Universidad
Rey Juan Carlos

PRÁCTICA 2

Computación de Altas Prestaciones

Paula Barroso Robleda 54406071 P

Marta Sacristan Villalba 09130815 E

Contenido

INTRODUCCIÓN

En esta práctica se nos ha pedido trabajar con ray tracing, una técnica de renderizado ampliamente utilizada por su capacidad para generar imágenes con un alto nivel de realismo. Sin embargo, esa precisión visual tiene un coste computacional muy elevado, lo cual se convierte en un reto especialmente significativo cuando se renderizan escenas complejas o secuencias de múltiples fotogramas.

Nuestro objetivo ha sido evaluar cómo diferentes técnicas de paralelización pueden mejorar el rendimiento de un motor de ray tracing que originalmente funciona de manera secuencial. Para ello, hemos partido de un código base proporcionado y lo hemos modificado para introducir paralelismo utilizando dos tecnologías muy conocidas en el campo de la computación de alto rendimiento: OpenMP y MPI. En concreto, hemos explorado la paralelización entre fotogramas con MPI y la división del renderizado de cada fotograma en porciones independientes con OpenMP, utilizando tanto filas como columnas como unidades de trabajo.

Hemos decidido centrarnos exclusivamente en estas dos tecnologías, dejando fuera CUDA, para poder profundizar mejor en las posibilidades y limitaciones del paralelismo en CPU. A lo largo de esta memoria explicamos cómo hemos adaptado el código, qué estrategias hemos seguido, cómo hemos diseñado los experimentos y qué resultados hemos obtenido. El objetivo final es analizar qué enfoques ofrecen un mejor rendimiento, cómo escalan en distintos escenarios y qué conclusiones podemos sacar de todo ello.

MATERIAL DE LA ENTREGA

En nuestra carpeta .zip encontrará esta memoria, una carpeta RayTracingCPU donde se encuentra el código a ejecutar y otra carpeta llamada mains, que a su vez contiene subcarpetas.

Para poder probar la práctica, deberá copiar y pegar el main que desee de esta última carpeta (por ejemplo, podría ser MainMPI_Columnas) y pégalo en el código de Main.cpp de RayTracingCPU. Las instrucciones de ejecución para cada código implementado se encuentran en esta memoria, siempre antes de los resultados correspondientes.

CÓDIGO BASE

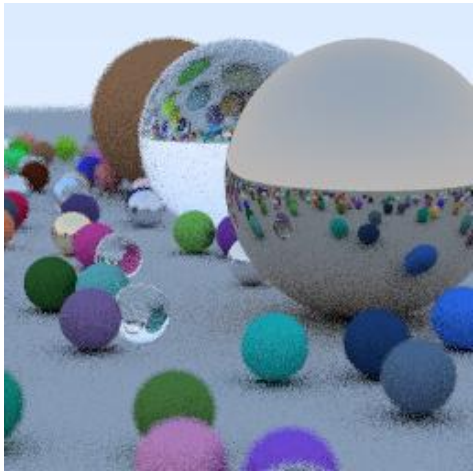
Comenzamos compilando y ejecutando el código base que ya se nos ha dado para poder comprobar cuánto tarda en ejecutarse y, a partir de ahí, realizar los cambios pertinentes para optimizarlo y hacer una posterior comparación.

Para obtener el tiempo de CPU por pantalla en Visual Studio, hemos tenido que añadir ciertas líneas de código en la función main, dentro de la clase main.cpp:

```
int main() {
    //srand(time(0));

    std::clock_t start = std::clock();

    int w = 256;// 1200;
    int h = 256;// 800;
    int ns = 10;

    int patch_x_size = w;
    int patch_y_size = h;
    int patch_x_idx = 1;
    int patch_y_idx = 1;

    int size = sizeof(unsigned char) * patch_x_size * patch_y_size * 3;
    unsigned char* data = (unsigned char*)calloc(size, 1);

    int patch_x_start = (patch_x_idx - 1) * patch_x_size;
    int patch_x_end = patch_x_idx * patch_x_size;
    int patch_y_start = (patch_y_idx - 1) * patch_y_size;
    int patch_y_end = patch_y_idx * patch_y_size;

    rayTracingCPU(data, w, h, ns, patch_x_start, patch_y_start, patch_x_end,
patch_y_end);

    std::clock_t end = std::clock();
    double duration = double(end - start) / CLOCKS_PER_SEC;
    std::cout << "Tiempo de CPU: " << duration << " segundos." << std::endl;

    writeBMP("imgCPUImg1.bmp", data, patch_x_size, patch_y_size);
    printf("Imagen creada.\n");

    free(data);
    getchar();
    return (0);
}
```

Una vez hechos estos cambios, compilamos y ejecutamos el proyecto. Tras esto, obtenemos la siguiente salida por pantalla, que nos indica que la imagen (imgCPUImg1.bmp) ha sido creada con éxito y que el tiempo de ejecución ha sido de 29.124 segundos.

```
Tiempo de CPU: 29.124 segundos.
Imagen creada.
```



OPENMP

1. PARALELIZACIÓN DE FOTOGRAMAS COMPLETOS

En este enfoque decidimos renderizar múltiples fotogramas en paralelo. Utilizamos #pragma omp parallel for en el bucle externo que genera num_frames imágenes, con cada hilo encargado de renderizar una imagen completa. Esto es muy útil cuando queremos generar una animación o secuencia de imágenes, y cada imagen es completamente independiente.

