Desarrollo de Software en Arquitecturas Paralelas

Práctica 7: Mandelbrot

Elza Sarrías Alieva

Contenido

1. Introducción

El objetivo de este trabajo es realizar la implementación paralela del algoritmo que calcula representaciones parciales del conjunto Mandelbrot. Este conjunto está formado por números complejos, $c = a + bi$, se puede definir la siguiente relación de concurrencia,

$$z_{k+1} = z_k^2 + c, \qquad k = 0,1,2 \ldots,$$

que converge a un número complejo finito, siendo $z_0 = 0$ la condición inicial. De esta manera, si existe un número m para el cual $|z_m| > 2$, la sucesión diverge y, por tanto, c, no pertenece al conjunto de Mandelbrot.

Para calcular si la sucesión diverge o no, se establece un número máximo de iteraciones, $z_{IterMax}$, de forma que si ningún $|z_i|$, $i < IterMax$es superior a 2, se considera que el punto c forma parte del conjunto.

De esta forma, si el punto c está dentro del conjunto el píxel correspondiente de la imagen se colorean como negro. En caso contrario, se selecciona un color basándose en el número de iteraciones necesarias para encontrar un número $m$tal que $|z_m| > 2$.

El algoritmo implementado creará dos imágenes que representan el conjunto de Mandelbrot, que se diferencian en los colores utilizados y la forma de determinar el color para los píxeles que no forman parte del conjunto.

Previo al cálculo, se establecen las características de la representación, como las dimensiones de la imagen, el número máximo de iteraciones, los nombres de archivos de salida, y el área del plano complejo.

La implementación secuencial del algoritmo recorre la imagen píxel por píxel para determinar si forman parte del conjunto o no.

El tiempo que se tarda en completar una fila es variable: depende del número de píxeles de esta fila que pertenecen al conjunto de Mandelbrot, ya que para estos píxeles como mínimo se completarán IterMax iteraciones. Por ello, para la implementación paralela dividir la imagen en partes iguales no es una buena solución.

Por tanto, la implementación paralela del algoritmo se basará en la asignación dinámica de tareas mediante un pool de procesos. De esta forma, los procesos irán haciendo una por una las filas de la imagen. Una vez completada una fila, la enviarán al proceso 0 que será encargado de asignarle una fila aún no calculada al proceso inactivo.

2. Implementación

A la hora de implementar la versión paralela del algoritmo, en primer lugar, se leen los datos de entrada que especifican las características de la representación parcial del conjunto Mandelbrot que se debe calcular. El proceso 0 se encarga de realizar este paso. Se pueden especificar los siguientes datos:

- Las dimensiones de la imagen: **pixelXmax** x **pixelYmax**, por defecto 1024 x 1024.
- El número máximo de iteraciones: **IterMax**, por defecto 1000.
- El nombre del archivo de las imágenes. Se especifica el nombre sin la extensión, por defecto img, de forma que los archivos creados se llamen imgA.pgm y imgB.pgm.
- El área del plano complejo en el que se realizarán los cálculos. En este caso se puede seleccionar los dominios de un conjunto predefinido. Como resultado, se conocerán los mínimos y los máximos de la parte compleja y la parte real: **RealMin**, **RealMax**, **ImMin**, **ImMax**. Por defecto, la parte real varía en [-2, 1] y la imaginaria en [-1.5, 1.5].

Una vez leídos los datos iniciales, el proceso 0 crea las matrices donde se realicen los cálculos, así como los archivos de salida.

Antes de poder empezar el algoritmo, todos los procesos deben conocer los datos necesarios para los cálculos: el dominio del plano complejo, alto y ancho de la imagen, así como el número máximo de iteraciones. Para ello, se realiza un broadcast de estas variables.

```
127     MPI_Bcast(&RealMin, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
128     MPI_Bcast(&RealMax, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
129     MPI_Bcast(&ImMin, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
130     MPI_Bcast(&ImMax, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
131     MPI_Bcast(&pixelXmax, 1, MPI_INT, 0, MPI_COMM_WORLD);
132     MPI_Bcast(&pixelYmax, 1, MPI_INT, 0, MPI_COMM_WORLD);
133     MPI_Bcast(&IterMax, 1, MPI_INT, 0, MPI_COMM_WORLD);
```

