Sistemas Concurrentes y Distribuidos

El conjunto de Mandelbrot con MPI

Antonio Coín Castro

26 de diciembre de 2017

El conjunto de Mandelbrot

El conjunto de Mandelbrot es un *conjunto fractal*, digamos M, definido en el plano complejo como sigue: un número c pertene a M si la sucesión $z_0 = 0,\ z_{n+1} = z_n^2 + c$ queda acotada, o equivalentemente, si $|z_n| \leq 2$ sin importar lo grande que sea n. Para más información se puede consultar [1].

Esquema de un algoritmo basado en paso de mensajes

Con el objetivo de construir un algoritmo para dibujar el conjunto de Mandelbrot, empleamos la técnica de paso de mensajes en un esquema maestro-esclavo siguiendo la idea de la diapositiva 59 del tema 3 de la asignatura.

Como primera observación, notamos que trabajaremos en un rectángulo del plano complejo, el equivalente a $[-2, 2] \times [-2, 2]$. En este rectángulo, necesitaremos dividir los complejos en elementos discretos, para así poder convertirlos en píxeles y crear una imagen.

Un esquema del código del proceso maestro es el siguiente:

```
function MAESTRO
    for i := 0 to n_e − 1 do
        send(fila, Esclavo[i]);
    end for
    while quedan filas sin colorear do
        select
        for j := 0 to n_e − 1 when receive(colores, Esclavo[j]) do
            if quedan filas sin mandar then
                send(fila, Esclavo[j]);
            else
                send(fin, Esclavo[j]);
            end if
        end for
    end while
    visualiza(colores);
end function
```

Como vemos, el maestro se encarga de enviar a los esclavos filas completas de números complejos a demanda. Además, recibe las filas ya procesadas, y espera a completar el proceso para visualizar la imagen.

Los esclavos, por su parte, van recibiendo filas de complejos y las devuelven al maestro ya coloreadas mediante un cierto algoritmo:

---

function ESCLAVO ($i : 0 \ldots n_e - 1$)
    receive(mensaje, Maestro);
    while mensaje $\neq$ fin **do**
        colores := calcula_colores(mensaje.fila);
        send(colores, Maestro);
        receive(mensaje, Maestro);
    end while
end function

Implementación en C++ con MPI

Para implementar el algoritmo en el lenguaje C++, utilizamos la librería de paso de mensajes MPI. Para generar la imagen obtenida del conjunto de Mandelbrot empleamos la librería `png++` [2], que se trata de un *wrapper* para la conocida librería `libpng`. Procedemos ahora a analizar los puntos clave del programa, que puede encontrarse en el fichero `mandelbrot.cpp`.

Convertir píxeles en números complejos

Para transformar los píxeles de nuestra imagen en números complejos, realizamos un simple trabajo de discretización sobre el rectángulo definido. Aquí entra en juego la **precisión** (*jump*) con la que queremos discretizar el campo complejo.

```
    const int width() const { return (2 * x_limit / jump) + 1; }
    const int height() const { return (2 * y_limit / jump) + 1; }

    Complex pixelToComplex(int i, int j) {
      int W = width();
      int H = height();
      return Complex(-x_limit + (2 * x_limit) / (W - 1) * i,
                     -y_limit + (2 * y_limit) / (H - 1) * j);
    }
```

Proceso maestro

El proceso maestro se comporta tal y como mostramos en el pseudocódigo, con la salvedad de que junto a cada fila envía un identificador de fila, para facilitar el proceso de reconstrucción de la imagen.

```
  // Send initial rows
  for (i = 0; i < num_slaves; i++) {
    MPI_Send(&i, 1, MPI_INT, i+1, tag_send, MPI_COMM_WORLD);
    MPI_Send(img[i], H, MPI_FLOAT, i+1, tag_send, MPI_COMM_WORLD);
  }
```

```
6
7   i--;
8
9   // Receive colored rows and send new ones dynamically
10  while (colored_rows < W) {
11    MPI_Recv(&row_id, 1, MPI_INT, MPI_ANY_SOURCE, tag_send, MPI_COMM_WORLD, &status);
12    id_slave = status.MPI_SOURCE;
13    MPI_Recv(img[row_id], H, MPI_FLOAT, id_slave, tag_send, MPI_COMM_WORLD, &status);
14    colored_rows++;
15
16    // Decide whether to send more rows or to terminate slave
17    if (i < W - 1) {
18      tag = tag_send;
19      i++;
20    }
21    else {
22      tag = tag_end;
23    }
24
25    MPI_Send(&i, 1, MPI_INT, id_slave, tag_send, MPI_COMM_WORLD);
26    MPI_Send(img[i], H, MPI_FLOAT, id_slave, tag, MPI_COMM_WORLD);
27  }
28
29  // Print resulting image
30  visualize(plane, img);
```

