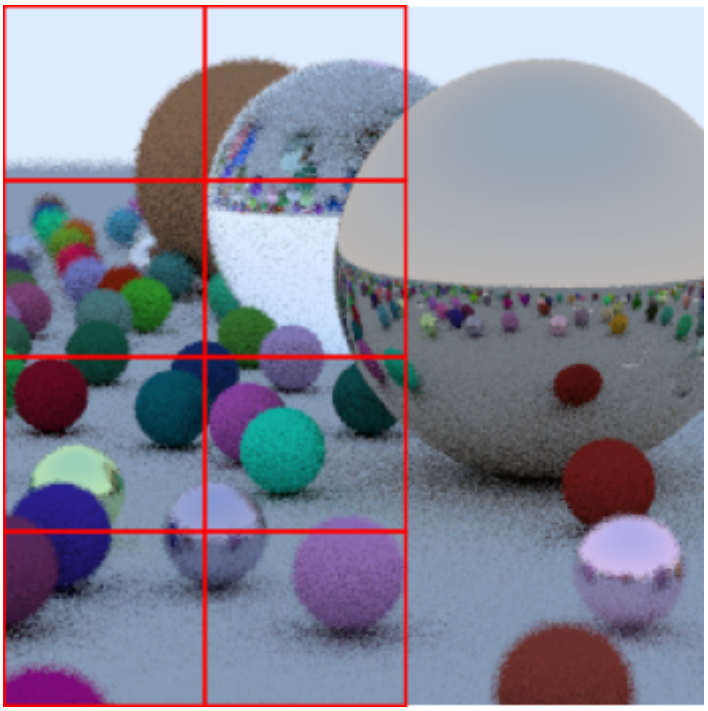
Universidad Rey Juan Carlos

Práctica 2. MPI - Renderizado de películas en ray tracing



Grupo C

Sohaib Tarfi Elidrissi

Daniel Briones

Índice

Introducción

El presente trabajo tiene como objetivo principal analizar y comparar las diferencias y ventajas de diversas tecnologías de paralelización, como MPI, OpenMP, en la aceleración de procesos paralelizables, específicamente en el renderizado de imágenes.

Además, se creará una estrategia para paralelizar estas tareas entre múltiples procesos y se implementará instrucciones de código adicionales para poder tener un análisis de tiempos de ejecución que nos permitan tener una vista más amplia de las mejoras realizadas.

En la memoria se analizará los resultados obtenidos explicando la utilidad de cada método implementado en diferentes casos.

Análisis del código

En este análisis se van a exponer diferentes métodos para la generación de imágenes por ray tracing, así comprobando la implementación y uso de diferentes tecnologías de paralelización como son MPI y OPENMP.

Todos estos modelos se han ejecutado con la optimización de -O2.

Versión 1: MPI - Repartir imágenes entre los nodos

Para comenzar definimos las variables que va a usar cada hilo, siendo estas, el tiempo a calcular, el rango, el número de procesos, el ancho y largo de la imagen, el número de rayos y la información que se va a comunicar entre padre e hijos:

```
int main(int argc, char** argv) {
    int nProcesses, rank;
    double startTime, endTime; //tiempo de inicio y fin de ejecucion

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);//Proceso actual
    MPI_Comm_size(MPI_COMM_WORLD, &nProcesses);//numero de procesos
    MPI_Status status;
    MPI_Request request;

    srand(time(0));

    int width = 256; // ancho de la imagen
    int height = 256; // alto de la imagen
    int rayNumber = 10; //brillo de la imagen

    fflush(stdout);
    int currentFrame, random, mensaje;
    int info[2]; //es el frame y el numero random del frame
```

Se envía de manera asíncrona desde el padre a cada hijo el número de la imagen que vamos a procesar. Se tiene como objetivo enviar una imagen a cada hijo para conseguir que a mayor número de imágenes, más dividido esté el trabajo:

```
// Master process
    if (rank == 0) {
        startTime = MPI_Wtime();

        for (int i = 1; i < nProcesses; i++) { //Enviamos los frames a los hijos
            srand(i);
            currentFrame = i;

            info[0] = currentFrame;
```

```
            info[1] = rand();
            printf("Enviamos currentFrame: %d y un numero random: %d al hijo %d\n", info[0],
info[1], i);

            fflush(stdout);

            MPI_Isend(info, 2, MPI_INT, i, 0, MPI_COMM_WORLD, &request); //Enviamos el frame y el
numero random al hijo
            printf("Enviado mensaje al proceso %d\n", i);
        }

        printf("Se han enviado todos los frames, esperando a ser resueltos\n");

        fflush(stdout);
```