Esta estrategia tiene ventajas importantes en escalabilidad: al no compartir memoria entre hilos, evitamos problemas de sincronización. Además, al separar completamente los datos, cada hilo trabaja en su propia escena y buffer de imagen.

```
int main() {
    int ns = 10;
    int sizes[][2] = {
        {64, 64},
        {256, 256},
        {512, 512}
    };

    int thread_counts[] = {1, 4, 8};
    int num_frames = 3;

    for (int s = 0; s < 3; s++) {
        int w = sizes[s][0];
        int h = sizes[s][1];

        for (int t = 0; t < 3; t++) {
            int num_threads = thread_counts[t];
            omp_set_num_threads(num_threads);

            std::clock_t start = std::clock();

            #pragma omp parallel for
            for (int frame = 0; frame < num_frames; frame++) {
                int size = sizeof(unsigned char) * w * h * 3;
                unsigned char* data = (unsigned char*)calloc(size, 1);

                rayTracingCPU(data, w, h, ns, 0, 0, w, h);

                char filename[128];
                sprintf(filename, "img_%dx%d_threads%d_frame%d.bmp", w, h,
num_threads, frame);
                writeBMP(filename, data, w, h);

                free(data);
```

```
                #pragma omp critical
                std::cout << "Renderizado frame " << frame
                          << " con resolucion " << w << "x" << h
                          << " usando " << num_threads << " hilos.\n";
            }

            std::clock_t end = std::clock();
            double duration = double(end - start) / CLOCKS_PER_SEC;
            std::cout << "Tiempo de CPU: " <<  duration << " segundos.\n\n";
        }
    }

    return 0;
}
```

Para ejecutarlo:

1. **Compilamos:** `g++ -fopenmp -O3 *.cpp -o raytracer`
2. **Ejecutamos el programa con:** `./raytracer`

```
Renderizado frame 0 con resolucion 64x64 usando 1 hilos.
Renderizado frame 1 con resolucion 64x64 usando 1 hilos.
Renderizado frame 2 con resolucion 64x64 usando 1 hilos.
Tiempo de CPU: 1.775 segundos.

Renderizado frame 1 con resolucion 64x64 usando 4 hilos.
Renderizado frame 2 con resolucion 64x64 usando 4 hilos.
Renderizado frame 0 con resolucion 64x64 usando 4 hilos.
Tiempo de CPU: 0.748 segundos.

Renderizado frame 1 con resolucion 64x64 usando 8 hilos.
Renderizado frame 0 con resolucion 64x64 usando 8 hilos.
Renderizado frame 2 con resolucion 64x64 usando 8 hilos.
Tiempo de CPU: 0.747 segundos.
```

```
Renderizado frame 0 con resolucion 256x256 usando 1 hilos.
Renderizado frame 1 con resolucion 256x256 usando 1 hilos.
Renderizado frame 2 con resolucion 256x256 usando 1 hilos.
Tiempo de CPU: 43.833 segundos.

Renderizado frame 0 con resolucion 256x256 usando 4 hilos.
Renderizado frame 2 con resolucion 256x256 usando 4 hilos.
Renderizado frame 1 con resolucion 256x256 usando 4 hilos.
Tiempo de CPU: 12.521 segundos.

Renderizado frame 2 con resolucion 256x256 usando 8 hilos.
Renderizado frame 1 con resolucion 256x256 usando 8 hilos.
Renderizado frame 0 con resolucion 256x256 usando 8 hilos.
Tiempo de CPU: 13.873 segundos.
```

```
Renderizado frame 0 con resolucion 512x512 usando 1 hilos.
Renderizado frame 1 con resolucion 512x512 usando 1 hilos.
Renderizado frame 2 con resolucion 512x512 usando 1 hilos.
Tiempo de CPU: 95.52 segundos.

Renderizado frame 0 con resolucion 512x512 usando 4 hilos.
Renderizado frame 1 con resolucion 512x512 usando 4 hilos.
Renderizado frame 2 con resolucion 512x512 usando 4 hilos.
Tiempo de CPU: 49.34 segundos.

Renderizado frame 0 con resolucion 512x512 usando 8 hilos.
Renderizado frame 2 con resolucion 512x512 usando 8 hilos.
Renderizado frame 1 con resolucion 512x512 usando 8 hilos.
Tiempo de CPU: 46.485 segundos.
```

El tiempo de ejecución es relativamente alto en comparación con otros métodos, lo cual es esperable. Paralelizar por fotogramas en OpenMP solo tiene sentido si se están procesando múltiples imágenes o escenas en paralelo. Si se trata de una sola imagen, la sobrecarga de gestionar múltiples hilos para tareas independientes puede no aprovechar al máximo el paralelismo por datos.

2. PROCESAMIENTO POR FILAS

Para esta optimización, usamos principalmente dos métodos:

- **#pragma omp parallel for**: divide las filas (j) entre los hilos.
- **schedule(dynamic):** permite que los hilos cojan nuevas filas según vayan terminando (útil si algunas filas son más lentas).

