Universidad de Alicante

Practica 3

PARALELISMO A NIVEL DE HILOS:

Paralelización mediante OpenMP y programación asíncrona del análisis forense de manipulación de imágenes digitales

Nicole Manresa Peiro
Jesús Ballesteros Navarro
Anja Raphaela Alum Millares

Índice

I. INTRODUCCIÓN

En esta práctica vamos a utilizar el paralelismo a nivel de hilos, aprendiendo a usar OpenMP y std::async en C++. La idea principal es aprovechar todos los núcleos del procesador para que un programa pueda hacer varias tareas al mismo tiempo y así tardar menos.

El ejercicio se basa en un análisis forense de imágenes digitales, donde el programa detecta si una imagen ha sido manipulada o editada (por ejemplo, con deepfakes o zonas recomprimidas). Primero se parte de una versión secuencial, que hace todo paso a paso, y después se modifica el código para crear una versión paralela, que ejecuta partes del proceso en paralelo.

Durante el desarrollo se estudian distintas formas de paralelizar el código, se hacen pruebas con diferentes números de hilos y se comparan los resultados de tiempo de ejecución para ver si realmente mejora el rendimiento. Al final, se analiza la ganancia en velocidad (speed-up), la eficiencia y se comentan las conclusiones sobre cuál método resulta más rápido y por qué.

II. OBJETIVO

El principal objetivo de esta práctica es aprender a paralelizar un programa usando diferentes herramientas en C++, concretamente OpenMP y std::async, para aprovechar mejor los múltiples núcleos del procesador. A partir de una versión secuencial del código que analiza imágenes manipuladas, se busca identificar qué partes del programa se pueden ejecutar en paralelo y modificarlas para mejorar el rendimiento. Además, se pretende comparar los tiempos de ejecución entre la versión secuencial y la paralela, calculando la ganancia en velocidad (speed-up) y la eficiencia de cada caso. También se quiere entender qué tipo de paralelismo es más adecuado para cada parte del código y cómo afecta el número de hilos al comportamiento del programa. En resumen, la práctica tiene como objetivo poner en práctica los conceptos de paralelismo vistos en clase, analizar los resultados obtenidos y sacar conclusiones sobre las ventajas y limitaciones de ejecutar un programa en paralelo.

III. TAREA 0: ENTRENAMIENTO PREVIO

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 */

}
```

0.1.1 ¿Para qué sirve la variable chunk?

Define el tamaño de bloques de iteraciones que se asignan a cada hilo. Con valor 100, cada hilo procesa 100 iteraciones antes de solicitar más trabajo. Mejora el balance entre eficiencia y sincronización

La variable chunk indica el tamaño del bloque de iteraciones del bucle for que se asigna a cada hilo cuando se reparte el trabajo.

- En este caso, CHUNKSIZE = 100 -> cada hilo procesará 100 elementos consecutivos del vector antes de que el OpenMP asigne un nuevo bloque.
- Si el bucle tiene N = 1000 iteraciones, y el chunk es 100, el bucle se divide en 10 bloques de 100 iteraciones.

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?

Crea una región paralela, donde varios hilos ejecutan el bloque de código simultáneamente.
shared(a,b,c,chunk): Estas variables son compartidas entre todos los hilos en la misma dirección de memoria.
a, b : vectores de entrada que todos leen.
C : vector de salida donde escriben en posiciones diferentes.
Chunk : configuración accesible para todos.
private(i) : Cada hilo tiene su propia copia de i. Evita condiciones de carrera en la variable de índice. Permite que cada hilo itere correctamente sobre su conjunto de iteraciones.

0.1.3 ¿Para qué sirve schedule? ¿Qué otras posibilidades hay?
schedule(dynamic,chunk): Asigna bloques de 100 iteraciones dinámicamente a hilos disponibles. Útil para carga desigual pero con mayor overhead.
Alternativas:
static: Distribuye equitativamente al inicio (bajo overhead, carga uniforme).
guided: Bloques grandes que disminuyen (balance intermedio).
auto: Decide automáticamente el compilador/runtime.

0.1.4 ¿Qué tiempos y otras medidas de rendimiento podemos medir en secciones de código paralelizadas con OpenMP?
Tiempos: *Tiempo total* con *omp_get_wtime(),* tiempo de overhead, tiempo por hilo.
Métricas: *Speedup* (tiempo secuencial/paralelo), *Eficiencia* (speedup/número hilos), *Load balance* (trabajo por hilo), Overhead paralelo (costo de sincronización).

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;
}
```

