ARQO

Memoria Práctica 4

Roberto Martín Alonso

Diego Forte Jara

Pareja 11

ÍNDICE

Ejercicio 1……………………………………………………………………………………...…………4

Ejercicio 2……………………………………………………………………………….…….…………5

Ejercicio 3……………………………………………………………………………….……………….6

Ejercicio 4……………………………………………………………………………….……………….7

Ejercicio 5……………………………………………………………………………….……………….8

# Aclaraciones previas:

La práctica actual se va a desarrollar haciendo uso del subsistema de Windows para Linux (WSL), con las siguientes especificaciones:

Texto

Descripción generada automáticamente

Los ficheros entregados están distribuidos de la siguiente forma:

* En el directorio “Codigo” están todos los ficheros fuente base de los programas desarrollados en la práctica, ficheros .c, .h y makefile.
* En los directorios Ex (Donde x es el número de ejercicio) se encuentran los ficheros .c, .dat, .png y los scripts correspondientes a cada ejercicio. **Para ejecutar un script se debe pasar dicho script al interior del directorio “Codigo” junto a los ficheros .c (solo aplica para los ejercicios 4 y 5)**
* En el directorio “Docs” se encuentra la memoria de la práctica.

# Ejercicio 0: Información sobre la topología del sistema V

Tras ejecutar el comando “cat /proc/cpuinfo” por terminal y volcarlo al fichero de texto “cpuinfoE0.txt” se obtienen los siguientes resultados:

Texto

Descripción generada automáticamente

Aquí se puede apreciar que el número de cores físicos es 8 (campo cpu cores), el número de cores lógicos es 16 (campo siblings) y su frecuencia es 4200.16MHz (campo cpu MHz). Podemos concluir que el hyperthreading está activado en el sistema debido a que el número de cores lógicos es mayor que el de cores físicos.

NOTA: En la carpeta E0 de la entrega se adjunta el fichero “cpuinfoE0.txt” donde se puede ver toda la información completa tras ejecutar el comando “cat /proc/cpuinfo”

# Ejercicio 1: Programas básicos de OpenMP V

# 1.1-

El sistema en el que se realiza la práctica dispone de 16 cores lógicos (16 hilos máximo). Se prueba a lanzar el programa “omp1.c” con 32 hilos y el programa funciona, por lo que es posible lanzar más hilos que cores lógicos.

Con respecto a si tiene sentido hacerlo, en general no debido a que el sistema operativo debe gestionar el cambio de contexto entre hilos, lo que produce sobrecarga en el sistema, pero en situaciones donde se produzcan un gran número de bloqueos si puede resultar beneficioso debido a que mientras que haya hilos bloqueados puede haber otros trabajando.

# 1.2-

El número de hilos a utilizar depende de la tarea que se quiera realizar. Se podría comenzar usando el número máximo de hilos del sistema (6 para el ordenador del laboratorio y 16 hilos para el ordenador propio) y posteriormente experimentar con más hilos y medir el rendimiento para ver si hay mejoras.

# 1.3-

Se modifica el programa “omp1.c” tal y como se pide en el enunciado y se ejecuta. A partir de esta ejecución se deduce que la prioridad del número de hilos es, de forma descendente:

nº hilos cláusula -> nº hilos con función -> nº hilos variable de entorno

NOTA: Si no se indica número de hilos mediante cláusula o función, OpenMP utiliza el valor de la variable de entorno OMP\_NUM\_THREADS, por lo que para que se ejecute con el valor de dicha variable de entorno no se le asigna número de hilos en la ejecución con el valor de la variable de entorno

# 1.4-

Se ejecuta el programa “omp2”:

Texto

Descripción generada automáticamente

Cuando se declara una variable privada en OpenMP ésta no se comparte con el resto de hilos, como es el caso de “a”, donde cada hilo tiene dicha variable (sin inicializar) pero cada uno tiene un valor distinto para ella.

