Pontificia Universidad Javeriana Cali

Facultad de Ingeniería y Ciencias

Parcial Práctico 1 – Programación Paralela

Trabajo realizado por:

Juan Diego Rojas Valdés

Docente:

Jefferson Peña Torres

Cali, 31 de agosto de 2025

1. Introducción:

La programación paralela es beneficiosa en gran cantidad de aplicaciones, debido a que permite una mayor eficiencia a la hora de correr ciertos programas, pues en lugar de ejecutar cada instrucción de manera secuencial, utiliza la arquitectura de una computadora al máximo y permite que las tareas sean divididas en los diferentes procesadores.

Uno de los usos que puede tener este enfoque es en el procesamiento y aplicación de filtros en imágenes. Es justamente de eso de lo que se trata esta práctica.

Unos de los muchos formatos de imágenes existentes son los PGM (Portable Graymap) y los PPM (Portable Pixmap). Estos consisten en una matriz de valores que equivalen a los colores que se pueden visualizar en la imagen.

En los PGM, que son en escala de grises, cada valor de la matriz corresponde con un valor entre 0 y un máximo establecido (usualmente 255), el cual, entre más cercano a 0, más oscuro será el tono de gris, y entre más cercano al máximo, más claro será el tono [2].

En los PPM sí hay presencia de color, por lo que el formato es un poco más complicado. Nuevamente se trata de una matriz, donde cada celda contiene tres valores, uno para el tono de rojo (R), uno para el tono de verde (G) y uno para el tono de azul (B) que contiene la imagen en ese pixel en particular. De esta manera, si en una celda está la combinación (0,0,0), el pixel se verá negro y si está (255, 255, 255) se verá blanco, pero si está (255,0,0), (0,255,0) o (0,0,255), se verá rojo, verde y azul, respectivamente, y los demás distintos colores pueden ser generados mediante combinaciones de estos tres [3].

En la práctica se solicitó desarrollar un programa que procese estos formatos de imágenes y aplique tres filtros sobre estos (Blur o borroso, Laplace y Sharpening o nítido) utilizando operaciones matriciales, primero de manera secuencial y luego utilizando los tres frameworks de programación paralela aprendidos en el curso: Pthreads, OpenMP y MPI, con el objetivo de llegar a una respuesta a una pregunta de interés que se planteó en un principio.

2. Pregunta de Interés:

¿Cómo influye la programación paralela, utilizando OpenMP o Pthreads y MPI (En un entorno simulado con docker), en el tiempo de ejecución del filtrado de imágenes PPM y PGM en comparación con una implementación secuencial?

Es la pregunta que se buscará resolver con los descubrimientos encontrados durante la práctica, y que serán expuestos en el presente informe. Sin mayor dilación, se procederá a explicar el procedimiento de resolución de la actividad.

3. Desarrollo de la práctica:

3.1. Secuencial:

El primer paso para desarrollar la práctica fue clonar el repositorio de github proporcionado por el profesor [1], para obtener las bases de dos de los códigos a desarrollar y el arreglo de imágenes en los formatos requeridos para testear y verificar la efectividad de los programas realizados.

El repositorio contenía dos archivos .cpp incompletos que debían complementarse para realizar, inicialmente, un programa que aplicase los filtros de manera secuencial.

- *processor.cpp*: Este programa tenía como función leer los archivos .ppm y .pgm y guardarlos como objetos de la clase imagen.
- *filterer.cpp*: Este programa tenía como función recibir un archivo .ppm o .pgm como entrada, aplicarle el filtro especificado en la ejecución y guardar la imagen con el filtro aplicado en otro archivo de salida, del mismo formato que el archivo de entrada.

Como se menciona en la descripción de *processor.cpp*, se va a utilizar el paradigma de programación orientada a objetos en la resolución de esta práctica. En ese orden de ideas, fueron creadas la clase abstracta “image” y sus clases concretas hijas “pgmimage” y “ppmimage”.

La clase *image* tiene como atributos su “magic number” (es decir, la línea en la que se específica si la imagen es ppm o pgm), su anchura, su altura y el valor máximo al que podrían asignarse los colores (como ya se dijo, se utilizaría 255).

Los métodos de la clase son los siguientes:

- *load(FILE* input)*: que carga el archivo.
- *save(FILE* output)*: que guarda la imagen procesada en otro archivo.
- *applyFilter(char* filterType)*: que se encarga de determinar el filtro a aplicar sobre la imagen.
- *loadFromData(char* magic, int width, int height, int maxColor, int* pixels)*: que se encarga de cargar el archivo dados los datos de manera individual (útil para implementaciones posteriores).
- *Image* createFromFile(char* filename)*: que es la que crea la imagen a partir del archivo .pgm y .ppm.

