UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf

Computación Paralela (CPA)

Práctica 1: Paralelización con OpenMP

Curso 2017/18

Índice

Introducción

Esta práctica consta de 3 sesiones. La siguiente tabla lista el material de partida para realizar cada uno de los apartados:

Sesión 1	Integración numérica	`integral.c`
Sesión 2	Procesamiento de imágenes	`imagenes.c`, `Lenna.ppm`
Sesión 3	Números primos	`primo_grande.c`, `primo_numeros.c`

Para compilar los códigos, en la mayoría de casos será necesario enlazar con la biblioteca matemática (porque se usan funciones como `sqrt`), es decir, añadir `-lm` en la línea de compilación. Recordar que la compilación de programas OpenMP requiere indicar la opción de compilación correspondiente, por ejemplo

```
$ gcc -fopenmp -o pintegral pintegral.c -lm
```

Para ejecutar con varios hilos, se puede hacer por ejemplo:

```
$ OMP_NUM_THREADS=4 ./pintegral 1
```

Para más detalles, consultar la documentación de uso del clúster de prácticas.

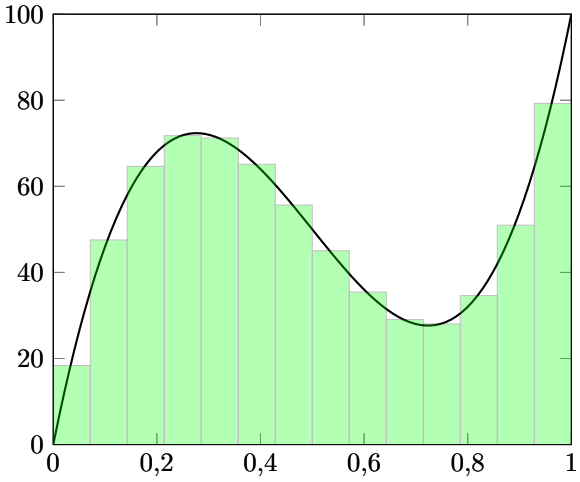

Figura 1: Interpretación geométrica de una integral.

1. Integración numérica

Consideraremos aquí el cálculo numérico de una integral de la forma

$$\int_a^b f(x)dx.$$

La técnica que se va a utilizar para calcular numéricamente la integral es sencilla, y consiste en aproximar mediante rectángulos el área correspondiente a la integral, tal y como puede verse en la Figura 1. Se puede expresar la aproximación realizada de la siguiente forma

$$\int_a^b f(x)dx \approx \sum_{i=0}^{n-1} f(x_i) \cdot h = h \cdot \sum_{i=0}^{n-1} f(x_i), \tag{1}$$

donde n es el número de rectángulos considerado, $h = (b-a)/n$ es la anchura de los rectángulos, y $x_i = a + h \cdot (i + 0{,}5)$ es el punto medio de la base de cada rectángulo. Cuanto mayor sea el número de rectángulos, mejor será la aproximación obtenida.

El código del programa secuencial que realiza el cálculo se encuentra en el fichero `integral.c`, del que puede verse un extracto en la Figura 2. En particular, podemos ver que hay dos funciones distintas para el cálculo de la integral, que corresponden a dos variantes con pequeñas diferencias entre sí. En ambas hay un bucle que realiza el cálculo de la integral, y que corresponde al sumatorio que aparece en la ecuación (1).

Lo que se pretende hacer en esta práctica es paralelizar mediante OpenMP las dos variantes del cálculo de la integral.

En primer lugar, compila el programa y ejecútalo. Al ejecutarlo le indicaremos mediante un argumento (1 o 2) cuál de las dos variantes del cálculo de la integral queremos utilizar. Por ejemplo, para utilizar la primera variante:

```
$ ./integral 1
```

El resultado de la integral saldrá por pantalla. El resultado será el mismo independientemente de la variante elegida para su cálculo. Opcionalmente, se puede indicar el valor de n como segundo argumento del programa.

```
/* Calculo de la integral de una funcion f. Variante 1 */
double calcula_integral1(double a, double b, int n)
{
   double h, s=0, result;
   int i;
   h=(b-a)/n;
   for (i=0; i<n; i++) {
      s+=f(a+h*(i+0.5));
   }
   result = h*s;
   return result;
}
/* Calculo de la integral de una funcion f. Variante 2 */
double calcula_integral2(double a, double b, int n)
{
   double x, h, s=0, result;
   int i;
   h=(b-a)/n;
   for (i=0; i<n; i++) {
      x=a;
      x+=h*(i+0.5);
      s+=f(x);
   }
   result = h*s;
   return result;
}
```

Figura 2: Código secuencial para calcular una integral.