Una vez finalizado el broadcast, todos los procesos calculan al alto y el ancho de cada píxel y crean la matriz **datos** donde se escribirán los datos a enviar. Los procesos hijos

trabajarán con dos filas a la vez: una de la imagen A y otra de la imagen B, y tendrán que enviar al padre dos esas dos filas a la vez. Para simplificar el envío de mensajes se ha optado por utilizar una matriz auxiliar en la que los hijos escribirán ambas filas calculadas.

```
135     AnchoPixel=(RealMax-RealMin)/(pixelXmax-1);
136     AltoPixel=(ImMax-ImMin)/(pixelYmax-1);
137
138     // matriz de datos con la que van a trabajar los hijos:
139     int* datos = (int*) malloc(pixelXmax*2*sizeof(int));
```

Una cuestión importante que considerar es el modo de envío de los datos. En concreto, es necesario tener en cuenta que cuando se envían mensajes grandes se puede producir un interbloqueo. MPI proporciona varias alternativas para resolver este problema. En este caso se ha optado por el uso de la función **MPI_Bsend**, que permite gestionar su propio buffer de comunicación. Además, esta operación no depende de una operación de recepción para finalizar: si no existe la recepción el mensaje se dirige al buffer para completar la llamada y continuar la ejecución.

Para poder enviar datos con este método se debe hacer uso de las funciones **MPI_Buffer_attach**, para apuntar al buffer creado, y **MPI_Pack_size** para calcular la cota superior del espacio requerido para un mensaje. A continuación, se puede ver, como se configura el buffer en el código implementado.

```
141     // int sizeBuffer;
142     // int* buffer;
143     MPI_Pack_size(pixelXmax*2, MPI_INT, MPI_COMM_WORLD, &sizeBuffer);
144     sizeBuffer = numproc*(sizeBuffer + MPI_BSEND_OVERHEAD);
145     buffer = (int*) malloc(sizeBuffer);
146     if (buffer == NULL)
147         printf("Error al reservar la memoria del buffer\n");
148     MPI_Buffer_attach(buffer, sizeBuffer);
```

La cota superior corresponde al tamaño de dos filas de la imagen (son dos filas debido a que se trabaja con dos imágenes a la vez). El tamaño del buffer equivale a la cota superior del tamaño de un mensaje multiplicado por el número de procesos, ya que es posible que todos los procesos realicen un envío de forma simultánea. Adicionalmente, se utiliza la constante **MPI_BSEND_OVERHEAD** que corresponde a la memoria extra utilizada por la función de envío.

Una vez finalizados todos los pasos previos se puede empezar la ejecución paralela del algoritmo. El proceso 0 se encargará de asignar dinámicamente las tareas, mientras que los procesos hijos irán calculando las filas de las imágenes A y B que se les asignen.

Para indicar qué fila se debe calcular el padre enviará al hijo correspondiente el número de fila (**pixelY**). Se enviarán las filas de la imagen en orden creciente empezando del 0 y finalizando en **pixelYmax** - 1, de forma que el proceso 0 siempre sepa qué fila es la que se debe enviar.

La etiqueta del mensaje podrá contener dos posibles valores: **START_TAG** (0) y **STOP_TAG** (99). Cuando los procesos hijos reciban un mensaje con **STOP_TAG**, no se quedarán a la espera de recibir más filas y finalizarán su ejecución.

En primer lugar, el padre enviará una fila a cada proceso hijo.

```
158         // enviar una fila a cada proceso
159         for (int i=1;i<numproc; ++i){
160             MPI_Send(&fila, 1, MPI_INT, i, START_TAG, MPI_COMM_WORLD);
161             fila++;
162         }
```