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

Especifica la política de lanzamiento de la tarea asíncrona. Con std::launch::async, obliga a que la función se ejecute inmediatamente en un nuevo hilo (ejecución verdaderamente paralela). Sin este parámetro, el comportamiento es indefinido según la implementación.

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?

Con std::launch::async: ~3000 ms (máximo de las dos tareas, ejecutadas en paralelo).
task1 (2000 ms) y task2 (3000 ms) se ejecutan simultáneamente.
El tiempo total es el de la tarea más larga.

Con std::launch::deferred: 5000 ms (suma secuencial de ambas tareas).
task1 se ejecuta cuando se llama task1.wait() (2000 ms).
task2 se ejecuta cuando se llama task2.get() (3000 ms).
Las tareas se ejecutan secuencialmente en el hilo principal, no en paralelo.
Diferencia: async paraleliza, deferred serializa el trabajo.

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

wait() solo espera a que la tarea termine, pero no devuelve ningún valor.
get() también espera, pero además devuelve el resultado de la tarea.
Después de llamar a get(), el valor del future ya no se puede volver a obtener.

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

std::async crea y gestiona los hilos automáticamente, sin necesidad de hacer join().
Devuelve un std::future para recoger el resultado de la función y manejar errores fácilmente.
Es más cómodo y seguro, ya que permite elegir si se ejecuta en paralelo o diferido y simplifica el control del flujo.

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;
}
```

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

La segunda forma es más eficiente porque:

Reserva la memoria de una vez (evita realocaciones).

No necesita copiar ni mover datos cuando crece el vector.

Complejidad:

push_back() -> O(n) amortizado, pero con reubicaciones.

Acceso directo v2[i] -> O(1).

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

Sí, si varios hilos acceden al mismo vector sin sincronización, puede haber: Condiciones de carrera si se modifican las mismas posiciones, Inconsistencia de datos si un hilo redimensiona el vector mientras otro accede.
Se podría paralelizar solo si cada hilo accede a índices diferentes y el tamaño está preasignado.

IV. TAREA 1: ENTRENAMIENTO PREVIO

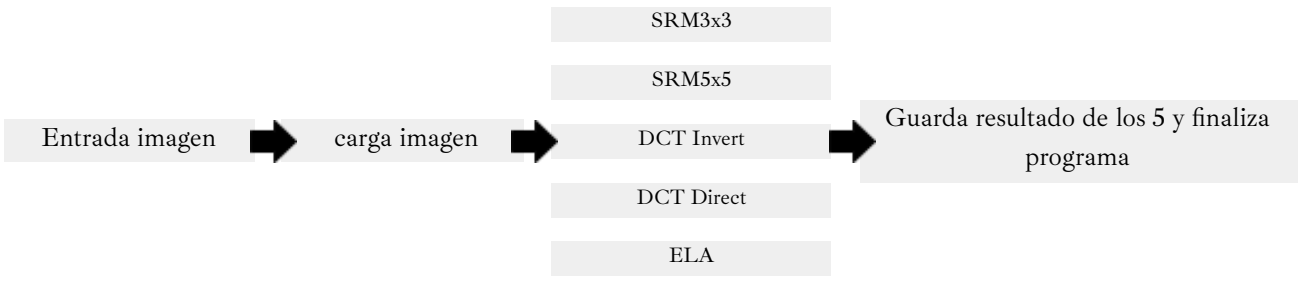
Tarea 1.1: Analiza el código e identifica los distintos procesos:

Sobre la imagen se aplican 5 procesos, filtro imagen dct direct, dct inverse, srm3x3, srm5x5 y ela. El código que se encarga de ello esta en el main y es paralelizable, es el siguiente