Proceso esclavo

De nuevo, el esclavo funciona de forma muy similar a la mostrada anteriormente, teniendo en cuenta la restricción adicional de que en cada comunicación intervienen dos mensajes: un identificador de fila, y después el contenido de la misma.

```
1   // Receive row_id and row
2   MPI_Recv(&row_id, 1, MPI_INT, id_master, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
3   MPI_Recv(row, H, MPI_FLOAT, id_master, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
4
5   while (status.MPI_TAG != tag_end) {
6     calculate_colors(plane, row, row_id);
7
8     // Send row_id and colored row
9     MPI_Send(&row_id, 1, MPI_INT, id_master, tag_send, MPI_COMM_WORLD);
10    MPI_Send(row, H, MPI_FLOAT, id_master, tag_send, MPI_COMM_WORLD);
11
12    // Receive more rows
13    MPI_Recv(&row_id, 1, MPI_INT, id_master, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
14    MPI_Recv(row, H, MPI_FLOAT, id_master, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
15  }
```

Algoritmo de tiempo de escape

Este algoritmo simplemente realiza iteraciones de la sucesión de Mandelbrot hasta un límite impuesto por el programador, y establece el color del píxel en cuestión en función del número de iteraciones realizadas.

```
for (int j = 0; j < plane.height(); j++) {
  Complex z(0);

  // Scale pixel (i,j) to a complex number in our plane
  Complex c = plane.pixelToComplex(i,j);

  n_iterations = 0;
  while (std::abs(z) <= radius && n_iterations < limit) {
    z = function_mandelbrot(z, c);
    n_iterations++;
  }
```

Para decidir el color de cada píxel disponemos de dos algoritmos, conocimos como *linear coloring* y *continuous coloring*. En el primero pintamos de negro los puntos que consideramos que pertenecen al conjunto, mientras que los puntos que probablemente no pertenecen se van aclarando hasta llegar al blanco. Esto se consigue mediante un escalado de las iteraciones hasta los posibles valores de RGB (0-255).

```
  // Compute color of the pixel from 0 to 255 (linear map)
  float scale = (float) (RGB_MAX-1) / (limit-1);
  img[j] = (limit - n_iterations) * scale;
```

En el segundo algoritmo, calculamos un número en el intervalo $[0, 1)$, y lo multiplicamos por 255 para obtener el color final del píxel. De esta forma se consigue un espacio de colores continuo. La justificación del funcionamiento de este algoritmo se puede consultar en [3].

```
  // A couple of extra iterations to improve the coloring algorithm
  const int EXTRA_ITER = 3;
  for (int k = 0; k < EXTRA_ITER; k++) {
    z = function_mandelbrot(z,c);
    n_iterations++;
  }

  // Compute color of the pixel from 0 to 255 (continuous coloring)
  if (n_iterations == limit + EXTRA_ITER)
    img[j] = 0.0;
  else
    img[j] = (n_iterations - log(log(abs(z)))/log(radius)) / n_iterations * RGB_MAX-1;
```

Mostrar la imagen final

Para construir la imagen, identificamos cada complejo del rectángulo definido con un píxel coloreado, y en la orientación que nos interesa. Además, empleamos una escala para que la imagen resultante tenga siempre un tamaño fijo.

```
png::image<png::rgb_pixel> img_png(img_W, img_H);
int W = plane.width();
int H = plane.height();
float scale_W = (float) (W-1) / (img_W-1);
float scale_H = (float) (H-1) / (img_H-1);

for (int i = 0; i < img_W; i++) {
  for (int j = 0; j < img_H; j++) {
    int x = i * scale_W;
    int y = j * scale_H;
```

```
      int color = img[x][y];
      img_png[j][i] = png::rgb_pixel(color, color, color);
    }
  }

  img_png.write("mandelbrot.png");
```

La imagen resultante se guarda en el directiorio actual como `mandelbrot.png`.

Resultados

Por último, mostramos el resultado de una ejecución con precisión p=0.001. El tiempo de ejecución en mi máquina (Intel Core i5-7200U @ 2.5 GHz) es de unos 90 segundos. El tamaño de imagen elegido es 1024×1024, y el límite de iteraciones es 1000.