# 1.5-

Cuando comienza la ejecución paralela, se hace una copia independiente para cada hilo de las variables privadas. Dichas copias están sin inicializar, por lo que contienen valores indefinidos.

# 1.6-

Cuando finaliza la región paralela el valor de esta variable privada no se mantiene. Es similar a cuando se declaran variables locales en una función y esta retorna.

# 1.7-

En el caso de las variables públicas si conservan su valor al finalizar la región paralela debido a que forman parte de la memoria original que se ha compartido entre los hilos al iniciar la región paralela. Es similar a cuando se le pasa por referencia una variable a una función y esta retorna

# Ejercicio 2: Paralelizar el producto escalar V

# 2.1-

El resultado del programa “pescalar\_serie” es el tamaño “M” del vector que se le pase como argumento a la función “generateVectorOne”, que crea un vector de tamaño “M” y lo inicializa a “1s”. Es decir, si los vectores son de tamaño 1000, el resultado del producto escalar del programa será 1000.

# 2.2-

El resultado que arroja el programa “pescalar\_par1” no es correcto. Esto es debido a que como la variable “sum” es compartida, todos los hilos acceden a ella y se producen condiciones de carrera, por lo que en cada ejecución muestra un resultado distinto.

# 2.3-

Se crea el programa “pescalar\_par2” para obtener el resultado correcto. Esta condición de carrera puede resolverse mediante ambas directivas, los cambios realizados son los siguientes:

Pantalla de computadora con letras

Descripción generada automáticamente con confianza media

En este caso la solución elegida es el pragma atomic, ya que es solo una línea la que se quiere proteger. En caso de que hubiese más instrucciones susceptibles a sufrir condiciones de carrera se usaría el pragma critical para proteger todo el bloque entre llaves.

# 2.4-

Se crea el programa “pescalar\_par3” para obtener el resultado correcto mediante el uso del pragma “omp parallel for reduction”. Los cambios realizados son:

Texto

Descripción generada automáticamente

En este caso la opción elegida sería este último pragma ya que es el que tiene mayor afinidad con el problema que queremos resolver.

# 2.5 (2.6)-

Tras realizar varias ejecuciones del programa del producto escalar con distintos tamaños de vectores se encuentra un posible valor umbral de vector, el cual es 2750000. Para considerarlo válido se ejecuta el programa del producto escalar para tamaños de vector:

ceil(0.8 x 2750000) = 2200000, ceil(1.2 x 2750000) = 3300000

Texto

Descripción generada automáticamente

Se comprueba que para ceil(0.8 x 2750000) el tiempo del programa en serie es menor que el del programa en paralelo y para ceil(1.2 x 2750000) = 3300000 el tiempo del programa en paralelo es menor que el del programa en serie.

A continuación, se comprueba la estimación del umbral con tamaños por encima y por debajo del óptimo:

Texto

Descripción generada automáticamente

Como puede apreciarse, por encima del umbral el tiempo de la ejecución paralela es menor al tiempo de la ejecución en serie y por debajo del umbral el tiempo de la ejecución en serie es menor al tiempo de la ejecución en paralelo.

# Ejercicio 3: Paralelizar el producto escalar V

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Tiempos de ejecución (s) | | | | |
| Versión\nhilos | 1 | 2 | 3 | 4 |
| Serie | 45,098091 | 45,098091 | 45,098091 | 45,098091 |
| Paralela-bucle1 | 47,703853 | 80,219340 | 77,515044 | 84,054902 |
| Paralela-bucle2 | 48,639896 | 24,923234 | 17,541910 | 14,180161 |
| Paralela-bucle3 | 45,712199 | 21,847603 | 14,316261 | 10,480383 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Aceleración | | | | |
| Version\nhilos | 1 | 2 | 3 | 4 |
| Serie | 1 |  |  |  |
| Paralela-bucle1 | 0,945376278 | 0,56218477 | 0,58179792 | 0,53653136 |
| Paralela-bucle2 | 0,92718313 | 1,8094799 | 2,57087689 | 3,18036523 |
| Paralela-bucle3 | 0,986565774 | 2,06421231 | 3,15013054 | 4,3030957 |

Tablas obtenidas con una dimensión de matriz N = 1800.