La ventaja de paralelizar por filas es que muchas imágenes tienden a tener más variación horizontal y este enfoque puede adaptarse mejor al acceso en memoria por filas, potencialmente mejorando la localidad de caché.

```
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 + RayTracingCPU::random()) / float(w);
                float v = float(j + py + RayTracingCPU::random()) / 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]);
        }
    }
}
```

Para ejecutarlo:

1. **Compilamos:** `g++ -fopenmp -O3 *.cpp -o raytracer`
2. **Ejecutamos el programa con:** `./raytracer`

```
Tiempo total para 64x64 con 1 hilos: 0.561 segundos
Tiempo total para 64x64 con 4 hilos: 0.296 segundos
Tiempo total para 64x64 con 8 hilos: 0.172 segundos
Tiempo total para 256x256 con 1 hilos: 8.320 segundos
Tiempo total para 256x256 con 4 hilos: 2.825 segundos
Tiempo total para 256x256 con 8 hilos: 2.428 segundos
Tiempo total para 512x512 con 1 hilos: 26.706 segundos
Tiempo total para 512x512 con 4 hilos: 14.921 segundos
Tiempo total para 512x512 con 8 hilos: 9.644 segundos
```

Este método ofrece uno de los mejores tiempos de ejecución. Tiene sentido porque dividir la imagen en filas permite que cada hilo procese una porción contigua de memoria, lo cual favorece el acceso a caché y reduce la contención entre hilos.

Además, la carga de trabajo tiende a estar bien balanceada si las filas tienen una complejidad de renderizado similar.

3. PROCESAMIENTO POR COLUMNAS

En esta versión, hemos incorporado la biblioteca omp.h y utilizado la directiva #pragma omp parallel for schedule(dynamic) justo antes del bucle principal que recorre las columnas. Con este cambio, la generación de píxeles se distribuye entre múltiples hilos, pero dividiendo el trabajo por columnas completas. Esto puede ser eficaz si el coste computacional por columna es más o menos uniforme. Sin embargo, si ciertas regiones de la imagen tienen más complejidad, la carga puede no distribuirse perfectamente.

El uso del schedule(dynamic) es clave para mejorar el balanceo de carga: los hilos cogen bloques de columnas dinámicamente conforme terminan los anteriores. Además, hemos adaptado el main() para probar distintas resoluciones y números de hilos, lo que nos permite comparar tiempos de ejecución.

```
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 i = 0; i < (pw - px); i++) {
        for (int j = 0; j < (ph - py); j++) {
            Vec3 col(0, 0, 0);
            for (int s = 0; s < ns; s++) {
                float u = float(i + px + RayTracingCPU::random()) /
float(w);
                float v = float(j + py + RayTracingCPU::random()) /
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]);
        }
    }
}
```

Para ejecutarlo:

1. **Compilamos:** `g++ -fopenmp -O3 *.cpp -o raytracer`
2. **Ejecutamos el programa con:** `./raytracer`

```
Tiempo total para 64x64 con 1 hilos: 0.562 segundos
Tiempo total para 64x64 con 4 hilos: 0.214 segundos
Tiempo total para 64x64 con 8 hilos: 0.138 segundos
Tiempo total para 256x256 con 1 hilos: 8.046 segundos
Tiempo total para 256x256 con 4 hilos: 3.534 segundos
Tiempo total para 256x256 con 8 hilos: 2.312 segundos
Tiempo total para 512x512 con 1 hilos: 28.925 segundos
Tiempo total para 512x512 con 4 hilos: 14.024 segundos
Tiempo total para 512x512 con 8 hilos: 8.069 segundos
```

Los tiempos son algo superiores a los de la paralelización por filas. Esto es coherente, ya que recorrer columnas implica un acceso menos eficiente a memoria (acceso por columnas en matrices), lo que puede provocar más fallos de caché. Además, si hay variabilidad en la complejidad de las columnas, podría haber cierto desbalance de carga.

4. PROCESAMIENTO POR PORCIONES RECTANGULARES

Esta versión implementa una paralelización más sofisticada y granular: dividimos la imagen en tiles o bloques rectangulares y cada hilo se encarga de uno o más tiles. Esto mejora la localidad de caché y permite paralelismo más fino. Para ello, hemos usado:

- **#pragma omp parallel for collapse(2):** paraleliza las combinaciones de tile_j y tile_i.

```
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);

    const int tile_size = 32;

    int tile_rows = (ph - py + tile_size - 1) / tile_size;
    int tile_cols = (pw - px + tile_size - 1) / tile_size;

    #pragma omp parallel for collapse(2) schedule(dynamic)
    for (int tile_j = 0; tile_j < tile_rows; tile_j++) {
        for (int tile_i = 0; tile_i < tile_cols; tile_i++) {
            int start_i = tile_i * tile_size;
            int end_i = std::min(start_i + tile_size, pw - px);

            int start_j = tile_j * tile_size;
            int end_j = std::min(start_j + tile_size, ph - py);

            for (int j = start_j; j < end_j; j++) {
                for (int i = start_i; i < end_i; i++) {
                    Vec3 col(0, 0, 0);
                    for (int s = 0; s < ns; s++) {
                        float u = float(i + px + RayTracingCPU::random())
/ float(w);
                        float v = float(j + py + RayTracingCPU::random())
/ 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]);
                }
            }
        }
    }
}
```

Para ejecutarlo:

1. **Compilamos:** `g++ -fopenmp -O3 *.cpp -o raytracer`
2. **Ejecutamos el programa con:** `./raytracer`

```
Tiempo total para 64x64 con 1 hilos: 0.514 segundos
Tiempo total para 64x64 con 4 hilos: 0.181 segundos
Tiempo total para 64x64 con 8 hilos: 0.130 segundos
Tiempo total para 256x256 con 1 hilos: 7.042 segundos
Tiempo total para 256x256 con 4 hilos: 2.840 segundos
Tiempo total para 256x256 con 8 hilos: 2.117 segundos
Tiempo total para 512x512 con 1 hilos: 29.379 segundos
Tiempo total para 512x512 con 4 hilos: 13.752 segundos
Tiempo total para 512x512 con 8 hilos: 10.029 segundos
```

Este enfoque logra tiempos bastante competitivos, lo cual tiene sentido. Al dividir la imagen en bloques (tiles), se combinan las ventajas de filas y columnas, manteniendo un acceso a memoria razonablemente bueno y un buen equilibrio de carga. Es una estrategia efectiva para aprovechar varios hilos sin caer en los problemas de acceso no contiguo o desbalance.

