UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf

Computación Paralela (CPA)

Práctica 1: Paralelización con OpenMP

Curso 2021/22

Índice

Introducción

Esta práctica consta de 3 sesiones, correspondientes a cada uno de los 3 apartados del boletín. 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`, `Lenna1k.ppm`
Sesión 3	Números primos	`primo_grande.c`, `primo_numeros.c`

Las prácticas de la asignatura están preparadas para ser realizadas sobre los ordenadores del laboratorio, usando el sistema operativo Linux, o bien accediendo al mismo entorno a través del escritorio remoto DSIC-LINUX desde `https://polilabs.upv.es/`. Será necesario también conectarse desde este entorno al clúster de cálculo `kahan` mediante `ssh`, como se comenta en el apartado 1.3.

Empezaremos por crear una carpeta en la unidad `W` para el código fuente de la práctica. Es aconsejable que la ruta de dicha carpeta sea corta y sin espacios, porque más adelante necesitaremos teclearla a menudo. Por ejemplo, podría ser `W/cpa/prac1` (en adelante supondremos que esa es la ruta de la carpeta). Guardaremos en esa carpeta los ficheros `.c` y `.ppm` de la práctica (directamente, sin subcarpetas).

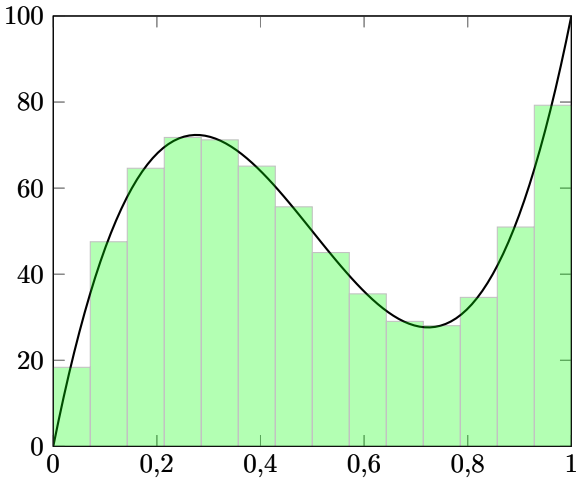

Figura 1: Interpretación geométrica de una integral.

1. Integración numérica

En este primer apartado aprenderás a compilar programas OpenMP y a realizar la paralelización de bucles sencillos. Verás cómo ejecutar programas, tanto sobre la máquina local como sobre el cluster de cálculo `kahan`.

Para ello, consideraremos el problema del 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. Para ello, abre una terminal sobre la carpeta donde se encuentra el fichero `integral.c` y ejecuta la orden:

```
$ gcc -Wall -o integral integral.c -lm
```

Al hacerlo, se habrá creado en la misma carpeta un programa ejecutable llamado `integral`. El significado de las opciones de compilación es el siguiente:

- `-o fichero_ejecutable`: especifica el nombre del fichero ejecutable o de salida (*output*).

```
/* 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.

- `-lm`: compilar con la librería matemática. Esta opción es necesaria si el programa utiliza funciones matemáticas como `sin`, `cos`, `pow`, `exp`...
- `-Wall` (opcional): mostrar todas las advertencias (*warnings*).

A continuación, ejecuta el programa. 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 (número de rectángulos) como segundo argumento del programa (por defecto son 100000 rectángulos). Por ejemplo:

```
$ ./integral 1 500000
```

1.1. Paralelización de la primera variante

En esta sección debes modificar el código de `integral.c` para realizar el cálculo de forma paralela utilizando OpenMP. En vez de modificar el fichero original, haz una copia con otro nombre (por ejemplo, `pintegral.c`).

Empieza por hacer que el programa simplemente muestre el número de hilos con los que se ejecuta. Para ello, modifica el programa como se muestra de forma esquemática en la Figura 3. Es decir, añade una