1.1. Paralelización de la primera variante

A continuación, modificaremos el fichero `integral.c` para realizar el cálculo de forma paralela utilizando OpenMP. Empezaremos por la primera variante (`calcula_integral1`). Podemos empezar colocando una directiva `parallel for` sin preocuparnos de si las variables son compartidas, privadas o de otro tipo, y seguidamente compilamos y ejecutamos el programa modificado, para ver qué es lo que ocurre.

Nos daremos cuenta de que el resultado es incorrecto. Para solucionarlo puede ser necesario indicar el tipo de algunas de las variables que intervienen en el bucle, mediante la utilización de cláusulas como `private` o `reduction`.

Una vez se haya modificado el fichero y se haya compilado para producir el ejecutable, se comprobará que el resultado de la integral es el mismo que en el caso secuencial, y que no varía al volver a ejecutar el programa, ni al cambiar el número de hilos.

Importante: Es conveniente asegurarse de que el programa está efectivamente usando el número de hilos que hemos indicado. Para ello, haz que el programa muestre, mediante un `printf`, el número de hilos con el que está trabajando. El mensaje con el número de hilos debería mostrarse una sola vez.

1.2. Paralelización de la segunda variante

Consideraremos ahora la paralelización de la segunda variante (`calcula_integral2`). Como vemos en la Figura 2, el código de esta segunda variante es prácticamente idéntico al de la primera. La única diferencia es que se utiliza una variable auxiliar `x`, lo que obviamente no afecta en nada al resultado del cálculo.

Paraleliza ahora esta segunda versión. De nuevo comprueba que el resultado es el mismo que en el caso secuencial, y que no varía al volver a ejecutar ni al cambiar el número de hilos.

```
#!/bin/sh
#PBS -l nodes=1,walltime=00:05:00
#PBS -q cpa
#PBS -d .

OMP_NUM_THREADS=3 ./pintegral 1
```

Figura 3: *Script* para ejecutar en el sistema de colas.

1.3. Ejecución en el cluster

Para ejecutar el programa en el clúster de prácticas, debemos compilar en la máquina `kahan` y lanzar un trabajo al sistema de colas, siguiendo las instrucciones. Básicamente, hay que escribir un *script* con las opciones del sistema de colas seguidas por los comandos que se quieren ejecutar. En la Figura 3 hay un ejemplo en el que se lanza un programa OpenMP con 3 hilos de ejecución con un tiempo máximo de 5 minutos, en la cola `cpa` y con la opción de ejecutar en el directorio actual (`.`). Para lanzar el trabajo al sistema de colas, suponiendo que el fichero del *script* se llama por ejemplo `jobopenmp.sh`, bastaría con ejecutar:

```
qsub jobopenmp.sh
```

Como alternativa a indicar el número de hilos en el *script*, es posible asignar la variable `OMP_NUM_THREADS` al lanzar el trabajo al sistema de colas:

```
qsub -v OMP_NUM_THREADS=3 jobopenmp.sh
```

Se recuerda que se puede consultar el estado de las colas con la orden `qstat`, y cancelar trabajos con la orden `qdel`.

2. Procesamiento de imágenes

Este apartado de la práctica se centra en la implementación en paralelo del filtrado de una imagen utilizando OpenMP. El objetivo es profundizar en el conocimiento de OpenMP y de la resolución de dependencias entre hilos.

