

# 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 conjunto 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  $f : C \rightarrow C$  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:

$$\text{Si } |z_0| < 1, \text{ entonces } f^n(z_0) \rightarrow 0$$

$$\text{Si } |z_0| > 1, \text{ entonces } f^n(z_0) \rightarrow \infty$$

$$\text{Si } |z_0| = 1, \text{ entonces } f^n(z_0) \rightarrow 1$$

## 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, caracterizado por su complejidad 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  $|z_n| > 2$ , 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:

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

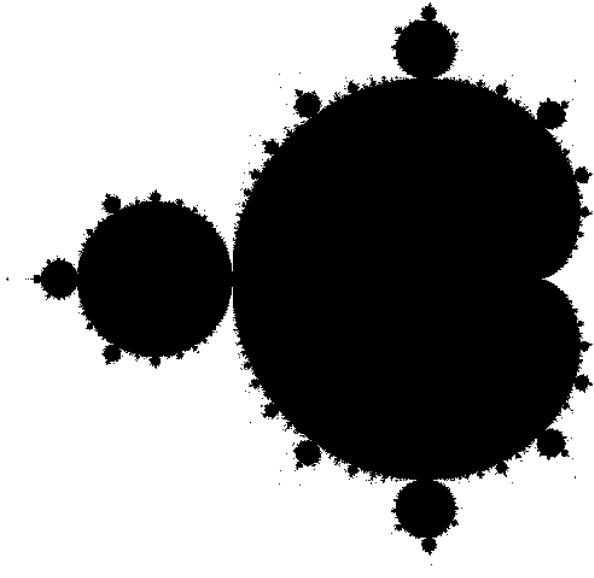
donde:

- $p$  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].

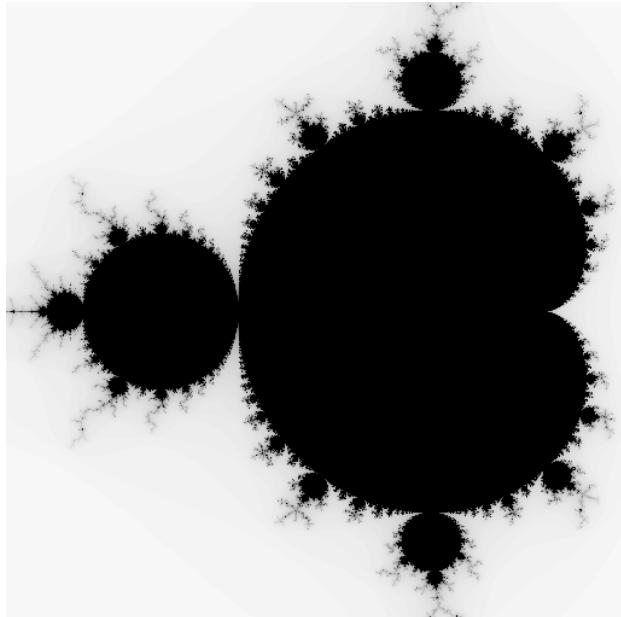
## 2.5 Coloreo de fractales

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.



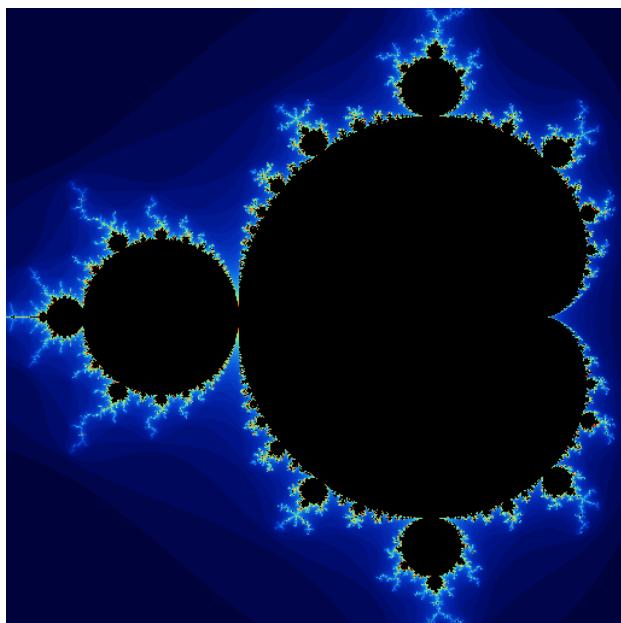
**Figura 1:** Representación en blanco y negro del conjunto de Mandelbrot.

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].