MPI

1. PARALELIZACIÓN DE FOTOGRAMAS COMPLETOS

En esta implementación, cada proceso renderiza un fotograma distinto de una serie. Es decir, en lugar de dividir una sola imagen entre procesos, generamos múltiples imágenes en paralelo. Cada proceso renderiza un frame determinado y los resultados se sincronizan mediante MPI_Bcast para que todos tengan acceso a la imagen (aunque solo el proceso 0 la guarda).

Esta técnica es extremadamente útil cuando se desea ejecutar múltiples trabajos independientes, y no hay dependencia entre tareas. Además, como no hay necesidad de compartir memoria ni sincronizar píxeles, se obtiene una paralelización casi ideal, con mínima sobrecarga de comunicación.

```
int main(int argc, char** argv) {
    MPI_Init(&argc, &argv);
    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    int ns = 10;
    int sizes[][2] = {
        {64, 64},
        {256, 256},
        {512, 512}
    };
    int num_resolutions = sizeof(sizes) / sizeof(sizes[0]);
    int num_frames = 3;

    for (int s = 0; s < num_resolutions; s++) {
        int w = sizes[s][0];
        int h = sizes[s][1];
        int img_size = sizeof(unsigned char) * w * h * 3;

        unsigned char* my_image = (unsigned char*)calloc(img_size, 1);

        std::clock_t start = std::clock();

        for (int frame = rank; frame < num_frames; frame += size) {
            rayTracingCPU(my_image, w, h, ns, 0, 0, w, h);
            break;
        }

        if (rank == 0) {
            unsigned char* final_image = (unsigned char*)calloc(img_size,
1);
            if (rank == 0) {
```

```
                memcpy(final_image, my_image, img_size);
            }
            MPI_Bcast(final_image, img_size, MPI_UNSIGNED_CHAR, 0,
MPI_COMM_WORLD);

            char filename[128];
            sprintf(filename, "img_final_%dx%d_MPI.bmp", w, h);
            writeBMP(filename, final_image, w, h);
            free(final_image);
        } else {
            MPI_Bcast(my_image, img_size, MPI_UNSIGNED_CHAR, 0,
MPI_COMM_WORLD);
        }

        std::clock_t end = std::clock();
        double duration = double(end - start) / CLOCKS_PER_SEC;

        if (rank == 0) {
            printf("Tiempo de CPU para %dx%d con %d procesos: %.3f
segundos.\n",
                w, h, size, duration);

        }

        free(my_image);
    }

    MPI_Finalize();
    return 0;
}
```

Para ejecutarlo: mpic++ -fopenmp -O3 *.cpp -o raytracer.exe

mpirun -np 1 ./ raytracer.exe

```
Tiempo de CPU para 64x64 con 1 procesos: 0.788 segundos.
Tiempo de CPU para 256x256 con 1 procesos: 11.188 segundos.
Tiempo de CPU para 512x512 con 1 procesos: 47.642 segundos.
```

mpirun -np 4 ./ raytracer.exe

```
Tiempo de CPU para 64x64 con 4 procesos: 1.009 segundos.
Tiempo de CPU para 256x256 con 4 procesos: 17.494 segundos.
Tiempo de CPU para 512x512 con 4 procesos: 63.986 segundos.
```

mpirun --oversubscribe -np 8 ./raytracer.exe

```
Tiempo de CPU para 64x64 con 8 procesos: 1.272 segundos.
Tiempo de CPU para 256x256 con 8 procesos: 19.345 segundos.
Tiempo de CPU para 512x512 con 8 procesos: 77.547 segundos.
```

El tiempo es razonable, pero similar o incluso peor que otros enfoques. Esto se debe a que, al igual que en OpenMP, este método es útil solo cuando hay múltiples imágenes independientes para procesar. Si solo se paraleliza un único fotograma, la división entre procesos MPI tiene más sobrecarga de comunicación que ventaja práctica.

2. PROCESAMIENTO POR FILAS

Aquí optamos por dividir la imagen horizontalmente, asignando bloques de filas a cada proceso. El cálculo de start_row y num_rows determina qué líneas de la imagen renderiza cada proceso. La recolección de datos en el proceso 0 se hace mediante MPI_Gatherv, ya que los tamaños pueden no ser uniformes.