Los ejercicios se realizarán partiendo de la versión secuencial de un programa que permite leer una imagen en formato PPM, aplicar varias etapas de filtrado basado en la media ponderada con radio variable y escribir la imagen resultante en un archivo con el mismo formato. Los alumnos tendrán que realizar una versión paralela mediante OpenMP de dicho programa, aprovechando los diferentes bucles en los que se estructura el método.

2.1. Descripción del problema

El filtrado de imágenes consiste en sustituir los valores de los píxeles de una imagen por valores dependientes de sus vecinos. El filtrado se puede utilizar para reducir ruido, reforzar contorno, enfocar o desenfocar una imagen, etc.

El filtrado de media consiste en sustituir el valor de cada píxel por la media de los valores de los píxeles vecinos. Por vecinos podemos entender a los píxeles que distan como mucho un cierto valor, llamado radio, en ambas direcciones cartesianas. El filtrado de media reduce notablemente el ruido aleatorio, pero produce un importante efecto de desenfoque. Generalmente, se utiliza una máscara que pondera el valor de los puntos cercanos a la imagen, siguiendo un ajuste parabólico o lineal. Este filtrado produce mejores efectos, aunque tiene un coste computacional más elevado. Más aún, el filtrado es un proceso iterativo que puede requerir varias etapas.

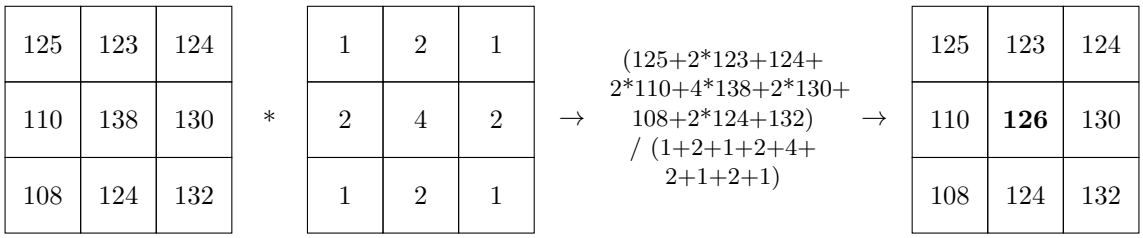

Figura 4: Modelo de aplicación de una media ponderada en el filtrado de imágenes.

2.2. Versión secuencial

El material para la práctica incluye el fichero `imagenes.c` con la implementación secuencial del filtrado. El filtrado utilizado aplica una máscara ponderada que proporciona más peso a los puntos más próximos al punto que se está procesando. La Figura 4 muestra un esquema. En esta figura, partimos de una imagen (izquierda) sobre la que estamos aplicando en su píxel central el filtrado, utilizando la máscara cuadrática a su derecha. Esta aplicación implica realizar el cálculo que se muestra, cuyo resultado aparece en negrita en la imagen resultante a la derecha. Este filtrado se realizará para cada uno de los puntos de la imagen.

El algoritmo por tanto recorre las dos dimensiones de la imagen, y para cada píxel, utiliza dos bucles internos que recorren los píxeles que distan como mucho el valor del radio en cada una de las dimensiones, evitando salir de los límites de las coordenadas. El proceso de filtrado se repite varias veces para toda la imagen, con lo que el proceso implica cinco bucles anidados: pasos, filas, columnas, radio por filas, radio por columnas, tal y como muestra la Figura 5.

Para la lectura y escritura de la imagen utilizamos el formato PPM, un formato simple basado en texto, que puede ser visualizado por diferentes programas, como `irfanview`[1] o `display` (disponible en el clúster de prácticas). El formato de la imagen se muestra en la Figura 6 y puede comprobarse viendo el contenido del fichero con `head`, `more` o `less`.

Por tanto, el programa leerá el contenido de un fichero de imagen cuyo nombre está especificado en `IMAGEN_ENTRADA`, aplicará el filtrado tantas veces como indica `NUM_PASOS` y con el radio `VAL_RADIO` y lo escribirá en el fichero `IMAGEN_SALIDA`. La reserva de memoria la realiza la función de lectura de la imagen, garantizando que todos los píxeles de la imagen se encuentran consecutivos.