```
Image<unsigned char> compute_srm(const Image<unsigned char> &image, int kernel_size) {
    auto begin = std::chrono::steady_clock::now();
    std::cout<<"Computing SRM "<<kernel_size<<"x"<<kernel_size<<"..."<<std::endl;
    Image<float> srm = image.to_grayscale().convert<float>();
    srm = srm.convolution(get_srm_kernel(kernel_size));
    srm = srm.abs().normalized();
    srm = srm * 255;
```

```
    Image<unsigned char> result = srm.convert<unsigned char>();

    auto end = std::chrono::steady_clock::now();
    std::cout<<"SRM elapsed time: "<<std::chrono::duration_cast<std::chrono::milliseconds>(end -
begin).count()<<"ms"<<std::endl;
    return result;
}

Image<unsigned char> compute_dct(const Image<unsigned char> &image, int block_size, bool invert) {
    auto begin = std::chrono::steady_clock::now();
    std::cout<<"Computing";
    if (invert) std::cout<<" inverse";
    else std::cout<<" direct";
    std::cout<<" DCT "<<block_size<<"x"<<block_size<<"..."<<std::endl;
    Image<float> grayscale = image.convert<float>().to_grayscale();
    std::vector<Block<float>> blocks = grayscale.get_blocks(block_size);

    for(int i=0;i<blocks.size();i++){
        float **dctBlock = dct::create_matrix(block_size, block_size);
        dct::direct(dctBlock, blocks[i], 0);
        if (invert) {
          for(int k=0;k<blocks[i].size/2;k++)
            for(int l=0;l<blocks[i].size/2;l++)
              dctBlock[k][l] = 0.0;
          dct::inverse(blocks[i], dctBlock, 0, 0.0, 255.);
        }else dct::assign(dctBlock, blocks[i], 0);
        dct::delete_matrix(dctBlock);
    }

    Image<unsigned char> result = grayscale.convert<unsigned char>();
    auto end = std::chrono::steady_clock::now();
    std::cout<<"DCT elapsed time: "<<std::chrono::duration_cast<std::chrono::milliseconds>(end -
begin).count()<<"ms"<<std::endl;
    return result;
}

Image<unsigned char> compute_ela(const Image<unsigned char> &image, int quality){
    std::cout<<"Computing ELA..."<<std::endl;
    auto begin = std::chrono::steady_clock::now();
    Image<unsigned char> grayscale = image.to_grayscale();
    save_to_file("_temp.jpg", grayscale, quality);
    Image<float> compressed = load_from_file("_temp.jpg").convert<float>();
    compressed = compressed + (grayscale.convert<float>()*(-1));
    compressed = compressed.abs().normalized() * 255;
    Image<unsigned char> result = compressed.convert<unsigned char>();
    auto end = std::chrono::steady_clock::now();
    std::cout<<"ELA elapsed time: "<<std::chrono::duration_cast<std::chrono::milliseconds>(end -
begin).count()<<"ms"<<std::endl;
    return result;
}
```

DIAGRAMA DE DEPENDENCIAS:



Los 5 se pueden paralelizar y ninguno de ellos depende de otro ya que solamente leen sin modificar , es decir, a la hora de procesar los filtros se pueden ejecutar simultáneamente paralelizando el código.

Tarea 1.2: Analiza el código de cada proceso y el tiempo

- **Compute_srm()**

```
Image<float> srm = image.to_grayscale().convert<float>();
srm = srm.convolution(get_srm_kernel(kernel_size));
srm = srm.abs().normalized();
```

to_grayscale():	rápido (recorre cada píxel una vez).
convolution():	muy costoso, recorre cada píxel y su vecino ($O(n \cdot k^2)$).
abs() y normalized():	medios (recorre todos los píxeles).
Conclusión:	el bucle de convolución es el punto clave para paralelizar con OpenMP.

- **Compute_ela()**