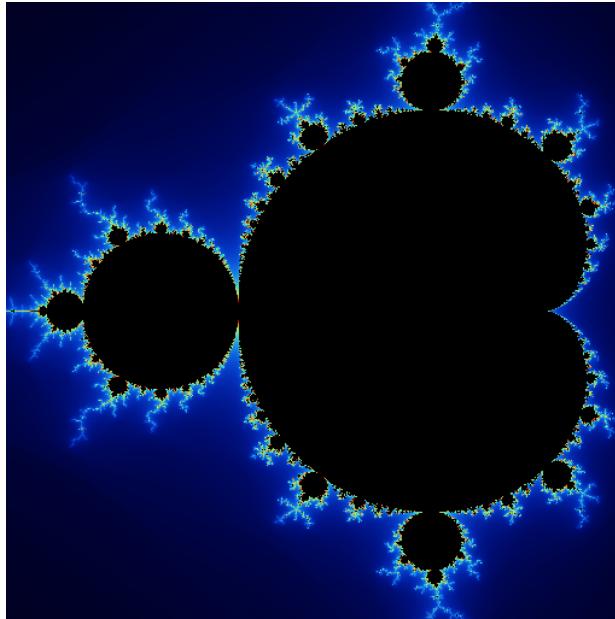
**Figura 2:** Representación en escala de grises del conjunto de Mandelbrot.

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 imágenes se resume en, la obtención del color correspondiente según la fórmula de fractal aplicada a cada pixel de la imagen. Para la cual se desarrolló una función 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 cámara
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 función 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 función, **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 paralelo, para observar las diferencias de ambas en términos 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 períodos 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_señal_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 llenarlo 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 Construcción:**
  - 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 Construcción

Previo a la ejecucion, 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 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 cual se obtuvo a partir del promedio de al menos 10 ejecuciones, con el objetivo de asegurar resultados representativos y confiables.

### 4.1 Parámetros utilizados

Los siguientes parámetros se mantienen constantes a lo largo de todos los experimentos.

Parámetro	Valor
<code>samples</code>	4
<code>cx</code>	-0.7454286
<code>cy</code>	0.1130089
<code>zoom</code>	327000
<code>color_mode</code>	5
<code>type</code>	<i>Mandelbrot</i>
<code>output_disabled</code>	-

Nótese que se utiliza el parámetro `output_disabled` ya que el fin de los experimentos radica en el cómputo del buffer con los colores de los pixeles. El proceso de creación de imágenes PNG y su escritura en el disco puede tomar una considerable cantidad de tiempo, especialmente en ejecuciones rápidas.

## 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 han considerado dos tipos de atributos, la resolución de la imagen y el tamaño de los bloques.

Los siguientes experimentos se ejecutan utilizando diferentes cantidades de procesos MPI: [2, 4, 8, 16, 32], con el fin de evaluar la **eficiencia** y el **speedup** de la versión paralela, en comparación con la versión secuencial.

### 4.2.1 Resolución de imagen

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

Además de los parámetros establecidos anteriormente, se ha fijado:

Parámetro	Valor
<code>block_size</code>	32
<code>iterations</code>	20000

#### 4.2.2 Tamaño de bloque

Permite analizar el impacto del tamaño de bloque en el balanceo de carga entre nodos, con el objetivo de encontrar un valor óptimo.

Parámetros extra constantes:

Parámetro	Valor
width	1080
height	1080
iterations	20000

Se ha fijado la resolución ya que en este caso nos interesa estudiar el impacto del tamaño de bloque.

Los tamaños de bloque evaluados son [2, 4, 8, 16, 32, 64, 128]

#### 4.3 Cantidad de iteraciones

Resulta de especial interés estudiar cómo impacta la cantidad de iteraciones no solo en los tiempos de ejecución, sino también en la calidad de las imágenes generadas.

La experimentación enfocada en la cantidad de iteraciones permite observar cómo influye el incremento del número máximo de iteraciones en el tiempo de ejecución, y analizar si este comportamiento es lineal, logarítmico o exponencial.

Este análisis se realiza exclusivamente sobre la versión paralela, dejando de lado la versión secuencial, dado que los experimentos previos han sido suficientes para la comparación, y este experimento en particular no aporta información adicional relevante.

Las iteraciones evaluadas son [200, 500, 1000, 2000, 3000, 4000, 5000, 10000, 15000, 20000]

Parámetros extra constantes:

Parámetro	Valor
width	1080
height	1080
block_size	32
np (Cantidad de nodos)	32

#### 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 ejecuciones 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 librería `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.

## 5 Resultados Obtenidos

En esta sección se presentarán los resultados obtenidos al realizar los experimentos planteados en la sección de diseño de experimentos. Es por este motivo que seguirá la misma estructura planteada anteriormente. Se presentarán tablas de datos y gráficos, los cuales serán analizados en detalle en la sección de análisis de resultados.

### 5.1 Versión secuencial contra paralela

A continuación se muestra el rendimiento de ambas versiones en función del tamaño de la imagen y la cantidad de nodos utilizados.

#### 5.1.1 Tamaño de imagen

Se estudió el efecto de modificar la resolución de imagen en el tiempo de ejecución.

##### 5.1.1.1 Tabla de datos de ejecución secuencial

A modo de comparación, se incluye el tiempo de ejecución para la versión secuencial con cada resolución.