Los procesos hijos, una vez recibido el número de fila, comprobarán la etiqueta del mensaje, y, en caso de que no sea **STOP_TAG** empezarán a calcular uno por uno los píxeles de la fila indicada. De esta forma, los hijos estarán constantemente a la espera de nuevas tareas hasta que les llegue un mensaje con la etiqueta **STOP_TAG**.

```
212         // PROCESOS HIJOS
213         while(1){
214             // recibir fila y comprobar la etiqueta
215             MPI_Recv(&pixelY, 1, MPI_INT, 0, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
216             if (status.MPI_TAG == STOP_TAG){
217                 MPI_Finalize();
218                 return 0;
219             }
220 
221             // realizar calculos
222             Cimg = ImMin + pixelY*AltoPixel;
223             for (pixelX=0;pixelX<pixelXmax;pixelX++){…
248             }
249 
250             // enviar resultado al padre
251             MPI_Bsend(&datos[0], 2*pixelXmax, MPI_INT, 0, pixelY, MPI_COMM_WORLD);
252         }
```

Escribirán ambas filas calculadas en la matriz **datos**. La fila de la imagen A ocupará las primeras **pixelXmax** posiciones, mientras que la fila de la imagen B ocupará las últimas **pixelXmax** posiciones. Esta matriz será enviada al proceso padre mediante la

función **MPI_Bsend** para evitar problemas de interbloqueo. Para que el proceso 0 pueda identificar la fila recibida, al enviar los datos el número de fila se especificará en la etiqueta del mensaje.

El proceso 0, tras enviar una tarea a cada hijo, se quedará a la espera de respuestas, enviando una nueva tarea a los procesos inactivos, hasta que se acaben las tareas (todas las filas enviadas).

```
164         // asignar nuevas tareas a procesos inactivos hasta que se acaben
165         while(fila < pixelYmax){
166             MPI_Recv(&datos[0], 2*pixelXmax, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
167             pixelY = status.MPI_TAG;
168             int source = status.MPI_SOURCE;
169             MPI_Send(&fila, 1, MPI_INT, source, START_TAG, MPI_COMM_WORLD);
170             fila++;
171             memcpy(&matriz[pixelY][0], &datos[0], pixelXmax*sizeof(int));
172             memcpy(&matriz2[pixelY][0], &datos[pixelXmax], pixelXmax*sizeof(int));
173         }
```

Cabe destacar que, al recibir un nuevo mensaje, el padre comprobará su etiqueta para saber en qué posiciones de las matrices de las imágenes insertar nuevos datos, así como el origen del mensaje para saber a qué proceso enviar una tarea nueva.

Cuando no queden más filas por enviar, quedarán pendientes por recibir algunas filas. En concreto, una fila de cada proceso, por lo que el proceso 0 esperará a la recepción de esas filas. Una vez recibidas todas las filas, el proceso 0 enviará la señal de finalizar a los procesos hijos.

```
175         // recibir las filas restantes
176         for (int i=1;i<numproc; ++i){
177             MPI_Recv(&datos[0], 2*pixelXmax, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
178             pixelY = status.MPI_TAG;
179             memcpy(&matriz[pixelY][0], &datos[0], pixelXmax*sizeof(int));
180             memcpy(&matriz2[pixelY][0], &datos[pixelXmax], pixelXmax*sizeof(int));
181         }
182 
183         // indicar a los hijos que finalicen la ejecución
184         for (int i=1;i<numproc; ++i){
185             MPI_Send(&fila, 1, MPI_INT, i, STOP_TAG, MPI_COMM_WORLD);
186         }
```

El tiempo que se tarda para completar el cálculo se mide desde que el proceso 0 empieza a enviar las tareas a los procesos hijos, y finaliza una vez enviada la señal STOP a todos los procesos.

```
155         // PROCESO 0
156         Tinicial = MPI_Wtime();
157
158         // enviar una fila a cada proceso
159         for (int i=1;i<numproc; ++i){…
162         }
163
164         // asignar nuevas tareas a procesos inactivos hasta que se acaben
165         while(fila < pixelYmax){…
173         }
174
175         // recibir las filas restantes
176         for (int i=1;i<numproc; ++i){…
181         }
182
183         // indicar a los hijos que finalicen la ejecución
184         for (int i=1;i<numproc; ++i){…
186         }
187
188         Tfinal = MPI_Wtime();
189         Ttotal = Tfinal - Tinicial;
```

