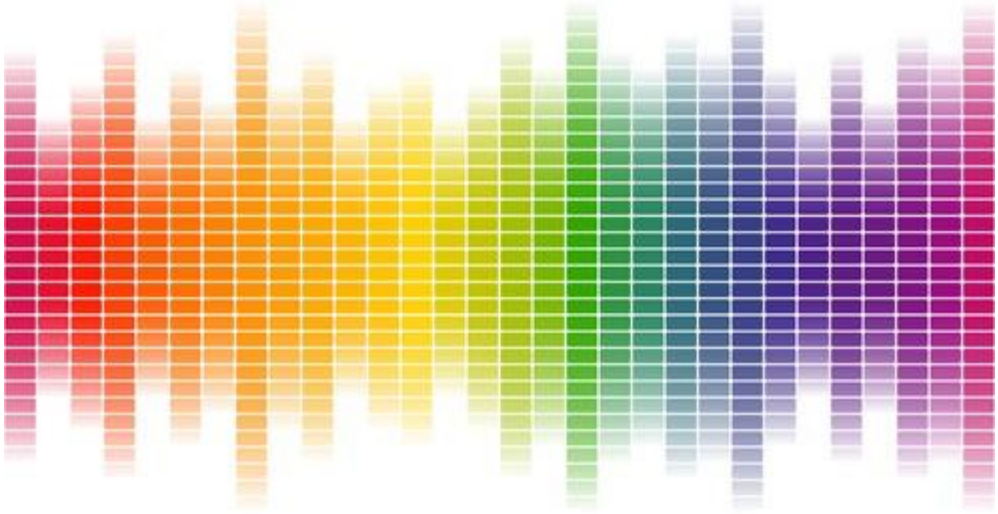
Universidad Nacional Autónoma de México

Facultad de Ingeniería

Estructuras de Datos y Algoritmos II

Proyecto Final

Ecualización de Histogramas en Imágenes



Profesor: Jesús Cruz Navarro

Grupo: 01

Integrantes: Velasco García Santiago, Zarza Zurita Axel Zahir

Números de lista: 42, 43

Semestre: 2024-1

Fecha de Entrega: 10/12/2023

- **Introducción:**

Durante la realización de este proyecto se pusieron a prueba los conocimientos generales en el lenguaje de programación C, en programación estructurada y sobre todo en los últimos dos temas ‘Archivos’ y ‘Programación Paralela’.

Se utilizaron librerías como omp, time, string y principalmente stb para la realización de un ecualizador de imágenes que reciba una imagen como argumento en la línea de comandos y genere a partir de esta, un histograma y una versión ecualizada del archivo, de forma serial y de forma paralela.

El código está comentado casi en su totalidad para seccionar los algoritmos y darle orden al flujo de programa, además de explicar lo que hacen y devuelven algunas funciones de librerías nuevas.

- **Desarrollo:**

Para la realización de este proyecto comenzamos por su desarrollo secuencial para entender el algoritmo, identificar procesos paralelizables y variables compartidas.

Con la finalidad de hacer más legible cada parte del método, el procedimiento secuencial está dividido en funciones que representan los pasos a seguir dados en la asignación del proyecto:

```
// Ecualización secuencial
startSeq = omp_get_wtime();
generateGrayscaleHistogram(image_data, width, height, histogram); // Obtener histograma de la imagen
generateGrayscaleCDF(histogram, cdf); // Generar el cdf (Cumulative Distributive Function)
cdfmin = findNonZeroMin(cdf); // Encontrar cfdmin
generateGrayscaleEqCDF(cdf, eqCdf, cdfmin, size); // Generar el nuevo cdf, que será el arreglo eqCdf
eqImage = generateGrayscaleEqImage(image_data, width, height, eqCdf, size); // Generar el arreglo de pixeles para la nueva imagen
generateGrayscaleHistogram(eqImage, width, height, eqHistogram); // Generar el nuevo histograma
endSeq = omp_get_wtime();
startIm = omp_get_wtime();
createGrayscaleImage(eqImage, width, height, imgPath, false); // Generar la nueva imagen
endIm = omp_get_wtime();
timeIm = endIm - startIm;
startCSV = omp_get_wtime();
generateCSV(histogram, eqHistogram, imgPath, false); // Genere el archivo csv
endCSV = omp_get_wtime();
timeCSV = endCSV - startCSV;
```