Resolución	Tiempo promedio (s)	Desviación estándar (s)
32x32	0.07977260379984731	0.00043435765082315455
64x64	0.31759674040004027	0.002131164715677036
128x128	1.2609421436000048	0.0013148213508798747
512x512	20.132811490099993	0.009180779629399193
1080x1080	89.59314089329982	0.024588972955427328
1920x1920	283.25439927300033	0.04803506827399954

##### 5.1.1.2 Tabla de datos de ejecución paralela

Se presenta el tiempo promedio y la desviación estándar para distintas resoluciones y cantidades de nodos.

Cantidad de nodos	Resolución	Tiempo promedio (s)	Desviación estándar (s)
2	32x32	0.7347766071998194	0.024293793933656622
4	32x32	0.7194581487994582	0.019995011884794463
8	32x32	0.7597506460006116	0.019692794415084734
16	32x32	0.755642992799767	0.014237480350074726
32	32x32	0.7530325819996506	0.014154650552929655
2	64x64	0.9698013217999687	0.04280733081779699
4	64x64	0.7844574396993267	0.02057444796819635
8	64x64	0.8701380547001463	0.20425526530434057
16	64x64	0.813580839000133	0.02785545076843234
32	64x64	0.8377403283004241	0.030037874078768456
2	128x128	1.8987524622003549	0.02253840710023415
4	128x128	1.0858486641998752	0.027295021328173408
8	128x128	1.0203170211996622	0.026902499139743397
16	128x128	1.0145710798999061	0.022382251243951167
32	128x128	1.0069084299993847	0.019242904410697554
2	512x512	20.83092403529954	0.022270186767527914
4	512x512	7.38089184599994	0.025637281621891578
8	512x512	3.6114315322007315	0.024159854440402008
16	512x512	2.126211233899812	0.03389213149233942
32	512x512	1.6320069547000458	0.03812131381339011
2	1080x1080	90.4301608312002	0.044303374630204846
4	1080x1080	30.63535752170028	0.022529494539971554
8	1080x1080	13.590946517599514	0.014405034421835603
16	1080x1080	6.808138592400428	0.0172290493620024
32	1080x1080	4.109694875600326	0.05180247169371324
2	1920x1920	284.35179361740086	0.028804059407490677
4	1920x1920	95.44321080139962	0.03332558576788434
8	1920x1920	41.52969608500025	0.017873206091913493
16	1920x1920	20.056586532099754	0.025654280626253977
32	1920x1920	11.296893179300605	0.25224913775009544

### 5.1.1.3 Tabla de datos de speedup y eficiencia

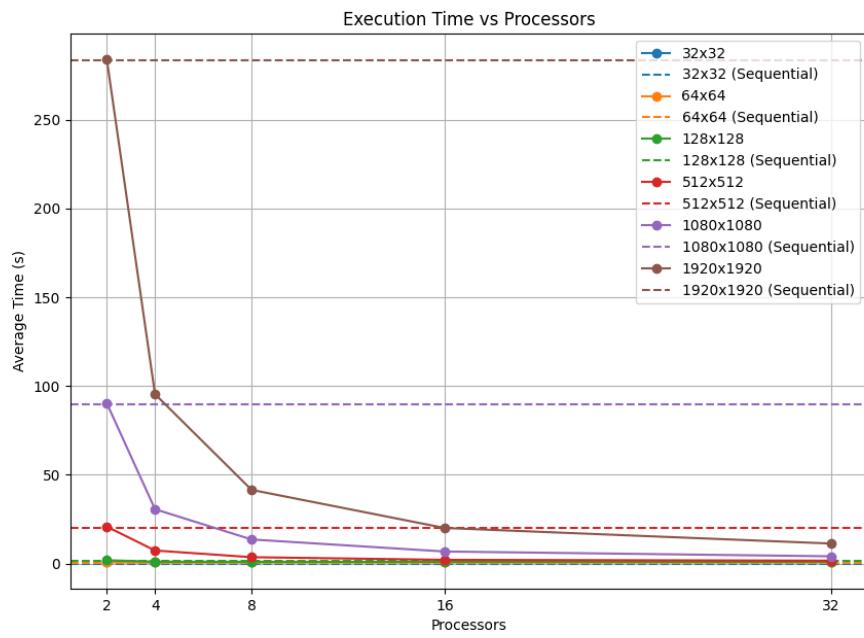
A partir de los tiempos anteriores, se calculó el Speedup y la Eficiencia de la versión paralela respecto a la secuencial.

