Renderizado de fractales con MPI

Trabajo Integrador Final: Programación Paralela y Distribuida

Autores

- Franco Yudica
- Martín Farrés

Abstract: Este trabajo presenta el desarrollo de una implementación paralela de renderizado de fractales bidimensionales, mostrando sus diferencias con la versión secuencial. Además se detalla sobre los experimentos realizados para determinar el rendimiento de la aplicación, en conjutno con conclusiones sobre tales resultados.

Keywords: Fractals, Mandelbrot Set, Julia Set, Programación paralela, MPI, Números Complejos, Sistemas Dinámicos, Gráficos de Computadora, Evaluación de Resultados.

1 Introducción

El renderizado eficiente de imágenes fractales representa un reto significativo en el ámbito de la computación gráfica, debido a la complejidad matemática involucrada y la alta demanda computacional que implica su visualización detallada. Los fractales, como el conjunto de Mandelbrot o los conjuntos de Julia, se caracterizan por su estructura auto-similar e infinita complejidad, lo que requiere un gran número de cálculos por píxel para generar imágenes precisas y atractivas.

Este informe presenta el desarrollo y análisis de un sistema de renderizado de fractales, implementado utilizando técnicas de paralelización y optimización computacional. El objetivo principal es reducir el tiempo de renderizado, aprovechando al máximo los recursos de hardware disponibles.

En primer lugar, se describe el proceso secuencial de renderizado de fractales, abordando las ecuaciones iterativas involucradas y los criterios de escape utilizados para determinar la convergencia o divergencia de cada punto. Luego, se introduce la versión paralela del sistema, explicando las estrategias utilizadas para distribuir el trabajo entre múltiples hilos o nodos, y las optimizaciones aplicadas para mejorar la eficiencia.

A continuación, se presentan los experimentos realizados para evaluar el rendimiento del sistema, analizando métricas como el speedup, la eficiencia y la escalabilidad en diferentes configuraciones. Se incluyen también comparaciones visuales y tiempos de renderizado, acompañados de una discusión sobre los resultados.

Finalmente, se exponen las conclusiones obtenidas a partir del análisis, resaltando las ventajas y limitaciones del enfoque propuesto. Se sugieren además posibles líneas de mejora, incluyendo la exploración de técnicas avanzadas como el uso de GPUs, algoritmos adaptativos de muestreo y representación dinámica en tiempo real.

2 Marco teórico

En esta sección se desarrolla el marco teórico fundamental de los fractales, realizando una breve introducción a la dinámica compleja [7], conjuntos de Julia y Fatou [9], fractales de Julia y Mandelbrot [2], y las técnicas de coloreo de fractales utilizadas en este proyecto.

2.1 Dinámica compleja

La dinámica compleja, también conocida como dinámica holomorfa [7], es una rama de la matemática que estudia el comportamiento de los sistemas dinámicos [8] obtenidos mediante la iteración de funciones analíticas en el plano complejo. A diferencia de los sistemas dinámicos en el plano real, la estructura adicional que proporciona la analiticidad en los números complejos introduce una rica variedad de comportamientos geométricos y topológicos que han sido ampliamente estudiados desde principios del siglo XX.

Una función compleja $f : C \to C$ se itera generando una secuencia de funciones

$$f^n(z) = f(f^{n-1}(z))$$

donde $n \in \mathbb{N}$. El objeto de estudio principal es el conjunto de órbita de un punto z, definido como la secuencia de sus imágenes sucesivas bajo iteraciones de f. Esta órbita puede exhibir distintos comportamientos: puede tender al infinito, converger a un punto fijo o seguir una trayectoria caótica.

Un ejemplo clásico y fundamental es la función cuadrática:

$$f(z) = z^2$$

Aunque simple, esta función exhibe una variedad de comportamientos interesantes dependiendo del punto de partida z_0

Por ejemplo:

$$Si|z_0| < 1, entonces f^n(z_0) \to 0$$

$$Si|z_0| > 1, entonces f^n(z_0) \to \infty$$

$$Si|z_0| = 1, entonces f^n(z_0) \to 1$$

2.2 Conjuntos de Julia y Fatou

El estudio de estas órbitas lleva a la clasificación del plano complejo en dos regiones fundamentales:

- El conjunto de Fatou, donde las órbitas tienen un comportamiento estable bajo pequeñas perturbaciones iniciales.
- El conjunto de Julia, que contiene puntos con un comportamiento altamente sensible a las condiciones iniciales, caractarizado por su complejiad fractal.

Estos conjuntos son complementarios y su frontera compartida representa el límite entre estabilidad y caos. En el caso de $f(z) = z^2$, el conjunto de Julia es el círculo unitario $|z| = 1$, mientras que el conjunto de Fatou está formado por el interior y el exterior de tal círculo.

2.3 Fractal Julia

Los conjuntos de Julia se generan utilizando números complejos. Estos poseen dos componentes, real e imaginaria, y pueden representarse como puntos en un plano bidimensional, lo que permite renderizar el fractal sobre una imagen 2D. Para cada píxel de la imagen, su coordenada (x, y) en el plano se utiliza como entrada en una función recursiva.

El fractal de Julia es un ejemplo clásico de fractal de tiempo de escape, lo que significa que el interés está en determinar si, tras aplicar repetidamente una función compleja, el valor resultante tiende al infinito o no.

La función recursiva que define el conjunto de Julia es:

$$z_{n+1} = z_n^2 + c$$

donde:

- z_n es el valor complejo en la iteración n,
- z_0 es la posición del píxel en el plano complejo, escrita como $z_0 = p_x + p_y i$
- c es una constante compleja, $c = c_x + c_y i$, que permanece fija durante toda la generación del fractal.