Después de descargar e implementar la librería STB, y de declarar todas las librerías y funciones que utilizamos en el proyecto, se desarrolló la función main, donde se recibe la imagen mediante los argumentos argc y argv, siendo el primero el número de argumento que contiene la ruta o nombre de la imagen y el segundo, esto mismo. El programa verifica que tengamos exactamente 2 argumentos que serán el nombre del ejecutable y la imagen, para posteriormente llamar a la función encargada de iniciar la ecualización, donde se trata de cargar la imagen a un arreglo de unsigned char mediante la función stbi_load que devuelve un puntero NULL si no puede cargar la imagen (ya sea porque no existe, se ingresó mal o cualquier otro error).

```c
int main(int argc, char *argv[]) {
    if (argc != 2) { /*El parámetro argument count debe ser exactamente igual a 2 (el nombre del
        ejecutable y la ruta o nombre de la imagen)*/
        printf("Error: Se espera un argumento con el nombre de la imagen.\n");
        return 1;
    }

    equalizeImage(argv[1]);

    return 0;

}

int equalizeImage(unsigned char *imgPath) {
    double startLoad, endLoad, timeLoad;
    int width, height, channels;
    startLoad = omp_get_wtime();
    unsigned char *image_data = stbi_load(imgPath, &width, &height, &channels, 0);
    endLoad = omp_get_wtime();
    timeLoad = endLoad - startLoad;

    if (image_data == NULL) { /*stbi_load devuelve un puntero NULL si no puede cargar la imagen
        correctamente desde el archivo especificado*/
        printf("Error: No se pudo cargar la imagen '%s'.\n", imgPath);
        return 1;
    }

    printf("Imagen cargada con éxito.\n");
```

Si la imagen se carga con éxito se imprimen los datos referentes a su resolución, se declaran las variables necesarias para calcular las métricas y se evalúa su número de canales.

Si la imagen es de un solo canal comienza con el algoritmo secuencial, cuya primera parte es llamar a la función constructora de histogramas para imágenes de un canal (en escala de grises):

```c
void generateGrayscaleHistogram(unsigned char *image, int width, int height, int *histogram) {
    // Inicializar el histograma a 0
    for (int i = 0; i < NUM_BINS; ++i) {
        histogram[i] = 0;
    }

    // Calcular el histograma directamente para la imagen de escala de grises
    for (int i = 0; i < width * height; ++i) {
        histogram[image[i]]++;
    }
}
```

Esta primera función inicializa el histograma a 0 para evitar inconsistencias ya que después se realiza un conteo igual al que se hace en Counting Sort.

Una vez teniendo el histograma se llama a la función generadora del CDF (función acumulativa) y a la función que encontrará al valor mínimo no nulo de este nuevo arreglo:

```
void generateGrayscaleCDF(const int *histogram, int *cdf) {
    cdf[0] = histogram[0]; // El primer valor del CDF es el primer valor del histograma

    // Calcular el CDF
    for (int i = 1; i < NUM_BINS; ++i) {
        cdf[i] = histogram[i] + cdf[i - 1];
    }
}

int findNonZeroMin(const int *cdf) {
    int minNonZero = cdf[0]; // Inicializa el mínimo con el primer valor del CDF

    // Busca el primer valor diferente de cero en el CDF
    for (int i = 1; i < NUM_BINS; ++i) {
        if (cdf[i] != 0) {
            minNonZero = cdf[i];
            break; // Se encontró el primer valor no cero, se puede detener el bucle
        }
    }

    return minNonZero;
}
```

Los datos que acabamos de generar los usamos ahora para generar el eqCDF que utiliza la fórmula:

$$h(v) = \text{round}\left(\frac{cdf(v) - cdf_{min}}{(M \times N) - cdf_{min}} \times (L - 2)\right) + 1$$

Simplemente creamos una función con los parámetros necesarios y codificamos la fórmula:

```
void generateGrayscaleEqCDF(const int *cdf, int *eqCdf, int cdfmin, int size) {
    for (int i = 0; i < NUM_BINS; ++i) {
        double eqcdfv = round(((double)(cdf[i] - cdfmin) / (size - cdfmin)) * ((double) NUM_BINS - 2)) + 1;
        eqCdf[i] = (int) eqcdfv;
    }
}
```

