones de memoria en la misma línea, lo que implica que cuando un hilo escriba producirá false sharing al otro

Por otro lado, en el programa 3, se calcula se obtiene el tamaño de línea de cache para que cada hilo disponga de su propia linea de caché y de esta forma no existe false sharing entre hilos

En el resto de las versiones, pi_par4 también hay false sharing, pi_par6 también tiene false sharing dado que tiene que escribir en memoria bloques a los que un hilo solicita acceder tras acceder otro hilo

Por ende, La diferencia entre los dos es que en pi_par5 y pi_par3 no existe false sharing (3 por lo explicado con anterioridad) y 5 no produce, ya que sum es privada y se emplea la directiva critical para pi lo que hace que solo puede acceder un hilo a la vez a la variable.

Relacionado con lo anterior, pasamos al siguiente punto explicando la directiva critical;

La directiva critical asegura la exclusión mutua del bloque, originando que se produzcan colas de espera en las que mientras un hilo accede a la variable, el resto deban esperar su turno lo que ralentiza los resultados sin embargo al no tener false sharing se equilibra con el resto de las versiones en cuanto a tiempos de ejecución.

Para el siguiente apartado ejecutaremos la versión pi_par6 para analizar resultados y definir que versión es la óptima;

Por un lado, pi_par6 la región paralela comienza con #pragma omp parallel default(shared) private(numThreads) y dentro de esta tiene #pragma omp for, es decir, cuando el hilo principal llega a la zona paralela se generan tantos hilos como se indiquen y seguidamente en #pragma omp for se dividen las iteraciones del for entre ellos.

Comparando con pi_par7 usa únicamente una directiva #pragma omp parallel for reduction(+: sum) private(i,x) default(shared).

Se lanzan tantos hilos como se haya indicado anteriormente, a estos se les asignan las iteraciones del for y se van añadiendo los resultados en la variable sum por reducción

Pi_par7 es más eficiente y esto se debe a la cláusula reduction(+: sum) que agiliza el proceso además de que pi_par6 cuenta con false sharing lo que hace más notable la diferencia.

Finalmente, los resultados de las ejecuciones muestran como pi_par4 es la que menor tiempo medio tiene seguida de cerca por pi_par7.

Esto es debido porque, a pesar de tener false sharing, este es mínimo en comparación a otros ya que cada hilo accede únicamente una vez a la variable compartida y en comparación con pi_par 7 al tener menos operaciones lógicas de multiplicaciones y un malloc menor es más rápido

6. Ejercicio 5: Optimización de programas de cálculo

Se nos pide para este ejercicio:

Compile y ejecute el programa usando como argumento una imagen de su elección. Examine los ficheros que se hayan generado y analice el programa entregado.

Al ejecutar el programa pasando por argumento la imagen SD.jpg se generan las imágenes SD_grey que es la imagen en tonos grises, también SD_grad que acentúa los bordes y finalmente SD_grad_denoised que es la misma que la anterior, pero reduciendo el ruido, se adjunta el resultado mostrado por terminal de la ejecución.

./exe/edgeDetector images/SD.jpg

[info] Processing ../images/SD.jpg

[info] ../images/SD: width=640, height=360, nchannels=3

[info] Using gaussian denoising...

Tiempo: 0.037119.

1->El programa incluye un bucle más externo que itera sobre los argumentos aplicando los algoritmos a cada uno de los argumentos (señalado como Bucle 0). ¿Es este bucle el óptimo para ser paralelizado? Responda a las siguientes cuestiones para complementar su respuesta. a. ¿Qué sucede si se pasan menos argumentos que número de cores? b. Suponga que va a procesar imágenes de un telescopio espacial que ocupan hasta 6GB cada una, ¿es la opción adecuada? Comente cuanta memoria consume cada hilo en función del tamaño en pixeles de la imagen de entrada

No es un bucle óptimo para paralelizar ya que es más adecuado procesar una imagen por vez y en cada imagen paralelizar los bucles del algoritmo

a) Si el número de imágenes es inferior al número de cores, habrá algún core que no esté ejecutando nada por lo que se perderá productividad al no paralelizar empleando el máximo útil y posible.
b) Como los hilos tienen memoria compartida, como cada hilo debería procesar una imagen, eso indica que cada imagen requiere de 6GB para poder tener toda la imagen, eso supone un consumo enorme de memoria principal, lo que reitera la respuesta anterior de que es mejor opción ir procesando imagen a imagen

2->Durante la práctica anterior, observamos que el orden de acceso a los datos es importante. ¿Hay algún bucle que esté accediendo en un orden subóptimo a los datos? Corríjalo en tal caso. a. Es imprescindible que el programa siga realizando el mismo algoritmo, por lo que solo se deberían realizar cambios en el programa que no cambien la salida. b. Explique por qué el orden no es el correcto en caso de cambiarlo.