```
...
#include <omp.h>
...
int main(int argc, char *argv[]) {
   ...
   printf("Numero de hilos: %d\n", funcion_openmp_adecuada());
   ...
}
```

Figura 3: Modificación (incompleta) para mostrar el número de hilos.

sentencia `printf` en la función `main` para mostrar el número de hilos, el cual se obtiene mediante la función de OpenMP adecuada. Se debe incluir además el fichero cabecera `omp.h`.

Para compilar el programa, debes añadir la opción `-fopenmp`, por ejemplo:

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

Y para ejecutar el programa con varios hilos, por ejemplo 4:

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

Ten en cuenta que **no debe haber ningún espacio** en "`OMP_NUM_THREADS=4`".

Al ejecutar el programa, se mostrará el número de hilos utilizado. ¿Muestra 1 solo hilo en vez de 4? Ten en cuenta que la función de OpenMP para obtener el número de hilos devuelve el número de hilos **activos**, y fuera de una región paralela solo hay 1 hilo activo. Debes solucionar este problema para que se muestre el número de hilos con los que realmente se ha ejecutado el programa. Además, debes hacer que el mensaje con el número de hilos se muestre una sola vez.

Una vez solucionado lo anterior, nos ocupamos de la paralelización de la primera variante de cálculo de la integral (`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.

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.

1.3. Ejecución en el cluster

En este apartado utilizarás el clúster de cálculo `kahan` para lanzar ejecuciones, lo que te permitirá hacer uso de un mayor número de *cores*.

`kahan` es un clúster compuesto por 4 nodos de cálculo, cada uno de ellos con 64 cores, y un nodo *front-end* al cual se conectan los usuarios para compilar y lanzar ejecuciones. Todos los *cores* de un mismo nodo

```
#!/bin/bash
#SBATCH --nodes=1
#SBATCH --time=5:00
#SBATCH --partition=cpa

OMP_NUM_THREADS=3 ./pintegral 1
```

Figura 4: Fichero de trabajo para ejecutar en el sistema de colas.

comparten la memoria de dicho nodo, pero no pueden acceder a la memoria de otros nodos. En el material de la asignatura puedes encontrar más detalles sobre el clúster `kahan`.

Para trabajar con `kahan`, debes conectarte al nodo *front-end* mediante `ssh`:

```
$ ssh -l login@alumno.upv.es kahan.dsic.upv.es
```

donde `login` es tu nombre de usuario UPV. Tras ejecutar la orden, estarás en tu directorio `home` de `kahan`.

Si ejecutas el comando `ls` comprobarás que hay un directorio `W`, que corresponde a tu unidad `W` de la UPV. Recuerda que los ficheros de la práctica deberían estar en una carpeta de `W`, por ejemplo `W/cpa/prac1`, como se indicó en el apartado de Introducción. Comprueba que la siguiente orden muestra un listado de los ficheros de código fuente de la práctica, entre los cuales estará el fichero `pintegral.c` creado anteriormente:

```
$ ls W/cpa/prac1
```

A continuación habrá que compilar el programa, igual que hemos hecho anteriormente, pero esta vez en `kahan`. Es importante tener en cuenta que el ejecutable generado no debe estar en la carpeta `W`, ya que dicha carpeta no es accesible desde los nodos de cálculo de `kahan`. Por ello, crearemos una carpeta (con `mkdir`) en la que dejar los ejecutables, nos situaremos en ella (`cd`) y compilaremos:

```
$ mkdir prac1
$ cd prac1
$ gcc -Wall -fopenmp -o pintegral ~/W/cpa/prac1/pintegral.c -lm
```

donde al compilar indicamos la ruta del fichero `pintegral.c` (el carácter `~` indica el directorio `home` y habitualmente se puede teclear con `AltGr+4`).

Ahora puedes ejecutar el programa, por ejemplo:

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

Acabas de ejecutar el programa en el *front-end*, **no en los nodos de cálculo** de `kahan`. Aunque es posible ejecutar un programa en el *front-end*, solo debe hacerse para ejecuciones muy cortas. El *front-end* solo debe usarse para conectarse por `ssh`, compilar y lanzar trabajos sobre el clúster de la manera que se explica a continuación.

La ejecución de trabajos en el clúster se debe hacer mediante el **sistema de colas SLURM**. Para ello crearemos un **fichero de trabajo**, el cual es un *script* con diferentes opciones del sistema de colas seguidas por los comandos que deseamos ejecutar. En la Figura 4 hay un ejemplo de un fichero de trabajo, en el que se ejecuta un programa OpenMP con 3 hilos de ejecución (última línea del fichero). Las líneas que empiezan por `#SBATCH` especifican distintas opciones del sistema de colas. En este caso el trabajo utilizará la cola (partición) denominada *cpa*, usando un nodo del clúster (con sus 64 cores) y con un tiempo máximo de 5 minutos para su ejecución.

Copia el texto de la Figura 4 en un fichero (por ejemplo `jobopenmp.sh`) y guárdalo en la carpeta `W/cpa/prac1`. Cambia el número de hilos con los que se ejecutará el programa, poniendo el valor que quieras. ¿Tendría sentido cambiar el número de nodos por un valor distinto de 1?

A continuación se debe lanzar el trabajo al sistema de colas, para lo cual se utiliza la orden `sbatch`. Suponiendo que el fichero de trabajo se llama por ejemplo `jobopenmp.sh`, bastaría con ejecutar en el *front-end* (desde el directorio donde está el ejecutable, en este caso `~/prac1`):