Además de los atributos de su clase padre, tanto la clase *pgmimage* como *ppmimage*, tenían un atributo adicional, la matriz de pixeles. Además de los métodos heredados por la clase padre, también poseen los siguientes método propios:

- *applyKernel(const float kernel[3][3])*: que es el que se encarga de realizar las operaciones matriciales para aplicar los filtros.
- *applyKernelToRegion(const float kernel[3][3], int startY, int endY, int startX, int endX, pthread_mutex_t* mutex)*: que se encarga de realizar las operaciones matriciales para aplicar los filtros pero dada una región específica de la imagen (útil para implementaciones posteriores).

Para que este último método tuviese sentido, se tuvieron también que crear las constantes con los kernels de operaciones matriciales a aplicar, que fueron los siguientes [4]:

- Blur:

$$\begin{matrix} \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \\ \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \\ \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \end{matrix}$$

- Laplace:

$$\begin{matrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{matrix}$$

- Sharpening:

$$\begin{matrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{matrix}$$

Una vez establecido todo lo que tiene que ver con el paradigma de POO, se procedió a modificar los códigos base proporcionados por el profesor para complementarlos de manera apropiada [1].

Siendo así, en *processor.cpp*, bastó con crear la imagen y utilizar los métodos de load() y save() de la clase para terminar.

```
    Image* image = Image::createFromFile(argv[1]);
    if (image == NULL) {
        std::cout << "Error, incorrect path or incorrect file." << std::endl;
        return 1;
    }
    FILE *file = fopen(argv[1], "r");
    if (file == NULL) {
        std::cout << "Error, could not open file." << std::endl;
        delete image;
        return 1;
    }
    image->load(file);
    fclose(file);

    FILE *output = fopen(argv[2], "w");
    if (output == NULL) {
        std::cout << "Error, could not open file." << std::endl;
        delete image;
        return 1;
    }
    image->save(output);
    fclose(output);

    delete image;
    return 0;
}
```

Para *filterer.cpp*, se necesitó más elaboración. Primero, se define cuál es el tipo de filtro a aplicar en base a lo ingresado por consola, luego, se realiza el mismo procedimiento de cargar el archivo que en *processor.cpp*, y ya con la imagen cargada, se llama al método de *applyFilter*, que a su vez llama al método de *applyKernel* y se aplica el filtro. Finalmente, guarda la imagen con el filtro aplicado en el archivo especificado por consola.

```
const char* filterType = nullptr;
for (int i = 3; i < argc; i++) {
    if (strcmp(argv[i], "--f") == 0 && i + 1 < argc) {
        filterType = argv[i + 1];
        break;
    }
}
if (filterType == NULL) {
  std::cout << "Error, must specify a filter with --f" << std::endl;
  return 1;
}
if (strcmp(filterType, "blur") != 0 &&
  strcmp(filterType, "laplace") != 0 &&
  strcmp(filterType, "sharpening") != 0) {
  std::cout << "Error, wrong filter. Use blur, laplace or sharpening" << std::endl;
  return 1;
}
```

```
Image* image = Image::createFromFile(argv[1]);
if (image == NULL) {
    std::cout << "Error, incorrect path or incorrect file." << std::endl;
    return 1;
}

FILE *file = fopen(argv[1], "r");
if (file == NULL) {
  std::cout << "Error, could not open the input file."<< std::endl;
  delete image;
  return 1;
}

image->load(file);
fclose(file);

clock_t cpu_start = clock();
image->applyFilter(filterType);
clock_t cpu_end = clock();
double cpu_time = double(cpu_end - cpu_start) / CLOCKS_PER_SEC;
```

```
FILE *output = fopen(argv[2], "w");
if (output == NULL) {
  std::cout << "Error, could not create the output file."<< std::endl;
  delete image;
  return 1;
}

image->save(output);
fclose(output);

auto wall_end = std::chrono::high_resolution_clock::now();
std::chrono::duration<double> wall_time = wall_end - wall_start;
```

3.2. PThread:

La implementación con PThread funcionaba de manera diferente. Lo que se pedía hacer era dividir la imagen de entrada en cuatro regiones y asignarle cada una de estas regiones a un hilo diferente para que paralelamente las cuatro regiones fuesen ejecutadas a la vez.