Está demostrado que si $\mid z_n \mid > 2$, entonces la sucesión diverge (tiende a infinito). En este contexto, el valor 2 se denomina bailout, y es el umbral utilizado para determinar la divergencia. [5].

2.4 Fractal Mandelbrot

El fractal de Mandelbrot es muy similar al de Julia, ya que también se trata de un fractal de tiempo de escape. La principal diferencia radica en la función recursiva y en los valores iniciales utilizados.

La función que define al conjunto de Mandelbrot es:

$$z_{n+1} = z_n^2 + p$$

donde:

- p es la posición del píxel en el plano complejo, de la forma $p = p_x + p_y i$,
- z_n inicia en 0, es decir, $z_0 = 0$, y se itera añadiendo el valor constante p en cada paso.

Al igual que en el caso del fractal de Julia, el criterio de escape se basa en si $|z_n| > 2$, utilizando el mismo valor de bailout. [2], [5].

2.5 Coloreo de fractales

Existen distintos métodos para colorear fractales, siendo el más básico el blanco y negro. En este esquema, los píxeles cuya posición, al ser utilizada como punto de partida en la iteración del fractal, tienden al infinito, se colorean de blanco. Por el contrario, aquellos que no divergen se colorean de negro.

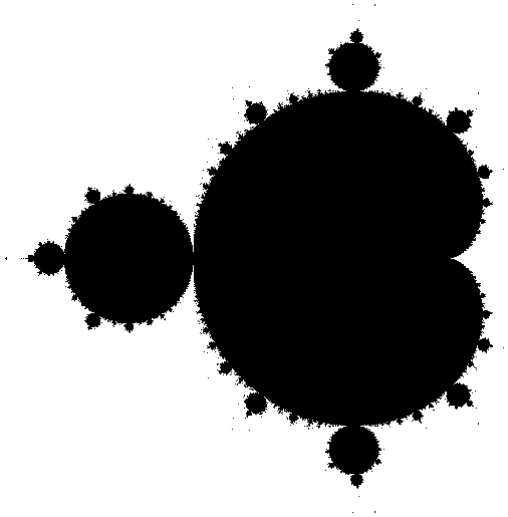


Figura 1: Representación en blanco y negro del conjunto de Mandelbrot.

Sin embargo, este método binario puede resultar limitado para visualizar la complejidad del sistema dinámico. Por ello, se utilizan técnicas más avanzadas como el coloreo por tiempo de escape (escape time coloring), donde se asignan colores según la cantidad de iteraciones que tarda un punto en escapar de un cierto radio. Esto permite generar imágenes con ricos gradientes de color que reflejan la velocidad de divergencia y destacan la estructura del borde del conjunto. [5].

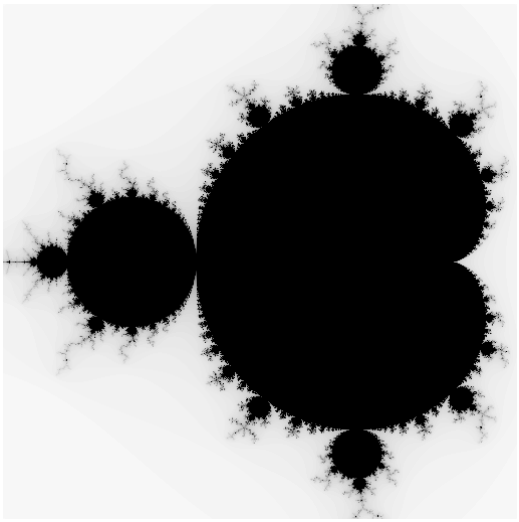


Figura 2: Representación en escala de grises del conjunto de Mandelbrot.

Pero también es posible mapear el número de iteraciones a una paleta de colores. Nótese que los puntos pertenecientes al conjunto de Mandelbrot toman un color uniforme, ya que alcanzan el número máximo de iteraciones sin divergir.

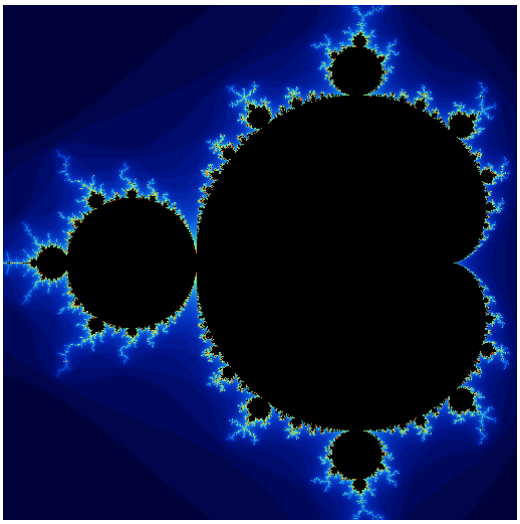


Figura 3: Mapeo de iteraciones a paleta de colores del conjunto de Mandelbrot.

En la figura 3 se pueden observar resultados mucho más interesantes. Al mirar con detalle, se aprecian transiciones abruptas entre los colores, un efecto comúnmente denominado *banding* en computación gráfica. Esto se debe a que el mapeo del color se realiza únicamente en función de la cantidad de iteraciones, que es un valor discreto.