Cuando se termina de enviarlas, se espera a que los hijos terminen de procesarlas:

```
// Esperamos a que los hijos terminen
        for (int i = 1; i < nProcesses; i++) { //Recibimos los mensajes de los hijos
            MPI_Recv(&mensaje, 1, MPI_INT, MPI_ANY_SOURCE, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            printf("Recibido mensaje de finalizado del hijo: %d\n", mensaje);
            fflush(stdout);
        }
```

Finalmente en el proceso padre, se envía un mensaje a cada hijo para que termine su ejecución y se calcula el tiempo total que ha tardado:

```
 // Enviamos mensaje de finalizacion a los hijos
        for (int i = 1; i < nProcesses; i++) {
            info[0] = -1; //Enviamos -1 para que los hijos sepan que deben finalizar
            MPI_Send(info, 2, MPI_INT, i, 0, MPI_COMM_WORLD);
        }

        endTime = MPI_Wtime();
        printf("Padre, Tiempo total de ejecucion: %f segundos\n", endTime - startTime); //Tiempo
total de ejecucion

        MPI_Finalize();
    }
```

Por otro lado, cada hijo realiza lo siguiente. Se recibe el mensaje del padre y si este es “-1” el hijo rompe el bucle y finaliza:

```
 else {
        // Proceso hijo
        while (1) {
            MPI_Recv(info, 2, MPI_INT, 0, 0, MPI_COMM_WORLD, &status); //Recibimos el frame y el
numero random del frame
            if (info[0] == -1) { //Si el frame es -1, finalizamos
                mensaje = rank;
                MPI_Send(&mensaje, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
                printf("Hijo %d,\tFinaliza\n", rank);
```

```
                break;
            }
```

En caso contrario, se guarda el frame y el número aleatorio usado para la aleatorización de la composición, y se divide la imagen en fragmentos:

```
            printf("Recibido frame numero: %d en el hijo: %d\n", currentFrame, rank);
            random = info[1];
            printf("Recibido random: %d en el hijo: %d\n", random, rank);

            fflush(stdout);

            srand(random);
            int size = sizeof(unsigned char) * width * height * 3; //3 canales de color
            unsigned char* data = (unsigned char*)calloc(size, 1); //inicializamos la imagen a 0

            int patch_x_start = 0;
            int patch_x_end = width; //ancho de la imagen
            int patch_y_start = 0;
            int patch_y_end = height; //alto de la imagen
```

Finalmente, creamos la imagen usando la función rayTracingCPU y creamos el archivo final:

```
printf("Hijo %d:\tcalculando frame %d\n", rank, currentFrame);
            fflush(stdout);

            startTime = MPI_Wtime(); //iniciamos el tiempo de ejecucion del hijo
            rayTracingCPU(data, width, height, rayNumber, patch_x_start, patch_y_start, patch_x_end,
patch_y_end);
            endTime = MPI_Wtime(); //finalizamos el tiempo de ejecucion del hijo

            printf("Hijo %d,\tTiempo de ejecucion: %f segundos\n", rank, endTime - startTime);

            char fileName[32]; //archivo de la imagen
            sprintf(fileName, "../imgCPUImg%d.bmp", currentFrame);
            writeBMP(fileName, data, width, height); //escribimos la imagen en un archivo
            printf("Hijo %d,\tImagen%d creada\n", rank, currentFrame);
            fflush(stdout);
            free(data);
```

Cuando se haya terminado, el hijo devolverá un mensaje al padre con su número de proceso:

```
 mensaje = rank;
            MPI_Send(&mensaje, 1, MPI_INT, 0, 0, MPI_COMM_WORLD); //Enviamos mensaje de finalizado
al padre
        }
        MPI_Finalize();
    }
    return 0;
}
```