Por último, antes de finalizar la ejecución se liberan la memoria de las matrices creadas y el buffer.

```
255     MPI_Buffer_detach(buffer, &sizeBuffer);
256     if(myrank == 0){
257         destruirMatriz(matriz, pixelYmax);
258         destruirMatriz(matriz2, pixelYmax);
259     }
260     free(datos);
261     free(buffer);
```

3. Resultados

3.1. Descripción de las pruebas

Una vez finalizada la implementación del algoritmo paralelo, se han hecho varias pruebas con el fin de comparar ambas versiones. En concreto, se van a comparar los tiempos de ejecución de ambas versiones con diferentes datos de entrada. Además, se van a calcular dos medidas: speedup y eficiencia.

Speedup es el factor de mejora de rendimiento que expresa la aceleración alcanzada con una cierta mejora, en este caso la paralelización del código. El speedup en un sistema con N núcleos se puede definir de la siguiente manera:

$$S(N) = \frac{T(1)}{T(N)},$$

donde $T(1)$ corresponde al código ejecutado con un sólo núcleo (versión secuencial); y $T(N)$ al código paralelo ejecutado con N núcleos.

Por otro lado, la eficiencia es la comparación entre el grado de speedup conseguido frente al valor máximo posible, $S(N) = N$. De esta forma, la eficiencia en un sistema con N núcleos se define como:

$$E(N) = \frac{S(N)}{N} = \frac{T(1)}{N \cdot T(N)}$$

Dado que $1 \leq S(N) \leq N$,tenemos $1/N \leq E(N) \leq 1$. El valor más bajo posible $E(N) = 0$, corresponde a la versión secuencial del sistema, mientras que el valor máximo de eficiencia $E(N) = 1$, se consigue cuando en el código paralelo se aprovechan al máximo todos los núcleos disponibles durante el tiempo completo de ejecución.

En este caso, las pruebas se han realizado con una computadora con 8 núcleos, por lo que los valores de speedup y eficiencia calculados corresponden a $S(8)$ y $E(8)$.

Para la ejecución del algoritmo secuencial se ha utilizado el código proporcionado en `mandelbrot.c`. Para la ejecución de la versión paralela implementada se puede utilizar el atajo `make run`, que ejecuta el siguiente comando:

```
make run ⇔ mpirun -np 8 mandelbrot_mpi
```

Para especificar los datos iniciales se puede utilizar el argumento `ARGS` de la siguiente manera:

```
make run ARGS="pixelXmax pixelYmax IterMax NombreImg dominio"
```

De la misma manera, el argumento `NP` sirve para especificar el número de procesos.

```
make run NP=4 ⇔ mpirun -np 4 mandelbrot_mpi
```

3.2. Resultados obtenidos

En la siguiente tabla se pueden ver los resultados de las pruebas ejecutadas. Cabe destacar que en todo caso las dimensiones de la imagen final son 1024 x 1024, pero se han realizado pruebas con diferentes dominios y diferente número de iteraciones máximas.

Dominio	Iteraciones	Secuencial (s)	Paralelo (s)	Speedup	Eficiencia
0	1000	3,796	0,943	4,025	0,503
0	2000	7,144	1,846	3,870	0,484
0	3000	10,384	2,795	3,715	0,464
0	4000	13,776	3,736	3,687	0,461
1	100	1,949	0,49	3,978	0,497
1	500	7,033	1,96	3,588	0,449
1	1000	13,159	3,561	3,695	0,462
1	2000	27,617	6,85	4,032	0,504
3	2000	7,462	1,884	3,961	0,495
4	3000	4,612	1,15	4,010	0,501
6	1000	7,327	1,902	3,852	0,482
7	4000	10,876	2,801	3,883	0,485
8	2000	18,515	5,251	3,526	0,441

En los resultados se puede ver, en primer lugar, que valor del speedup es cercano al 4. En concreto, el promedio del speedup en las pruebas realizadas es 3,83. Esto significa que la versión paralela del programa, ejecutada con 8 núcleos es 3,83 veces más rápida que la versión secuencial.