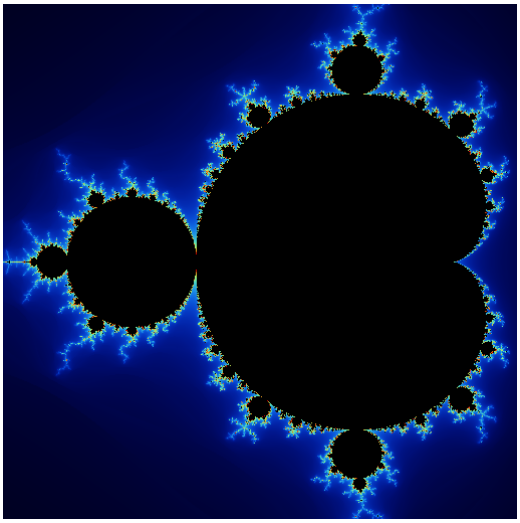


Figura 4: Mapeo de iteraciones a colores con transición suave.

Para renderizar la figura 4, se ha utilizado el número de iteraciones, en conjunto con $|z_n|$, lo cual permite realizar un mapeo continuo a la paleta de colores, eliminando el efecto de *banding*. El desarrollo matemático se encuentra en la referencia [5].

3 Desarrollo

El algoritmo para el desarrollo de dichas imagenes se resume en, la obtencion del color correspondiente segun la formula de fractal aplicada a cada pixel de la imagen. Para la cual se desarrollo una funcion de renderizado **render_block**;

```
def render_block(x_inicial, y_inicial, ancho, alto):

    for j in range(alto):

        for i in range(ancho):

            pixel_x = x_inicial + i
            pixel_y = y_inicial + j

            # Calcular coordenadas normalizadas [-1.0, 1.0]
            nx, ny = calcular_ndc(pixel_x, pixel_y)

            # Convertir (nx, ny) a coordenadas del mundo (wx, wy) usando la camara
            wx, wy = camera.to_world(nx, ny)

            # Evaluar el fractal, obteniendo las iteraciones normalizadas [0.0, 1.0]
```

```
t = fractal(wx, wy)
r, g, b = color(t)

# Guardar el color en el buffer
guardar_buffer(r, g, b, pixel_x, pixel_y)
```

La funcion se encarga del procesamiento de la imagen fractal. Para cada píxel, el algoritmo toma varias muestras con un pequeño desplazamiento aleatorio (antialiasing) para disminuir el ruido obtenido en la imagen final. Luego cada muestra se transforma en coordenadas del mundo con una cámara virtual. Las mismas son evaluadas por **funcion_fractal** para obtener un valor que, luego es transformados a valores rgb utilizando la funcion, **funcion_color**. Finalmente, dichos valores se promedian obteniendo así el color para cada pixel de la imagen.

Para el desarrollo del problema se hicieron dos versiones, secuencial y paralalelo, para observar las diferencias de ambas en terminos de tiempo y costo computacional.

3.1 Secuencial

El código secuencial implementa un enfoque lineal para resolver el problema de renderizado. El algoritmo recibe varios parámetros de configuración, tales como el ancho y alto de la imagen, la posición y el nivel de zoom de la cámara, el tipo de fractal a calcular, entre otros. A partir de esta información, se invoca directamente la función de renderizado, y una vez finalizado el proceso, se guarda la imagen resultante.

El procesamiento es completamente secuencial: cada píxel de la imagen es calculado uno por uno, sin ningún tipo de paralelismo o concurrencia. Esto lo convierte en una implementación sencilla pero poco eficiente para imágenes de alta resolución o fractales complejos.

3.2 Paralelo

Dado que el renderizado de fractales es una tarea altamente demandante en términos computacionales, se exploró una versión paralela del algoritmo con el objetivo de reducir significativamente el tiempo de ejecución.

3.3 Identificación del paralelismo

El renderizado de fractales es un problema naturalmente paralelizable. Cada píxel de la imagen puede calcularse de forma independiente, ya que no requiere información de los píxeles vecinos ni de ningún otro elemento de la imagen. Esta independencia permite dividir la carga de trabajo entre múltiples procesos o hilos de ejecución sin necesidad de sincronización compleja, lo que lo convierte en un caso ideal para aplicar técnicas de paralelismo.

3.4 Secciones secuenciales y paralelizables

El algoritmo presenta tanto secciones secuenciales como paralelizables.

Las secciones secuenciales incluyen la etapa de inicialización, en la cual se configura el entorno de ejecución, se inicializa la biblioteca MPI y se definen las tareas o bloques de la imagen que serán distribuidos a los procesos workers. La etapa de finalización también es secuencial, ya que implica recopilar los bloques renderizados, ensamblar la imagen final y guardarla en disco. Estas etapas requieren acceso centralizado a ciertos recursos y coordinación general, lo que limita su paralelización.

Por otro lado, la sección paralelizable corresponde al renderizado de los bloques de imagen. Dado que cada bloque puede ser procesado de forma independiente, esta etapa se distribuye entre los distintos procesos para acelerar significativamente el tiempo total de ejecución.

3.5 Estrategia de descomposición

El renderizado de fractales representa un caso típico para aplicar una estrategia de descomposición de dominio. Esta técnica consiste en subdividir el dominio del problema, en este caso, la imagen a renderizar, en múltiples subregiones independientes. Concretamente, la imagen se divide en bloques rectangulares, cada uno definido por una tupla de la forma (x, y, ancho, alto), que indica la posición y dimensiones del bloque dentro de la imagen global.

3.6 Modelo de algoritmo paralelo

Se adopta un modelo master-worker. En este esquema, el nodo master se encarga de dividir la imagen en bloques y distribuir el trabajo entre los distintos procesos workers. Además, coordina las solicitudes de tareas, asigna bloques disponibles de forma dinámica y recibe los resultados procesados por cada worker.

Una vez que los bloques son completados, el master se encarga de ensamblar los resultados parciales en un búfer central, que luego se utiliza para generar la imagen final.