Versión 2 : OpenMP - Repartir imágenes entre los nodos

La versión de openMP es bastante sencilla. Primero podemos modificar la función de rayTracing para así paralelizar el bucle exterior que recorre las filas. El uso de schedule(dynamic) permite a OpenMP distribuir las iteraciones del bucle de manera dinámica entre los hilos disponibles, lo que puede mejorar el equilibrio de carga si algunas filas son más costosas de procesar que otras.

```
void rayTracingCPU(unsigned char* img, int w, int h, int ns = 10, int px = 0, int py = 0, int pw =
-1, int ph = -1) {
    if (pw == -1) pw = w;
    if (ph == -1) ph = h;
    int patch_w = pw - px;
    Scene world = randomScene();
    world.setSkyColor(Vec3(0.5f, 0.7f, 1.0f));
    world.setInfColor(Vec3(1.0f, 1.0f, 1.0f));

    Vec3 lookfrom(13, 2, 3);
    Vec3 lookat(0, 0, 0);
    float dist_to_focus = 10.0;
    float aperture = 0.1f;

    Camera cam(lookfrom, lookat, Vec3(0, 1, 0), 20, float(w) / float(h), aperture, dist_to_focus);

#pragma omp parallel for schedule(dynamic)
    for (int j = 0; j < (ph - py); j++) {
        for (int i = 0; i < (pw - px); i++) {
            Vec3 col(0, 0, 0);
            for (int s = 0; s < ns; s++) {
                float u = float(i + px + customRandom()) / float(w);
                float v = float(j + py + customRandom()) / float(h);
                Ray r = cam.get_ray(u, v);
                col += world.getSceneColor(r);
            }
            col /= float(ns);
            col = Vec3(sqrt(col[0]), sqrt(col[1]), sqrt(col[2]));

            img[(j * patch_w + i) * 3 + 2] = char(255.99 * col[0]);
            img[(j * patch_w + i) * 3 + 1] = char(255.99 * col[1]);
            img[(j * patch_w + i) * 3 + 0] = char(255.99 * col[2]);
        }
    }
}
```

Se definen las variables como en el anterior ejemplo:

```
int main(int argc, char** argv) {
    double startTime, endTime;

    int width = 256; // ancho de la imagen
    int height = 256; // alto de la imagen
    int rayNumber = 10; //brillo de la imagen

    startTime = omp_get_wtime();
```

Se guarda el número de hilos con el que se va a ejecutar el programa, donde todo el código dentro de este bloque se ejecutará en paralelo usando los 100 hilos configurados previamente

```
omp_set_num_threads(100);
#pragma omp parallel
```

El objetivo del padre es el mismo que en la anterior versión es dividir cada imagen por cada hilo, donde se calcula el tamaño de la imagen en bytes y se asigna memoria para ella, inicializando todo a cero. Además que luego se asigna una imagen por cada nodo, que en el código siguiente se representa como parche.

```
omp_set_num_threads(100);
#pragma omp parallel
    {
        int thread_num = omp_get_thread_num();
        printf("Thread %d started.\n", thread_num);

        time_t seed = time(0) + thread_num;
        srand(seed); // Semilla única para cada hilo

        int size = sizeof(unsigned char) * width * height * 3; //3 canales de color
        unsigned char* data = (unsigned char*)calloc(size, 1); //inicializamos la imagen a 0

        int patch_x_start = 0;
        int patch_x_end = width; //ancho de la imagen
        int patch_y_start = 0;
        int patch_y_end = height; //alto de la imagen

        rayTracingCPU(data, width, height, rayNumber, patch_x_start, patch_y_start, patch_x_end,
patch_y_end);

        char fileName[32]; //archivo de la imagen
        sprintf(fileName, "./imgCPUImg%d.bmp", thread_num);
        writeBMP(fileName, data, width, height); //escribimos la imagen en un archivo
        free(data);

        printf("Thread %d finished creating image.\n", thread_num);
    }
```

Finalmente se muestra el tiempo de ejecución total.

```
 endTime = omp_get_wtime();
    printf("Tiempo total de ejecucion: %f segundos\n", endTime - startTime); //Tiempo total de
ejecucion
```