El alumno deberá comprobar el correcto funcionamiento del programa y ajustar los valores del radio y el número de pasos para la imagen que quiera filtrar. Se adjunta una imagen de prueba, procedente de un famoso benchmark[2], ver Figura 7. El alumno podrá utilizar sus propias imágenes, recomendándose unas dimensiones superiores a los 4 Mpíxeles.

2.3. Implementación paralela

Existen diferentes aproximaciones para realizar la implementación paralela mediante OpenMP. El trabajo se centrará en analizar los cinco bucles y decidir (y probar) qué bucles son susceptibles de ser paralelizados y qué variables deberían ser compartidas o privadas. Para ello, el alumno deberá:

- Analizar si las diferentes iteraciones tienen alguna dependencia entre sí (p.e. si la segunda iteración utiliza datos generados por la primera) y si estas dependencias pueden resolverse directamente con alguna cláusula de OpenMP (p.e. en el caso de sumatorios).
- Una vez analizados los bucles susceptibles de ser paralelizados, se procederá a identificar las variables que deberán ser privadas a cada hilo o compartidas entre todos.
- Una vez definida la aproximación paralela, se implementará utilizando las directivas OpenMP correspondientes.

[1] `http://www.irfanview.com`
[2] `http://en.wikipedia.org/wiki/Standard_test_image`

```
for (p=0;p<pasos;p++) {
  for (i=0;i<n;i++) {
    for (j=0;j<m;j++) {
      resultado.r = 0;
      resultado.g = 0;
      resultado.b = 0;
      tot=0;
      for (k=max(0,i-radio);k<=min(n-1,i+radio);k++) {
         for (l=max(0,j-radio);l<=min(m-1,j+radio);l++) {
           v = ppdBloque[k-i+radio][l-j+radio];
           resultado.r += ppsImagenOrg[k][l].r*v;
           resultado.g += ppsImagenOrg[k][l].g*v;
           resultado.b += ppsImagenOrg[k][l].b*v;
           tot+=v;
         }
      }
      resultado.r /= tot;
      resultado.g /= tot;
      resultado.b /= tot;
      ppsImagenDst[i][j].r = resultado.r;
      ppsImagenDst[i][j].g = resultado.g;
      ppsImagenDst[i][j].b = resultado.b;
    }
  }
  memcpy(ppsImagenOrg[0],ppsImagenDst[0],n*m*sizeof(struct pixel));
}
```

Figura 5: Bucles principales para el procesamiento de imágenes.

```
P3       <- Cadena constante que indica el formato (ppm, color RGB)
512 512  <- Dimensiones de la imagen (número de columnas y número de filas)
255      <- Mayor nivel de intensidad
224 137 125 225 135 ... <- 512x512x3 valores. Cada punto son tres valores consecutivos (R,G,B)
```

Figura 6: Formato de un fichero de imágenes PPM.

Figura 7: Imagen de referencia (`Lenna.ppm`) antes (izquierda) y después (derecha) de aplicar un filtrado de un paso y radio 5.

- Los resultados variarán dependiendo de los bucles que se paralelicen y por tanto se deberá realizar una experimentación que analice la mejor aproximación. Deberá instrumentarse el código para que se pueda medir el tiempo (`omp_get_wtime`). Para ello, se medirá el tiempo necesario para realizar el proceso con las diferentes combinaciones. Se recomienda utilizar una tabla en la que se recoja el tiempo de proceso para cada uno de los bucles por separado.

3. Números primos

En esta sesión se va a trabajar con el archiconocido problema de ver si un número es primo o no. Aunque para este problema hay diversos algoritmos (algunos de ellos más eficientes pero algo complicados), se va a utilizar el algoritmo secuencial típico.

En este caso, la paralelización no es "trivial", es decir, no siempre se puede obtener un buen algoritmo paralelo con OpenMP añadiendo tan sólo un par de directivas. Cuando se pueda, así debe hacerse tanto por comodidad como por claridad e independencia de la plataforma. Pero en ocasiones (y esta es una de ellas), hay que pararse a pensar e indicar explícitamente un reparto de la carga entre hilos. Esto es lo que va a haber que realizar en esta práctica.