3.6.1 Asignación de tareas y balanceo de carga

Esta aproximación inicial ya demuestra mejoras en el tiempo total de cómputo, aunque revela un desbalanceo de carga cuando algunos bloques requieren más cómputo que otros, siendo esta una característica común en el renderizado de fractales, dejando procesos inactivos mientras otros siguen trabajando.

Por ejemplo, si las ocho tareas tienen duraciones (en ms) [10, 10, 10, 10, 20, 30, 40, 50] y se reparten estáticamente en dos nodos:

- **Nodo1** recibe las cuatro primeras tareas: $10 + 10 + 10 + 10 = 40$ ms de trabajo y permanece inactivo los 100 ms restantes.
- **Nodo2** recibe las cuatro últimas: $20 + 30 + 40 + 50 = 140$ ms, completando todo el render en 140 ms.

Para resolver este desbalanceo se implementó un balanceo de carga dinámico basado en asignación bajo demanda. En lugar de asignar bloques estáticamente, el master mantiene una cola de tareas y cada worker solicita un nuevo bloque tan pronto como finaliza el anterior. De este modo, el tiempo

de inactividad de los procesos se reduce significativamente y se optimiza el uso de los recursos de cómputo. Con la misma serie de duraciones y balanceo dinámico:

- **Nodo1** procesa: $10 + 10 + 10 + 10 + 50 = 90$ ms.
- **Nodo2** procesa: $20 + 30 + 40 = 90$ ms,

logrando que ambos nodos terminen en 90 ms y minimizando los períodos ociosos.

3.6.2 Sincronismo - Asincronismo

El sistema implementado utiliza un modelo de comunicación sincrónico. Los mensajes intercambiados entre el nodo master y los workers se gestionan mediante llamadas bloqueantes, donde tanto el emisor como el receptor deben estar sincronizados para que la operación de envío o recepción se complete.

Este enfoque simplifica la lógica de coordinación y garantiza un flujo de ejecución controlado, aunque puede introducir ciertos periodos de espera innecesarios si alguno de los procesos se encuentra inactivo temporalmente.

En este contexto, los beneficios de un modelo asincrónico serían mínimos, ya que el tiempo de comunicación es muy bajo en comparación con el tiempo de cómputo, siendo este último dominado por el proceso de renderizado.

3.7 Pseudocódigo de `master`

```
def master(num_procs, settings):
    # Crear buffer de imagen
    imagen = crear_buffer_imagen(settings)

    # Dividir imagen en bloques de trabajo
    worker_tasks = dividir_en_tareas(imagen)

    sent = 0
    done = 0

    while done < len(worker_tasks):

        mensaje, origen = mpi_esperar_mensaje()

        if mensaje.tag == "REQUEST":
            if sent < len(worker_tasks):
                tarea = worker_tasks[sent]
                mpi_enviar(tarea, destino=origen, tag="TASK")
                sent += 1
            else:
                mpi_enviar(None, destino=origen, tag="TERMINATE")
```

```
        elif mensaje.tag == "RESULT":
            bloque = recibir_mensaje(fuente=origen)
            copiar_bloque_en_buffer(imagen, bloque)
            done += 1

    # Enviar TERMINATE a todos los workers
    for rank in range(1, num_procs):
        mpi_broadcast(tag="TERMINATE")

    # Guardar imagen final
    guardar_imagen(imagen)
```

La función master comienza reservando un búfer para la imagen completa y dividiendo el área de renderizado en bloques de tamaño fijo, que se almacenan en una lista de tareas. A continuación, mantiene dos contadores: uno para las tareas enviadas y otro para las tareas completadas. En un bucle principal, espera mensajes de los masteres; cuando recibe una petición de trabajo, comprueba si aún quedan bloques sin asignar y, en caso afirmativo, envía el siguiente bloque, o bien envía una señal de terminación si ya no hay más. Cuando recibe el resultado de un bloque, copia los píxeles de ese fragmento en la posición correspondiente del búfer global y actualiza el contador de tareas completadas. Este proceso se repite hasta que todas las tareas han sido procesadas, momento en el cual el master envía una señal de terminación a cada worker, detiene el temporizador y muestra el tiempo total de cómputo. Finalmente, invoca al manejador de salida para guardar el búfer como imagen.

3.8 Pseudocódigo de `worker`

```
def worker(rank, config_imagen, config_fractal, camara):
    while True:

        # Worker listo, solicita tarea
        enviar_mensaje(destino=master, tag="REQUEST")
        mensaje = recibir_mensaje(master)

        if mensaje.tag == "TASK":

            # Recibe el bloque a renderizar
            x, y, ancho, alto = recibir_tarea(fuente=master)
            buffer = crear_buffer(ancho * alto * 3)

            # Renderiza
            render_block(
              buffer,
              config_imagen,
              config_fractal,
              camara,
```

```
            x,
            y,
            ancho,
            alto)

        # Envía el bloque renderizado
        mpi_enviar(
            buffer,
            destino=master,
            tag="RESULT")

    elif mensaje.tag == "TERMINATE":
        recibir_senal_terminacion()
        break
```

La función worker arranca enviando al master una petición de tarea y se bloquea hasta recibir una respuesta. Cuando llega una tarea, el worker crea un búfer para la sección asignada, invoca render_block para rellenarlo con los píxeles fractales correspondientes y luego devuelve tanto la descripción de la tarea como su contenido al proceso master. Este ciclo de petición–procesamiento–envío se repite hasta que el master indica la terminación, momento en el cual el worker sale del bucle y finaliza su ejecución.

3.9 Parámetros de Funcionamiento

En esta sección se describen en detalle los comandos de ejecución de la aplicación DistributedFractals, tanto en modo secuencial como distribuido, los parámetros de entrada disponibles y las condiciones necesarias del entorno para su correcto funcionamiento.