```
$ sbatch ~/W/cpa/prac1/jobopenmp.sh
```

En la terminal se mostrará el número del trabajo.

Una vez lanzado, el sistema de colas se encarga de asignar a nuestro trabajo los recursos que necesite (en este caso un nodo del clúster) cuando estos estén disponibles, manteniendo a nuestro trabajo en espera hasta entonces. De esta manera, se asegura que los nodos asignados a un trabajo no son usados por ningún otro trabajo.

¿Qué ocurre con los mensajes que debería mostrar el programa? Esos mensajes no se muestran en la terminal, sino que se almacenan en un fichero. Por ejemplo, si el número de trabajo es 620, tras su ejecución se creará el fichero `slurm-620.out`. Podemos mostrar su contenido con la orden `cat`, por ejemplo:

```
$ cat slurm-620.out
Número de hilos: 16
Valor de la integral = 1.000000000041
```

También podemos copiar el fichero generado a la carpeta `W/cpa/prac1`, y una vez allí visualizarlo con cualquier editor de texto:

```
$ cp slurm-620.out ~/W/cpa/prac1
```

Se puede consultar el estado de las colas con la orden `squeue`, por ejemplo:

```
$ squeue
      JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
        620       cpa jobopenmp     john  R       0:01      1     kahan01
```

Para cada trabajo se muestra su estado (ST), entre los que destacan: en cola o pendiente (PD), en ejecución (R) o finalizado con éxito (CD).

También se puede cancelar un trabajo con la orden `scancel`:

```
$ scancel 620
```

1.4. Toma de tiempos

En muchos casos interesa medir el tiempo de ejecución del programa, o de una parte de él, para poder compararlo con el tiempo del programa secuencial, y determinar la mejora obtenida.

Para medir el tiempo de un fragmento de un programa utilizaremos la función `omp_get_wtime()` de OpenMP, que devuelve el tiempo transcurrido (en segundos) desde un instante inicial fijo. La forma de usar esta función se ilustra en la Figura 5.

Mide el tiempo de ejecución del programa paralelo en el clúster usando un número de rectángulos de 500000000 (500 millones), con 1 hilo y con 16 hilos, comprobando la reducción de tiempo obtenida.

¿Podemos aumentar el número de hilos indefinidamente, mejorando siempre el tiempo de ejecución? Si no es así, ¿hasta qué número de hilos piensas que habría mejora?

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.

```
...
#include <omp.h>
...

int main(int argc, char *argv[]) {
   double t1, t2;
   ...
   t1 = omp_get_wtime();
   ...  /* Fragmento de codigo a cronometrar */
   t2 = omp_get_wtime();
   printf("Tiempo: %f\n", t2-t1);
}
```

Figura 5: Toma de tiempo de un fragmento de código.

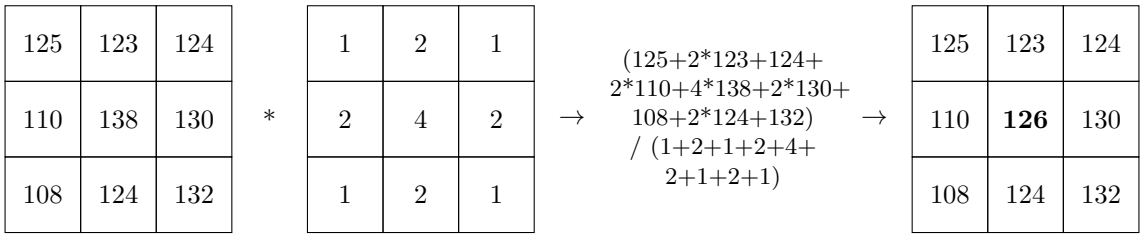

Figura 6: Modelo de aplicación de una media ponderada en el filtrado de imágenes.

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. Tendrás 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 una cierta distancia, llamada 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 vecinos, 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.

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 6 muestra un esquema. En esta figura partimos de una imagen (izquierda) sobre cuyo pixel central (con el valor de color 138) se aplica 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.

```
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 7: 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 8: Formato de un fichero de imágenes PPM.

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 7.

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 8 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` usando 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.