Nuevamente se utilizaron las mismas clases ya establecidas, y se copió el código de base para crear un *pth_filterer.cpp* que cumpliera con lo solicitado para este apartado.

Se comenzó creando una estructura para los datos del hilo, que contenía la imagen, el filtro a aplicar, el inicio en X y en Y de la región, el final en X y en Y de la región y el mutex, necesario para evitar problemas de condiciones de carrera.

```
pthread_mutex_t pixelsMutex = PTHREAD_MUTEX_INITIALIZER;

struct ThreadData {
    Image* image;
    const char* filterType;
    int startY;
    int endY;
    int startX;
    int endX;
    pthread_mutex_t* mutex;
};
```

Luego se creó la función worker llamada *applyFilterToRegion(void* arg)*, llamada igual que el método de las clases *pgmimage* y *ppmimage*, en la que básicamente se llamaba a este método específico (de la clase correspondiente según el archivo de entrada) dependiendo del filtro a aplicar, y mandando como argumentos los datos de la estructura de datos del hilo.

```
void* applyFilterToRegion(void* arg) {
    ThreadData* data = (ThreadData*)arg;
    if (PGMImage* pgmImage = dynamic_cast<PGMImage*>(data->image)) {
        if (strcmp(data->filterType, "blur") == 0) {
            pgmImage->applyKernelToRegion(BLUR_KERNEL, data->startY, data->endY,
                                          data->startX, data->endX, data->mutex);
        } else if (strcmp(data->filterType, "laplace") == 0) {
            pgmImage->applyKernelToRegion(LAPLACE_KERNEL, data->startY, data->endY,
                                          data->startX, data->endX, data->mutex);
        } else if (strcmp(data->filterType, "sharpening") == 0) {
            pgmImage->applyKernelToRegion(SHARPEN_KERNEL, data->startY, data->endY,
                                          data->startX, data->endX, data->mutex);
        }

    } else if (PPMImage* ppmImage = dynamic_cast<PPMImage*>(data->image)) {
        if (strcmp(data->filterType, "blur") == 0) {
            ppmImage->applyKernelToRegion(BLUR_KERNEL, data->startY, data->endY,
                                          data->startX, data->endX, data->mutex);
        } else if (strcmp(data->filterType, "laplace") == 0) {
            ppmImage->applyKernelToRegion(LAPLACE_KERNEL, data->startY, data->endY,
                                          data->startX, data->endX, data->mutex);
        } else if (strcmp(data->filterType, "sharpening") == 0) {
            ppmImage->applyKernelToRegion(SHARPEN_KERNEL, data->startY, data->endY,
                                          data->startX, data->endX, data->mutex);
        }
    }

    return nullptr;
}
```

Ya en el main, el proceso era similar al descrito en la versión secuencial del programa, salvo que luego de cargar la imagen, se hacía la división de la misma por regiones, que se almacenaba en un arreglo de tamaño cuatro de tipo estructura de datos de hilo, en la que se especificaban los datos de hilo de cada

una de las cuatro regiones (recordar que parte de los datos de hilo eran el inicio y fin de la región tanto en el eje X como en el eje Y).

```
int width = image->getWidth();
int height = image->getHeight();
int midX = width / 2;
int midY = height / 2;
ThreadData regions[4] = {
    {image, filterType, 0, midY, 0, midX, &pixelsMutex},
    {image, filterType, 0, midY, midX, width, &pixelsMutex},
    {image, filterType, midY, height, 0, midX, &pixelsMutex},
    {image, filterType, midY, height, midX, width, &pixelsMutex}
};
```

Hecho esto, se crean los hilos, uno por cada región y llamando a la función worker con cada una de las regiones, hasta que la final, se unen nuevamente cuando las cuatro regiones hayan terminado su ejecución.

```
pthread_t threads[4];
for (int i = 0; i < 4; i++) {
    pthread_create(&threads[i], nullptr, applyFilterToRegion, &regions[i]);
}

for (int i = 0; i < 4; i++) {
    pthread_join(threads[i], nullptr);
}
```

El proceso de guardado funciona de la misma manera que en la versión secuencial.

3.3. OpenMP:

Nuevamente, la versión de OpenMP se elaboró en base a la versión secuencial original, aplicando las diferencias solicitadas. Para esta versión, no había que dividir la imagen en regiones como en PThread, sino que había que ejecutar los tres filtros de manera paralela sobre un mismo archivo de entrada, por lo que ya no había que especificar el filtro a usar, sino que tocaba crear tres archivos de salida diferentes, uno para cada filtro.