3.9.1 Requisitos y Condiciones del Entorno

Para garantizar la reproducibilidad de los experimentos y el correcto funcionamiento de la plataforma de renderizado distribuido, el entorno de ejecución debe satisfacer los siguientes requisitos hardware, software y de configuración:

- **Sistema Operativo**:
  - Linux (distribuciones basadas Debian GNU/Linux)
- **Herramientas de Construccion**:
  - **CMAKE** version >= 3.14
  - **Compilador C++** con soporte para estandar C++17 (p. ej., `g++ 7.5+`, `clang++ 8+`)
- **Implementacion MPI**
  - **MPICH** >= 3.2 o **OpenMPI** >= 4.0
  - Variables de entorno configuradas (`MPI_HOME`, `PATH`, `LD_LIBRARY_PATH`)
  - Acceso a los binarios `mpirun` y/o `mpiexec`
- **Bibliotecas de tiempo de ejecución**
  - Librerías estándar de C++17 (`libstdc++`, `libm`)

- Librerías MPI (`openmpi-bin`, `openmpi-bin`, `openmpi-common`)

3.9.2 Instrucción de Construccion

Previo a la ejecuccion, es necesario construir el ejecutable. Para ello, primero instalar las dependencias necesarias:

```
sudo apt install openmpi-bin openmpi-bin openmpi-common
```

Luego, dentro de la carpeta `DistributedFractals` ejecutar:

```
mkdir build
cd build
cmake ..
make
```

3.9.3 Parámetros de Entrada

La aplicación admite los siguientes parámetros de entrada:

Parámetro	Descripción	Valor por defecto
`--width`	Ancho de la imagen (píxeles)	800
`--height`	Alto de la imagen (píxeles)	600
`--zoom`	Nivel de zoom	1.0
`--camera_x`	Posición X del centro de la cámara	0.0
`--camera_y`	Posición Y del centro de la cámara	0.0
`--iterations`	Máximo número de iteraciones	100
`--type`	Identificador de tipo de fractal (0 = Mandelbrot, 1 = Julia, …)	0
`--color_mode`	Modo de coloreado	0
`--block_size`	(MPI) Tamaño de bloque en píxeles	64
`--samples`	(MPI) Número de muestras MSAA	1
`--output_disk`	Ruta de salida para guardar la imagen en disco	`output.png`
`--Julia-cx`	Componente real de la constante C (solo Julia)	0.285
`--Julia-cy`	Componente imaginaria de la constante C (solo Julia)	0.01

3.9.4 Ejecución Secuencial

La versión secuencial de la aplicación permite generar imágenes fractales utilizando un único proceso de cómputo. El ejecutable asociado se denomina `sequential`.

```
./sequential [OPCIONES]
```

3.9.5 Ejecución Distribuida (MPI)

La versión paralela aprovecha MPI para repartir bloques de cálculo entre varios procesos. El ejecutable se denomina `fractal_mpi`.

```
mpirun -np <N> ./fractal_mpi [OPCIONES]
```

donde `<N>` es el número de procesos MPI.

Ejemplo de Uso

```
# Con 8 procesos MPI y parámetros personalizados
mpirun -np 8 ./fractal_mpi \
  --width 1080 \
  --height 720 \
  --zoom 1.5 \
  --camera_x -0.7 \
  --camera_y 0.0 \
  --iterations 256 \
  --type 0 \
  --block_size 64 \
  --samples 4 \
  -output_disk mandelbrot_distribuido.png
```

4 Estudio experimental

En esta sección, se desarrolla el diseño de experimental, sus resultados y análisis de los mismos.

4.1 Diseños de Experimentos

Primero, se plantea un estudio comparativo entre la versión secuencial y la paralela bajo distintas cantidades de nodos, con el fin de evaluar la eficiencia y el speedup.

Luego, se analiza en profundidad el comportamiento de la versión paralela frente a diferentes configuraciones de parámetros.

Es importante aclarar que cada una de las mediciones se basa en el tiempo de ejecución medio, el cuál se obtuvo a partir del promedio de al menos 10 ejecuciones, con el objetivo de asegurar resultados representativos y confiables. Además, ya que el fin de los experimentos radica en el cómputo del buffer con los colores de los pixeles, se ha eliminado de la experimentación el guardado de la imagen.

4.2 Versión secuencial contra paralela

Con el objetivo de realizar una comparación exhaustiva entre la versión secuencial y la versión paralela de la aplicación, se evaluó el rendimiento medio de ambas bajo los siguientes parámetros fijos:

Parámetro	Valor
`iterations`	512
`samples`	16
`block size` (Aplica a la versión paralela)	32

Un factor determinante es la resolución de la imagen, especificada por los parámetros width y height. Para simplificar, se utilizaron imágenes cuadradas con $width = height$. Se realizaron ejecuciones para los siguientes tamaños:

Caso	Resolución
0	128 x 128
1	512 x 512
2	1080 x 1080
3	1920 x 1920

Con el fin de evaluar la **eficiencia** y el **speedup** de la versión paralela, en comparación con la versión secuencial, cada combinación de los parámetros anteriores se ejecutó utilizando diferentes cantidades de procesos MPI: $[2, 4, 8, 16, 32]$.