Comprueba el correcto funcionamiento del programa. Para ello, compílalo y ejecútalo en la máquina local. Observarás que el programa ha creado el fichero `lenna-fil.ppm` (el nombre especificado en `IMAGEN_SALIDA`)

[1] `http://www.irfanview.com`

Figura 9: Imagen de referencia (`Lenna.ppm`) antes (izquierda) y después (derecha) de aplicar un filtrado de un paso y radio 5.

con el resultado de aplicar el filtro sobre la imagen del fichero `Lenna.ppm`, obtenida de un conocido benchmark[2] (ver Figura 9).

Haz una copia del fichero `lenna-fil.ppm` que has obtenido, llamándole `ref.ppm`. Ese fichero contiene la salida del programa original, que usaremos como referencia para compararla más adelante con la salida de los programas modificados a desarrollar, para comprobar si funcionan correctamente.

A continuación, modifica el programa para que muestre el número de hilos con los que se ejecuta, tal como hiciste en la sesión anterior, y también el tiempo de ejecución empleado por la función que realiza el filtrado de la imagen.

Compílalo y ejecútalo en el cluster. Para ello, como se explicó en el apartado 1.3, debes conectarte mediante `ssh` al nodo *front-end* de `kahan`, cambiar al directorio `~/prac1` y compilar. En este caso también deberás copiar los ficheros de imágenes desde la carpeta del código fuente (`~/W/cpa/prac1`) a la carpeta de ejecutables (`~/prac1`), por ejemplo mediante la orden:

```
$ cp ~/W/cpa/prac1/*.pmm ~/prac1/
```

A continuación, crea un fichero de trabajo y lanza el trabajo mediante `sbatch` (de nuevo, ver apartado 1.3).

2.3. Implementación paralela

Existen diferentes aproximaciones para realizar la implementación paralela mediante OpenMP, dependiendo del bucle que se elija para su paralelización. El trabajo se centrará en analizar los cinco bucles de la Figura 7 y decidir (y probar) qué bucles son susceptibles de ser paralelizados y qué variables deberían ser compartidas o privadas. Para ello, para cada uno de los cinco bucles, deberás seguir los siguientes pasos:

1. Decidir si el bucle se puede paralelizar. Para ello, analizar si las diferentes iteraciones del bucle en cuestión 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).

[2] `http://en.wikipedia.org/wiki/Standard_test_image`

```
Tiempo secuencial: _____________

Tiempo de versiones paralelas:
            Version 1  Version 2  ...
  2 hilos   _________  _________  ...
  8 hilos   _________  _________  ...
 32 hilos   _________  _________  ...
```

Figura 10: Plantilla para la toma de tiempos.

Si el bucle no se puede paralelizar, los siguientes pasos no se realizarían.

2. Escribir la(s) directiva(s) para paralelizar el bucle, prestando atención a qué variables deberán ser privadas a cada hilo y cuáles compartidas entre todos.
3. Ejecutar el programa modificado (preferiblemente en el clúster).
4. Comprobar que el fichero generado por el programa modificado (`lenna-fil.ppm`) es exactamente igual que el producido por el programa original (`ref.ppm`). Para ello, ejecuta la orden:

   ```
   $ cmp lenna-fil.ppm ref.ppm
   ```

   Si los ficheros son iguales, la orden anterior no muestra ningún mensaje.

A continuación se proporcionan algunas orientaciones que pueden ser de utilidad:

- Puede ser más sencillo empezar considerando el bucle más interno, dado que hay menos variables implicadas, luego el bucle inmediatamente por encima, y así sucesivamente hasta acabar con el bucle más externo.
- No se puede utilizar la cláusula `reduction` sobre un miembro de una variable de tipo `struct`. Si lo necesitas, prescinde del `struct` y, en su lugar, declara una variable por cada miembro del mismo.
- Algunas versiones paralelas, aunque sean correctas, pueden ser extremadamente lentas, e incluso empeorar al aumentar el número de hilos. Por esa razón, se recomienda utilizar inicialmente solo 2 hilos al ejecutar las versiones paralelas.
- Ten en cuenta que el clúster se usa de forma compartida por todos los alumnos, por lo que debes evitar llenar la cola con tus trabajos. Puedes ejecutar varias órdenes dentro de un mismo fichero de trabajo.

Conviene que te hagas la siguiente pregunta: ¿tiene sentido paralelizar dos o más bucles a la vez?

Trata de obtener algo parecido a lo que se muestra en la Figura 10, en la que se recoge el tiempo de ejecución del programa secuencial, junto con el tiempo correspondiente a cada una de las versiones paralelas desarrolladas, para distinto número de hilos. Si una versión paralela es más lenta con 2 hilos que el programa secuencial, no es recomendable que la ejecutes con más hilos, porque solo empeorará.