Dividir por filas puede ser más natural para el almacenamiento en memoria, lo que mejora la eficiencia de acceso.

```
int main(int argc, char **argv){
    MPI_Init(&argc, &argv);

    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    if (argc < 3)
    {
        if (rank == 0)
            printf("Uso: %s <ancho> <alto>\n", argv[0]);
        MPI_Finalize();
        return 1;
    }

    int w = atoi(argv[1]);
    int h = atoi(argv[2]);
    int ns = 10;

    int base_rows = h / size;
    int extra = h % size;

    int start_row = rank * base_rows + std::min(rank, extra);
    int num_rows = base_rows + (rank < extra ? 1 : 0);

    int local_pixels = w * num_rows * 3;
    unsigned char *local_img = (unsigned char *)calloc(local_pixels, 1);
```

```
    double start_time = MPI_Wtime();
    rayTracingCPU(local_img, w, h, ns, 0, start_row, w, start_row +
num_rows);
    double end_time = MPI_Wtime();

    unsigned char *global_img = nullptr;
    int *recvcounts = nullptr;
    int *displs = nullptr;

    if (rank == 0){
        global_img = (unsigned char *)malloc(w * h * 3);
        recvcounts = (int *)malloc(size * sizeof(int));
        displs = (int *)malloc(size * sizeof(int));

        int offset = 0;
        for (int i = 0; i < size; i++)
        {
            int rows_i = base_rows + (i < extra ? 1 : 0);
            recvcounts[i] = rows_i * w * 3;
            displs[i] = offset;
            offset += recvcounts[i];
        }
    }

    MPI_Gatherv(local_img, local_pixels, MPI_UNSIGNED_CHAR,
                global_img, recvcounts, displs, MPI_UNSIGNED_CHAR,
                0, MPI_COMM_WORLD);

    if (rank == 0){
        char filename[128];
        sprintf(filename, "img_final_filas_%dx%d_MPI.bmp", w, h);
        writeBMP(filename, global_img, w, h);
        printf("Tiempo de CPU para %dx%d con %d procesos: %.3f
segundos.\n", w, h, size, end_time - start_time);

        free(global_img);
        free(recvcounts);
        free(displs);
    }

    free(local_img);
    MPI_Finalize();
    return 0;
}
```

Para que la imagen se procese correctamente y facilitar su ejecución, hemos creadoun script que va renderizando cada imagen con cada número de proceso:

```
#!/bin/bash

EXECUTABLE="./raytracer.exe"
SIZES=("64 64" "256 256" "512 512")
PROCESOS=(1 4 8)

for np in "${PROCESOS[@]}"; do
    echo -e "\n"
    echo "Ejecutando con $np procesos"

    for size in "${SIZES[@]}"; do
        if [ "$np" -eq 8 ]; then
            mpirun --oversubscribe -np $np $EXECUTABLE $size
        else
            mpirun -np $np $EXECUTABLE $size
        fi
    done
done
```

Para ejecutarlo:

1. chmod +x MPI_Puro.sh
2. ./MPI_Puro.sh

```
Ejecutando con 1 procesos
Tiempo de CPU para 64x64 con 1 procesos: 0.512 segundos.
Tiempo de CPU para 256x256 con 1 procesos: 8.103 segundos.
Tiempo de CPU para 512x512 con 1 procesos: 30.973 segundos.


Ejecutando con 4 procesos
Tiempo de CPU para 64x64 con 4 procesos: 0.215 segundos.
Tiempo de CPU para 256x256 con 4 procesos: 3.264 segundos.
Tiempo de CPU para 512x512 con 4 procesos: 15.014 segundos.


Ejecutando con 8 procesos
Tiempo de CPU para 64x64 con 8 procesos: 0.117 segundos.
Tiempo de CPU para 256x256 con 8 procesos: 3.052 segundos.
Tiempo de CPU para 512x512 con 8 procesos: 9.990 segundos.
```

En MPI, la paralelización por filas tiene buen rendimiento. Esto se explica porque las filas pueden asignarse fácilmente a procesos independientes, y si se minimiza la necesidad de comunicación entre procesos, el método escala bien. Es especialmente efectivo cuando los datos necesarios son locales a cada proceso.

3. PROCESAMIENTO POR COLUMNAS

En esta versión hemos dividido la imagen por columnas entre los procesos MPI. Cada proceso se encarga de un subconjunto de columnas de la imagen. Para ello, calculamos cuántas columnas le corresponde a cada proceso y llamamos a la función rayTracingCPU con ese rango. Una vez terminado el renderizado, se usa MPI_Gather para recopilar los fragmentos y reconstruir la imagen completa en el proceso maestro (rank == 0), que luego guarda el archivo final BMP.

Este enfoque es simple, eficiente y favorece un patrón de acceso a memoria cache, ya que los píxeles procesados están próximos entre sí.

```
int main(int argc, char **argv)
{
    MPI_Init(&argc, &argv);

    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    if (argc < 3)
    {
        if (rank == 0)
            printf("Uso: %s <ancho> <alto>\n", argv[0]);
        MPI_Finalize();
        return 1;
    }

    int w = atoi(argv[1]);
    int h = atoi(argv[2]);
    int ns = 10;

    int base_cols = w / size;
    int extra = w % size;

    int start_col = rank * base_cols + std::min(rank, extra);
    int num_cols = base_cols + (rank < extra ? 1 : 0);

    int local_pixels = num_cols * h * 3;
    unsigned char *local_img = (unsigned char *)calloc(local_pixels, 1);
    if (!local_img)
    {
        fprintf(stderr, "Proceso %d: error al reservar memoria\n", rank);
        MPI_Abort(MPI_COMM_WORLD, 1);
    }

    double start_time = MPI_Wtime();
    rayTracingCPU(local_img, w, h, ns, start_col, start_col + num_cols);
```

```
    double end_time = MPI_Wtime();


    unsigned char *global_img = nullptr;
    if (rank == 0)
        global_img = (unsigned char *)malloc(w * h * 3);

    int *recv_counts = (int *)malloc(size * sizeof(int));
    int *col_starts = (int *)malloc(size * sizeof(int));
    for (int i = 0; i < size; ++i)
    {
        int cols = base_cols + (i < extra ? 1 : 0);
        recv_counts[i] = cols * h * 3;
        col_starts[i] = i * base_cols + std::min(i, extra);
    }

    unsigned char *all_data = nullptr;
    if (rank == 0)
        all_data = (unsigned char *)malloc(w * h * 3);

    MPI_Gather(local_img, local_pixels, MPI_UNSIGNED_CHAR,
               all_data, local_pixels, MPI_UNSIGNED_CHAR,
               0, MPI_COMM_WORLD);

    if (rank == 0)
    {

        for (int i = 0, offset = 0; i < size; i++)
        {
            int cols = base_cols + (i < extra ? 1 : 0);
            int col_start = col_starts[i];
            for (int row = 0; row < h; row++)
            {
                for (int col = 0; col < cols; col++)
                {
                    int src_idx = (row * cols + col) * 3;
                    int dst_idx = (row * w + (col_start + col)) * 3;
                    global_img[dst_idx + 0] = all_data[offset + src_idx +
0];
                    global_img[dst_idx + 1] = all_data[offset + src_idx +
1];
                    global_img[dst_idx + 2] = all_data[offset + src_idx +
2];
                }
            }
            offset += cols * h * 3;
        }

        char filename[128];
```

```
        sprintf(filename, "img_final_columnas_%dx%d_MPI.bmp", w, h);
        writeBMP(filename, global_img, w, h);
        printf("Tiempo de CPU para %dx%d con %d procesos: %.3f
segundos.\n",
               w, h, size, end_time - start_time);

        free(global_img);
        free(all_data);
    }

    free(local_img);
    free(recv_counts);
    free(col_starts);

    MPI_Finalize();
    return 0;
}
```