4.3 Análisis de versión paralela

Existen otros parámetros relevantes, además del tamaño de imagen, que se han considerado fundamentales para el análisis. A continuación, se detallan los parámetros estudiados junto con la justificación de su inclusión y los distintos valores sobre los cuales se realizaron las mediciones:

Parámetro	Justificación	Valores
`block_size`	Permite analizar el impacto del tamaño de bloque en el balanceo de carga entre nodos, con el objetivo de encontrar un valor óptimo.	$[2, 4, 8, 16, 32, 64, 128]$
`tipo de fractal`	Evalúa si el tipo de fractal influye en el rendimiento computacional.	$[Mandelbrot, Julia]$
`iteraciones`	Permite observar cómo afecta el aumento en el número máximo de iteraciones al tiempo de ejecución, y analizar si su comportamiento es lineal, logarítmico o exponencial.	$[100, 500, 1000]$

Estos experimentos se realizaron con 32 nodos computacionales,

4.4 Consideraciones sobre experimentos

Con el fin de garantizar la validez de los resultados, se han tomado en cuenta los siguientes criterios:

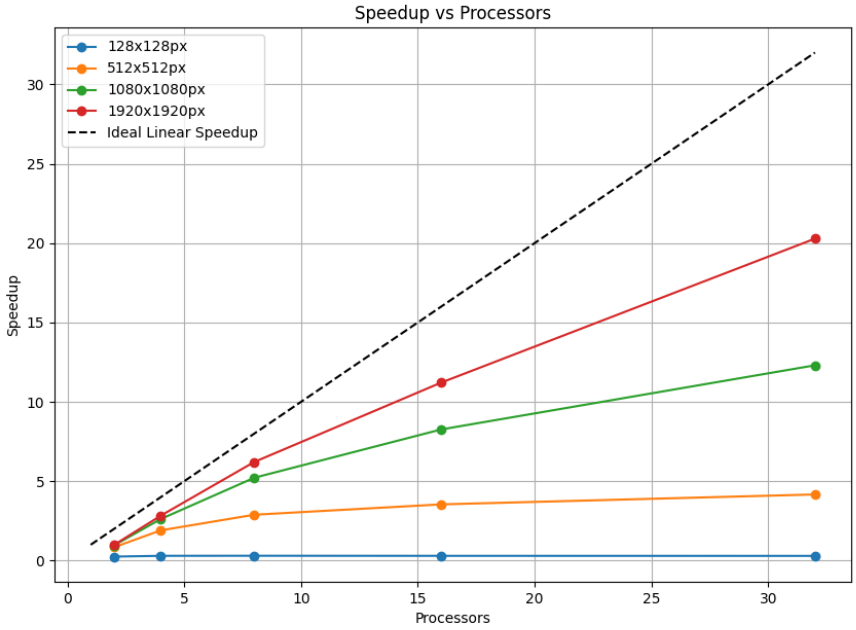
- Todas las corridas se realizan sobre la misma configuración de hardware, un cluster de nodos Debian con CPU de cuatro núcleos físicos, los cuales forman un total de 32 nodos computacionales.
- Misma versión de **OpenMPI (4.1.4)**.
- Las compilaciones se efectúan con optimización `-O3`.
- Aquellos parámetros que no se hayan especificado, toman su valor por defecto.
- La función de temporización utilizada, `perf_counter()` de la libreria `time` en python, se invoca de manera uniforme en todas las pruebas.

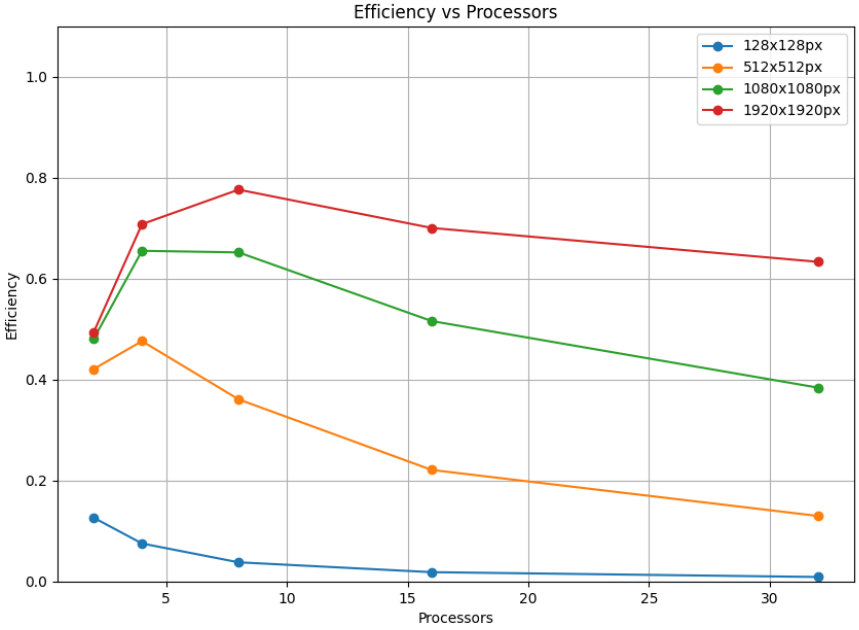
Con este riguroso control de variables, los resultados obtenidos reflejan de forma confiable el impacto de los factores estudiados sobre el tiempo de renderizado, el speedup y la eficiencia de la versión paralela, permitiendo extraer conclusiones sólidas sobre sus límites de escalabilidad y sus puntos de inflexión en el rendimiento.

