Práctica 3
Paralelismo a nivel de hilos: Paralelización mediante OpenMP y programación asíncrona del revelado digital de fotografías

Objetivos:

- Aprender a paralelizar una aplicación en una máquina paralela de memoria centralizada mediante hilos (*threads*) usando el estilo de *variables compartidas*.
- Estudiar la API de OpenMP y aplicar distintas estrategias de paralelismo en su aplicación.
- Estudiar la API de C++ de programación asíncrona para explotar el paralelismo funcional.
- Aplicar métodos y técnicas propios de esta asignatura para estimar las ganancias máximas y la eficiencia del proceso de paralelización.
- Aplicar todo lo anterior a un problema de complejidad y envergadura suficiente.

Desarrollo:
En esta práctica, tendréis que paralelizar, usando OpenMP y programación asíncrona, la solución a un problema dado para aprovechar los distintos núcleos de los que dispone cada ordenador de prácticas. Se paralelizará por tanto para un sistema multiprocesador (máquina paralela de memoria centralizada), en el que todos los núcleos de un mismo encapsulado ven la misma memoria, es decir, un puntero en un núcleo es el mismo puntero para el resto de los núcleos del microprocesador.

Tarea 0.1 Entrenamiento previo OpenMP:

Observa el siguiente programa en C donde se suman dos vectores de *floats* empleando OpenMP para paralelizar el cálculo.

```
#include <omp.h>
#define N 1000
#define CHUNKSIZE 100

main(int argc, char *argv[]) {

    int i, chunk;
    float a[N], b[N], c[N];

    /* Inicializamos los vectores */
    for (i=0; i < N; i++)
         a[i] = b[i] = i * 1.0;
    chunk = CHUNKSIZE;

    #pragma omp parallel shared(a,b,c,chunk) private(i)
    {
         #pragma omp for schedule(dynamic,chunk) nowait
         for (i=0; i < N; i++)
                c[i] = a[i] + b[i];
    }   /* end of parallel region */

}
```

Revisa la documentación de OpenMP (API, tutoriales, este enlace: https://computing.llnl.gov/tutorials/openMP/, etc...) y responda:

0.1.1 ¿Para qué sirve la variable **`chunk`**?

0.1.2 Explica **completamente** el *pragma* :

```
#pragma omp parallel shared(a,b,c,chunk) private(i)
```

- ¿Por qué y para qué se usa **`shared(a,b,c,chunk)`** en este programa?
- ¿Por qué la variable `i` está etiquetada como **`private`** en el pragma?

0.1.3 ¿Para qué sirve **`schedule`**? ¿Qué otras posibilidades hay?

0.1.4 ¿Qué tiempos y otras medidas de rendimiento podemos medir en secciones de código paralelizadas con OpenMP?

Tarea 0.2: Entrenamiento previo std::async

Observa el siguiente programa en c++ donde se llama a dos funciones con std::async:

```
#include <iostream>
#include <future>
#include <chrono>

int task(int id, int millis) {
    std::this_thread::sleep_for(std::chrono::milliseconds(millis));
    std::cout<<"Task "<<id<<" completed"<<std::endl;
    return id;
}

int main() {
    auto start = std::chrono::high_resolution_clock::now();
    std::future<int> task1 = std::async(std::launch::async, task, 1, 2000);
    std::future<int> task2 = std::async(std::launch::async, task, 2, 3000);

    task1.wait();
    int taskId = task2.get();

    auto end = std::chrono::high_resolution_clock::now();
    auto elapsed = std::chrono::duration_cast<std::chrono::milliseconds>(end-start);

    std::cout<<"Completed in: "<<elapsed.count()<<"ms"<<std::endl;
}
```

Para compilarlo usa g++ y enlaza con la librería pthreads con -lpthread.
Revisa la documentación de std::async y contesta a las siguientes preguntas:

0.2.1 ¿Para qué sirve el parámetro `std::launch::async`?