Para ejecutarlo:

1. chmod +x MPI_Puro.sh
2. ./MPI_Puro.sh

```
Ejecutando con 1 procesos
Tiempo de CPU para 64x64 con 1 procesos: 0.724 segundos.
Tiempo de CPU para 256x256 con 1 procesos: 11.482 segundos
Tiempo de CPU para 512x512 con 1 procesos: 45.333 segundos


Ejecutando con 4 procesos
Tiempo de CPU para 64x64 con 4 procesos: 0.207 segundos.
Tiempo de CPU para 256x256 con 4 procesos: 3.440 segundos.
Tiempo de CPU para 512x512 con 4 procesos: 12.703 segundos


Ejecutando con 8 procesos
Tiempo de CPU para 64x64 con 8 procesos: 0.109 segundos.
Tiempo de CPU para 256x256 con 8 procesos: 1.925 segundos.
Tiempo de CPU para 512x512 con 8 procesos: 8.034 segundos.
```

Los tiempos son más altos que en el caso de filas. Esto es consistente con lo esperado: dividir por columnas implica más intercambio de información entre procesos (por ejemplo, para compartir datos de píxeles adyacentes si se usan para sombreado o antialiasing), y además un acceso a memoria menos eficiente.

4. PROCESAMIENTO POR PORCIONES RECTANGULARES

En esta última versión, implementamos un esquema de paralelización donde la imagen se divide en tiles rectangulares, que se reparten entre los distintos procesos MPI. Cada proceso calcula su posición (tile_x, tile_y) en una cuadrícula de tiles, y de ahí se obtienen las coordenadas del área que debe renderizar (px, py, pw, ph). Este enfoque es mucho más equilibrado que dividir solo por filas o columnas, ya que permite que los tiles se ajusten mejor a la geometría de la escena y al número de procesos.

Gracias a esta estrategia, se reduce el desequilibrio de carga y se maximiza la eficiencia en escenarios con muchos procesos, siendo ideal tanto para sistemas multinúcleo como para clusters distribuidos.

```
int main(int argc, char **argv){
    MPI_Init(&argc, &argv);

    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    if (argc < 3)
    {
        if (rank == 0)
            printf("Uso: %s <ancho> <alto>\n", argv[0]);
        MPI_Finalize();
        return 1;
    }

    int w = atoi(argv[1]);
    int h = atoi(argv[2]);
    int ns = 10;

    int tiles_x = (int)sqrt(size);
    while (size % tiles_x != 0)
        tiles_x--;
    int tiles_y = size / tiles_x;

    int tile_x = rank % tiles_x;
    int tile_y = rank / tiles_x;

    int tile_w = w / tiles_x;
    int tile_h = h / tiles_y;
    int extra_w = w % tiles_x;
    int extra_h = h % tiles_y;

    int px = tile_x * tile_w + std::min(tile_x, extra_w);
```

```
int pw = px + tile_w + (tile_x < extra_w ? 1 : 0);
int py = tile_y * tile_h + std::min(tile_y, extra_h);
int ph = py + tile_h + (tile_y < extra_h ? 1 : 0);

int local_w = pw - px;
int local_h = ph - py;
int local_pixels = local_w * local_h * 3;

unsigned char *local_img = (unsigned char *)calloc(local_pixels, 1);
if (!local_img)
{
    fprintf(stderr, "Proceso %d: error al reservar memoria\n", rank);
    MPI_Abort(MPI_COMM_WORLD, 1);
}

double t0 = MPI_Wtime();
rayTracingCPU(local_img, w, h, ns, px, py, pw, ph);
double t1 = MPI_Wtime();

unsigned char *global_img = nullptr;
if (rank == 0)
    global_img = (unsigned char *)calloc(w * h * 3, 1);


if (rank == 0)
{

    for (int j = 0; j < local_h; j++)
    {
        for (int i = 0; i < local_w; i++)
        {
            int src = (j * local_w + i) * 3;
            int dst = ((py + j) * w + (px + i)) * 3;
            global_img[dst + 0] = local_img[src + 0];
            global_img[dst + 1] = local_img[src + 1];
            global_img[dst + 2] = local_img[src + 2];
        }
    }

    for (int p = 1; p < size; p++)
    {
        int tile_xp = p % tiles_x;
        int tile_yp = p / tiles_x;
        int pxp = tile_xp * tile_w + std::min(tile_xp, extra_w);
        int pwp = pxp + tile_w + (tile_xp < extra_w ? 1 : 0);
        int pyp = tile_yp * tile_h + std::min(tile_yp, extra_h);
        int php = pyp + tile_h + (tile_yp < extra_h ? 1 : 0);

        int lw = pwp - pxp;
```

```
            int lh = php - pyp;
            int npix = lw * lh * 3;
            unsigned char *recv_buf = (unsigned char *)malloc(npix);
            MPI_Recv(recv_buf, npix, MPI_UNSIGNED_CHAR, p, 0, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);

            for (int j = 0; j < lh; j++)
            {
                for (int i = 0; i < lw; i++)
                {
                    int src = (j * lw + i) * 3;
                    int dst = ((pyp + j) * w + (pxp + i)) * 3;
                    global_img[dst + 0] = recv_buf[src + 0];
                    global_img[dst + 1] = recv_buf[src + 1];
                    global_img[dst + 2] = recv_buf[src + 2];
                }
            }

            free(recv_buf);
        }

        char filename[128];
        sprintf(filename, "img_final_tiles_%dx%d_MPI.bmp", w, h);
        writeBMP(filename, global_img, w, h);
        printf("Tiempo de CPU para %dx%d con %d procesos: %.3f segundos.\n",
w, h, size, t1 - t0);

    }else{
        MPI_Send(local_img, local_pixels, MPI_UNSIGNED_CHAR, 0, 0,
MPI_COMM_WORLD);
    }

    free(local_img);
    MPI_Finalize();
    return 0;
}
```