El valor promedio de la eficiencia correspondiente es 0,49. Esto quiere decir que los núcleos se aprovechan aproximadamente al 50%. Cabe destacar que este valor nunca podría ser 100%, ya que uno de los procesos únicamente se encarga de asignar tareas al resto de los procesos y no realiza cálculos. Además, se pierde rendimiento en el envío de los mensajes.

3.3. Ejemplos de salida

A continuación, se visualizan algunos de los ejemplos de las imágenes producidas.

En primer lugar, podemos observar el efecto que tiene el número máximo de iteraciones sobre el resultado final. Las siguientes imágenes corresponden a la Imagen A del dominio 1. Se puede observar, con un número de iteraciones mayor la imagen resultante es más exacta y tiene menos zonas de colores intermedios.

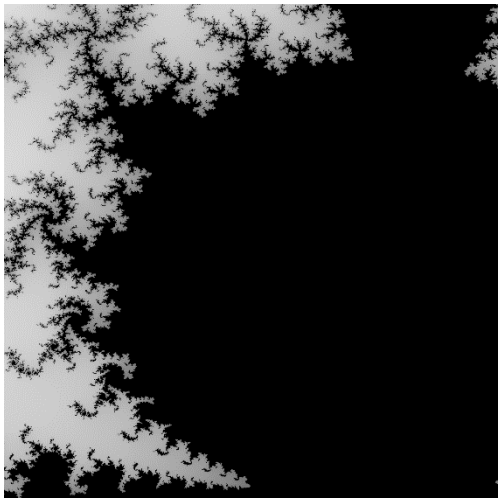

Figura 1: 100 iteraciones

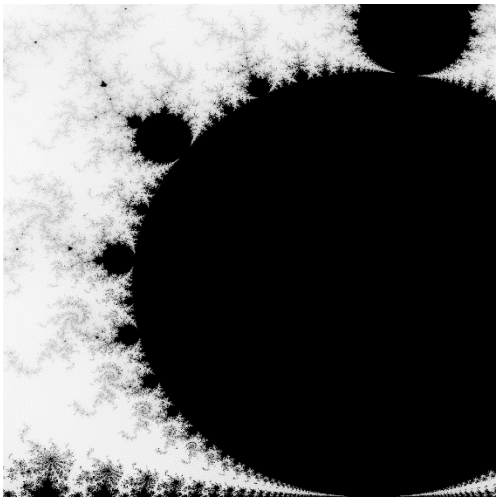

Figura 2: 500 iteraciones

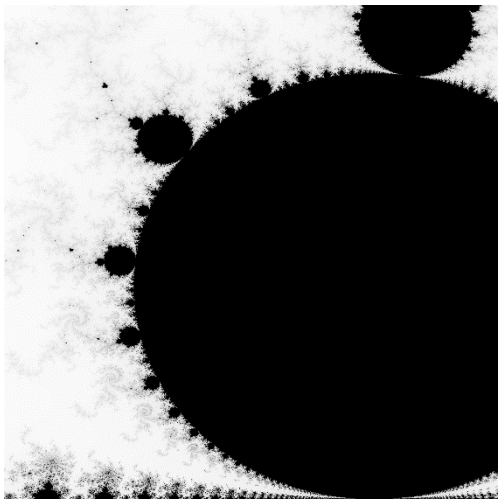

Figura 3: 1000 iteraciones

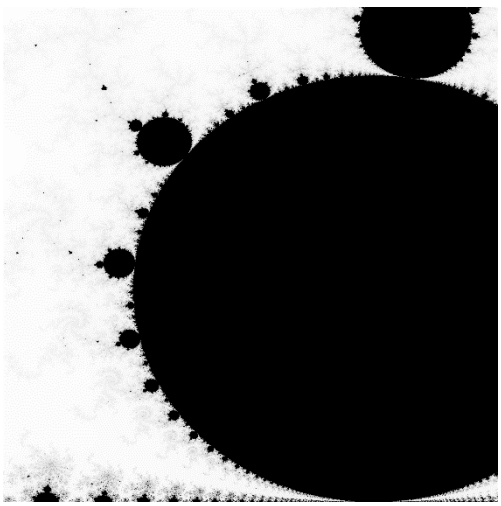

Figura 4: 2000 iteraciones

Por otro lado, en la tabla de los resultados vemos que, en algunos casos con el mismo número de iteraciones y las mismas dimensiones de la imagen, algunos dominios tardan más en completarse que otros. Esto se debe a que para determinar que un píxel pertenece al conjunto Mandelbrot es necesario realizar el número máximo de iteraciones. Por tanto, los dominios en los que pocos puntos forman parte del conjunto tardarán menos en completarse.