La siguiente función reserva memoria para el nuevo arreglo de pixeles, verifica que no haya habido problemas en la asignación y comienza a copiar la nueva distribución de pixeles en el arreglo, para finalmente devolverlo.

```c
unsigned char* generateGrayscaleEqImage(unsigned char *srcImage, int width, int height, int *eqCdf, int size) {

    // Verificar si eqCdf es un puntero válido y srcImage no es NULL
    if (eqCdf == NULL || srcImage == NULL) {
        return NULL;
    }

    // Crear el arreglo para la nueva imagen eqImage
    unsigned char *eqImage = malloc(size);

    // Verificar si la asignación de memoria fue exitosa
    if (eqImage == NULL) {
        return NULL;
    }

    // Generar la nueva imagen eqImage utilizando el CDF ecualizado eqCdf
    for (int i = 0; i < size; ++i) {
        eqImage[i] = (unsigned char)eqCdf[srcImage[i]];
    }

    return eqImage;
}
```

Desarrollamos una función que escriba la nueva imagen y otra que genere los csv donde vendrá la información de los histogramas. Estas funciones realizan un procedimiento **EXTRA** para nombrar los archivos generados de forma consistente conforme al nombre de la imagen cargada. Este proceso consiste en encontrar el punto que va antes de la extensión de la imagen. Mediante la función strrchr creamos un apuntador a este carácter y lo sustituimos por el carácter nulo \0 para truncar la cadena ahí y tener únicamente el nombre de la imagen.

Es decir, si el archivo se llama ‘imagen.jpg’, la función strrchr encuentra un apuntador hacia el punto antes de la extensión y coloca ahí el carácter nulo quedando como ‘imagen\0’, para posteriormente concatenar otra cadena que hace referencia al archivo que se esta generando (esto también se auxilia de la bandera ‘parallel’ que es un parámetro booleano que nos ayudará a saber cual será el nombre correcto del archivo generado, ‘_secuencial’ o ‘_paralelo’.

Posteriormente, en la función que genera la imagen, se escribe el archivo con la función stbi_write_jpg y posteriormente se libera la memoria utilizada:

```c
void createGrayscaleImage(unsigned char *eqImage, int width, int height, const char *original_filename, bool parallel) {
    char *output_filename = malloc(strlen(original_filename) + strlen("_eq_secuencial.jpg") + 1);
    strcpy(output_filename, original_filename);

    // Encontrar la extensión del archivo (asumiendo que el nombre de archivo original tiene la extensión)
    char *dot = strrchr(output_filename, '.');
    if (NULL != dot) {
        // Si se encuentra una extensión, se agrega el sufijo antes de la extensión
        *dot = '\0';
        if(parallel) {
            strcat(output_filename, "_eq_paralelo.jpg");
        } else {
            strcat(output_filename, "_eq_secuencial.jpg");
        }
    }

    // Escribir la imagen utilizando la librería STB
    stbi_write_jpg(output_filename, width, height, 1, eqImage, 100);

    // Liberar la memoria del arreglo eqImage
    stbi_image_free(eqImage);
    free(output_filename);
}
```

Y para la función que crea los CSV se copia en la primera columna los valores, en la segunda, sus correspondientes en el histograma de la imagen original, y en la tercera, sus correspondientes en el histograma de la imagen ecualizada, para finalmente cerrar el archivo:

```c
void generateCSV(int *histogram_original, int *histogram_equalized, const char *original_filename, bool parallel) {

    char *output_filename = malloc(strlen(original_filename) + strlen("_histo_secuencial.csv") + 1);
    strcpy(output_filename, original_filename);
    // Encontrar la extensión del archivo (asumiendo que el nombre de archivo original tiene la extensión)
    char *dot = strrchr(output_filename, '.');
    if (NULL != dot) {
        // Si se encuentra una extensión, se agrega el sufijo antes de la extensión
        *dot = '\0';
        if(parallel) {
            strcat(output_filename, "_histo_paralelo.csv");
        } else {
            strcat(output_filename, "_histo_secuencial.csv");
        }
    }
    FILE *csv_file = fopen(output_filename, "w"); // Abre el archivo en modo escritura

    if (csv_file == NULL) {
        printf("Error al abrir el archivo.");
        return;
    }

    fprintf(csv_file, "valor, histo, eqHisto\n");

    for (int i = 0; i < 256; ++i) {
        fprintf(csv_file, "%d, %d, %d\n", i, histogram_original[i], histogram_equalized[i]);
    }

    fclose(csv_file); // Cierra el archivo
}
```