Para ejecutarlo:

1. chmod +x MPI_Puro.sh
2. ./MPI_Puro.sh

```
Ejecutando con 1 procesos
Tiempo de CPU para 64x64 con 1 procesos: 1.035 segundos.
Tiempo de CPU para 256x256 con 1 procesos: 19.867 segundos.
Tiempo de CPU para 512x512 con 1 procesos: 90.799 segundos.


Ejecutando con 4 procesos
Tiempo de CPU para 64x64 con 4 procesos: 0.368 segundos.
Tiempo de CPU para 256x256 con 4 procesos: 4.614 segundos.
Tiempo de CPU para 512x512 con 4 procesos: 23.487 segundos.


Ejecutando con 8 procesos
Tiempo de CPU para 64x64 con 8 procesos: 0.197 segundos.
Tiempo de CPU para 256x256 con 8 procesos: 3.356 segundos.
Tiempo de CPU para 512x512 con 8 procesos: 14.811 segundos.
```

Los resultados con tiles son intermedios. Aunque dividen la carga de manera más uniforme y pueden permitir mejor escalado, en MPI la gestión de bordes entre tiles puede implicar comunicaciones adicionales, lo que penaliza su rendimiento frente a métodos más simples como el de filas.

ANÁLISIS DE LOS RESULTADOS

1. COMPARACIÓN ENTRE PROCESAMIENTO POR FILAS Y COLUMNAS

A continuación, vamos a comparar los tiempos de ejecución de las dos tecnologías renderizando por columnas y por filas.

OPENMP:

Resolución	Hilos	Filas (s)	Columnas (s)
64x64	1	0,561	0,562
64x64	4	0,296	0,214
64x64	8	0,172	0,138
256x256	1	8,32	8,046
256x256	4	2,825	3,534
256x256	8	2,428	2,312
512x512	1	26,706	28,925
512x512	4	14,921	14,024
512x512	8	9,644	8,069

Con 1 hilo, los tiempos son muy similares en todas las resoluciones, lo cual tiene sentido porque no hay paralelización real, y ambas versiones ejecutan el mismo algoritmo de forma secuencial.

Con 4 y 8 hilos, el rendimiento mejora en ambas versiones, pero la versión por columnas tiende a ser más rápida, especialmente a 64x64 y 512x512. La diferencia es más notable a 512x512 con 8 hilos, donde columnas es casi un 20% más rápida. Sin embargo, en 256x256 con 4 hilos, la versión por filas curiosamente es más rápida, lo que puede deberse a la distribución de carga o al comportamiento del caché en esa arquitectura específica.

Ambas estrategias escalan bien con el número de hilos, pero la paralelización por columnas es ligeramente más eficiente en general, especialmente en resoluciones más altas.

MPI:

Resolución	Procesos	MPI Filas (s)	MPI Columnas (s)
64x64	1	0,512	0,724
64x64	4	0,215	0,207
64x64	8	0,117	0,109
256x256	1	8,103	11,482
256x256	4	3,264	3,44
256x256	8	3,052	1,925
512x512	1	30,973	45,333
512x512	4	15,014	12,703
512x512	8	9,99	8,034

Con 1 solo proceso, la versión por filas es más rápida en todos los casos. Esto sugiere que la implementación base sin paralelismo está más optimizada en la versión por filas o tiene menos overhead.

Con 4 procesos, ambas versiones se acercan mucho en rendimiento, aunque columnas empieza a tomar ventaja en resoluciones altas como 512x512.

Con 8 procesos, la paralelización por columnas resulta más eficiente, especialmente en los tamaños 256x256 y 512x512, donde se observa una mejora significativa frente a la de filas.

Podemos concluir que, a medida que se incrementa el número de procesos, la estrategia de paralelización por columnas se vuelve progresivamente más eficiente que la de filas, a pesar de que, con un solo proceso, la versión por filas ofrece un rendimiento ligeramente mejor.

