Práctica 4: Explotar el potencial de las arquitecturas modernas

NOTA: Leer detenidamente el enunciado de cada ejercicio antes de proceder a su realización.

El objetivo de esta práctica es experimentar y aprender a sacar partido a las arquitecturas *multicore (o manycore)* a través del *framework* de programación OpenMP [1].

Todo el trabajo asociado a esta práctica se realizará en entorno Linux. Para ello el estudiante puede emplear los ordenadores de los laboratorios, o bien el clúster de docencia de la asignatura. También será posible emplear un ordenador propio. En cualquiera de los casos, es necesarios comunicar en qué equipo/s se han desarrollado las pruebas. Siempre que no se mencione lo contrario, se deberá usar una configuración de número de hilos razonable que quedará a criterio del alumno. Se valorará positivamente el realizar la prueba para diferentes configuraciones.

MATERIAL DE PARTIDA

Junto con el enunciado de la práctica, se entrega un fichero, "`MaterialP4.zip`", que contiene los siguientes ficheros que serán referenciados a lo largo del enunciado:

- `omp1.c` – fichero con el código de ejemplo de un programa que crea hilos utilizando OpenMP. El número de hilos creados se pasa como argumento al programa.
- `omp2.c` – fichero con el código de ejemplo de un programa que muestras las diferencias en la declaración de la privacidad de variables en OpenMP.
- `pescalar_serie.c` – fichero con un código serie que realiza el producto escalar de dos vectores.
- `pescalar_par1.c` – ficheros con el código de ejemplo de programas que realizan paralelización de bucles con OpenMP
- `Makefile` – fichero makefile utilizado para compilar todos los ficheros de código fuente. Debe ser modificado para que también se compilen los programas que se piden.
- `pi_serie.c` – fichero con un código serie que realiza el cálculo de pi por integración numérica.
- `pi_par1.c` a `pi_par7.c` – ficheros con el código de variantes en OpenMP para calcular pi por integración numérica.
- `edgeDetector.c` – programa que aplica un algoritmo de procesamiento de imágenes para detectar bordes.

IMPORTANTE

A lo largo de este documento se hará referencia a un número P que afectará a los experimentos y resultados que se obtendrán. El valor de P que debe utilizar debe ser igual al número de pareja módulo 8 más 1 (es decir será un número entre 1 y 8).

Ejercicio 0: información sobre la topología del sistema

En primer lugar, deberíamos de ser capaces de obtener la información relativa a la arquitectura de la máquina sobre la que estamos trabajando. Esto es sencillo en un sistema Linux y podemos hacerlo ejecutando cualquiera de los siguientes comandos:

```
> cat /proc/cpuinfo

> dmidecode
```

NOTA: este comando requiere permisos de superusuario, por lo que NO podremos utilizarlo en la instalación de los laboratorios. Puede utilizar en su lugar el comando lstopo. En el cluster está instalado en /share/apps/tools/hwloc/bin/lstopo