Asi es como se generan los archivos de forma secuencial. Para generarlos de forma paralela comenzamos identificando los procesos que son paralelizables, que en este caso son la inicialización y generación de los histogramas, la generación del eqCDF y la generación del arreglo de la nueva imagen. La generación de CDF y encontrar el cdfmin son procesos dependientes de otros así que esos los trabajamos

atendiendo a la recomendación de usar la clausula single. Para reducir aún más los tiempos hicimos uso de la clausula reduction y diseñamos el código de forma tal que creáramos la menor cantidad de regiones paralelas posibles. El resultado de este análisis finalmente se codificó en una función única que va desde la generación del primer histograma hasta la generación del histograma de la imagen ecualizada:

```
unsigned char* equalizeImageParallel(unsigned char *image_data, int width, int height, int size,
                                     int *histogram, int *cdf, int *eqCdf, int *eqHistogram, int cdfmin) {

    // Obtener el histograma en paralelo usando reduction
    #pragma omp parallel
    {
        #pragma omp for
        for(int i=0; i < NUM_BINS; i++)
            histogram[i] = 0;

        #pragma omp barrier

        #pragma omp for reduction(+:histogram[:NUM_BINS])
        for(int i=0; i < size; i++)
            histogram[image_data[i]]++;
    }

    // Calcular el CDF y el cdfmin en secuencia con un solo hilo
    #pragma omp single
    {
        cdf[0] = histogram[0];
        for (int i = 1; i < NUM_BINS; ++i) {
            cdf[i] = histogram[i] + cdf[i - 1];
        }

        cdfmin = cdf[0];
        for (int i = 1; i < NUM_BINS; ++i) {
            if (cdf[i] != 0) {
                cdfmin = cdf[i];
                break;
            }
```

```
        }
    }

    // Generar el CDF ecualizado en paralelo
    #pragma omp for nowait
    for (int i = 0; i < NUM_BINS; ++i) {
        double eqcdfv = round(((double)(cdf[i] - cdfmin) / (size - cdfmin)) * ((double) NUM_BINS - 2)) + 1;
        eqCdf[i] = (int) eqcdfv;
    }

    // Generar la nueva imagen en paralelo
    #pragma omp for nowait
    for (int i = 0; i < size; ++i) {
        image_data[i] = (unsigned char)eqCdf[image_data[i]];
    }

    // Generar el nuevo histograma en paralelo usando reduction
    #pragma omp parallel
    {
        #pragma omp for
        for(int i=0; i < NUM_BINS; i++)
            eqHistogram[i] = 0;

        #pragma omp barrier

        #pragma omp for reduction(+:eqHistogram[:NUM_BINS])
        for(int i = 0; i < size; i++)
            eqHistogram[image_data[i]]++;
    }
    return image_data;
}
```

- **Capturas de los datos de entrada de prueba y salidas del programa:**

Compilación del proyecto:

```
PS C:\Users\Santiago\OneDrive\Documentos\Dev\EDA II\Proyecto> gcc velasco_garcia_santiago_proyecto_codig
o.c -o Equalizer.exe -fopenmp
```

Prueba con imágenes proporcionadas por el profesor:

imaGray_1

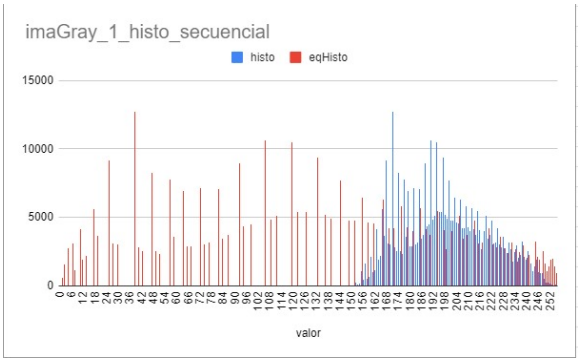
Entradas y salidas en consola:

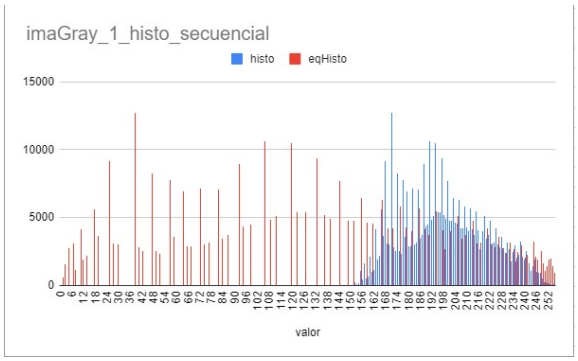
```
PS C:\Users\Santiago\OneDrive\Documentos\Dev\EDA II\Proyecto> ./Equalizer.exe imagenesTest/imaGray_1.jpg

Imagen cargada con ├®xito.

Resoluci├│n de la imagen:
Ancho: 750
Alto: 500
Tama├▒o de la imagen: 375000 bytes
Canales: 1

Sobre la ecualizaci├│n:
N├║mero de procesadores: 8
Numero de hilos usados: 8

M├®tricas:
Tiempo de ejecuci├│n en serie: 0.000000 [s]
Tiempo de ejecuci├│n en paralelo: 0.019000 [s]
Speedup: 0.000000
Eficiencia: 0.000000
Tiempo de Overhead: 8.000000 [s]

Otros tiempos:
Tiempo de carga de imagen: 0.015000
Tiempo de generaci├│n de imagen: 0.040000
Tiempo de generaci├│n CSV: 0.000000
PS C:\Users\Santiago\OneDrive\Documentos\Dev\EDA II\Proyecto>
```

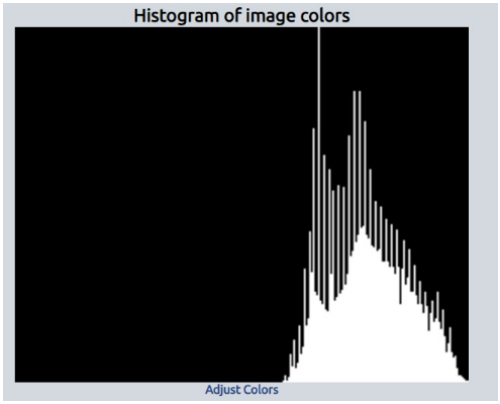
Imagen original:

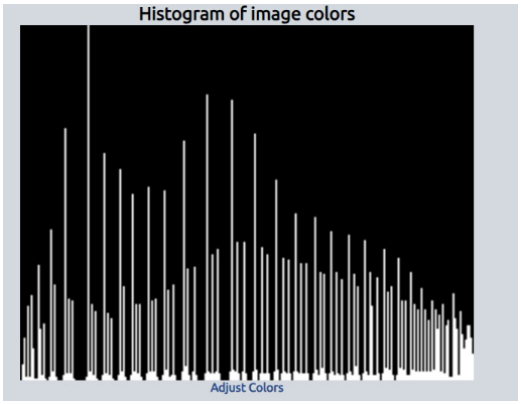
Imágenes ecualizadas:

Gráficas:









Análisis: El tiempo secuencial se reporta como 0 [s] **debido a que las dimensiones de la imagen proporcionada son muy pequeñas**, lo que hace que el tiempo en paralelo sea mayor en este caso particular, siendo así que el speedup que es una medida que compara el rendimiento de un programa o algoritmo ejecutado en paralelo con respecto a su versión secuencial sea 0, su eficiencia que es la medida en que un sistema o algoritmo utiliza sus recursos de manera efectiva para realizar un trabajo en paralelo sea 0 (ya que en su cálculo aparece el speedup como factor) lo que hace parecer que el tiempo de overhead fue de 8 segundos.

imaGray_3

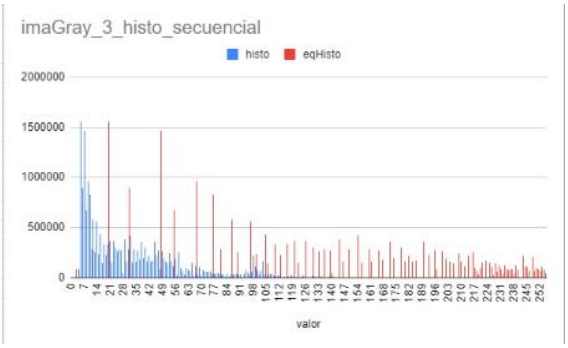
Entradas y salidas en consola:

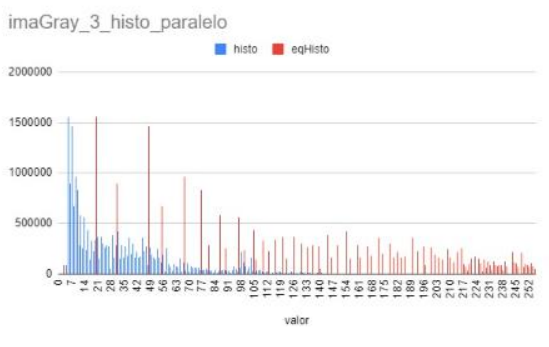
```
PS C:\Users\Santiago\OneDrive\Documentos\Dev\EDA II\Proyecto> ./Equalizer.exe imagenesTest/imaGray_3.jpg

Imagen cargada con ├®xito.

Resoluci├│n de la imagen:
Ancho: 5723
Alto: 3815
Tama├▒o de la imagen: 21833245 bytes
Canales: 1

Sobre la ecualizaci├│n:
N├║mero de procesadores: 8
Numero de hilos usados: 8

M├®tricas:
Tiempo de ejecuci├│n en serie: 0.137000 [s]
Tiempo de ejecuci├│n en paralelo: 0.080000 [s]
Speedup: 1.712503
Eficiencia: 0.214063
Tiempo de Overhead: 6.287497 [s]

Otros tiempos:
Tiempo de carga de imagen: 0.303000
Tiempo de generaci├│n de imagen: 1.957000
Tiempo de generaci├│n CSV: 0.000000
PS C:\Users\Santiago\OneDrive\Documentos\Dev\EDA II\Proyecto>
```

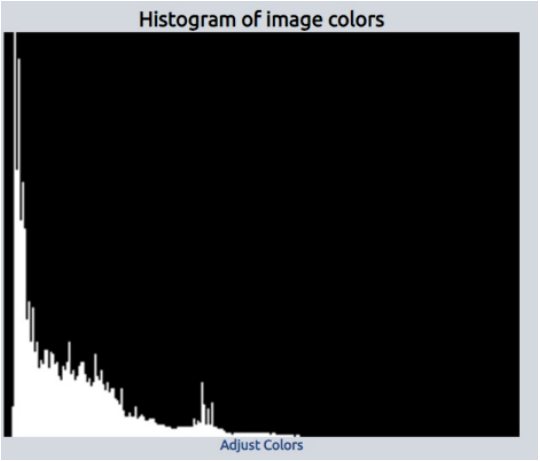
Imagen original:

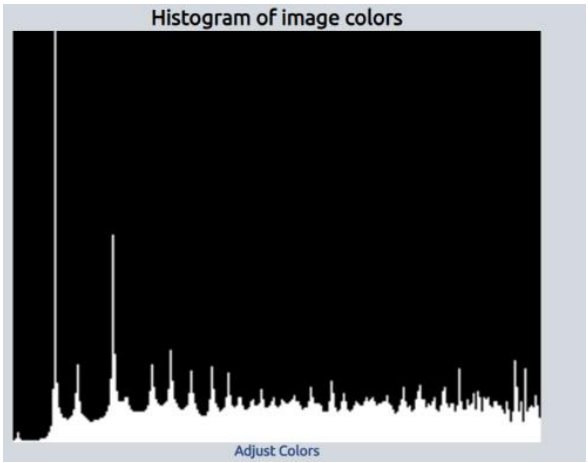
Imágenes ecualizadas:

Gráficas:









Análisis: En este caso al ser una imagen de dimensiones mas grandes apreciamos con más claridad la superioridad de la paralelización, obteniendo un tiempo casi el doble de tardado para la ejecución del algoritmo secuencial, lo que nos da un speedup de casi 2, que dividido en 8 nos da una eficiencia aproximadamente de 0.2 y un tiempo de overhead de 6 segundos, que aún es grande pero ya no es 8 como en el ejemplo anterior.

Pruebas con imágenes propias:

ave

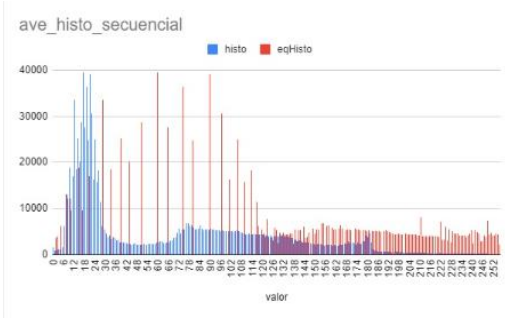
Entradas y salidas en consola:

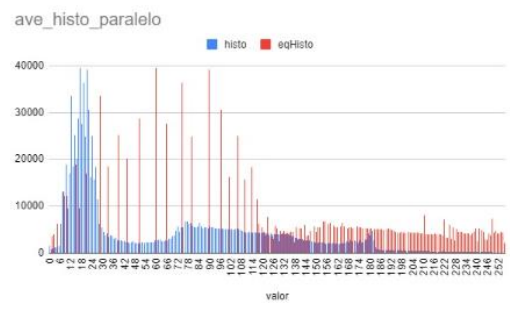
```
PS C:\Users\Santiago\OneDrive\Documentos\Dev\EDA II\Proyecto> ./Equalizer.exe imagenesTest/ave.jpg
Imagen cargada con ├®xito.

Resoluci├│n de la imagen:
Ancho: 1280
Alto: 853
Tama├▒o de la imagen: 1091840 bytes
Canales: 1

Sobre la ecualizaci├│n:
N├║mero de procesadores: 8
Numero de hilos usados: 8

M├®tricas:
Tiempo de ejecuci├│n en serie: 0.010000 [s]
Tiempo de ejecuci├│n en paralelo: 0.006000 [s]
Speedup: 1.666653
Eficiencia: 0.208332
Tiempo de Overhead: 6.333347 [s]

Otros tiempos:
Tiempo de carga de imagen: 0.000000
Tiempo de generaci├│n de imagen: 0.086000
Tiempo de generaci├│n CSV: 0.000000
PS C:\Users\Santiago\OneDrive\Documentos\Dev\EDA II\Proyecto>
```

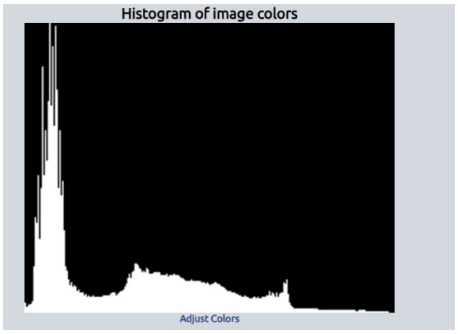
Imagen original:

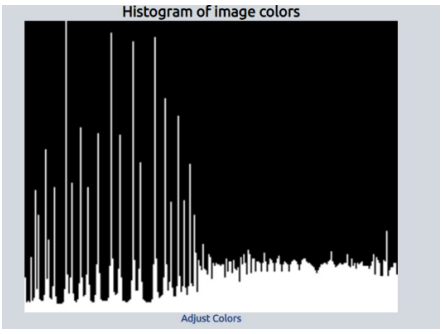
Imágenes ecualizadas:

Gráficas:









Análisis: Tenemos nuevamente una imagen de dimensiones “pequeñas” aunque de tamaño mas promedio. En este caso observamos resultados métricos similares a la imagen anterior a pesar de la diferencia de tamaños. Vemos que ecualizando ‘ave’ obtenemos una diferencia de tiempos notoria y un speedup, eficiencia y overhead similares a la imagen anterior.

playa1

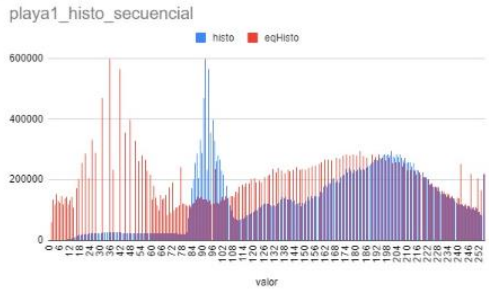
Entradas y salidas en consola:

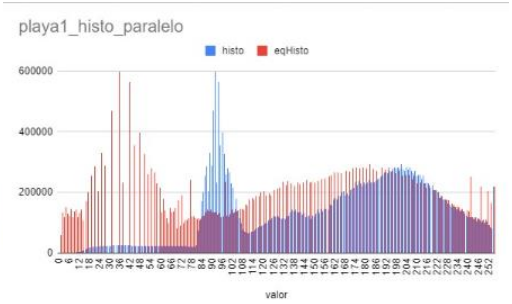
```
PS C:\Users\Santiago\OneDrive\Documentos\Dev\EDA II\Proyecto> ./Equalizer.exe imagenesTest/playa1.jpg
Imagen cargada con ├®xito.

Resoluci├│n de la imagen:
Ancho: 7108
Alto: 4744
Tama├▒o de la imagen: 33720352 bytes
Canales: 1

Sobre la ecualizaci├│n:
N├║mero de procesadores: 8
Numero de hilos usados: 8

M├®tricas:
Tiempo de ejecuci├│n en serie: 0.218000 [s]
Tiempo de ejecuci├│n en paralelo: 0.107000 [s]
Speedup: 2.037383
Eficiencia: 0.254673
Tiempo de Overhead: 5.962617 [s]

Otros tiempos:
Tiempo de carga de imagen: 0.455000
Tiempo de generaci├│n de imagen: 2.953000
Tiempo de generaci├│n CSV: 0.000000
PS C:\Users\Santiago\OneDrive\Documentos\Dev\EDA II\Proyecto>
```