Los datos de tiempos han sido obtenidos mediante la ejecución de un script creado para este propósito y la aceleración ha sido calculada en una hoja de Excel. Ambos ficheros (E3.xlsx y ej3.sh) se adjuntan en los ficheros entregados

# 3.1-

La versión que obtiene el peor rendimiento es la paralela del bucle 1 con 4 hilos. Esto se debe a que consume más tiempo en gestionar todo lo referente a los hilos que en computar los resultados. Como se abren 4 hilos en el bucle interno, en total los hilos que se abren son 1000x1000x4, es decir 4000000 hilos.

La versión que obtiene el mejor rendimiento es la paralela del bucle 3 con 4 hilos. Esto se debe a que se abren y se cierran menos hilos durante toda la ejecución del programa, por lo que se pierde menos tiempo en la gestión de los hilos. Como los hilos se abren en el bucle externo, el total de hilos abiertos es 4.

# 3.2-

Depende del algoritmo a implementas será preferible la paralelización de grano fino o de grano grueso.

Si el algoritmo no necesita demasiada coordinación entre hilos (como es el caso de la multiplicación de matrices) es preferible utilizar paralelismo de grano grueso, ya que así se reduce el coste de lanzar hilos (overhead) y cada hilo realiza una cantidad significativa de trabajo.

En cambio, si se necesita coordinación constante entre hilos (como en el caso de la suma de elementos de dos arrays en uno nuevo) será preferible utilizar paralelismo de grano fino, aumentando el coste de lanzar hilos.

# 3.3-

Se crea un script para automatizar la toma de tiempos (ej3.3.sh) y crear las gráficas con gnuplot, obteniéndose los siguientes resultados:

Gráfico, Gráfico de líneas

Descripción generada automáticamenteGráfico, Gráfico de líneas

Descripción generada automáticamente

En la gráfica del tiempo de ejecución se observa como el tiempo de ejecución del programa en serie aumenta mas rápido que el tiempo de ejecución del programa paralelo a medida que se va aumentando el tamaño N de la matriz.

En la gráfica de la aceleración se observa como la aceleración del programa paralelo con respecto al programa en serie está estable a medida que se aumenta el tamaño N de la matriz.

Como no se obtiene una gráfica cuya aceleración disminuya con el incremento del tamaño de la matriz se repiten las mediciones con tamaños mayores de N (N inicial = 512, N final = 4096 y pasos de 512), resultando:

Gráfico, Gráfico de líneas

Descripción generada automáticamenteGráfico, Gráfico de líneas

Descripción generada automáticamente

Ahora, a pesar de los picos que presenta la nueva gráfica de aceleración, se puede apreciar como esta tiene una tendencia descendente.

# Ejercicio 4: Pérdida de rendimiento por el falso compartir (False sharing) en OpenMP V

# 4.1-

En la versión del programa serie, el número de rectángulos que se dan son 100000000, por lo que el valor h (ancho del rectángulo) es de 1/100000000

# 4.2-

Se ejecuta el programa en serie y todos los paralelos mediante el script ej4.sh, obteniéndose los siguientes resultados:

Texto

Descripción generada automáticamente

Pasando los datos a tabla y calculando su aceleración, se obtiene:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Tiempos de ejecución (s) | | | | | | | | |
| Datos\Versión | serie | pi\_par1 | pi\_par2 | pi\_par3 | pi\_par4 | pi\_par5 | pi\_par6 | pi\_par7 |
| Tiempos (s) | 0,277151 | 0,254153 | 0,251406 | 0,040708 | 0,028056 | 0,033620 | 0,233791 | 0,027823 |
| Aceleración | 1,000000 | 1,090489 | 1,102404 | 6,808269 | 9,878493 | 8,243635 | 1,185465 | 9,961219 |