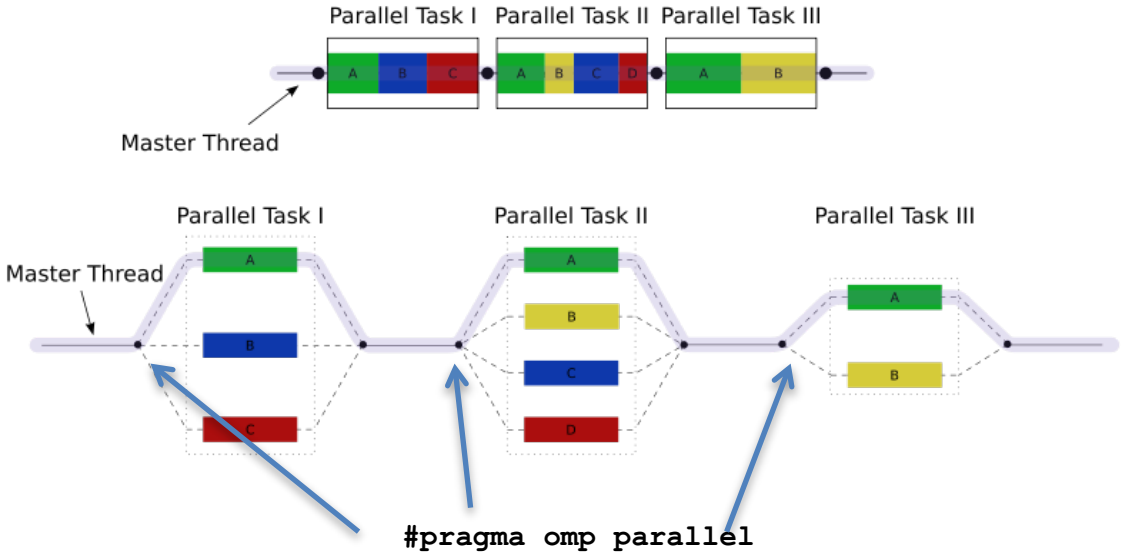
En particular, si nos fijamos en la información que nos da el fichero `cpuinfo` podemos deducir cuántos *cores* tiene nuestro equipo y si tiene o no habilitado la opción de *hyperthreading* (el parámetro `cpu_cores` indica cuántas CPUs físicas hay en equipo en total, y el parámetro `siblings` hace referencia al número de CPUs virtuales). En un equipo con mas de un procesador, el parámetro `physical_id` diferencia el procesador donde esta cada *core*.

A partir de la información expuesta, indique en la memoria asociada a la práctica los datos relativos a las cantidad y frecuencia de los procesadores disponibles en el equipo empleado, y a si la opción de *hyperthreading* está o no activa.

Ejercicio 1: Programas básicos de OpenMP (1 p)

En este ejercicio, vamos a realizar una primera toma de contacto con programas basados en OpenMP. Para realizar la compilación de estos programas utilizaremos como compilador `gcc`, añadiendo las banderas `-lgomp` y `-fopenmp` para indicarle que queremos utilizar las librerías y extensiones de OpenMP.

OpenMP nos permite lanzar de una forma muy sencilla un número de hilos (o *threads*) para que lleven a cabo tareas de forma paralela (cada hilo ejecutando en su CPU correspondiente). La creación de un equipo de hilos se lleva a cabo cada vez que se incluye en la directiva `#pragma omp parallel` en el programa, y la sincronización y finalización de los hilos se realiza al terminar la región paralela señalada con el `parallel`.



En el material de partida, se pide compilar y ejecutar el programa `omp1.c` para comprobar cómo se lleva a cabo el lanzamiento de equipos de hilos, y familiarizarse con las funciones `omp_get_num_threads()` y `omp_get_thread_num()` que nos permiten identificar a cada uno de los *threads* en activo.

1.1 ¿Se pueden lanzar más *threads* que *cores* tenga el sistema? ¿Tiene sentido hacerlo?
1.2 ¿Cuántos *threads* debería utilizar en los ordenadores del laboratorio? ¿y en el clúster? ¿y en su propio equipo?

El número de *threads* que se lanza al abrir una región paralela se puede seleccionar de las siguientes maneras:

Con una variable de entorno:

```
OMP_NUM_THREADS
```

con una función:

```
omp_set_num_threads(int num_threads);
```

con una clausula dentro de la región paralela

```
#pragma omp parallel num_threads(numthr)
```

1.3 Modifique el programa omp1.c para utilizar las tres formas de elegir el número de threads y deduzca la prioridad entre ellas.

Una característica fundamental a la hora de programa utilizando OpenMP es definir correctamente qué variables han de ser públicas (accesibles a todos los hilos, es el valor por defecto) o privadas (cada hilo accede a su propia copia). Un ejemplo de la utilización de estas características puede encontrarse en el programa `omp2.c`.

Se pide ejecutar el programa omp2 e incluir la salida en la memoria de la práctica. En base a la salida obtenida (y tras leer y comprender el código fuente que la genera) conteste a las siguientes preguntas:

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?
1.7 ¿Ocurre lo mismo con las variables públicas?