Nuevamente, el proceso inicial era igual al de la versión secuencial, con las diferencias apareciendo luego de haber cargado la imagen.

Esta vez no había una sola imagen, sino que se crearon tres objetos de la clase *image*, uno para cada filtro a aplicar, pero en un inicio todos derivados de la misma imagen de entrada.

Una vez estaba cargada la imagen del archivo en los tres objetos *image*, se procedía a la implementación de OpenMP.

Se abría el bloque de *#pragma omp parallel sections*, y dentro de él, tres *#pragma omp section*, que serían equivalentes a los hilos. Dentro de cada una de estas secciones, cada objeto correspondiente a un filtro llamaba a su método *applyFilter*, junto con su respectivo filtro.

```
#pragma omp parallel sections
{
  #pragma omp section
  {
    blur->applyFilter("blur");
  }
  #pragma omp section
  {
    laplace->applyFilter("laplace");
  }
  #pragma omp section
  {
    sharpen->applyFilter("sharpening");
  }
}
```

Luego de esto, se guardaba las imágenes con los filtros aplicados en los tres archivos de salida especificados desde la consola.

3.4. MPI:

Para el modelo de paso de mensajes, implementado con MPI, el proceso nuevamente fue bastante distinto. Se repitió la solicitud presentada con OpenMP, de aplicar los tres filtros de manera paralela sobre una imagen inicial.

Claramente, lo primero apenas se inició el main fue inicializar MPI, definiendo el rank y el size de los procesos, que debía ser de 4 (uno maestro y tres esclavos encargados de cada filtro).

```
MPI_Init(&argc, &argv);
int rank, size;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);

if(size<4){
  if(rank == 0)
  {
    std::cout << "There needs to be 4 processes." << std::endl;
  }

  std::cout << "Usage:" << argv[0] << " input_image.pgm output_blur.pgm output_laplace.pgm output_sharpening.pgm" << std::endl;
  std::cout << "or "<< argv[0] << "input_image.ppm output_blur.ppm output_laplace.ppm output_sharpening.ppm" << std::endl;
  MPI_Finalize();
  return 1;
}
```

Al rank 0, se le asignaba el rol de maestro, y desde allí se cargaba la imagen del archivo de entrada como ya es costumbre. Luego, con un ciclo for, se le enviaba a los demás procesos (los esclavos), los datos de la imagen.

```
for (int i = 1; i < 4; i++) {
    MPI_Send(&width, 1, MPI_INT, i, TAG_WORK, MPI_COMM_WORLD);
    MPI_Send(&height, 1, MPI_INT, i, TAG_WORK, MPI_COMM_WORLD);
    MPI_Send(&maxColor, 1, MPI_INT, i, TAG_WORK, MPI_COMM_WORLD);

    int magic_len = strlen(magic) + 1;
    MPI_Send(&magic_len, 1, MPI_INT, i, TAG_WORK, MPI_COMM_WORLD);
    MPI_Send(magic, magic_len, MPI_CHAR, i, TAG_WORK, MPI_COMM_WORLD);
}

if (strcmp(magic, "P2") == 0) {
    PGMImage* pgm = dynamic_cast<PGMImage*>(image);
    int* pixels = pgm->getPixels();
    int pixelCount = width * height;

    for (int i = 1; i < 4; i++) {
        MPI_Send(pixels, pixelCount, MPI_INT, i, TAG_WORK, MPI_COMM_WORLD);
    }
} else if (strcmp(magic, "P3") == 0) {
    PPMImage* ppm = dynamic_cast<PPMImage*>(image);
    int* pixels = ppm->getPixels();
    int pixelCount = width * height * 3;

    for (int i = 1; i < 4; i++) {
        MPI_Send(pixels, pixelCount, MPI_INT, i, TAG_WORK, MPI_COMM_WORLD);
    }
}
```

Una vez hecho esto, aún dentro del proceso maestro, se creaba un arreglo de imágenes dónde recibir las imágenes con los filtros aplicados, y posterior a esto, se utilizaba este arreglo para recibir las imágenes resultantes de cada uno de los procesos esclavos y luego se guardaban respectivamente en su correspondiente archivo de salida especificado en la ejecución por consola.

```
Image* results[3];
for (int i = 1; i < 4; i++) {
    MPI_Status status;
    int result_size;
    MPI_Recv(&result_size, 1, MPI_INT, i, TAG_RESULT, MPI_COMM_WORLD, &status);

    int* result_pixels = new int[result_size];
    MPI_Recv(result_pixels, result_size, MPI_INT, i, TAG_RESULT, MPI_COMM_WORLD, &status);

    if (strcmp(magic, "P2") == 0) {
        PGMImage* result_img = new PGMImage();
        result_img->loadFromData(magic, width, height, maxColor, result_pixels);
        results[i-1] = result_img;
    } else {
        PPMImage* result_img = new PPMImage();
        result_img->loadFromData(magic, width, height, maxColor, result_pixels);
        results[i-1] = result_img;
    }

    delete[] result_pixels;
}
```

Luego más abajo, estaban especificadas las funciones workers de los hilos esclavos.