Como puede verse en la salida de las ejecuciones, los resultados de todos los programas son correctos

# 4.3-

No tiene sentido declarar la variable sum como privada ya que esta no se modifica, sino que se modifican los datos a los que apunta el puntero. Si una variable de tipo puntero se declara como privada, esta se encuentra disponible en cada hilo pero no está inicializada, por lo que se debería usar firstprivate para obtener su valor original y que sea privada a cada hilo.

# 4.4-

En pi\_par5 los hilos van sumando sus resultados a la variable pi, la cual está protegida la directiva critical.

En pi\_par1 se crea un array sum que tiene nº de elementos = nº de cores del equipo donde se ejecuta y cada hilo va sumando a su elemento del array denotado por su numero de hilo y después de terminar la región paralela se suman todos los elementos de ese array a la variable pi.

En pi\_par3 es parecido a pi\_par1 pero el array sum que tiene nº de elementos = nº de cores del equipo \* nº de doubles que caben en el tamaño de la línea de caché (padding) y al cual cada hilo accede a sum[tid \*padsz]

El false sharing es una situación que se produce cuando los hilos acceden a distintas variables pero estas se encuentran en un mismo bloque de caché, por lo que cuando un hilo actualiza una variable en el bloque caché y otro quiere acceder a otra variable distinta pero en el mismo bloque, este bloque debe escribirse en memoria antes de utilizarse.

En pi\_par3 se obtiene el tamaño de la línea de caché para crear un array donde cada elemento que actualiza cada hilo esté en un bloque caché distinto, por lo que no se produce false sharing.

# 4.5-

Al utilizar la directiva critical la región de código afectada solo puede ser accedida por un hilo a la vez, evitando así las condiciones de carrera.

Al restringir el acceso, el tiempo de ejecución aumenta ya que cada hilo tiene que esperar a que se libere la zona crítica para poder acceder a ella.

# 4.6-

Al ejecutarse pi\_par6 se observa que es de las ejecuciones que tarda mas tiempo en comparación de pi\_par3, por ejemplo. Esto se debe a que por la forma en la que está diseñado el array en pi\_par6 sufre los efectos de false sharing, aumentando el tiempo de ejecución y en cambio pi\_par3 no se ve afectado.

# 4.7-

La versión óptima según los tiempos de ejecución y memoria utilizada es pi\_par7 debido a que es el programa con menor tiempo de ejecución, no necesita reservar memoria adicional gracias al uso de la directiva reduction y el uso esta directiva se ajusta mas al problema a resolver.

# Ejercicio 5: Optimización de programas de cálculo

# 5.0-

El programa entregado recibe de argumentos una lista de nombres de imágenes, y mediante un bucle for detecta los bordes de la imagen primero pasándola a escala de grises, después obteniendo los bordes de la imagen con ruido y por último aplicándola una reducción de ruido para obtener la imagen final

# 5.1-

El bucle más externo puede ser el óptimo para ser paralelizado si se quiere hacer una paralelización de grano grueso, es decir, si lo que se busca es que cada hilo trabaje en una imagen distinta. Si solo se va a trabajar con una imagen o un grupo muy reducido de imágenes lo óptimo sería hacer una paralelización de grano fino donde los hilos trabajan en una misma imagen.

# 5.1a-

Se ejecuta el programa con más hilos que imágenes a procesar mediante la directiva omp parallel for, obteniéndose:

Texto

Descripción generada automáticamente

Como se puede observar, aunque se establece que se lancen 16 hilos, como solo se le pasan dos imágenes como argumento se abren solo 2 hilos, uno para cada imagen en lugar de los 16.