Ejercicio 2: Paralelizar el producto escalar (1 p)

Tomaremos como código de referencia el del producto escalar, cuya versión serie y un intento de código paralelizado se entregan como material de partida de la práctica. OpenMP está especialmente pensado para realizar la paralelización del trabajo de un bucle *for* entre los hilos del equipo. Cuando se quiere repartir el trabajo de un bucle de esta forma se utiliza la cláusula `#pragma omp parallel for`. Al indicar esta cláusula, OpenMP ya hace trabajo por nosotros como por ejemplo declarar como privada la variable que se utiliza como índice del bucle.

2.1 Ejecute la versión serie y entienda cual debe ser el resultado para diferentes tamaños de vector.

2.2 Ejecute el código paralelizado con el pragma openmp y conteste en la memoria a las siguientes preguntas:

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

Antes de intentar reducir el tiempo de ejecución es imprescindible que el resultado sea correcto. Para ello debe garantizar que la acumulación que se utiliza en este código no tenga problemas por condiciones de carrera.

2.3 Modifique el código anterior y denomine el programa pescalar_par2. Esta versión deber dar el resultado correcto utilizando donde corresponda alguno de los siguientes pragmas

```
#pragma omp critical
```

```
#pragma omp atomic
```

- ¿Puede resolverse con ambas directivas? Indique las modificaciones realizadas en cada caso.
- ¿Cuál es la opción elegida y por qué?

2.4 Modifique el código anterior y denomine el programa resultante pescalar_par3. Esta versión deber dar el resultado correcto utilizando donde corresponda alguno de los siguientes pragmas

```
#pragma omp parallel for reduction
```

- Comparando con el punto anterior ¿Cuál será la opción elegida y por qué?