```
save_to_file("_temp.jpg", grayscale, quality);
Image<float> compressed = load_from_file("_temp.jpg").convert<float>();
compressed = compressed + (grayscale.convert<float>()*(-1));
```

Escritura y lectura del archivo JPEG:	operación I/O lenta (no paralelizable fácilmente).
Resta de imágenes (compressed + ...):	bucles de píxeles paralelizables.
abs() y normalized():	paralelizables.
Conclusión:	solo las operaciones por píxel se paralelizan; la lectura/escritura de disco no.

- **Compute_dct()**

```
std::vector<Block<float>> blocks = grayscale.get_blocks(block_size);

#pragma omp parallel for
for(int i=0;i<blocks.size();i++){
    float **dctBlock = dct::create_matrix(block_size, block_size);
    dct::direct(dctBlock, blocks[i], 0);
    if (invert) {
```

División en bloques :	ligera.
Bucle de DCT:	muy costoso, cada bloque es independiente.
Conclusión:	este bucle for(int i=0; i<blocks.size(); i++) es ideal para OpenMP (#pragma omp parallel for).

DIAGRAMAS DE FLUJO

SRM

[Inicio]
⬇
Convertir a escala de grises
⬇
Convolución con kernel SRM (3x3 o 5x5)
⬇
Calcular valor absoluto
⬇
Normalizar y escalar
⬇
Guardar imagen SRM
⬇
[Fin] **Parte paralelizable:** convolución, abs, y normalización.

ELA

[Inicio]
⬇
Convertir a escala de grises
⬇
Guardar JPEG con calidad reducida
⬇
Cargar imagen comprimida
⬇
Restar imágenes y normalizar
⬇
Guardar resultado
⬇
[Fin] **Parte paralelizable**: resta y normalización (procesamiento de píxeles).

DCT

[Inicio]
⬇
Convertir a escala de grises
⬇
Dividir imagen en bloques (8x8)
⬇
PARA cada bloque: aplicar DCT directa (si invert: eliminar altas frecuencias) aplicar DCT inversa
⬇
Reconstruir y guardar imagen
⬇
[Fin] **Parte paralelizable**: bucle por bloques (DCT directa/inversa).

TIEMPO DE CADA PROCESO

```
alumno@clLS15I-38:~/Escritorio/source/detect$ ./detect ../vegeta_2048x2048.jpg
Computing SRM 3x3...
SRM elapsed time: 1055ms
Computing SRM 5x5...
SRM elapsed time: 2316ms
Computing ELA...
ELA elapsed time: 647ms
Computing inverse DCT 8x8...
DCT elapsed time: 12189ms
Computing direct DCT 8x8...
DCT elapsed time: 7350ms
alumno@clLS15I-38:~/Escritorio/source/detect$
```

Tarea 1.3: Paralelización

Para paralelizar el código seguiremos la siguiente estrategia de paralelismo, realizando un paralelismo a dos niveles:
1º Paralelismo funcional usando std::async para los 5 procesos independientes en el main (Visto en Tarea 1.1)
2º Paralelismo de datos con OpenMP dentro de las funciones específicas (Visto en Tarea 1.2)

1º PARALELIZACIÓN USANDO STD:ASYNC EN MAIN:

```
int threads = omp_get_max_threads(); // Detectar hilos logicos disponibles
std::cout << "Using " << threads << " threads" << std::endl;
omp_set_num_threads(threads); // Configurar OpenMP para usar todos los hilos logicos disponibles
```

Obtenemos los hilos lógicos de los que dispone el sistema para usar el máximo posible.