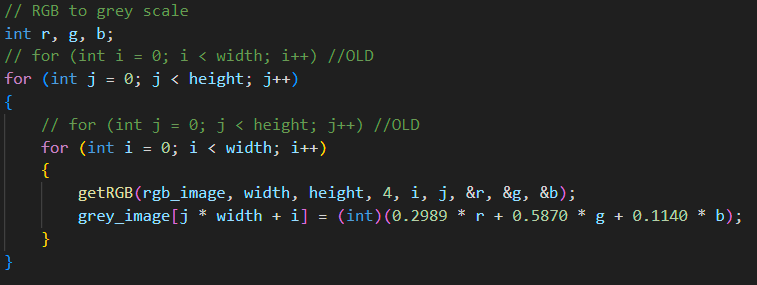
# 5.1b-

Para procesar una única imagen de gran tamaño esta opción no es la adecuada debido a que solo habría un hilo trabajando en dicha imagen. Cada hilo consume AxBxCx4, donde A el número de píxeles de ancho de la imagen, B es el número de píxeles de alto de la imagen, C es el tamaño en bytes de cada pixel (8 bits por canal = 24 bits para RBG = 3 bytes) y el 4 final debido a que se reserva memoria para la imagen original, para la de escala de grises, para la de detección de bordes con ruido y para la de detección de bordes sin ruido.

# 5.2-

# 5.2a-

Se encuentran los bucles con acceso subóptimo a los datos y se corrigen en el nuevo fichero edgeDetectorParallel:



Calendario

Descripción generada automáticamente con confianza media

Texto

Descripción generada automáticamente

Texto

Descripción generada automáticamente

Posteriormente se ejecuta y se comprueba que las imágenes obtenidas son idénticas a las que se producían con el bucle original. También se observa que para imágenes de gran tamaño (la imagen 8K) los tiempos de ejecución se ven reducidos.

# 5.2b-

El orden de acceso a los datos era subóptimo debido a que en lugar de acceder a los datos del array secuencialmente (posición 0, 1, 2…) se accedían de la forma “posición 0\*width, 1\*width, 2\*width…” lo que provocaba que no se aprovechase el principio de localidad espacial de los datos y aumentase la cantidad de fallos de caché, lo que acababa resultando en tiempos de ejecución mayores.

# 5.3-

# 5.3a-

Se realizan las paralelizaciones en los bucles mas externos de edgeDetectorParallel haciendo privadas las variables necesarias y usando directivas como reduction donde se requiera quedando los bucles:

Pantalla de computadora con letras

Descripción generada automáticamente con confianza media

Imagen que contiene Interfaz de usuario gráfica

Descripción generada automáticamente

Texto

Descripción generada automáticamente

Texto

Descripción generada automáticamente

# 5.3b-

Los bucles que deberían tener mejor rendimiento al ser paralelizados deberían ser los externos, debido a que como se ha comprobado anteriormente en la práctica, se abren menos hilos en total y trabajan de forma mas eficiente, haciendo que el tiempo de ejecución se vea reducido ya que se ahorra tiempo al gestionar menos hilos.

# 5.4-

Para este apartado se crea un script (ej5.4.dat) que automatiza la tarea de la toma de tiempos y los exporta a un fichero .dat, posteriormente estos datos se pasan a una hoja de Excel y se obtienen los siguientes resultados:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | SD | HD | FHD | 4K | 8K |
| Tiempo serie | 0,028695818 | 0,109735636 | 0,246258182 | 1,187596909 | 4,90861709 |
| Tiempo paralelo | 0,010671182 | 0,023896364 | 0,035755455 | 0,123505909 | 0,479786 |
| Aceleración | 2,68909467 | 4,592147916 | 6,887289932 | 9,615709229 | 10,2308469 |
| FPS Serie | 34,84828335 | 9,112809965 | 4,060778784 | 0,842036547 | 0,20372337 |
| FPS Paralelo | 93,710333 | 41,84737129 | 27,96776083 | 8,096778592 | 2,08426257 |