2.6 Análisis de tiempos de ejecución.

Es bastante frecuente que cuando desarrollamos un algoritmo queramos que este tenga un rendimiento óptimo independientemente del tamaño de las entradas. Esto implica que la paralelización tiene que ser limitada a casos en los que el tamaño de los bucles es suficiente para compensar el coste añadido u *overhead* de lanzar los hilos. 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)
```

Si M es el tamaño del vector, se pide estimar la cantidad denominada *threshold* o umbral. Para ello, se debe realizar un recorrido iterativo bajando o subiendo el tamaño del vector (argumento del programa) según vamos avanzando para encontrar el punto a partir del cual compensa paralelizar.

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 $ceil(1.2T)$.

Se pide estimar un valor para T y para ese valor medir el tiempo serie y paralelo para vectores de tamaño $ceil(0.8T)$, T y $ceil(1.2T)$ de tal manera que se vea claramente como se cumplen las dos condiciones anteriores.
Compruebe su estimación de T, para ello modifique el valor de T a un valor por debajo del estimado como optimo y evalúe las condiciones. Repita modificando el valor de T a un valor por encima del optimo y compruebe las condiciones.

2.7 (Opcional) Análisis exhaustivo (Hasta 1 punto extra)

Se espera que este proceso se realice manualmente y en pocas iteraciones el alumno encuentre un valor del umbral en pocas iteraciones por lo que no es necesario hacerlo de manera automática. No obstante, se valorará la obtención automática de este umbral (bien sea iterando o bien sea probando una cantidad suficiente de tamaños de vector), así como añadir cualquier gráfica que sirva para ilustrar el valor de T elegido.

¿Es el valor de umbral distinto para tu ordenador que para el clúster? ¿Depende del número de hilos que se lancen?

Ejercicio 3: Paralelizar la multiplicación de matrices (3 p)

Para este ejercicio se tomará como código de partida el de la multiplicación de matrices que cada alumno ha desarrollado a lo largo de la práctica 3 de la asignatura.

En este caso, el programa se compone de tres bucles anidados, por lo que a la hora de llevar a cabo la paralelización del trabajo a realizar es razonable preguntarse: ¿cuál de los bucles interesa paralelizar?

Se pide desarrollar y entregar (además de la versión serie de la práctica 3) tres versiones paralelas de la multiplicación de matrices paralelizando los tres bucles existentes. Las versiones entregadas han de funcionar de forma correcta.

Se pide rellenar las siguientes tablas realizando la ejecución para matrices de tamaño suficientemente grande para que los valores de tiempo de ejecución permitan ver diferencias entre los diferentes casos que se estudien (por ejemplo, empiece evaluando un tamaño de 1000x1000 que tarden unos 10 segundos y pruebe al menos otro tamaño que tarde 60 segundos):

Tiempos de ejecución (s)

Versión\# hilos	1	2	3	4
Serie				
Paralela – bucle1				
Paralela – bucle2				
Paralela – bucle3				

Speedup **(tomando como referencia la versión serie)**

Versión\# hilos	1	2	3	4
Serie	1			
Paralela – bucle1				
Paralela – bucle2				
Paralela – bucle3				

NOTA: se considera que el bucle 1 es el más interno, el bucle 2 es el bucle intermedio, y el bucle 3 es el bucle más externo. En el caso serie completar sólo la columna para un hilo.

Teniendo en cuenta los resultados obtenidos, conteste de manera razonada a las siguientes preguntas:

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?

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?

Tomando como referencia los tiempos de ejecución de la versión serie y el de la mejor versión paralela obtenida anteriormente (la mejor combinación entre las tres versiones de código, y las C posibilidades para el número de hilos paralelos). Tome tiempos en un fichero y realice una gráfica de la evolución del tiempo de ejecución y la aceleración (de la versión paralela vs serie) al ir variando el tamaño de las matrices de NxN para N entre 512+P y 1024+512+P (con incrementos en N de 64). Tras incluir estas dos gráficas en la memoria, incluya un párrafo describiendo y justificando el comportamiento observado en las mismas.

3.3 Si en la gráfica anterior no obtuvo un comportamiento de la aceleración en función de N que se estabilice o decrezca al incrementar el tamaño de la matriz, siga incrementando el valor de N hasta conseguir una gráfica con este comportamiento e indique para que valor de N se empieza a ver el cambio de tendencia.

Ejercicio 4: Ejemplo de integración numérica (2 p)

El ejemplo con el que vamos a trabajar a lo largo de esta sesión consiste en obtener una aproximación del valor del número π mediante un método de integración numérica denominado regla del punto medio (o intermedio).

Explicación del método