Cantidad de nodos	Resolución	Speedup	Efficiency
2	32x32	0.10856715227211022	0.05428357613605511
4	32x32	0.11087872718234111	0.027719681795585278
8	32x32	0.10499840206754242	0.013124800258442803
16	32x32	0.10556917030922001	0.006598073144326251

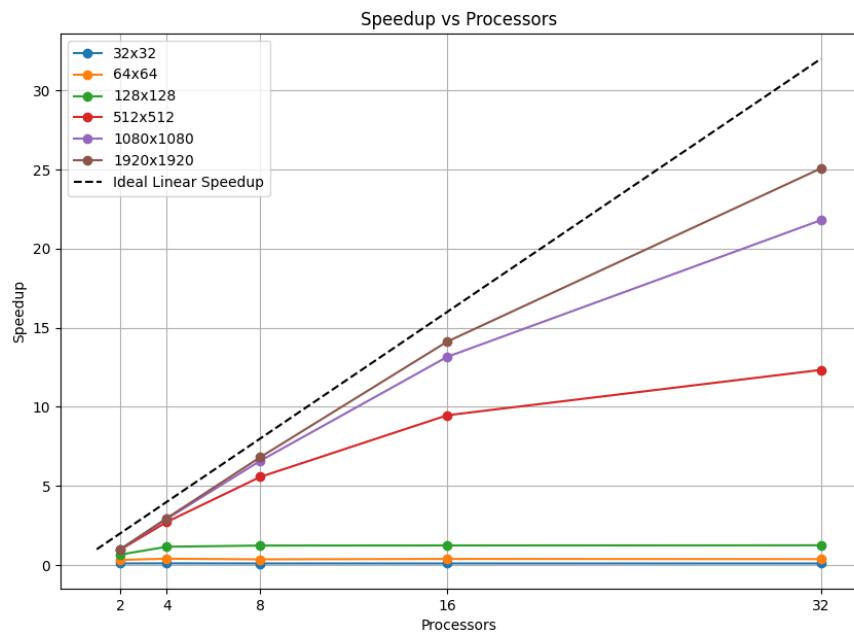
Cantidad de nodos	Resolución	Speedup	Efficiency
32	32x32	0.10593512911222787	0.003310472784757121
2	64x64	0.3274863967091476	0.1637431983545738
4	64x64	0.404861658934322	0.1012154147335805
8	64x64	0.3649958057626679	0.045624475720333485
16	64x64	0.3903690022866779	0.02439806264291737
32	64x64	0.3791111991043436	0.011847224972010737
2	128x128	0.6640898003833378	0.3320449001916689
4	128x128	1.1612503520729105	0.2903125880182276
8	128x128	1.2358336844341005	0.15447921055426256
16	128x128	1.2428327286092216	0.07767704553807635
32	128x128	1.252290780404704	0.039134086887647
2	512x512	0.9664867221436484	0.4832433610718242
4	512x512	2.727693605348103	0.6819234013370258
8	512x512	5.57474544667097	0.6968431808338712
16	512x512	9.46886704815927	0.5918041905099544
32	512x512	12.336228979980234	0.3855071556243823
2	1080x1080	0.9907440180332888	0.4953720090166444
4	1080x1080	2.924501234556748	0.731125308639187
8	1080x1080	6.592119303632144	0.824014912954018
16	1080x1080	13.159711671161924	0.8224819794476202
32	1080x1080	21.80043618936855	0.6812636309177672
2	1920x1920	0.9961407159404907	0.49807035797024535
4	1920x1920	2.9677794459618765	0.7419448614904691
8	1920x1920	6.820526658640936	0.852565832330117
16	1920x1920	14.122762057225598	0.8826726285765999
32	1920x1920	25.073654745360415	0.783551710792513

#### 5.1.1.4 Gráficos de rendimiento

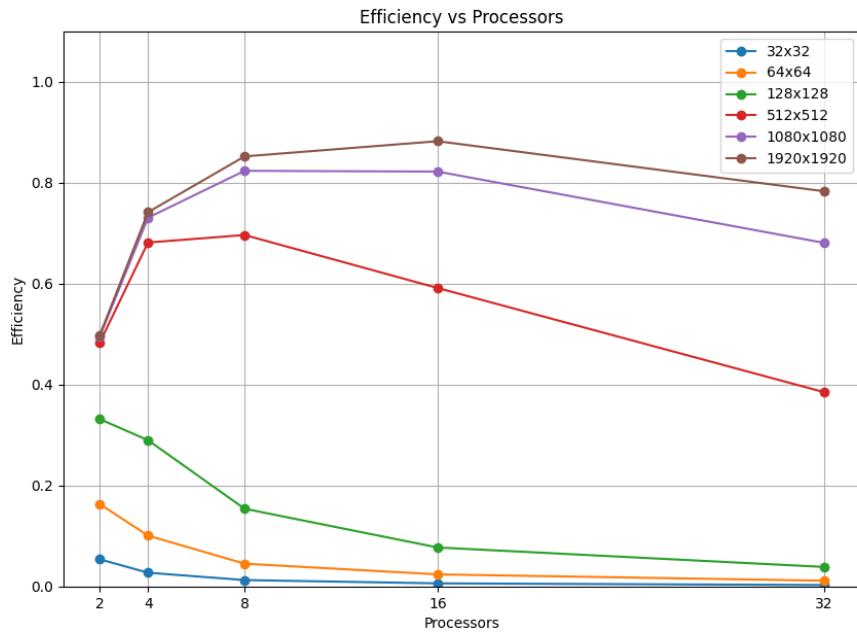
A continuación, se presentan gráficos realizados con los datos obtenidos previamente.