```
auto total_begin = std::chrono::steady_clock::now();
//async ejecuta en otro thred y guarda el resultado de la ejecucion
auto future_srm3 = std::async(std::launch::async, [&]() {return compute_srm(image, 3);});

auto future_srm5 = std::async(std::launch::async, [&]() {return compute_srm(image, 5);});

auto future_ela = std::async(std::launch::async, [&]() {return compute_ela(image, 90);});

auto future_dct_inv = std::async(std::launch::async, [&]() {return compute_dct(image, block_size, true);});

auto future_dct_dir = std::async(std::launch::async, [&]() {return compute_dct(image, block_size, false);});
```

Mediante std::async guardamos el resultado de ejecución que se realiza en hilos independientes de forma paralela en variables que posteriormente usaremos para mostrar el resultado con .get()

```
save_to_file("srm_kernel_3x3.png", future_srm3.get());
save_to_file("srm_kernel_5x5.png", future_srm5.get());
save_to_file("ela.png", future_ela.get());
save_to_file("dct_invert.png", future_dct_inv.get());
save_to_file("dct_direct.png", future_dct_dir.get());
```

```
auto total_end = std::chrono::steady_clock::now();
std::cout << "TOTAL TIME: " << std::chrono::duration_cast<std::chrono::milliseconds>(total_end - total_begin).count() << "ms"
```

Creamos un contador de tiempo total para apreciar la mejora ya que en la ejecución secuencial podemos obtenerla simplemente sumando pero en esta paralelización, al ser independientes y ejecutarse simultáneamente la suma no es una medida que podamos usar para obtener valores correctos.

2º Paralelización con OpenMP en image.h

Simplemente añadimos la opción omp parallel for con el parámetro collapse(número a anidar) para en los bucles que ocupan la mayor cantidad de tiempo anidarlos y que se ejecuten de manera simultánea e independiente, es decir, paralela en vez de secuencial ejemplos de este uso de la función se pueden apreciar en la mayoría de bucles for de la práctica los cuales han sido parametrizados siguiendo esta métrica.

```
#pragma omp parallel for collapse(3) //Paraleliza de forma anidada los 3 bucles
for(int j=0;j<height;j++)
{
    for(int i=0;i<width;i++){
        for(int c=0;c<channels;c++){
            new_image.set(j, i, c, this->get(j, i, c) * other.get(j, i, c));
        }
    }
}
```

```
template <class T> Image<T> Image<T>::operator+(const Image<T>& other) const {
    assert(width == other.width && height == other.height && channels == other.channels);
    Image<T> new_image(width, height, channels);

    #pragma omp parallel for collapse(3)
    for(int j=0;j<height;j++)
    {
        for(int i=0;i<width;i++){
            for(int c=0;c<channels;c++){
                new_image.set(j,i,c, this->get(j, i,c)+other.get(j, i, c));
            }
        }
    }

    return new_image;
```

En la función normalized si se usa de forma diferente ya que además se le añaden los parámetro reduction(max:max_value) y reduction(min:min_value) los cuales lo que hacen es que generan en cada bucle para cada hilo una copia privada de tanto max_value y min_value y al terminar todas las ejecuciones reduce las variables a el max_value global y el min_value global quedando así el mayor y el menor de todos.

```
template <class T> Image<float> Image<T>::normalized() const {
    Image<float> new_image(width, height, channels);
    float max_value = -999999999;
    float min_value = 999999999;