0.2.2 Calcula el tiempo que tarda el programa con `std::launch::async` y `std::launch::deferred`. ¿A qué se debe la diferencia de tiempos?

0.2.3 ¿Qué diferencia hay entre los métodos `wait` y `get` de `std::future`?

0.2.4 ¿Qué ventajas ofrece `std::async` frente a `std::thread`?

Tarea 0.3: Entrenamiento previo std::vector

Observa el siguiente programa en c++ donde se inicializa un vector stl y se rellena con valores:

```
#include <vector>
#include <iostream>
```

```
#include <chrono>

int main() {
        // default initialization and add elements with push_back
        auto start = std::chrono::high_resolution_clock::now();

        std::vector<float> v1;
        for (int i = 0; i < 10000; i++)
                v1.push_back(i);

        auto end = std::chrono::high_resolution_clock::now();
        auto elapsed =
        std::chrono::duration_cast<std::chrono::milliseconds>(end-start);
        std::cout<<"Default initialization: "<<elapsed.count()<<"ms"<<std::endl;

        // initialized with required size and add elements with direct access
        start = std::chrono::high_resolution_clock::now();

        std::vector<float> v2(10000);
        for (int i = 0; i < 10000; i++)
                v1[i] = i;
        end = std::chrono::high_resolution_clock::now();

        elapsed = std::chrono::duration_cast<std::chrono::milliseconds>(end-start);
        std::cout<<"Initialization with size: "<<elapsed.count()<<"ms"<<std::endl;
}
```

Responde a las siguientes preguntas:

0.3.1: ¿Cuál de las dos formas de inicializar el vector y rellenarlo es más eficiente? ¿Por qué?

0.3.2: ¿Podría ocurrir algún problema al paralelizar los dos bucles for? ¿Por qué?

Tarea 1: Estudio del API OpenMP [Parte individual obligatoria (25% de la nota)]

Se deberá estudiar el API de OpenMP y su uso con GNU GCC, comprobando el correcto funcionamiento de algunos de los ejemplos que hay disponibles en Internet (p.ej. https://lsi.ugr.es/jmantas/ppr/ayuda/omp_ayuda.php)

Cada miembro del grupo* deberá realizar un pequeño tutorial a modo de "libro de recetas de paralelismo con OpenMP" con **distintos ejemplos de aplicación** del paralelismo que ofrece OpenMP en función de distintas **estructuras de software**.

Tarea 2: Paralelización del revelado digital de fotografías

Todos los grupos de cada turno de prácticas deben acometer la paralelización del problema planteado. Para ello se analizará la solución secuencial facilitada siguiendo las indicaciones del enunciado y los profesores. Posteriormente, se transformará ésta, utilizando OpenMP y programación asíncrona, para que incorpore paralelismo a nivel de hilos.

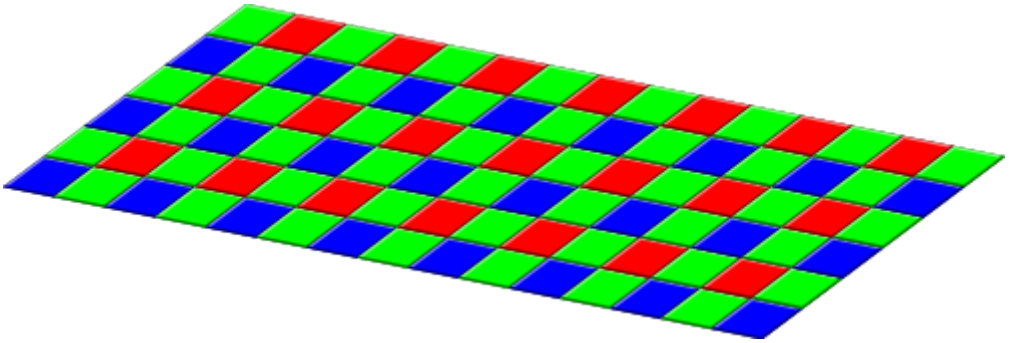
Problema:

El problema planteado es el revelado digital de fotografías.