**Figura 5:** Tiempo medio paralelo y secuencial para cada configuración de cantidad de nodos y resolución de imagen



**Figura 6:** Speedup para cada configuración de cantidad de nodos y resolución de imagen



**Figura 7:** Eficiencia para cada configuración de cantidad de nodos y resolución de imagen

### 5.1.2 Tamaño de bloque

Se estudió el efecto de modificar el tamaño de bloque en el tiempo de ejecución.

#### 5.1.2.1 Tabla de datos de ejecución paralela

Cantidad de nodos	Tamaño de bloque	Tiempo promedio (s)	Desviación estándar (s)
2	2x2	91.43973352199973	0.05611679641446187
4	2x2	31.363201925400062	0.035844511996024966
8	2x2	17.032129470899235	0.16132378141367337
16	2x2	9.80977562420012	0.135481889137545
32	2x2	7.282809931899829	0.41415118461081607
2	4x4	90.62620872690096	0.030289694627595516
4	4x4	30.74810164060109	0.028864040552125478
8	4x4	14.44186259309863	0.021789357986847967
16	4x4	7.489232776599965	0.02892393616505963
32	4x4	4.734652727498178	0.1061245533777847
2	8x8	90.39643993430218	0.04501368269027079

Cantidad de nodos	Tamaño de bloque	Tiempo promedio (s)	Desviación estándar (s)
4	8x8	30.60676306569949	0.021960356024083506
8	8x8	13.767168696501177	0.016930315679874557
16	8x8	6.886950576501112	0.023923186072850967
32	8x8	4.030236448300275	0.028466196545113925
2	16x16	90.33003026119987	0.0241139508851428
4	16x16	30.56708972500055	0.02466749615701291
8	16x16	13.5533268640007	0.019474163858620858
16	16x16	6.769189039500634	0.011171972498731777
32	16x16	3.9328893263009377	0.029974776402533564
2	32x32	90.32909833449958	0.0367866529867054
4	32x32	30.55001384190109	0.025399354470591846
8	32x32	13.534426990599604	0.023568102796302857
16	32x32	6.729872261199489	0.02568826934816551
32	32x32	4.0352310534013665	0.06224605207022667
2	64x64	90.31254809760067	0.018955403618388288
4	64x64	30.53957045689749	0.014845691365458883
8	64x64	13.52947120099925	0.023050388169619195
16	64x64	6.8264736725002875	0.04820564927923986
32	64x64	4.748673433499789	0.2314093086447298
2	128x128	90.3150229341998	0.027878257268597707
4	128x128	30.56140734679939	0.02336154178751105
8	128x128	14.937172039599682	0.019059549888477616
16	128x128	8.014758754597278	0.07856331140741309
32	128x128	6.913615914300317	0.6075389399688722

## 5.2 Cantidad de iteraciones

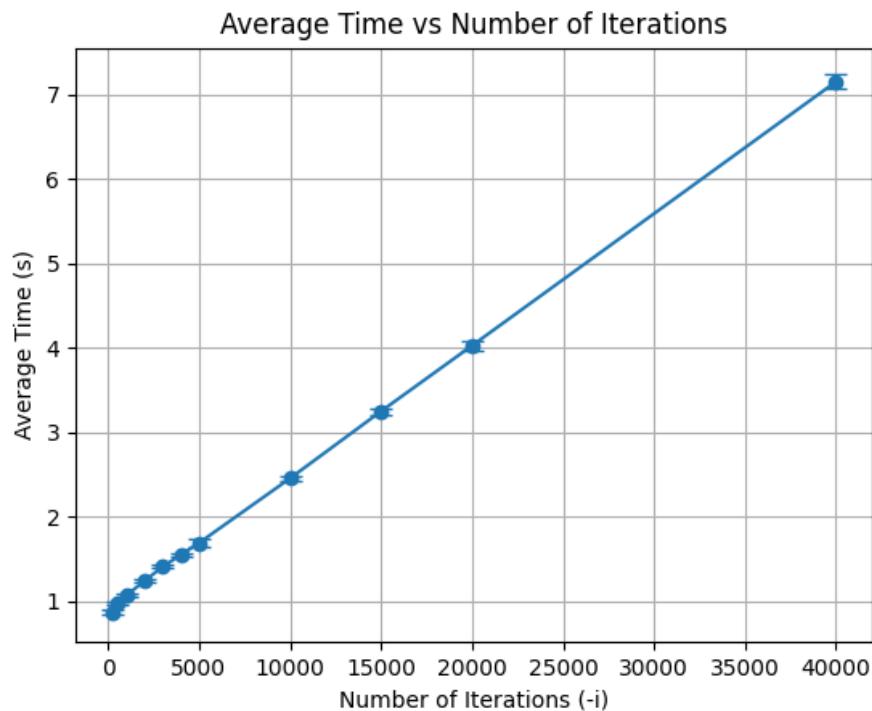
Se estudió el efecto de modificar el número de iteraciones en el tiempo de ejecución.