Es fácil ver que hay algunas versiones paralelas mucho más eficientes que otras. ¿Cuáles son las versiones más eficientes y por qué?

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

```
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 11: Algoritmo secuencial para determinar si n es primo.

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 11. 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 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 12.

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

En este apartado debes 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?

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

Figura 12: Algoritmo a paralelizar: busca el mayor primo que cabe en un entero sin signo de 8 bytes.

```
#pragma omp parallel ...
{
   for (i = ...; p && i <= ...; i += ...)
      if (n % i == 0) p = 0;
}
```

Figura 13: Esquema de paralelización del bucle de la función `primo`.

Efectivamente, OpenMP no permite usar la directiva `parallel for` en este caso. Recuerda que la directiva `parallel for` equivale a una directiva `parallel` seguida de una directiva `for`. Esta segunda directiva se encarga de repartir automáticamente las iteraciones del bucle entre los hilos, pero para poder hacerlo es necesario que el inicio, final e incremento del bucle estén perfectamente delimitados. En la función `primo`, el inicio y el incremento de la variable del bucle (`i`) están claros, pero el valor final es desconocido de antemano: puede que llegue a `s` o puede que el bucle acabe antes porque `p` pase a ser falso. Por tanto, OpenMP no permite usar la directiva `for` (ni tampoco `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? 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. Es decir, hacer de forma explícita lo que OpenMP hace de forma automática mediante la directiva `for`.

La paralelización del bucle tendrá la forma que puede verse en la Figura 13, donde cada hilo de la región paralela debe realizar solo un subconjunto de las iteraciones del bucle original. Las iteraciones que realiza cada hilo están determinadas por el valor inicial de `i`, su incremento y su valor final. Por tanto, debes modificar algunos de estos elementos para conseguir que cada hilo procese un subconjunto de iteraciones diferente.

Por ejemplo, si suponemos `s`=19, los valores de la variable `i` de las distintas iteraciones del bucle completo son: 3, 5, 7, 9, 11, 13, 15, 17, 19. Una manera de repartir las iteraciones sería asignar a cada hilo un bloque de iteraciones consecutivas. Por ejemplo, para 3 hilos:

Hilo 0	3	5	7
Hilo 1	9	11	13
Hilo 2	15	17	19

Otra manera de repartir las iteraciones es hacerlo de forma cíclica:

Hilo 0	3	9	15
Hilo 1	5	11	17
Hilo 2	7	13	19

Este reparto cíclico es más fácil de implementar, por lo que es el que se recomienda que uses en este caso. Lógicamente, el reparto deberá funcionar correctamente para cualquier valor de `s` y del número de hilos.

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

Necesitarás usar funciones de OpenMP para obtener el número de hilos y el identificador de cada hilo. Evita llamar a estas funciones en cada iteración del bucle, por cuestiones de eficiencia. Desarrolla esta nueva versión paralela y mide el tiempo de ejecución necesario.

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.

Nota: Es 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 14. Prueba 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 si un número es primo.

Sin embargo, esa paralelización no conduce a buenas prestaciones. La razón es que los números primos iniciales son muy pequeños y cada uno de ellos supone muy poco trabajo, de forma que no hay ganancia al repartir el trabajo entre múltiples hilos.

Una estrategia mejor sería paralelizar el bucle del programa principal (que en este caso sí que parece fácilmente paralelizable mediante directivas OpenMP) y usar la función `primo` secuencial original. Desarrolla una nueva versión paralela basada en ese enfoque y mide su tiempo de ejecución. Por último, saca tiempos para múltiples planificaciones de reparto del bucle, usando al menos las siguientes planificaciones de OpenMP:

- Estática sin especificar tamaño de *chunk*.
- Estática con *chunk* 1.
- Dinámica.

Recuérdese que en la directiva `for` de OpenMP, la planificación se puede indicar de dos maneras alternativas:

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

1. Especificando directamente la planificación con una cláusula `schedule` en la directiva `for` de OpenMP (por ejemplo, `schedule(static,6)`).

2. Usando la cláusula `schedule(runtime)` en la directiva `for`, y dándole valor a la variable de entorno `OMP_SCHEDULE` al ejecutar el programa (por ejemplo, `OMP_SCHEDULE="static,6"`). Esta opción tiene la ventaja de que se puede cambiar la planificación sin tener que recompilar el programa.