En la fotografía analógica tradicional, la cámara capta la imagen a través del objetivo y la proyecta sobre una lámina fotosensible. Esta lámina se llama "negativo" ya que se captura

una imagen de intensidad invertida. Una vez capturados los negativos, para revelar la imagen capturada en un papel fotográfico se utiliza un proceso químico que hace reaccionar al papel ante la exposición a la luz proyectada a través del negativo.

En la fotografía digital, el proceso de captura y revelado de fotografías es similar conceptualmente pero muy diferente en la práctica. En el caso de una cámara digital, la luz que llega a través del objetivo es proyectada en un sensor digital conocido como CCD. Este sensor digital es un array de píxeles que producen una intensidad eléctrica directamente proporcional a la intensidad de la luz recibida. La intensidad eléctrica de cada uno de estos píxeles es leída por el driver de la cámara para generar una imagen digital que se guarda en la tarjeta de memoria. Este proceso que en principio parece sencillo, se complica cuando se quiere obtener una imagen en color, ya que los píxeles del sensor únicamente reaccionan a la intensidad de la luz y no a la longitud de onda (color). Para solucionar este problema, se utiliza el denominado filtro de Bayer que consiste en colocar filtros de los tres colores primarios, rojo, verde y azul, de forma alternativa en cada uno de los píxeles del sensor:



Como se puede observar, existe una proporción mayor de píxeles verdes que rojos y azules. Esto es debido a que el ojo humano es más sensible al color verde. No todas las cámara utilizan el mismo patrón, ni siquiera los mismos colores primarios, pero el más común es el que aparece en la imagen anterior y es conocido como patrón RGGB o Bayer.

Por tanto, la imagen que sale del sensor de la cámara, es una matriz bidimensional, del mismo tamaño en píxeles que el sensor, con las intensidades de color RGB intercaladas siguiendo el patrón Bayer. Esta imagen se conoce como imagen RAW (cruda) y es el equivalente al negativo en la fotografía analógica. Esta imagen RAW contiene las intensidades registradas por cada uno de los píxeles del sensor sin ningún otro procesado adicional. Para obtener la imagen final, se aplican una serie de procesos a esa imagen RAW.

Por lo general, cuando realizamos una fotografía con una cámara digital, lo que recibimos en la tarjeta de memoria es la imagen ya procesada en formato JPEG o TIFF. La mayoría de cámaras de gama media-alta (incluso algunos teléfonos móviles o GoPro) proporcionan la posibilidad de guardar tanto la imagen procesada JPEG como la imagen RAW. Trabajar con la imagen RAW proporciona una serie de ventajas ya que da mayor libertad al fotógrafo para editar, pudiendo obtener un resultado final de mayor calidad. La imagen JPEG también puede editarse con editores de fotografía como Photoshop o GIMP, sin embargo, esta imagen ya procesada está comprimida y en un rango dinámico

mucho menor que la imagen RAW, 8 bits por pixel (256 niveles de gris) del JPEG frente a hasta 16 bits por píxel (65536 niveles de gris) de la imagen RAW. Esto permite aplicar filtros más agresivos sin perder calidad de imagen y reduciendo la aparición de artefactos como el banding.

Los procesos que se aplican a la imagen RAW dependen del driver de la cámara o del software de revelado digital utilizado (Lightroom, Darktable, RawTherapee, …) pero por lo general incluyen los siguientes pasos:

- **Debayer o demosaico**: este es el proceso por el cual se separan los píxeles rojos, verdes y azules en un canal individual y se interpolan los píxeles faltantes para obtener la imagen RGB.
- **Balance de blancos**: es el proceso por el cual se ajusta la temperatura del color de forma que los colores blancos se vean como tal y no con tono rojizo o azulado.
- **Corrección de gamma**: la imagen que se obtiene del sensor viene dada por la sensibilidad de los píxeles que reaccionan de forma lineal a la intensidad de la luz. Sin embargo, nuestros ojos no reaccionan así, son menos sensibles a las zonas más oscuras o más brillantes por lo que la imagen lineal salida del sensor es percibida por el ojo humano como una imagen plana sin contraste. Este proceso, pues, convierte la imagen lineal a una imagen no lineal más similar a como el ojo humano percibe la luz.
- **Sharpening**: la imagen resultante del sensor suele carecer de nitidez ya que la intensidad de un píxel suele estar afectada por la intensidad recibida por los píxeles vecinos debido al patrón de Bayer. Por tanto, se suele aplicar un proceso que aumente la nitidez de la imagen final.

Tarea 2.1: Analiza el código e identifica los distintos procesos

En esta tarea deberéis analizar el código utilizando las estrategias vistas en la práctica anterior para identificar los distintos procesos que se aplican sobre la imagen RAW y elaborar un diagrama de dependencias.
Identifica si se puede ejecutar algunos procesos en paralelo y las dependencias entre ellos.

Tarea 2.2: Analiza el código de cada proceso

En esta tarea deberéis analizar el código correspondiente a cada proceso identificado en la tarea 3.1. Realiza un diagrama de flujo de cada uno e identificar posibles puntos paralelizables.

Tarea 2.3 Paralelización

Identificadas las partes paralelizables del código, ahora toca aplicar el paralelismo con OpenMP y programación asíncrona. Haced pruebas paralelizando diferentes partes y observad la ganancia de rendimiento.
Responded a las siguientes preguntas:

- ¿Se obtiene un mejor rendimiento paralelizando todos las partes posibles?
- ¿Se degrada el rendimiento al paralelizar ciertas partes?
- Si es así, ¿a qué creéis que se debe esa degradación?

Objetivos:

- Obtener una versión paralela con OpenMP y std::async de la solución secuencial proporcionada
- Analizar y explotar en todos los casos los diferentes tipos posibles de paralelismo que se perciban en la solución secuencial dada.
- Realizar pruebas suficientes que permitan estimar valores de rendimiento de la versión paralela frente a la secuencial.

Pasos:

a. Analizar la solución secuencial facilitada para asegurar la comprensión del problema. Prestar atención a detalles vistos en prácticas anteriores que tengan importancia para diseñar la solución paralela: variables, talla del problema, etc.
b. Realice una representación gráfica en forma de diagrama de dependencias del funcionamiento de la solución secuencial dada.
c. Utilizando OpenMP y std::async y a la vista de los detalles anteriores, diseñe una solución paralela a nivel de hilo a partir de la versión secuencial
d. Realice pruebas de su versión paralela para diferentes casos y configuraciones tomando medidas de tiempos y ganancias.
e. Realice una representación gráfica en forma de grafo de control de flujo del funcionamiento de la solución paralela basada en OpenMP y std::async.
f. Comente los siguientes aspectos tanto de la solución secuencial como de la paralela según se indique:

Sobre el código implementado

- Comentar las porciones de código de más interés tanto de la solución secuencial como de la paralela implementadas:
  - Aspectos que definan la talla del problema.
  - Estructuras de control del código de especial interés en la solución al problema.
  - Es importante que se justifique **lo más detalladamente** posible los cambios que se hayan realizado con respecto a la versión secuencial para paralelizar la solución.
  - Instrucciones y bloques de OpenMP o std::async utilizados para la paralelización del código.

Sobre el paralelismo explotado

- **Revisando la documentación de la "*Unidad 3. Computación paralela*".** Indique lo siguiente con respecto a su práctica:
  - Tipos de paralelismo usado.
  - Modo de programación paralela.
  - Qué alternativas de comunicación (explícitas o implícitas emplea su programa).
  - Estilo de la programación paralela empleado.
  - Tipo de estructura paralela del programa

Sobre los resultados de las pruebas realizadas y su contexto

- Caracterización de la máquina paralela en la que se ejecuta el programa. (p.ej. Número de nodos de cómputo, sistema de caché, tipo de memoria, etc.).