### 5.2.1 Tabla de datos

Iteraciones	Tiempo promedio (s)	Desviación estándar (s)
200	0.8658388962008757	0.031467115018176055
500	0.9789770936957212	0.020520204910293226
1000	1.068946530300309	0.020387310065660757
2000	1.2362676242002635	0.021568939323107487
3000	1.4084850567989633	0.021316243515446965
4000	1.5473312993985018	0.020788707928570592
5000	1.6892066827014787	0.042628523113125386
10000	2.454260219701973	0.034755324130547174

Iteraciones	Tiempo promedio (s)	Desviación estándar (s)
15000	3.2485190514998976	0.037265794864773556
20000	4.026260491098219	0.058440292592556196
40000	7.153266767101013	0.08296469529306266

#### 5.2.1.1 Gráfico de rendimiento



**Figura 8:** Tiempo de ejecución medio por cantidad de iteraciones

## 6 Análisis de los Resultados

En esta sección, se realiza un análisis de los resultados obtenidos en la sección anterior.

## 6.1 Análisis de versión secuencial contra paralela

### 6.1.1 Tamaño de imagen

La *figura 5*, ilustra claramente que existe una mejora significativa al usar el algoritmo paralelo. Se puede observar que las versiones paralelas y secuenciales toman aproximadamente el mismo tiempo cuando la cantidad de nodos es de 2, ya que solo un procesador trabaja. Luego, a medida que aumenta la cantidad de nodos, el tiempo paralelo decrece logarítmicamente, lo cuál se ve con mayor claridad en grandes resoluciones, especialmente 1920 x 1920. Es decir que se pueden obtener los mismos resultados en menor tiempo, tal como era esperado.

En cuanto a la *figura 6*, se observa que existe una relación entre el speedup obtenido y la resolución de la imagen. Si la imagen a renderizar cuenta con muy pocos pixeles, tal como la resolución 128x128, entonces podemos decir que no resulta conviente la utilización del algoritmo paralelo. Esto se debe a la sección secuencial inicial presente en la versión paralela, la cuál corresponde a la inicialización de mpi, a través de `MPI_Init`, toma aproximadamente 350ms. Al aumentar la resolución, aumenta la cantidad de pixeles a renderizar, haciendo que aumente la porción paralelizable, y es por este motivo que el speedup aumenta al renderizar imágenes con más pixeles, se aprovecha el paralelismo.

Al considerar la *figura 7*, se observa que a mayor cantidad de pixeles, es decir, en imágenes de mayor resolución, el valor que toma la eficiencia con dos nodos es de 0.5. Esto no debería sorprender considerando el modelo de algoritmo paralelo que se ha seleccionado, siendo este el master-worker. Al haber dos nodos, uno toma el rol de master y el otro de worker, pero en realidad solo un nodo se encarga del renderizado. En este contexto, si asumimos que el tiempo secuencial es igual al tiempo de cómputo:

$$T_{Secuencial} = T_{Paralelo}$$

Entonces el speedup:

$$Speedup(2) = T_{Secuencial}/T_{Paralelo} = 1$$

Luego, calculando la eficiencia

$$Eficiencia(2) = Speedup(2)/N_{nodos} = 1/2$$

Lo cuál demuestra matemáticamente que el valor de eficiencia obtenido con 2 nodos es correcto.

Además, si consideramos las gráficas correspondientes a las distintas resoluciones mostradas en la Figura 7, podemos afirmar que el punto máximo de eficiencia depende tanto de la resolución de la imagen como de la cantidad de nodos utilizados.

Para las versiones de  $32 \times 32$ ,  $64 \times 64$  y  $128 \times 128$ , la eficiencia máxima se alcanza con  $N_{Nodos} = 2$ . Esto no debería sorprender, ya que, al observar nuevamente la Figura 6, se puede ver que estas resoluciones mantienen un speedup constante a partir de  $N_{Nodos} = 2$ .

Por otro lado, al analizar la resolución de  $512 \times 512$ , la eficiencia máxima se obtiene con  $N_{\text{Nodos}} = 8$ , alcanzando un valor de eficiencia de:

$$Eficiencia(8) = 0.6968431808338712$$

De manera similar, para una resolución de  $1080 \times 1080$ , la eficiencia máxima también se da con  $N_{\text{Nodos}} = 8$ , con un valor de:

$$Eficiencia(8) = 0.824014912954018$$

Finalmente, para una resolución de  $1920 \times 1920$ , el punto máximo de eficiencia se alcanza con  $N_{\text{Nodos}} = 16$ , alcanzando:

$$Eficiencia(16) = 0.8826726285765999$$

Si relacionamos los gráficos de speedup y eficiencia, se puede observar que la relación de orden entre las distintas resoluciones de imagen y su rendimiento se mantiene. Esto se debe a que la eficiencia se calcula a partir de el speedup y la cantidad de nodos, siendo este un factor constante.