Se recibían los datos enviados por el proceso maestro por cada uno de los procesos esclavos, y se creaban las imágenes resultantes, cargando los datos enviados por el proceso maestro de manera individual, usando el método *loadFromData*.

```
MPI_Recv(&width, 1, MPI_INT, 0, TAG_WORK, MPI_COMM_WORLD, &status);
MPI_Recv(&height, 1, MPI_INT, 0, TAG_WORK, MPI_COMM_WORLD, &status);
MPI_Recv(&maxColor, 1, MPI_INT, 0, TAG_WORK, MPI_COMM_WORLD, &status);
MPI_Recv(&magic_len, 1, MPI_INT, 0, TAG_WORK, MPI_COMM_WORLD, &status);

char* magic = new char[magic_len];
MPI_Recv(magic, magic_len, MPI_CHAR, 0, TAG_WORK, MPI_COMM_WORLD, &status);

int pixelCount;
if (strcmp(magic, "P2") == 0) {
    pixelCount = width * height;
} else {
    pixelCount = width * height * 3;
}

int* pixels = new int[pixelCount];
MPI_Recv(pixels, pixelCount, MPI_INT, 0, TAG_WORK, MPI_COMM_WORLD, &status);

Image* image;
if (strcmp(magic, "P2") == 0) {
    image = new PGMImage();
} else {
    image = new PPMImage();
}

image->loadFromData(magic, width, height, maxColor, pixels);
```

Ya con la imagen cargada, se procedía a aplicar los filtros dependiendo del rank del proceso (ya que cada rank estaba encargado de un filtro diferente), y luego, una vez aplicado el filtro, los datos de la imagen resultante eran enviados de vuelta al proceso maestro para que los guardara como ya se describió.

```
if (rank == 1) image->applyFilter("blur");
else if (rank == 2) image->applyFilter("laplace");
else if (rank == 3) image->applyFilter("sharpening");

int* result_pixels;
if (strcmp(magic, "P2") == 0) {
    result_pixels = dynamic_cast<PGMImage*>(image)->getPixels();
} else {
    result_pixels = dynamic_cast<PPMImage*>(image)->getPixels();
}

MPI_Send(&pixelCount, 1, MPI_INT, 0, TAG_RESULT, MPI_COMM_WORLD);
MPI_Send(result_pixels, pixelCount, MPI_INT, 0, TAG_RESULT, MPI_COMM_WORLD);
```

4. Resultados:

Ahora sí, lo interesante. Habiendo ya explicado todo el proceso de realización del código, lo siguiente es ejecutarlos todos con miras a responder la pregunta de interés planteada al inicio. Con fines comparativos, se ejecutará el mismo filtro sobre la misma imagen en todos los casos. Sin más preámbulos, se iniciará con la presentación de los resultados obtenidos.

4.1. Secuencial:

Los resultados temporales de la ejecución secuencial (se aplicó el filtro blur sobre la imagen de lena.ppm) fueron los siguientes:

```
CPU Time (applying the filter only): 0.002054 seconds
Total Execution Time: 0.116234 seconds
```

Los resultados visuales de la imagen fueron los siguientes:

a. *lena.ppm* (sin filtros, solo procesada):

b. *lenablur.ppm* (con el filtro blur aplicado de manera secuencial):

4.2. PThread:

Los resultados temporales de la ejecución con PThread fueron los siguientes:

```
CPU Time (applying the filter only): 0.002947 seconds
Total Execution Time: 0.0542972 seconds
```