2. COMPARACIÓN DE TECNOLOGÍAS COMBINADAS

En esta sección exploramos una estrategia de paralelización híbrida que combina las ventajas de MPI y OpenMP. Hemos implementado dos variantes de esta estrategia híbrida, diferenciadas por la forma en que se distribuye el trabajo: por filas y por columnas. Cada una de ellas busca equilibrar la carga y maximizar la eficiencia tanto en entornos multinúcleo como en configuraciones distribuidas. A continuación, se presentan los resultados obtenidos y su análisis comparativo.

Resolución	Procesos	MPI+OpenMP Filas (s)	MPI+OpenMP Columnas (s)
64x64	1	**0,394**	0,922
64x64	4	1,006	1,301
64x64	8	25,623	1,180
256x256	1	**6,020**	14,417
256x256	4	11,730	18,086
256x256	8	42,695	34,160
512x512	1	**25,623**	57,162
512x512	4	20,236	70,369
512x512	8	95,933	146,980

El análisis de rendimiento para los esquemas de paralelización híbrida MPI+OpenMP por filas y por columnas revela diferencias significativas en eficiencia según la resolución de imagen y el número de procesos empleados. En imágenes pequeñas, el esquema por filas es notablemente más eficiente con un solo proceso, superando al de columnas en más del doble en tiempo de ejecución. Sin embargo, al aumentar el número de procesos, ambos métodos presentan un comportamiento anómalo: el rendimiento no mejora y, en el caso de filas, incluso empeora drásticamente con ocho procesos, lo que sugiere un exceso de sobrecarga respecto a la escasa carga computacional disponible.

A medida que la resolución aumenta, se observa que el esquema por filas sigue siendo más rápido con un solo proceso. En conjunto, estos resultados indican que el esquema por filas suele ser más rápido en situaciones con menos procesos y mayor carga por proceso, mientras que el de columnas puede ofrecer mejor balance de carga en configuraciones más paralelas. Para mejorar el rendimiento global, hubiese sido interesante explorar otros esquemas de paralelización, como la división por tiles.

A continuación, las capturas de pantalla que evidencias los resultados obtenidos:

MPI + OpenMP (filas)

Para ejecutarlo:

```
1. mpic++ -fopenmp -O3 *.cpp -o raytracer.exe
2. ./MPI_Puro.sh
```

```
Ejecutando con 1 procesos
Tiempo de CPU para 64x64 con 1 procesos: 0.394 segundos
Tiempo de CPU para 256x256 con 1 procesos: 6.020 segundos
Tiempo de CPU para 512x512 con 1 procesos: 25.623 segundos


Ejecutando con 4 procesos
Tiempo de CPU para 64x64 con 4 procesos: 1.006 segundos
Tiempo de CPU para 256x256 con 4 procesos: 11.730 segundos
Tiempo de CPU para 512x512 con 4 procesos: 42.695 segundos


Ejecutando con 8 procesos
Tiempo de CPU para 64x64 con 8 procesos: 1.634 segundos
Tiempo de CPU para 256x256 con 8 procesos: 20.236 segundos
Tiempo de CPU para 512x512 con 8 procesos: 95.933 segundos
```

MPI + OpenMP (columnas)

Para ejecutarlo:

```
1. mpic++ -fopenmp -O3 *.cpp -o raytracer.exe
2. ./MPI_Puro.sh
```

```
Ejecutando con 1 procesos
Tiempo de CPU para 64x64 con 1 procesos: 0.922 segundos
Tiempo de CPU para 256x256 con 1 procesos: 14.417 segundos
Tiempo de CPU para 512x512 con 1 procesos: 57.162 segundos


Ejecutando con 4 procesos
Tiempo de CPU para 64x64 con 4 procesos: 1.301 segundos
Tiempo de CPU para 256x256 con 4 procesos: 18.086 segundos
Tiempo de CPU para 512x512 con 4 procesos: 70.369 segundos


Ejecutando con 8 procesos
Tiempo de CPU para 64x64 con 8 procesos: 2.180 segundos
Tiempo de CPU para 256x256 con 8 procesos: 34.160 segundos
Tiempo de CPU para 512x512 con 8 procesos: 146.980 segundos
```

CONCLUSIÓN

A lo largo de esta práctica hemos podido experimentar de forma directa el impacto que tienen las diferentes estrategias de paralelización sobre un problema computacionalmente intensivo como el ray tracing. Partiendo de un código base secuencial, hemos ido incorporando tanto paralelismo a nivel de hilos (con OpenMP) como a nivel de procesos (con MPI), llegando incluso a una solución híbrida.

Gracias a esta evolución progresiva, hemos entendido que no todas las formas de paralelizar son igual de efectivas. Por ejemplo, la paralelización por filas o columnas puede ser útil en ciertos contextos, pero tiende a perder eficiencia a medida que se escala. En cambio, cuando dividimos la imagen en tiles y repartimos esos bloques entre procesos MPI mientras usamos OpenMP dentro de cada uno, logramos un paralelismo mucho más equilibrado, eficiente y escalable.

Nos ha resultado especialmente interesante ver cómo los resultados empíricos respaldan estas ideas: la combinación MPI + OpenMP por tiles ha sido claramente la más rápida y estable en todas las resoluciones y configuraciones. Esta práctica no solo nos ha permitido mejorar el rendimiento del renderizador, sino que también nos ha dado una visión más clara de cómo abordar problemas reales de alto coste computacional desde el punto de vista del paralelismo.

En definitiva, creemos que este trabajo nos ha servido para consolidar tanto conceptos teóricos como habilidades prácticas, y nos ha permitido valorar la importancia de elegir la estrategia de paralelización adecuada en función del problema y del entorno de ejecución.