Es claro que la versión paralela no alcanzará el speedup superlineal, pero si tiende a alcanzar un speedup lineal. Hay una fuerte relación entre la porción paralelizable, la cuál aumenta al renderizar mayor cantidad de pixeles.

Sin embargo, existe un umbral a partir del cual no resulta conveniente emplear la versión paralela, ya que los tiempos secuenciales comienzan a representar una proporción significativa en relación con la parte paralelizable del problema.

A pesar de ello, al aumentar el tamaño de la imagen, la utilización de la versión paralela se vuelve cada vez más justificada y eficiente.

### 6.1.2 Tamaño de bloque

Como ilustra la figura X, existe una clara relación entre el speedup y el tamaño de los bloques.

Se observa que para bloques de  $2 \times 2$ , el speedup se ve acotado. A partir de los 16 nodos. Esto se debe a que contamos con un único nodo master, y al tener una granularidad muy fina, es decir tareas pequeñas y muchas tareas, el programa paralelo pasa mucho tiempo realizando comunicaciones, y el master se satura, convirtiéndose en un cuello de botella.

Los tamaños de  $4 \times 4$  y de  $8 \times 8$  presentan mejoras en comparación a  $2 \times 2$ .

La configuración óptima para el tamaño de bloque resulta ser  $32 \times 32$ , un total de 1024 pixeles por nodo. Este es el punto en el cuál se logra un balance entre tamaño de bloque, cantidad de nodos y tiempos de comunicaciones.

Nótese que al aumentar el tamaño de los bloques, para  $64 \times 64$  y especialmente  $128 \times 128$ , los tiempos de ejecución aumentan y el speedup se reduce.

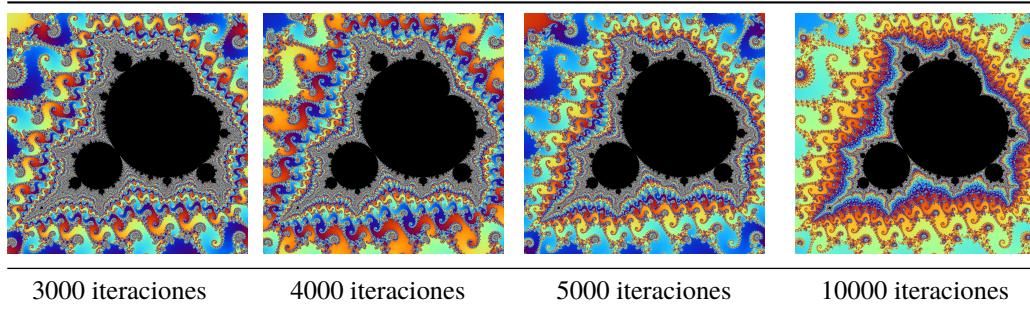
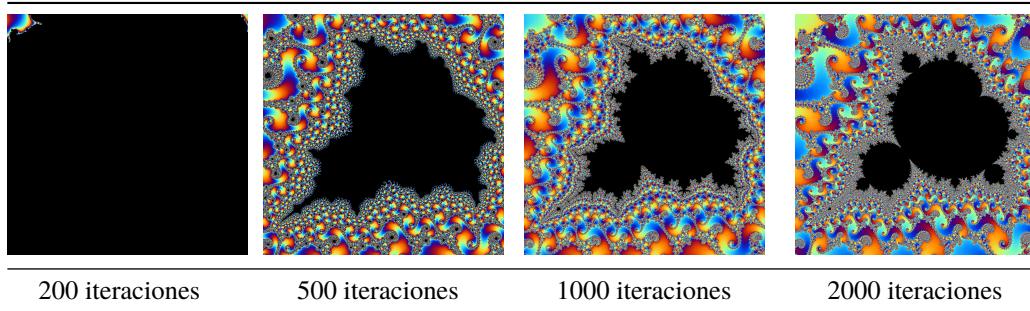
En cuanto a la eficiencia, nos encontramos la misma situación presentada en el análisis comparativo anterior cuando  $N_{\text{nodos}} = 2$ , siendo incluso más claro.