En particular, vamos a calcular el área encerrada por una función f en el intervalo $[0, 1]$ aproximando dicha área mediante rectángulos. Como ejemplo, para el primer rectángulo que está en el intervalo $[0, h]$, realizamos la siguiente estimación:

$$\int_0^h f(x)dx \approx h \cdot f\left(0 + \frac{h}{2}\right)$$

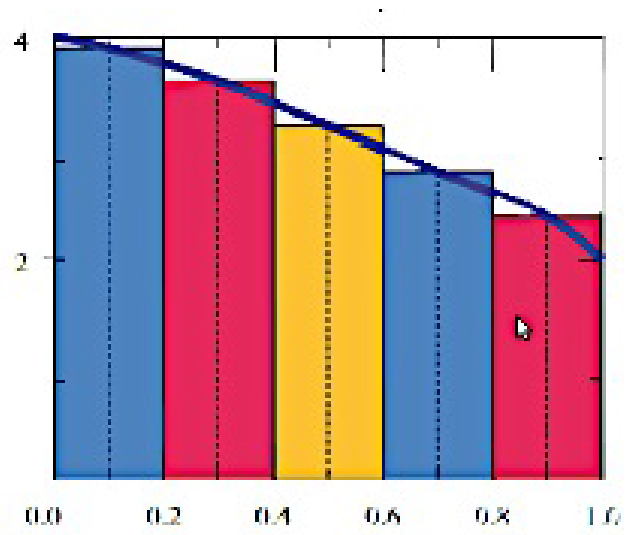
Repitiendo está estimación para todos los rectángulos, obtendríamos:

$$\int_0^1 f(x)dx = \sum_i \int_{h\cdot i}^{h\cdot(i+1)} f(x)dx \approx \sum_i h \cdot f\left(h \cdot i + \frac{h}{2}\right)$$

En particular, si elegimos $f(x) = \frac{4}{1+x^2}$, se puede ver que

$$\int_0^1 \frac{4}{1+\mathrm{x}^2}dx = 4\int_0^1 \frac{1}{1+\mathrm{x}^2}dx = 4[\arctan(1) - arctan(0)] = 4\left[\frac{\pi}{4} - 0\right] = \pi$$

Se incluye en la siguiente figura una representación gráfica del algoritmo.



Cuanto más estrechos sean estos rectángulos, esto es, menor sea h, mayor será la aproximación al resultado real y por tanto al número π. Un ejemplo de programa que realiza esta tarea puede encontrarse en el fichero pi_serie.c que se encuentra en la carpeta de material de para la práctica.

Perdida de rendimiento por el efecto de falso compartir (*False sharing*) en OpenMP

Pregunta 4.1: ¿Cuántos rectángulos se utilizan en la versión del programa que se da para realizar la integración numérica? ¿Cuál es entonces el valor de h (ancho del rectángulo)?

Pregunta 4.2: Ejecute todas las versiones paralelas y la versión serie. Analice el rendimiento (en términos de tiempo medio de ejecución y la aceleración o *speedup*) y si el resultado obtenido es correcto o no, reflejando estos datos en una tabla. En las versiones que el resultado no sea correcto, incluya una breve explicación de por qué no es correcto.

Ejercicio 4.3: En la versión pi_par2, ¿tiene sentido declarar sum como una variable privada? ¿Qué sucede cuando se declara una variable de tipo puntero como privada?

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?

Ejercicio 4.5: Explique el efecto de utilizar la directiva *critical*. ¿Qué diferencias de rendimiento se aprecian? ¿A qué se debe este efecto?

Ejercicio 4.6: Ejecute la versión pi_par6 del programa. ¿Qué diferencias de rendimiento se aprecian? ¿A qué se debe este efecto?

Ejercicio 4.7: ¿Qué versión es la óptima y por qué?

Ejercicio 5: Optimización de programas de cálculo (hasta 4 puntos)

Una de las ventajas de OpenMP es su extrema facilidad de aplicarse a programas de cálculo intensivo sin requerir un cambio completo de la arquitectura o el diseño de este. En este último ejercicio, se propone aplicar todos los conocimientos obtenidos durante la asignatura para optimizar y paralelizar un programa de propósito aplicado.

En el fichero `edgeDetector.c` se incluye un programa que aplica un algoritmo de procesamiento de imágenes para detectar bordes. Este algoritmo nos ha sido proporcionado, a modo de prueba de concepto, con la idea de que finalmente procese en tiempo real imágenes de un *streaming* de video. Para ello, se recuerda a los alumnos que un video suele estar compuesto de aproximadamente 30 fps (*frames per second*). Esto es, el programa tendría que procesar 30 imágenes en un segundo.

Responda razonadamente a las preguntas:

0. 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.
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
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.
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.
4. Rellene una tabla con resultados de tiempos y *speedup* respecto a la versión serie para imágenes de distintas resoluciones (SD, HD, FHD, UHD-4k, UHD-8k). Añada a su vez una columna que sea la tasa de fps a la que procesaría el programa.
5. Algo que hemos dejado de lado es utilizar las optimizaciones del compilador. Repita el apartado anterior añadiendo al comando gcc la bandera -O3. Investigue sobre las optimizaciones que implementa el compilador (y como podrían afectar a nuestra paralelización). ¿Implementa el desenrollado de bucles? ¿Está activada esta opción? ¿Implementa algún tipo de vectorización? **Nota**: la opción -O3 puede generar *warnings* al ser compilado. Compruebe que la salida sigue siendo la misma en todo caso.

MATERIAL A ENTREGAR

Memoria con las respuestas a las preguntas a lo largo del enunciado.
En subdirectorios independientes, los ficheros con los datos obtenidos en los experimentos de los ejercicios que así lo indiquen, y los scripts utilizados para la obtención de estos datos (si se ha usado alguno).
Códigos fuente desarrollados o modificados para la resolución de los ejercicios.

La entrega se realizará a través de Moodle, en la fecha indicada para esta entrega.

MATERIAL DE REFERENCIA

[1] **OpenMP: Tutorials and technical articles** http://openmp.org/wp/resources/#Tutorials