Versión 3: MPI y OMP - Repartir imágenes entre los nodos

Para la versión de MPI con OpenMP se usará el mismo código la versión 1 con MPI. Por otro lado, usaremos openMP para paralelizar la parte de la función rayTracingCPU. En este caso usaremos pragma parallel for on collapse(2) para paralelizar los dos bucles anidados y separarlos entre los hilos disponibles:

```
void rayTracingCPU(unsigned char* img, int w, int h, int ns = 10, int px = 0, int py = 0, int pw =
-1, int ph = -1) {
    if (pw == -1) pw = w;
    if (ph == -1) ph = h;
    int patch_w = pw - px;
    Scene world = randomScene();
    world.setSkyColor(Vec3(0.5f, 0.7f, 1.0f));
    world.setInfColor(Vec3(1.0f, 1.0f, 1.0f));

    Vec3 lookfrom(13, 2, 3);
    Vec3 lookat(0, 0, 0);
    float dist_to_focus = 10.0;
    float aperture = 0.1f;

    Camera cam(lookfrom, lookat, Vec3(0, 1, 0), 20, float(w) / float(h), aperture, dist_to_focus);

#pragma omp parallel for collapse(2)
    for (int j = 0; j < (ph - py); j++) {
        for (int i = 0; i < (pw - px); i++) {

            Vec3 col(0, 0, 0);
            for (int s = 0; s < ns; s++) {
                float u = float(i + px + customRandom()) / float(w);
                float v = float(j + py + customRandom()) / float(h);
                Ray r = cam.get_ray(u, v);
                col += world.getSceneColor(r);
            }
            col /= float(ns);
            col = Vec3(sqrt(col[0]), sqrt(col[1]), sqrt(col[2]));

            img[(j * patch_w + i) * 3 + 2] = char(255.99 * col[0]);
            img[(j * patch_w + i) * 3 + 1] = char(255.99 * col[1]);
            img[(j * patch_w + i) * 3 + 0] = char(255.99 * col[2]);
        }
    }
}
```

La directiva **#pragma omp parallel for collapse(2)** instruye al compilador para que paralelice ambos bucles anidados como si fueran un único bucle plano. Esto significa que en lugar de distribuir sólo el bucle más externo (j) entre los hilos, ambos bucles (j e i) se combinan y se distribuyen de manera más uniforme entre los hilos.

La parte de la función main no se explicará ya que se reutiliza la misma lógica que el código de la versión 1 con MPI.

Versión 4 : MPI - Selección de imagen y distribución

En este caso, se dividirá la imagen en parches en este caso esos parches serán columnas en función de los n procesos con la que se quiera ejecutar (siendo n el número de procesos). Lo siguiente es una representación de cómo trabajaría con 4 procesos (3 hijos y 1 padre):

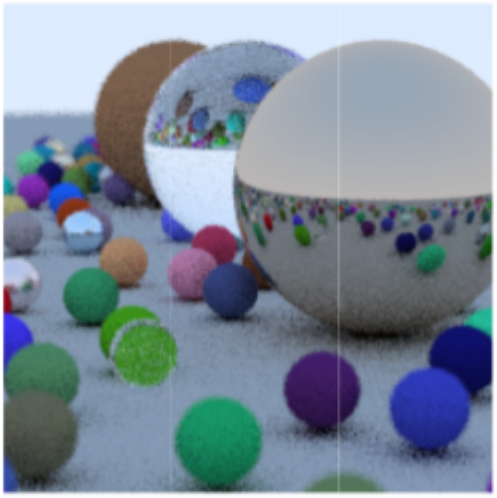


Figura 1. Imagen final sin división

FIgura 2. División por parches

Se definen las variables y la forma en cómo se van a dividir las columnas:

```
int main(int argc, char** argv) {
    double start_time, end_time, start_time_child, end_time_child;

    int nProcesses, rank;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &nProcesses);
    MPI_Status status;
    MPI_Request request;
    srand((unsigned int)time(0));

    int w = 256;
    int h = 256;
    int ns = 10;

    int nHijos = nProcesses - 1;
    int patchWidth = w / nHijos;
    patchWidth = ((w / nHijos) + 7) & ~7; // Redondear al múltiplo de 8 más cercano
    int patchHeight = h / nHijos;
```

Se envía a cada hijo la posición, ancho y altura de la columna que deberán generar:

```
if (rank == 0) { // Proceso padre
        start_time = omp_get_wtime();
        int hijo = 1;
                for (int i = 0; i < nHijos; ++i) { // Enviar los parches de imagen a los hijos
                        int mensaje2[4] = { i * patchWidth, 0, (i + 1) * patchWidth, h }; //
start_x, start_y, end_x, end_y = parche de imagen
            printf("Main:\tenvia a hijo%d start = (%d, %d) / end = (%d, %d)\n", hijo, mensaje2[0],
mensaje2[1], mensaje2[2], mensaje2[3]);
            fflush(stdout);
                        MPI_Send(mensaje2, 4, MPI_INT, hijo, 0, MPI_COMM_WORLD); // Enviar el
parche de imagen al proceso hijo
            ++hijo;
        }
```