3.1. Algoritmo secuencial

El algoritmo secuencial clásico para ver si un número es primo se muestra en la Figura 8. Consiste en mirar si es divisible por algún número inferior a él (diferente del 1). Si es divisible de forma exacta por algún número inferior a él y distinto de la unidad, el número no es primo.

Comprobar si un número es primo o no siguiendo este algoritmo tiene un coste pequeño, siempre que el número no sea muy grande. Nótese que el bucle deja de ejecutarse si se descubre que el número es compuesto (en ese caso no hace falta seguir mirando). Por esto, es obvio que el peor caso (el mayor coste) sucederá cuando el número a comprobar sea primo o sea no primo pero compuesto por factores grandes.

Con la intención de obtener un código que tenga un coste algo más elevado, se va a proceder a buscar un número primo grande. No tiene sentido paralelizar una tarea cuyo coste sea muy pequeño (excepto para

```
Función primo(n)
  Si  n es par  y  no es el número 2  entonces
    p <- falso
  si no
    p <- verdadero
  Fin si
  Si  p  entonces
    s <- raíz cuadrada de n
    i <- 3
    Mientras  p  y  i <= s
      Si  n es divisible de forma exacta por i  entonces
        p <- falso
      Fin si
      i <- i + 2
    Fin mientras
  Fin si
  Retorna p
Fin función
```

Figura 8: Algoritmo secuencial para determinar si n es primo.

```
Función primo_grande
  n <- entero más grande posible
  Mientras n no sea primo
    n <- n - 2
  Fin mientras
  Retorna n
Fin función
```

Figura 9: Algoritmo a paralelizar: busca el mayor primo que cabe en un entero sin signo de 8 bytes.

ilustrar o enseñar cosas básicas).

El problema inicial a resolver en este apartado de la práctica va a ser, por tanto, no el ver si un número es primo (esto será un componente fundamental) sino buscar el número primo más grande que quepa en un entero sin signo de 8 bytes. El proceso para obtener este número supone partir del mayor número que quepa en ese tipo de datos e ir hacia abajo hasta encontrar un número que sea primo, utilizando el algoritmo anterior para ver si cada número es primo o no. Habitualmente el número más grande será impar (todo a unos en binario) y para buscar primos se puede ir decrementando de dos en dos para no mirar los pares, que no hace falta. El algoritmo se muestra en la Figura 9.

Revisa el programa proporcionado, `primo_grande.c`. Este programa utiliza los algoritmos antes propuestos para buscar y mostrar por pantalla el número primo más grande que cabe en una variable entera sin signo de 8 bytes.

3.2. Algoritmo paralelo

Implementar una versión paralela con OpenMP de la función que averigua si un número es primo o no. Como la parte costosa de esa función es un bucle `for`, parece inmediata la paralelización mediante el uso de `parallel for`. Prueba a hacerlo. ¿Qué sucede?

Efectivamente, OpenMP no permite la paralelización directa de un bucle `for` a menos que esté perfectamente delimitado su inicio, final e incremento. Como en la función `primo` el bucle puede terminar antes (cuando ya se ha dado cuenta de que el número no es primo), no es un bucle factible de paralelizar con `parallel for`. En realidad, lo que no permite que OpenMP pueda paralelizar el bucle es el incluir la comprobación de primo dentro de la condición del bucle. ¿Qué pasaría si se quitara esa parte de la condición?

```
Función cuenta_primos(último)
  n <- 2 (por el 1 y el 2)
  i <- 3
  Mientras i <= último
    Si  i es primo  entonces
      n <- n + 1
    Fin si
    i <- i + 2
  Fin mientras
  Retorna n
Fin función
```

Figura 10: Algoritmo que cuenta la cantidad de números primos que hay entre 1 y un valor dado.

La función seguiría siendo perfectamente correcta, pero ¿qué inconveniente tiene? [Nota: Si no se es capaz de ver el inconveniente, se puede probar el programa eliminando esa parte de la condición, incluso en su versión paralela, ya que en ese caso sí se puede paralelizar de forma muy fácil. Pero hay que tener en cuenta que el tiempo de ejecución del programa se incrementará muchísimo.]