    #pragma omp parallel for collapse(3) reduction(max:max_value) reduction(min:min_value) //
    for(int j=0;j<height;j++)
    {
        for(int i=0;i<width;i++){
            for(int c=0;c<channels;c++){
                if (this->get(j,i,c) > max_value) max_value = this->get(j,i,c);
                if (this->get(j,i,c) < min_value) min_value = this->get(j,i,c);
            }
        }
    }
```

Responded a las siguientes preguntas:

- **¿Se obtiene un mejor rendimiento paralelizando todos las partes posibles?**

Si, se obtiene un gran rendimiento, comparando se obtiene aproximadamente un speedup de

- En el ordenador de clase de 6 núcleos y 12 hilos lógicos: 23,557/3771 = 6,246
- En portátil de 10 núcleos y 16 hilos lógicos: 32592/4349 = 7,494

Lo más seguro es que en el portatil tardase más ya que se ejecutó desde un wsl y con varios procesos más activos de fondo a diferencia del ordenador de clase que usa ubuntu nativo y no tenía nada más abierto.

- **¿Se degrada el rendimiento al paralelizar ciertas partes?**

Si ya que en std:async se puede ver como los primeros procesos tardan más que en el secuencial.

- **Si es así, ¿a qué creéis que se debe esa degradación?**

La degradación se debe a dos factores principales:

- Overhead: Crear hilos tiene su coste en cuanto a tiempo, tanto la creación como gestión y si bien es cierto que mejora el resultado final en algunas funciones repercute en negativo.
- Contención de Recursos: Hay más hilos lógico ejecutándose que núcleos físicos lo que provoca que se "compita" por el tiempo de CPU, la caché y el bus de la memoria RAM.

- **¿Cúal es la mejor estrategia de paralelización?**

Basándose en lo anteriormente comentado considero la mejor estrategia de paralelización la paralelización en varios niveles utilizando diferentes hilos conforme el sistema tenga de procesadores lógicos, utilizándolos en funciones con alto coste computacional, ya que si este fuera bajo provocaría mayor overhead que ganancia.
Por lo que se concluye que una paralelización híbrida es lo ideal.

Tarea 1.4: Resultados obtenidos

Tiempo sin paralelizar en ordenador de clase: (23,557 segundos)

```
alumno@clLS15I-38:~/Escritorio/source/detect$ ./detect ../vegeta_2048x2048.jpg
Computing SRM 3x3...
SRM elapsed time: 1055ms
Computing SRM 5x5...
SRM elapsed time: 2316ms
Computing ELA...
ELA elapsed time: 647ms
Computing inverse DCT 8x8...
DCT elapsed time: 12189ms
Computing direct DCT 8x8...
DCT elapsed time: 7350ms
alumno@clLS15I-38:~/Escritorio/source/detect$
```

Tiempo paralelizado en ordenador de clase:

```
alumno@clLS15I-38:~/Escritorio/ic-2025-gr6-p3-grupo4-master/source/detect$ ./detect ../../vegeta_2048x2048.jpg
Using 12 threads
Computing SRM 3x3...
Computing SRM 5x5...
Computing inverse DCT 8x8...
Computing direct DCT 8x8...
Computing ELA...
SRM elapsed time: 820ms
ELA elapsed time: 831ms
SRM elapsed time: 1723ms
DCT elapsed time: 3169ms
DCT elapsed time: 3612ms
TOTAL TIME: 3771ms
```

Especificaciones ordenador de clase:

```
alumno@clLS15I-38:~/Escritorio/source/detect$ lscpu
Arquitectura:                         x86_64
  modo(s) de operación de las CPUs:   32-bit, 64-bit
  Address sizes:                      39 bits physical, 48 bits virtual
  Orden de los bytes:                 Little Endian
CPU(s):                               12
  Lista de la(s) CPU(s) en línea:     0-11
ID de fabricante:                     GenuineIntel
  Nombre del modelo:                  12th Gen Intel(R) Core(TM) i5-12400
    Familia de CPU:                   6
    Modelo:                           151
    Hilo(s) de procesamiento por núcleo: 2
    Núcleo(s) por «socket»:           6
    «Socket(s)»                       1
    Revisión:                         5
    CPU(s) scaling MHz:               18%
    CPU MHz máx.:                     4400,0000
    CPU MHz mín.:                     800,0000
    BogoMIPS:                         4992,00
```

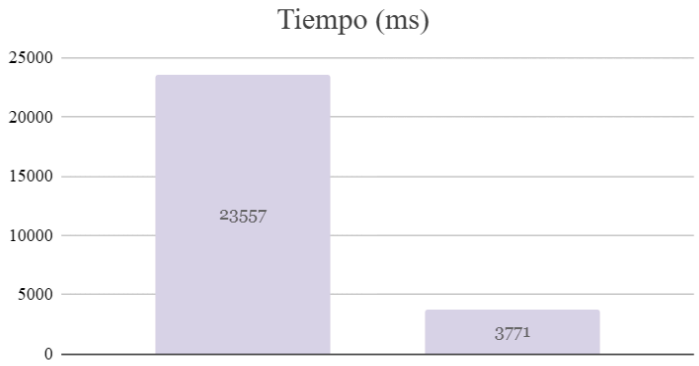
```
alumno@clLS15I-38:~/Escritorio/source/detect$ free -h
               total        usado        libre  compartido   búf/caché  disponible
Mem:            15Gi       3,9Gi       8,8Gi       349Mi       3,4Gi        11Gi
Inter:            0B           0B           0B
```

Salidas:

Tarea 1.5: Gráfica de ganancia:

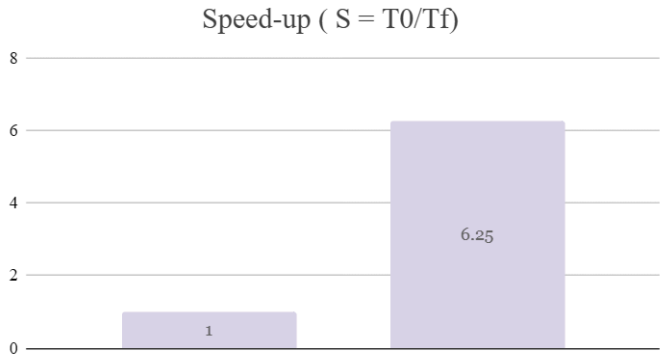
Gráfica 1 : Tiempo

Hilos	Tiempo (ms)
1	23557
12	3771



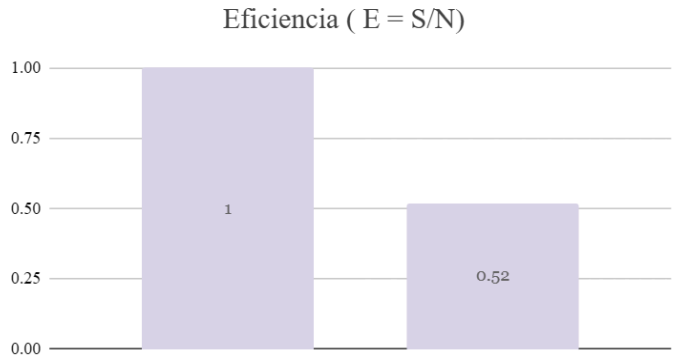
Gráfica 2 : Speed Up

Hilos	Speed-up
1	1
12	6,25



Gráfica 3: Eficiencia

Hilos	Eficiencia
1	1
12	0,52



VI. CONCLUSIÓN

En esta práctica hemos aprendido a aplicar el paralelismo a nivel de hilos usando OpenMP y std::async en C++. A partir de un programa secuencial que analizaba imágenes manipuladas, conseguimos crear una versión paralela capaz de aprovechar varios núcleos del procesador para reducir el tiempo de ejecución.

Durante las pruebas pudimos comprobar que el rendimiento mejora notablemente al aumentar el número de hilos, aunque no de forma lineal. En nuestro caso, el programa pasó de tardar unos 23,5 segundos en la versión secuencial a solo 3,7 segundos usando 12 hilos, lo que supone un *speed-up* aproximado de 6,25x y una eficiencia del 52 %. Esto demuestra que la paralelización fue efectiva, pero también que existen ciertos límites debidos a la sobrecarga de gestión de hilos y a las partes del código que no se pueden ejecutar en paralelo.

En resumen, con esta práctica hemos entendido mejor cómo funciona la programación paralela, cómo medir su rendimiento y qué factores influyen en la eficiencia. Además, pudimos comprobar de forma práctica que dividir el trabajo entre varios hilos puede mejorar mucho el rendimiento de un programa, siempre que se haga un buen análisis de las dependencias y de las tareas que realmente se pueden ejecutar al mismo tiempo.