A cada hijo se envia que termine para asegurarnos de que finaliza al generar la columna:

```
int mensaje2[4] = { -1, -1, -1, -1 }; // Fin de la comunicación
                for (int i = 1; i <= nHijos; ++i) { // Enviar mensaje de fin a los hijos
            MPI_Isend(mensaje2, 4, MPI_INT, i, 0, MPI_COMM_WORLD, &request);
        }
        printf("Main:\tTodos los parches enviados a los hijos.\n");
        fflush(stdout);

        // Crear un buffer para la imagen completa
        int sizeTotal = sizeof(unsigned char) * w * h * 3;
        unsigned char* img = (unsigned char*)calloc(sizeTotal, 1);
```

Después, el proceso padre irá juntando columna por columna la imagen generada para poder mostrarla al final de la ejecución del programa:

```
   // Recibir las imágenes de todos los hijos
        for (int i = 1; i <= nHijos; ++i) {
```

```
            int size;
            MPI_Recv(&size, 1, MPI_INT, i, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            // Recibir los datos del proceso hijo
            unsigned char* data = (unsigned char*)calloc(size, 1);
            MPI_Recv(data, size, MPI_UNSIGNED_CHAR, i, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            printf("Main:\tImagen recibida del hijo %d\n", i);
            fflush(stdout);
            // Combinar las imágenes de los hijos en la imagen completa
            // Copiar los datos de la imagen en el buffer de la imagen completa
            int patch_x_size = patchWidth;
            int patch_y_size = patchHeight;
            for (int y = 0; y < patch_y_size; ++y) {
                for (int x = 0; x < patch_x_size; ++x) {
                                        for (int c = 0; c < 3; ++c) { // R, G, B
                                                img[((y * w) + (x + (i - 1) * patch_x_size)) * 3 +
c] = data[(y * patch_x_size + x) * 3 + c]; // Copiar los datos de la imagen en el buffer de la
imagen completa
                    }
                }
            }
            // Escribir la imagen completa en un archivo
            char fileName[32];
            sprintf_s(fileName, sizeof(fileName), "../imgCPUFinal.bmp");
            writeBMP(fileName, img, w, h);
            printf("Main:\tImagen final creada\n");
            fflush(stdout);

            free(data);
        }
```

Cada hijo entra en un bucle donde espera recibir coordenadas del parche de imagen, procesa el parche, guarda la imagen generada, y envía los datos al padre. Este proceso se repite hasta que reciben un mensaje de finalización, permitiendo así que múltiples parches de imagen se generen en paralelo, acelerando la creación de la imágen completa.

```
else {
        // Proceso hijo
        while (1) {
            MPI_Recv(info, 2, MPI_INT, 0, 0, MPI_COMM_WORLD, &status); //Recibimos el frame y el
numero random del frame

            if (info[0] == -1) { //Si el frame es -1, finalizamos
                mensaje = rank;
                MPI_Send(&mensaje, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
                printf("Hijo %d,\tFinaliza\n", rank);
                break;
            }

            currentFrame = info[0];
            printf("Recibido frame numero: %d en el hijo: %d\n", currentFrame, rank);
            random = info[1];
            printf("Recibido random: %d en el hijo: %d\n", random, rank);

            fflush(stdout);

            srand(random);
            int size = sizeof(unsigned char) * width * height * 3; //3 canales de color
```

```
            unsigned char* data = (unsigned char*)calloc(size, 1); //inicializamos la imagen a 0

            int patch_x_start = 0;
            int patch_x_end = width; //ancho de la imagen
            int patch_y_start = 0;
            int patch_y_end = height; //alto de la imagen

            printf("Hijo %d:\tcalculando frame %d\n", rank, currentFrame);
            fflush(stdout);

            startTime = MPI_Wtime(); //iniciamos el tiempo de ejecucion del hijo
            rayTracingCPU(data, width, height, rayNumber, patch_x_start, patch_y_start, patch_x_end,
patch_y_end);
            endTime = MPI_Wtime(); //finalizamos el tiempo de ejecucion del hijo

            printf("Hijo %d,\tTiempo de ejecucion: %f segundos\n", rank, endTime - startTime);

            char fileName[32]; //archivo de la imagen
            sprintf(fileName, "../imgCPUImg%d.bmp", currentFrame);
            writeBMP(fileName, data, width, height); //escribimos la imagen en un archivo
            printf("Hijo %d,\tImagen%d creada\n", rank, currentFrame);
            fflush(stdout);
            free(data);

            mensaje = rank;
            MPI_Send(&mensaje, 1, MPI_INT, 0, 0, MPI_COMM_WORLD); //Enviamos mensaje de finalizado
al padre
        }
        MPI_Finalize();
    }
    return 0;
}
```

Análisis de tiempos

En la siguiente tabla se representan los tiempos de los diferentes códigos donde para las **versiones 1, 2 y 3**, el número de procesos coincide con el número de imágenes generadas, sin embargo, la **versión 4** sólo se genera una imagen, pero es procesada por el número de procesos, siendo el número de procesos el número de parches que se divide una imagen para ser procesada.