Una vez que se haya comprendido la inconveniencia de realizar así la versión paralela, habrá que buscar otra forma de hacerlo. En esta ocasión no va a ser tan trivial como la paralelización de otros programas. Una forma de hacerlo es realizar un reparto explícito del bucle entre los hilos del equipo. Habrá que usar primitivas y funciones de OpenMP que permitan obtener el número de hilos y ejecutar en paralelo el bucle, pero haciendo que cada hilo realice sólo una parte de él. Desarrolla esta nueva versión paralela y mide el tiempo de ejecución necesario. Sería interesante probar tanto repartiendo el bucle de forma cíclica entre los hilos como repartiéndolo por bloques consecutivos.

Es importante conservar la condición de salida del bucle cuando se ve que el número no es primo. De esta manera, (si se hace correctamente) cuando cualquier hilo descubra que el número no es primo, todos (tarde o temprano) dejarán de procesar el bucle.

Obsérvese que el código debe realizarse de forma que:

1. Su funcionamiento debe ser correcto independientemente del número de hilos con que se ejecute. En particular, debe funcionar bien con un solo hilo.

2. A ser posible, debe poder compilarse y ejecutarse correctamente aún sin usar OpenMP. Para ello, hay que recurrir a la compilación condicional (directiva `#ifdef`), de manera que se haga uso de funciones OpenMP solo si se usa OpenMP al compilar (o sea, si el símbolo `_OPENMP` está definido).

Nota: Puede que resulte conveniente añadir el modificador de C `volatile` a la variable que se utiliza como control del bucle: `volatile int p;` El modificador `volatile` del lenguaje C le indica al compilador que no optimice el acceso a esa variable (que no la cargue en registros y que cualquier acceso a ella se haga efectivamente sobre memoria), con lo que su modificación por parte de un hilo será visible antes en el resto de hilos[3].

3.3. Contando primos

Ya que se está trabajando con números primos, planteamos este nuevo problema relacionado: contar la cantidad de números primos que hay entre el 1 y un número grande, por ejemplo 100000000. [Nota: Si tarda mucho, tomar un número final menor. Idealmente debería tardar 1 o 2 minutos el algoritmo secuencial.]

El algoritmo para realizar este proceso sería el mostrado en la Figura 10. Probar la versión secuencial de este algoritmo, realizando medida del tiempo de ejecución.

Dado que se dispone de una versión paralela con OpenMP de la función para ver si un número es primo, es trivial realizar una versión paralela del nuevo problema sin más que utilizar la versión paralela para ver

[3]Este tipo de comportamiento puede conseguirse también usando la sentencia `flush` de OpenMP.

si un número es primo. Desarrolla este algoritmo paralelo para resolver el problema y mide los tiempos de ejecución.

No termina de funcionar bien. Los números primos iniciales son muy pequeños como para suponer una carga de trabajo suficiente que garantice ganancia al repartir el trabajo entre múltiples hilos. Esto se puede aliviar un poco añadiendo a la región paralela una cláusula `if`, para que se ejecute en paralelo sólo cuando el tamaño del problema sea mayor. Si bien al empezar a trabajar en paralelo con un problema mayor se va a repartir mejor el trabajo, esto también supone que todos los números anteriores a ese valor se habrán calculado de forma secuencial. Esta solución sigue sin comportarse especialmente bien.

Una estrategia alternativa sería paralelizar el bucle del programa principal, que en este caso sí que parece fácilmente paralelizable mediante directivas OpenMP. Desarrollar una nueva versión paralela para este problema que haga ese bucle en paralelo, usando para la función `primo` la función secuencial original. Medir tiempos de ejecución para este nuevo algoritmo. Por último, sacar tiempos para múltiples planificaciones de reparto del bucle, usando al menos las siguientes planificaciones de OpenMP:

- Estática sin *chunk*.
- Estática con *chunk* 1.
- Dinámica.

Recuérdese que puede resultar cómodo especificar la planificación con la correspondiente variable de entorno, para lo cual se debe indicar en el código `runtime` como tipo de planificación.