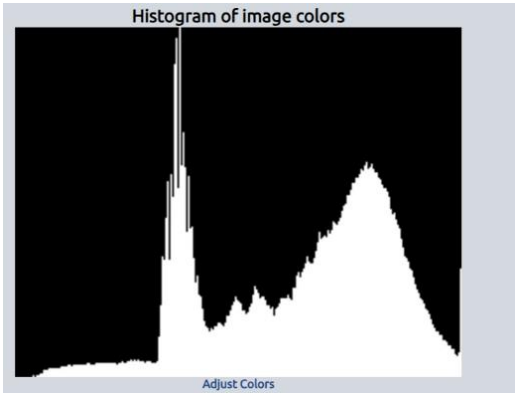
Imagen original:

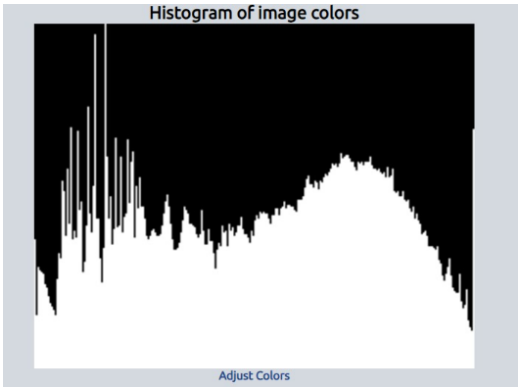
Imágenes ecualizadas:

Gráficas:









Análisis: Esta es, por mucho, la imagen mas grande de todas las pruebas. Obteniendo la mayor diferencia entre el tiempo secuencial y el tiempo paralelo siendo de mas del doble, obteniendo un speedup de un poco más de 2, dividido en 8 para una eficiencia de un poco mas de 0.25 y el único tiempo de overhead menor a 6. El tiempo de generación de la imagen fue el mas grande por lo evidente, a pesar de que las imágenes ecualizadas no son muy diferentes a la imagen original.

- **Conclusiones relevantes:**

Como podemos observar, hay una relación entre el tamaño de la imagen y las métricas obtenidas. Podemos observar que aparentemente, mientras las dimensiones de la imagen sean mas grandes, obtendremos mejores resultados métricos de la programación paralela en comparación con la programación secuencial, aunque esto no es del todo cierto.

En general, se espera que el rendimiento, medido por el speedup y la eficiencia, mejore para imágenes más grandes en ciertas condiciones. Sin embargo, esto puede depender de varios factores. Si la cantidad de datos es lo suficientemente grande para aprovechar al máximo los recursos paralelos disponibles, es probable que se obtenga un mejor rendimiento.

Aunque el margen de mejora no es infinito, ya que, dentro del algoritmo paralelo, hay regiones secuenciales que implican una cota para el speedup, es decir, existen límites en cuanto a la escalabilidad. A medida que el tamaño de la imagen sigue aumentando, es posible que el rendimiento no siga mejorando linealmente debido a posibles cuellos de botella, como la gestión de la memoria, la comunicación entre hilos o la sobrecarga de coordinación en entornos paralelos.

Como conclusión del análisis de los datos en conjunto con la teoría vista en clase, para ciertas condiciones y dependiendo de la implementación y la arquitectura del sistema, es posible que se observe un mejor speedup y eficiencia al trabajar con imágenes más grandes en algoritmos paralelos como el ecualizador de imágenes, siempre y cuando se puedan manejar eficientemente los recursos y la granularidad del trabajo.

En cuanto a los detalles del proyecto, este únicamente puede trabajar con imágenes de un canal, sin embargo, el diseño de código tiene consistencia en el nombre de las funciones utilizadas y un orden lo suficientemente legible como para que las funcionalidades correspondientes a ecualizar imágenes de 3 canales puedan ser implementados en el futuro. Aún así el proyecto cubre satisfactoriamente el objetivo de trabajar con hilos y archivos, lee los parámetros del main desde la línea de comandos e implementa una función extra para mantener consistencia en los nombres entre las imágenes recibidas y los archivos generados.