4.5 Resultados Obtenidos

(DESARROLLO PENDIENTE)

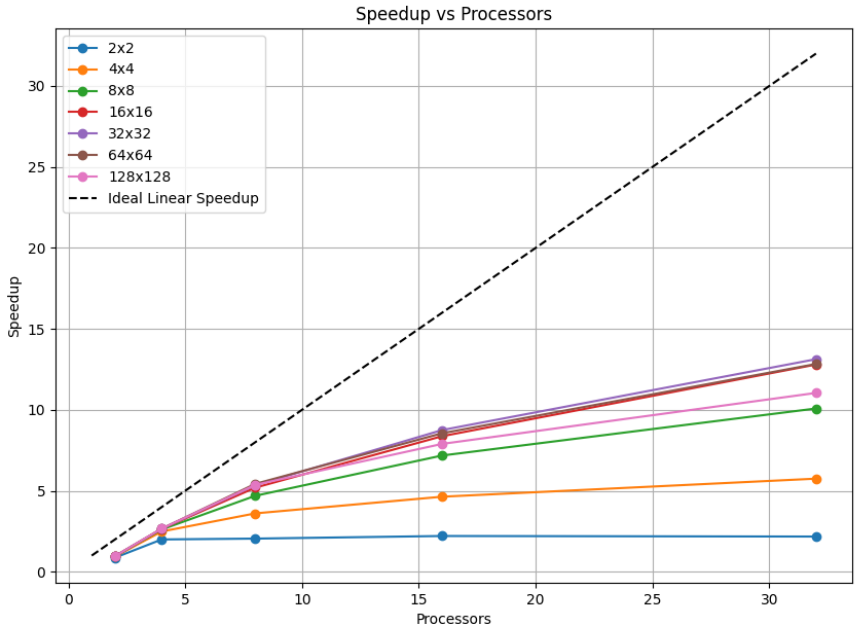
4.6 Versión secuencial contra paralela

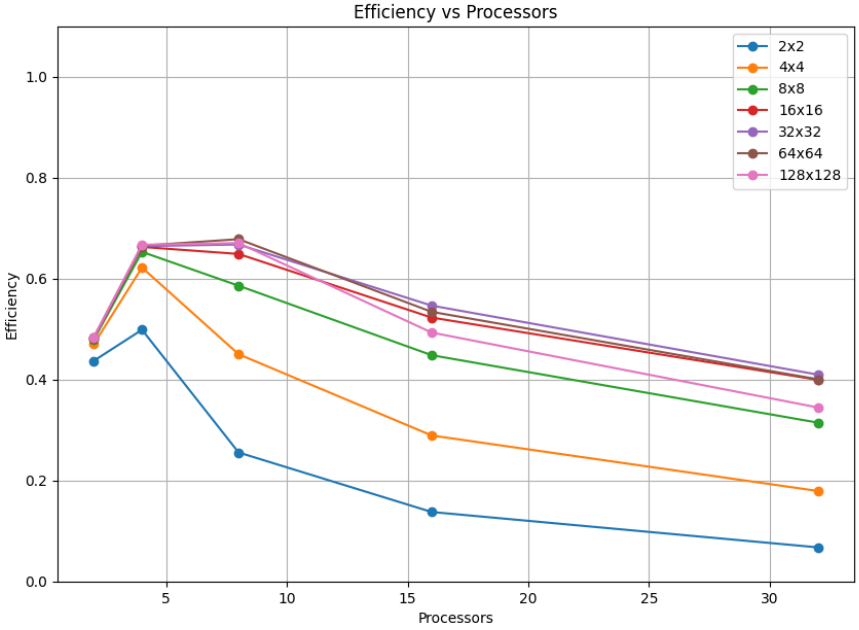



Efficiency vs Processors
128x128px
512x512px
1080x1080px
1920x1920px
Efficiency
Processors
0.0
0.2
0.4
0.6
0.8
1.0
5
10
15
20
25
30

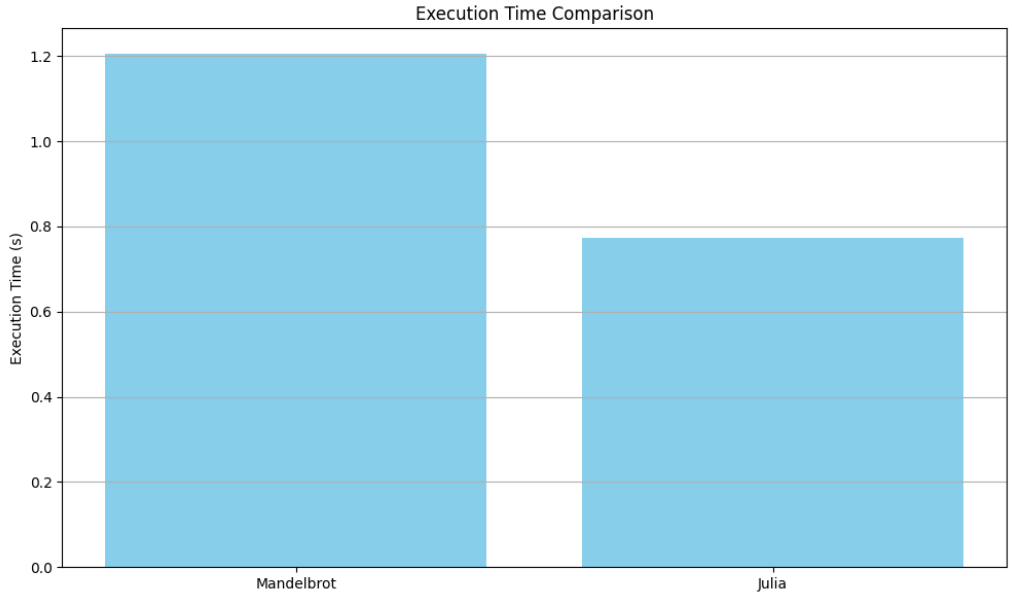
4.7 Versión paralela con distintos parámetros