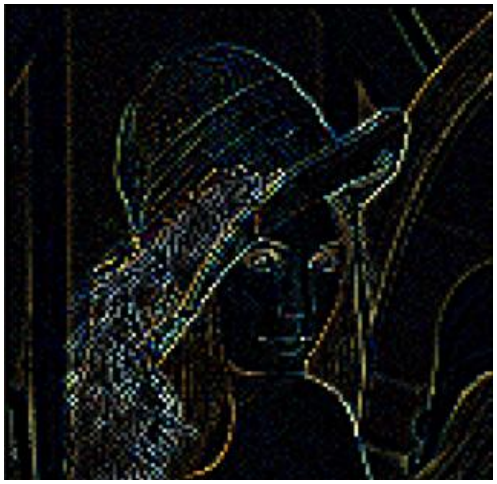
4.3. OpenMP:

Los resultados temporales de la ejecución con OpenMP fueron los siguientes:

```
CPU Time (applying the filter only): 0.005474 seconds
Total Execution Time: 0.127383 seconds
```

Como en este caso se aplicaron los tres filtros, se procederá a mostrar los resultados visuales de los no previamente presentados.

c. *lenalaplace.ppm* (con el filtro Laplace aplicado con OpenMP):



d. *lenasharpen.ppm* (con el filtro sharpening aplicado con OpenMP):

4.4. MPI:

Los resultados temporales de la ejecución con OpenMP fueron los siguientes:

```
MPI Total Time: 0.0776497 seconds
CPU Time (applying the filter only): 0.02622 seconds
Total Execution Time: 0.0776492 seconds
```

Una vez presentados todos los resultados, es hora de compararlos.

El algoritmo secuencial tardó 0.002054 segundos en CPU y 0.116234 segundos en ejecución total, únicamente aplicando el filtro sobre una sola imagen.

El algoritmo de PThread tardó 0.002947 segundos en CPU y 0.054297 segundos en ejecución total, por lo que aunque tardó unas diezmilésimas de segundo más que el secuencial en CPU, probablemente debido a la asignación de los hilos, sincronización, y cuestiones derivadas al mutex implementado, tardó casi la mitad del tiempo en ejecución total haciendo el mismo trabajo (aplicar un único filtro sobre una imagen).

El algoritmo de OpenMP tardó 0.005474 segundos en CPU y 0.127383 segundos en ejecución total, lo que significa que fue más lento que el secuencial en CPU por unas cuantas milésimas de segundo, y también un poco más lento en la ejecución total por poco más de una centésima de segundo, aunque realizando el triple de trabajo que el algoritmo secuencial (aplicó los tres filtros a la misma imagen). La diferencia entre los tiempos de ejecución es despreciable, pero a cambio el algoritmo de OpenMP realizó el triple de tareas que el secuencial, demostrando su eficiencia.

Finalmente, el algoritmo de MPI tardó 0.02622 segundos en CPU y 0.0776492 segundos en ejecución total. Nuevamente, fue significativamente más lento en CPU (aproximadamente diez veces más lento que el secuencial), seguramente debido a que como se trataba de un modelo de intercambio de mensajes, la interacción no era tan sencilla como en los otros dos, sino que tenía que enviar y recibir datos entre procesos, lo cual afecta el rendimiento en CPU. Sin embargo, su tiempo de ejecución total fue nuevamente bastante rápido, siendo más rápido incluso que el secuencial y que PThread, a pesar de, nuevamente, estar ejecutando el triple de tareas.

5. Conclusiones:

Luego de haber presentado los resultados, es hora de responder la pregunta de interés planteada al inicio:

¿Cómo influye la programación paralela, utilizando OpenMP o Pthreads y MPI (En un entorno simulado con docker), en el tiempo de ejecución del filtrado de imágenes PPM y PGM en comparación con una implementación secuencial?

La programación paralela, utilizando OpenMP, PThread y MPI influye positivamente el tiempo de ejecución del filtrado de imágenes PPM y PGM en comparación con la implementación secuencial. Esto es debido a que, aunque no siempre el tiempo de ejecución total fuese menor en los resultados obtenidos (como fue el caso de la

ejecución de OpenMP), siempre había alguna ventaja de eficiencia, ya que cuando esto ocurría, se logró producir el triple del trabajo en un tiempo con una diferencia despreciable.

Por tanto, en el filtrado de imágenes, la implementación de la programación paralela es altamente beneficial, debido a que se trata de un proceso que secuencialmente puede resultar repetitivo y que de manera paralela se logra reducir el tiempo de ejecución total y aprovechar mejor los recursos disponibles.

6. Referencias:

[1] https://github.com/japeto/netpbm_filters.git

[2] https://netpbm.sourceforge.net/doc/pgm.html

[3] https://netpbm.sourceforge.net/doc/ppm.html

[4] https://en.wikipedia.org/wiki/Kernel_(image_processing)