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

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 Docs 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: Información sobre la topología del sistema

# 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 y produce sobrecarga en el sistema, pero en situaciones donde se produzcan un gran número de bloqueos si 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 ( “” 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

# 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 vectorial 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 mas 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)-

# Ejercicio 3: Paralelizar el producto escalar





# 3.1-

La versión que obtiene el peor rendimiento es la paralela del bucle 1 con 4 hilos. Esto se debe a que consume mas 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 2 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 se abren 4 hilos en el bucle intermedio, en total los hilos que se abren son 1000x4, es decir 4000 hilos.

# 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.