4.8 Tamaño de bloques

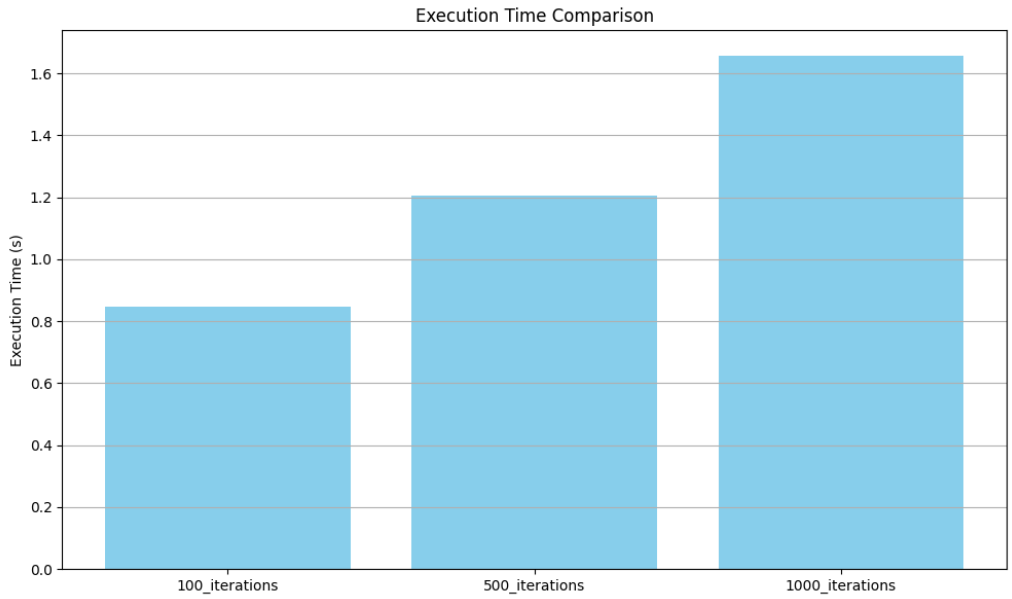



Efficiency vs Processors
2x2
4x4
8x8
16x16
32x32
64x64
128x128
Efficiency
Processors
0.0
0.2
0.4
0.6
0.8
1.0
5
10
15
20
25
30

4.9 Tipo de fractal



4.10 Cantidad de iteraciones



4.11 Analisis de los Resultados

(DESARROLLO PENDIENTE)

5 Conclusiones

5.1 Planteo de Mejora

(DESARROLLO PENDIENTE)

Aunque el esquema master–trabajador implementado en DistributedFractals consigue un balanceo de carga dinámico eficiente, el proceso master se convierte en un cuello de botella cuando el sistema escala a un gran número de workers. En la versión actual, el master atiende de forma secuencial dos tareas críticas: recibir bloques de píxeles procesados y copiarlos uno a uno en el búfer global. Cada recepción y posterior copia obliga al master a esperar a que se complete la escritura en memoria antes de poder responder a la siguiente petición de resultados, generando tiempos ociosos en los workers y limitando el speedup alcanzable.

Para mitigar esta contención, proponemos reemplazar la sección monohilo de recepción y ensamblado por una arquitectura multihilo dentro del master. En esta nueva versión, un hilo dedicado gestionaría exclusivamente la recepción de mensajes MPI entrantes, almacenándolos inmediatamente en un pool de buffers preasignados. Mientras tanto, uno o más hilos workers internos realizarían la copia asíncrona de cada bloque al búfer global, operando sobre regiones independientes de la imagen. De esta forma, la llamada a MPI_Recv no bloquearía la escritura en memoria, y los hilos de copia podrían ejecutarse en paralelo con las operaciones de recepción y la lógica de despacho de nuevas tareas.

El diseño multihilo se apoyaría en un patrón productor-consumidor: el hilo de recepción actúa como productor de unidades de trabajo (bloques recibidos), mientras que el(los) hilo(s) de ensamblado consumen dichos bloques para integrarlos en la imagen. La sincronización entre hilos se coordinaría mediante colas de bloqueo ligero (lock-free queues) o semáforos de bajo coste, garantizando seguridad de memoria y eliminando la latencia asociada a locks pesados. Asimismo, introduciendo doble búfer —un búfer en uso por la copia mientras otro está siendo llenado—, se conseguiría un solapamiento aún mayor entre comunicación y cómputo.

Adicionalmente, convendría explorar el uso de comunicaciones MPI no bloqueantes (MPI_Irecv/MPI_Isend), de manera que los hilos puedan iniciar recepciones anticipadas y comprobar su finalización de forma periódica, en vez de depender de bloqueos completos. Este enfoque híbrido MPI+threads aprovecha la independencia de los bloques fractales para maximizar el solapamiento, reduce los tiempos de espera del master y permite escalar más eficientemente al incrementar el número de procesos y el tamaño de los problemas. En conjunto, estas modificaciones prometen reducir drásticamente los intervalos ociosos en los workers y acercar el rendimiento observado al límite teórico dictado por la ley de Amdahl.

6 Bibliografía

[1] https://solarianprogrammer.com/2013/02/28/Mandelbrot-set-cpp-11/

[2] https://en.wikipedia.org/wiki/Mandelbrot_set

[3] https://youtu.be/FFftmWSzgmk?si=KPTdCiAoU7zeQ5VQ

[4] https://youtu.be/LqbZpur38nw?si=QAimXxeVIlmIqf3I

[5] https://cglearn.eu/pub/advanced-computer-graphics/fractal-rendering

[6] https://www.mpi-forum.org/docs/mpi-4.1/mpi41-report.pdf

[7] https://en.wikipedia.org/wiki/Complex_dynamics

[8] https://es.wikipedia.org/wiki/Sistema_din%C3%A1mico

[9] https://en.wikipedia.org/wiki/Julia_set

6.1 Proyectos de referencia

[8] https://github.com/lucaszm7/Mandel2Us

[9] https://github.com/Sudo-Rahman/Fractalium