En Linux use la orden: `cat /proc/cpuinfo` para acceder a esta información o `lscpu`.

- ¿Qué significa la palabra `ht` en la salida de la invocación de la orden anterior? ¿Aparece en su ordenador de prácticas?
- Calcular lo siguiente (con gráficas asociadas y su explicación breve):
  - Ganancia en velocidad (*speed-up*) en función del número de unidades de cómputo (*threads* en este caso) y en función de los parámetros que modifiquen el **tamaño del problema** (p.ej. dimensiones de una matriz, número de iteraciones considerado, etc.…).
    - Comente los resultados de forma razonada. ¿Cuál es la ganancia en velocidad máxima teórica?
      Nota: Puede entregar gráficas en 2D o 3D según resulte más ilustrativo.
  - **Responda de forma justificada:** ¿Cuál es la implementación más eficiente de las 2?

Tarea 3: Redacción de una memoria en la que se analice el diseño utilizando OpenMP y std::async y los resultados obtenidos.
Se redactará una memoria en la que se comenten y expliquen diferentes aspectos tanto de la versión secuencial como de la implementación paralela propuesta. También se analizarán los resultados obtenidos probándolas. La memoria deberá dar clara respuesta a las cuestiones planteadas en cada tarea. Cada profesor indicará la forma en que les será entregadas las partes individuales de cada miembro de los grupos.

Notas generales a la práctica:

- Las respuestas y la documentación generadas de las tareas se integrarán en la memoria conjunta y estructurada, incluyendo los apartados individuales como anexo, que se entregará al final de la práctica 3.
- Para el desarrollo y la memoria se utilizará la plataforma de desarrollo GitHub. Esta permitirá una gestión de un **repositorio común de trabajo** que deberá crear cada grupo incluyendo a sus miembros y al profesor. Este dará a conocer su usuario en dicha plataforma para ser añadido.
- La implementación realizada tendrá que poder ejecutarse bajo el sistema operativo Linux del laboratorio de prácticas.
- El código adjuntado utiliza la librería OpenCV para ejecutar el procesamiento de la imagen. En los PCs de los laboratorios ya viene instalada esta librería, en cambio, para hacerlo funcionar en vuestros PCs, es necesario compilar e instalar OpenCV. Para ello seguid las instrucciones que aparecen en este enlace. Otra opción es utilizar la máquina virtual de la EPS.
- Para compilar, el código adjuntado incluye un archivo CMakeLists.txt que contiene la información de dependencias de librerías externas y cómo construir los fuentes. Este archivo sirve para construir un Makefile utilizando CMake. Para ello, crea una carpeta llamada build en el mismo directorio y ejecuta el comando `cmake -S . -B ./build`. Una vez ejecutado el comando, dentro de la carpeta build habrá un fichero Makefile con el que poder ejecutar el comando make para construir el

ejecutable. Dentro de la carpeta build, ejecuta el comando make -j4 para compilar el ejecutable.

- Para ejecutar el proceso de revelado hay que usar el comando `./unraw <fichero_raw> <fichero_de_salida>`, por ejemplo, `./unraw ../_DSC2078.NEF output.jpg`.
- La información referente a OpenMP se encuentra en www.openmp.org. Para compilar use el make generado por cmake que ya incluye el flag -fopenmp.
- La práctica se deberá entregar mediante el método que escoja su profesor de prácticas tras 5 sesiones.

***Nota:**

Los trabajos teóricos/prácticos realizados han de ser originales. La detección de copia o plagio supondrá la calificación de “0” en la prueba correspondiente. Se informará la dirección de Departamento y de la EPS sobre esta incidencia. La reiteración en la conducta en esta u otra asignatura conllevará la notificación al vicerrectorado correspondiente de las faltas cometidas para que estudien el caso y sancionen según la legislación.