Además la magnitud de los tiempos se realiza en **segundos.**

Tabla de tiempos

Número de procesos	versión 1 : MPI	versión 2 : OpenMp	versión 3 : MPI + OpenMp	versión 4 : MPI división/parches
100	205.718.290	245.348.000	263.062.218	9.609.259
12	22.122.567	29.580.000	28.590.731	3.054.350

4	**5.929.600**	**10.684.000**	**7.261.230**	**2.378.955**
2	**2.180.380**	**7.913.000**	**2.523.992**	**2.309.021**

Se comenzará analizar los tiempos de cada código para finalmente dar una conclusión en conjunto.

Versión 1: MPI - Repartir imágenes entre los nodos

La implementación utilizando MPI (Message Passing Interface) en la versión 1 es eficiente para distribuciones de tareas en sistemas con múltiples nodos de CPU. Se observa que a medida que aumenta el número de procesos, el tiempo de ejecución disminuye significativamente. Sin embargo, al usar 100 procesos, la eficiencia disminuye debido a la sobrecarga de comunicación entre procesos. Esto sugiere que, aunque MPI es muy potente, hay un punto de saturación donde los beneficios adicionales de paralelizar más se ven contrarrestados por la complejidad de la comunicación.

Versión 2 : OpenMP - Repartir imágenes entre los nodos

Para la versión 2 se observa que OpenMP es adecuado para la paralelización en un solo nodo con múltiples núcleos. La reducción en los tiempos de ejecución es significativa al aumentar el número de hilos, aunque no tan marcada como con MPI. Esto hace que OpenMP sea ideal para aplicaciones donde la memoria compartida puede ser gestionada eficientemente entre hilos. Es una solución más sencilla de implementar en comparación con MPI, pero su escalabilidad es limitada a un hardware de 1 solo nodo.

Versión 3: con MPI y OMP - Repartir imágenes entre los nodos

La combinación de MPI y OpenMP aprovecha las ventajas de ambos paradigmas y es adecuada para sistemas con múltiples nodos y múltiples núcleos por nodo. Esta estrategia proporciona aunque es mínima, una mejora en los tiempos de ejecución al combinar la distribución de tareas entre nodos (MPI) y la paralelización dentro de cada nodo (OpenMP).

Versión 4 : MPI - Selección de imagen y distribución

Esta técnica, procesa una sola imagen dividiéndola en parches entre múltiples procesos, muestra ser la menos eficiente en términos de tiempo de ejecución. Aunque se intenta balancear la carga de trabajo, la sobrecarga de coordinación y comunicación entre procesos es alta, lo que resulta en tiempos de ejecución superiores comparados con otras estrategias. Es la opción menos favorable cuando se busca maximizar la eficiencia y reducir tiempos de procesamiento.

Conclusiones

Tras realizar las diferentes pruebas se llega a ver que todas las versiones tienen sus ventajas y desventajas, sin embargo, cada versión tiene su mejor uso según la tarea y el hardware disponible:

La versión 1 (MPI) es ideal para tareas que requieren la distribución de trabajo en sistemas con múltiples nodos,, evitando un número excesivo de procesos que cause sobrecarga de comunicación, y priorizando el resultado final ya que al tener una menor carga de gestión y un menor intercambio de mensajes, completa el total de tareas en un menor tiempo;

La versión 2 (OpenMP) es más adecuada para tareas paralelizadas en un solo nodo con múltiples núcleos, aprovechando la memoria compartida y siendo más sencilla de implementar;

La versión 3 (MPI + OpenMP) es óptima para tareas que necesitan combinar la distribución de trabajo entre múltiples nodos y la paralelización dentro de cada nodo, ideal para sistemas heterogéneos;

La versión 4 (MPI con división en parches) resulta menos eficiente debido a la alta sobrecarga de coordinación y comunicación, siendo generalmente la opción menos favorable para maximizar la eficiencia y reducir tiempos de procesamiento. Aunque es el más adecuado cuando se busca conseguir menos latencia entre resultados o un flujo de datos. ya que se procesan trozos de imagen y no la imagen completa y se puede obtener la imagen y la información de cómo se procesa la imagen de manera gradual.