Las siguientes imágenes corresponden a la Imagen B de diferentes dominios, todas completadas con 2000 iteraciones. Podemos ver que, efectivamente, para las imágenes con más zonas que forman parte del conjunto Mandelbrot se tardó más en completar los cálculos.

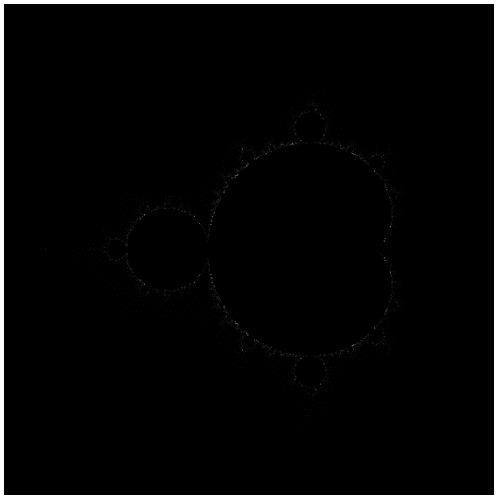

Figura 5: Dominio 0 – 1,85s

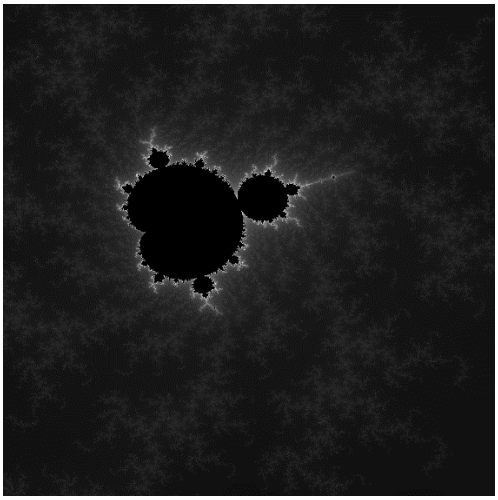

Figura 6: Dominio 3 – 1,89s

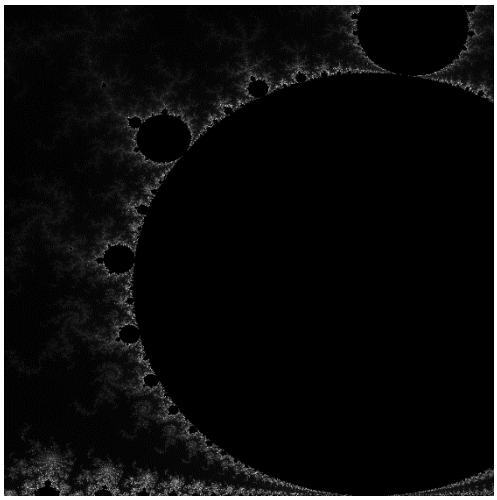

Figura 7: Dominio 1 – 8,85s

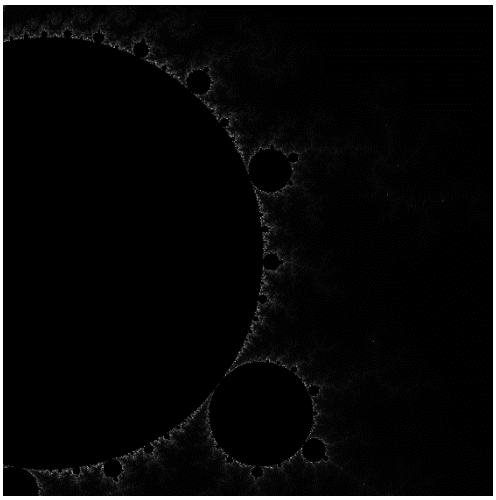

Figura 8: Dominio 8 – 5,25s