Existen bucles no óptimos como puede ser el bucle del rgb en el que se accede por columnas y como ya vimos en la práctica anterior, es mejor cambiar el orden, bastaría con cambiar de pos las i y las j

Esto se debe al principio de localidad la cual es aprovechada por las cachés pues la memoria reservada para un vector es contigua en memoria.

Es por ello por lo que es mucho más eficiente acceder a los elementos en memoria siguiendo el orden en el que se encuentran en esta.

```
for (int i = 0; i < width; i++)
    {
       for (int j = 0; j < height; j++)
       {
          getRGB(rgb_image, width, height, 4, i, j, &r, &g, &b);
          grey_image[j * width + i] = (int)(0.2989 * r + 0.5870 * g + 0.1140 * b);
       }
    }
```

```
for (int j = 0; j < height; j++)
    {
       for (int i = 0; i < width; i++)
       {
          getRGB(rgb_image, width, height, 4, i, j, &r, &g, &b);
          grey_image[j * width + i] = (int)(0.2989 * r + 0.5870 * g + 0.1140 * b);
       }
    }
```

El resultado es correcto tras realizar los cambios ya que el resultado no dependía del orden

El orden si es correcto en el sentido de que el resultado es el apropiado, sin embargo, si se refiere al orden de ejecución, como ya hemos visto no al ser ineficiente ya que se accede por columnas

También en el segundo bucle identificado como ineficiente, dentro del bucle en grey_image[j * width + i] si incrementamos la i estaremos accediendo a posiciones adyacentes, pero si incrementa la j, al estar en un producto con width, no se acceden a posiciones contiguas lo que provocará más fallos de caché.

3-> Obviando el Bucle 0, pruebe diferentes paralelizaciones con OpenMP comentando cuales deberían obtener mejor rendimiento. a. Es imprescindible que el programa siga realizando el mismo algoritmo, por lo que solo se deberían realizar cambios en el programa que no cambien la salida. b. No es necesaria la exploración completa de todas las posibles paralelizaciones, es necesario utilizar los conocimientos obtenidos en la práctica para acotar cuales serían las mejores soluciones. Los razonamientos que utilice deben ser incluidos en la memoria.

Para este ejercicio se han ido probando diferentes opciones anotando resultados para decidir que opción es la mejor.

Como las imágenes de prueba son más anchas que altas, se ha preferido paralelizar la altura ya que de esta forma a cada hilo se le asignan menos iteraciones y es por ende más rápido

De esta forma paralelizamos los bucles internos manteniendo el orden de los bucles del apartado 5.2 , empleamos el first private ya que requerimos tener inicializados los valores de widht, r, g y b con los datos inicializados anteriormente y cuyos valores requerimos en el bucle y shared para grey_image al tener que ser compartido (también serviría un first private como vimos anteriormente)

Fichero	Average Timer Serie	Average time Par	SpeedUp	FPS Serie	FPS Par
SD.jpg	.057271	.038287	1.495834	17.460841	26.11
HD.jpg	.278379	.171925	1.619188	3.5922	5.814
FHD.jpg	.765384	.462956	1.653254	1.3065	2.16
4k.jpg	3.335933	1.590801	2.097014	0.299766	0.63
8k.jpg	12.365836	4.077042	3.033041	0.0809	0.2453

Por último, responderemos a las cuestiones de optimización en compilación.

Al utilizar la bandera -O3 lo tiempos se reducen un poco y mejoran con respecto al programa sin emplearla y también por ende mejorado la tasa de FPS notándose especialmente en las imágenes más pesadas compilando con esta no cambia la salida pero si el tiempo de ejecución, tras búsquedas por internet y en el manual, hemos entendido que la bandera se encarga de emplear vectorización (O2 lo hace y O3 es una versión superior) junto con otras técnicas como desenrollado de bucles que hace que disminuya el tiempo a costo de aumentar el tamaño del archivo.

Al principio, podemos ver que llega incluso a ser más rápido el programa serie, sin embargo, rápidamente se impone el programa paralelo llegando a lograr una mejora de 3 veces más velocidad.

Estos son nuevamente los datos obtenidos esta vez compilando tanto la versión serie como la versión paralelizada con -O3

Como se puede ver hay una mejoría muy notable para ambos programas, en especial para la versión serie alcanzando y superando los 30fps que buscábamos.

Fichero	Average Timer Serie	Average time Par	SpeedUp	FPS Serie	FPS Par
SD.jpg	.017077	.019610	.870831	58.56	50.9944
HD.jpg	.131333	.388431	.338111	7.61423252	2.57446
FHD.jpg	.375125	.558232	.671987	2.6657781	1.8
4k.jpg	2.009585	.932869	2.154198	0.49762	1.072
8k.jpg	9.067595	2.799062	3.239512	0.1103	0.357

[FINAL DE DOCUMENTO]