Este resultado es muy similar al obtenido con una precisión 10 veces mayor, que tarda en la misma máquina unos 110 minutos (supone evaluar casi 16 millones más de números complejos).

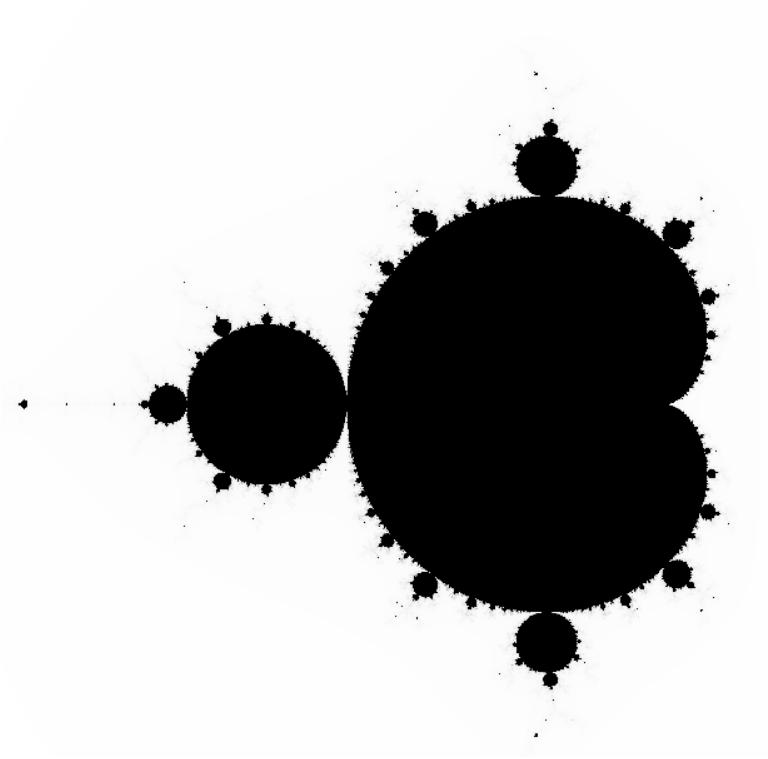

Figura 1: Conjunto de Mandelbrot con coloreado lineal

La misma ejecución con el algoritmo de coloreado continuo resulta en esta imagen:

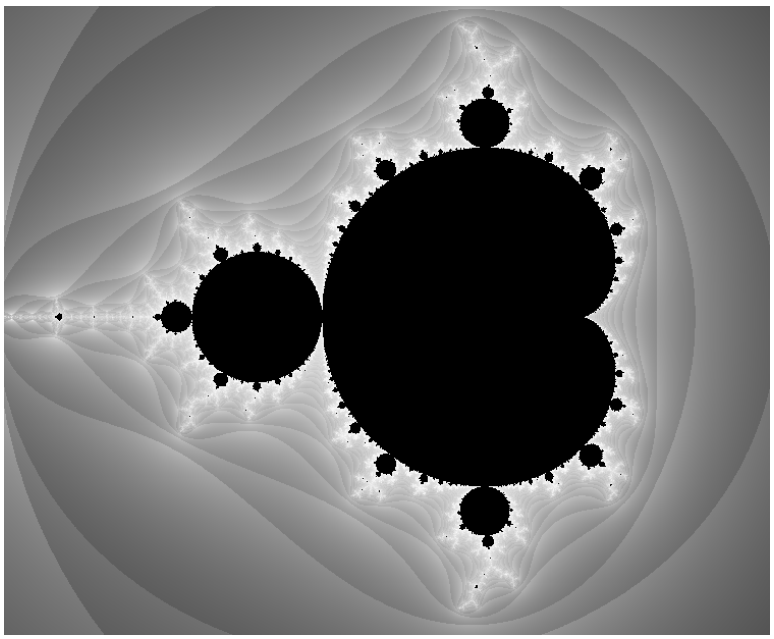

Figura 2: Conjunto de Mandelbrot calculado con coloreado continuo

Compilación

Se proporciona un `makefile` para compilar y ejecutar el programa. Es necesario trabajar bajo un entorno `Linux` y tener instalada las librerías `libpng` y `openMPI`.

Referencias

[1] Mandelbrot set on Wikipedia.

[2] Wrapper for libpng: png++.

[3] Continuous coloring algorithm.