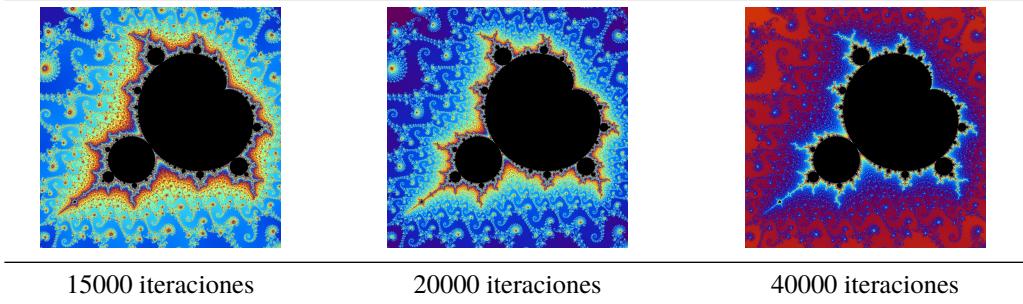
## 6.2 Cantidad de iteraciones

A pesar de que el algoritmo de tiempo de escape utilizado por el fractal de Mandelbrot pueda terminar con una menor cantidad de iteraciones que la cantidad de iteraciones máximas, establecida por el parámetro, se puede observar un comportamiento lineal en los tiempos de ejecución, ilustrados en la *figura 8*.

Este comportamiento lineal se debe a que en la imagen que se ha renderizado, existe una gran proporción de pixeles que llegan al límite de iteraciones máximo establecido por el parámetro. Estos pixeles son los que toman el color negro en las siguientes imágenes comparativas.

Parte del renderizado de factales, consiste en poder determinar que número de iteraciones utilizar, tratando de que este se minimice con el objetivo de reducir los tiempos de cómputo, pero maximizando la calidad de imagen. Es por este motivo que se ha renderizado la imagen correspondiente a cada número de iteraciones planteado:





Existe una diferencia notable en los resultados. Estos resultados varían tanto en las formas como en los colores y el ruido en la imagen. Cabe aclarar que con ruido se refiere a frecuencia de variación de color de pixeles adyacentes.

En cuanto a los colores, la diferencia radica en que el color se asigna al mapear la cantidad de iteraciones a las que llegó el pixel a una paleta de colores. Al aumentar o disminuir la cantidad de iteraciones máximas, el mapeo del número de iteraciones a un valor normalizado también cambia. Por ejemplo, supongamos que la función del fractal de Mandelbrot determina que un pixel  $P = (P_X, P_Y)$  no pertenece al conjunto con 300 iteraciones.

- $IteracionesMax = 1000$ , entonces  $t = 300/1000 = 0.3$
- $IteracionesMax = 40000$ , entonces  $t = 300/40000 = 0.00075$

Siendo  $t$  el valor utilizado para muestrear la paleta de colores y asignarle el color al pixel  $P$ .

Ahora, si analizamos las formas, especialmente las imágenes obtenidas con 200, 500 y cualquier otra con muchas iteraciones, tal como la de 20000, se puede observar que las secciones coloreadas en negro difieren y son mayores a menor cantidad de iteraciones. Esto se debe a que 200, y 500 iteraciones son muy pocas para el zoom utilizado, haciendo que el algoritmo considere a ciertos puntos como pertenecientes al conjunto de Mandelbrot, cuando en realidad no pertenecen.

Esto nos dice que existe una relación directa entre el zoom utilizado y la cantidad de iteraciones máximas. Haciendo que se requieran más iteraciones a mayor zoom.

El ruido es otro factor muy notable en las imágenes con una baja cantidad de iteraciones. Los píxeles cercanos al borde del conjunto de Mandelbrot son muy sensibles a pequeñas variaciones en la cantidad máxima de iteraciones. Con un valor bajo de iteraciones, muchos de estos píxeles se consideran escapados prematuramente, incluso si en realidad pertenecen al conjunto o están muy cerca de él. Esto provoca una coloración inconsistente entre píxeles vecinos, lo que genera una apariencia ruidosa. Aumentar la cantidad de iteraciones reduce esta incertidumbre y suaviza la imagen.

A pesar de que la imagen más precisa de todas es la de 40000 iteraciones, se considera que para estas configuraciones de cámara, las imágenes de 15000 y 20000 iteraciones son las ideales, al balancear los tiempos de ejecución, 3.2485190514998976 segundos y 4.026260491098219 segundos respectivamente, y la calidad de imagen obtenida.

## 7 Conclusiones

### 7.1 Planteo de Mejora

(DESARROLLO PENDIENTE)

Aunque el esquema master-worker 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.

## 8 Bibliografía

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

[2] [https://en.wikipedia.org/wiki/Mandelbrot\\_set](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](https://en.wikipedia.org/wiki/Complex_dynamics)
- [8] [https://es.wikipedia.org/wiki/Sistema\\_din%C3%A1mico](https://es.wikipedia.org/wiki/Sistema_din%C3%A1mico)
- [9] [https://en.wikipedia.org/wiki/Julia\\_set](https://en.wikipedia.org/wiki/Julia_set)

## 8.1 Proyectos de referencia

- [8] <https://github.com/lucaszm7/Mandel2Us>
- [9] <https://github.com/Sudo-Rahman/Fractalium>