UNIVERSIDAD CATOLICA SAN PABLO - AREQUIPA
FACULTAD DE CIENCIAS DE LA COMPUTACIÓN

Estructura de Datos Avanzados

Informe

Hilbert-curve

Advisor: Erick Mauricio Gomes Nieto
Carlos Eduardo Atencio Torres

Student: Loayza Huarachi Angel Josue

AREQUIPA, JUNE 2023

Contents

1 Introducción

La curva de Hilbert, descubierta por el matemático alemán David Hilbert, la describe a esta como una "curva fractal continúa" con el objetivo de recubrir o recorrer un plano en 2D. Años después, tras la aceptación de dicho descubrimiento, se hicieron mejoras para poder aplicarlo a un espacio 3D y mantener las mismas características.
La curva de Hilbert tiene muchas aplicaciones, entre estas está la criptografía, teoremas matemáticos, etc., pero la más conocida es para el tratamiento de imágenes; este tratamiento también se le conoce como "dithering". Esta técnica ofrece una alternativa al escaneo de una imagen línea a línea, con el objetivo de conseguir difuminados o degradados de mejor calidad; así eliminamos el problema de la adyacencia de puntos que posee un escaneo en líneas horizontales. [2]
Para este trabajo, se presentará la implementación de dicha estructura en el lenguaje de programación C++, pero en una versión paralela usando la librería OpenMP. Se detallará la metodología usada para usar threads con el objetivo de mejorar el código.
Para finalizar, se compartirá el link del repositorio de GitHub para futuras pruebas y mejoras: https://github.com/angel452/HilbertCurve_Paralelo

2 Metodología y Lógica

2.1 Version Lineal

A continuación, se describirá como se llevó a cabo esta implementación lineal, detallando las funciones y la lógica de la curva de Hilbert.
En el código presentado, contamos con 1 función principal que hace la lógica de la curva.

```
1  void hilbert_curve(int n, int angle) {
2      if( n==0 ) return;
3      fprintf(out,"turtle.right(%d)\n", angle);
4
5      hilbert_curve(n-1, -angle);
6
7      fprintf(out,"turtle.forward(%d)\n", STEP_SZ);
8      fprintf(out,"turtle.left(%d)\n", angle);
9
10     hilbert_curve(n-1, angle);
11
12     fprintf(out,"turtle.forward(%d)\n", STEP_SZ);
13
14     hilbert_curve(n-1, angle);
15
16     fprintf(out,"turtle.left(%d)\n", angle);
17     fprintf(out,"turtle.forward(%d)\n", STEP_SZ);
18
19     hilbert_curve(n-1, -angle);
20
21     fprintf(out,"turtle.right(%d)\n", angle);
22 }
```

Este recibe como parámetros iniciales los valores de $1, 2, \ldots$ para "n" que indica el orden de la curva que queremos, y 90 para el ángulo inicial.
La lógica de esta función es llamar recursivamente a la función Hilbert 4 veces en un orden menor (-1). Ya que teóricamente, si pedimos una curva de orden 4, el algoritmo debería crear 4 curvas de orden 3 una tras otra; y para crear una curva de orden 3, el algoritmo deberá crear 4 curvas de orden 2 una tras otra, por eso es que en las 5, 10, 14 y 19, las

líneas parámetro “n” le restamos 1. Esto lo seguiremos haciendo hasta que el orden sea 0 y será el punto de parada como vemos en la línea 2. Pero, como se ve visualmente en esta curva, algunos cuadrantes se tienen que rotar y unir con el siguiente. Por eso es que también le pasamos el parámetro “angle” que me ayudara a controlar dicha rotación. La siguiente imagen ayudará a comprenderlo mejor:

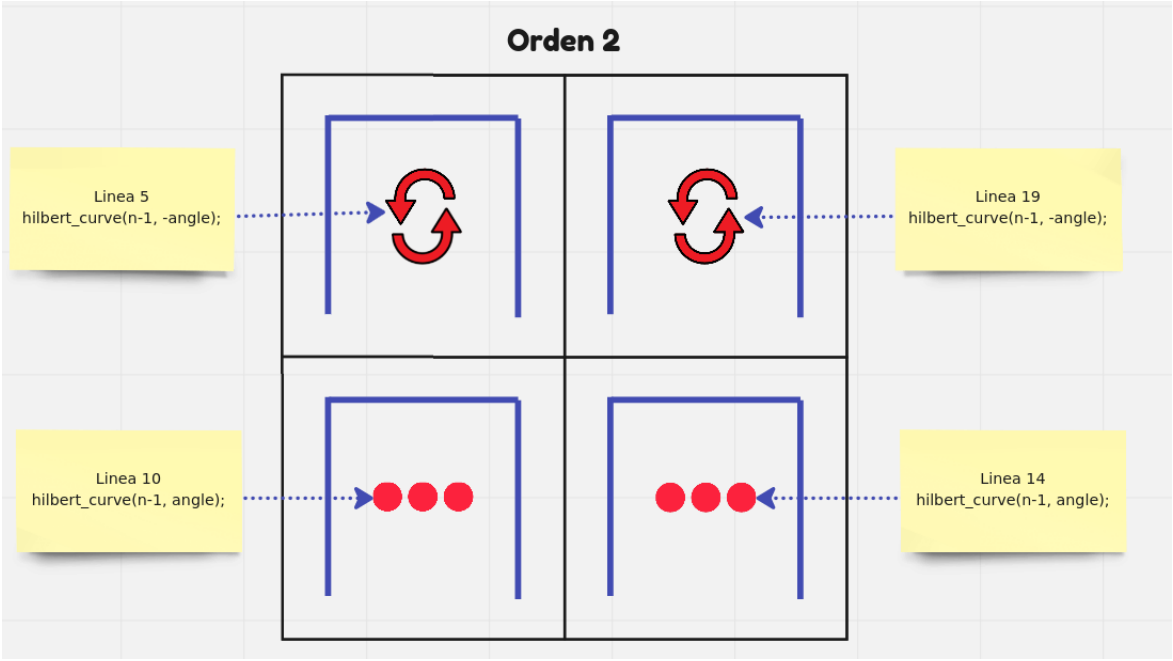

Figure 2.1: Ejemplo 1

Esta parte es esencial entenderla; pues, aprovecharemos esta lógica para hacer la parte paralela.

2.2 Versión Paralela

El primer paso para poder resolver este dilema fue averiguar como repartir la tarea de crear estos recorridos para que al final solo le asignemos un thread a cada uno. Entonces, se usó la lógica de la versión lineal; en el main, en vez de llamar a **hilbert_curve(2,90)**, llamamos manualmente 4 veces a la función Hilbert con un orden -1 y colocando el ángulo en su correcta posición.

```
1 int main(){
2     int order = 4;
3     int angle = 90;
4 
5     hilbert_curve(order-1, -90, id);
6     hilbert_curve(order-1, 90, id);
7     hilbert_curve(order-1, 90, id);
8     hilbert_curve(order-1, -90, id);
9 }
```

Aquí se sigue cumpliendo la misma lógica mostrada en la Figura2.1: Ejemplo 1. Solo quedaría añadirle los enlaces entre cuadrantes para completar con la curva.
Ahora que está dividido, solo es cuestión de asignarle threads; pero aún existe el problema de la concurrencia al escribir información en un espacio de memoria (en este caso el archivo .py). Entonces se usó la metodología de asignarle un espacio de memoria propio a cada thread (en este caso 4 threads), así cada uno escribirá en el lugar que le corresponde sin interferir con el trabajo del otro.
Para cumplir con esto, se usó un array de 4 vectores de tipo string que guardaran las instrucciones del cuadrante 1, 2, 3 y 4, **vector<string> data[4];**. Y en vez de usar fprintf para escribir directamente en el archivo .py, copiamos dicha instrucción en la variable data.

```
void hilbert_curve(int n, int angle) {
    if( n == 0 ){
        return;
    }

    data[idthread].push_back("turtle.right(" + to_string(angle) +
")");

    hilbert_curve(n-1, -angle, idthread);

    data[idthread].push_back("turtle.forward(" + to_string(STEP_SZ) +
")");
    data[idthread].push_back("turtle.left(" + to_string(angle) + ")");

    hilbert_curve(n-1, angle, idthread);

    data[idthread].push_back("turtle.forward(" + to_string(STEP_SZ) +
")");

    hilbert_curve(n-1, angle, idthread);

    data[idthread].push_back("turtle.left(" + to_string(angle) + ")");
    data[idthread].push_back("turtle.forward(" + to_string(STEP_SZ) +
")");

    hilbert_curve(n-1, -angle, idthread);

    data[idthread].push_back("turtle.right(" + to_string(angle) +
")");
}
```

Como vemos, cada instrucción lo guardamos en la posición data[idthread], controlado por el ID del thread que pasamos por parámetro.
Ahora veamos como se implementó el main.

```
int main(int argc, char const *argv[]) {
    int order = 4;
    int angle = 90;

    omp_set_num_threads(4);
    #pragma omp parallel
    {
        int id = omp_get_thread_num();

        #pragma omp for
        for(int i = 0; i < 4; i++){
            if( id == 0 ){
                data[id].push_back("turtle.right(" + to_string(90) +
") #id0");
                hilbert_curve(order-1, -90, id);
            }
            else if( id == 1 ){
                data[id].push_back("turtle.forward(" +
to_string(STEP_SZ) + ") #id1");
                data[1].push_back("turtle.right(-" + to_string(90) +
") #id1");
                hilbert_curve(order-1, 90, id);
            }

            else if( id == 2 ){
                data[id].push_back("turtle.forward(" +
to_string(STEP_SZ) + ") #id2");
                hilbert_curve(order-1, 90, id);
            }

            else if( id == 3 ){
                data[id].push_back("turtle.right(-" + to_string(90) +
") #id3");
```

```
29                  data[id].push_back("turtle.forward(" +
to_string(STEP_SZ) + ") #id3");
30                  hilbert_curve(order-1, -90, id);
31              }
32          }
33      }
34      return 0;
35  }
```

Indicaciones:

- En la línea 5, indicamos al programa que se ejecutara únicamente 4 threads.
- En la línea 8, obtenemos el ID de cada thread que se lanza.
- En la línea 10, el **#pragma omp for**, lanza dichos threads en paralelo
- En la línea 12, 16, 22 y 27, filtramos una tarea específica para cada thread utilizando su ID como controlador.
- Dentro de los 4 condicionales, llamamos a las 4 funciones mencionadas en el anterior código.

3 Resultados

A continuacion se mostrarán los resultados al ejecutar la curva de Hilbert en un orden 4 tanto en la versión paralela como en la versión lineal.

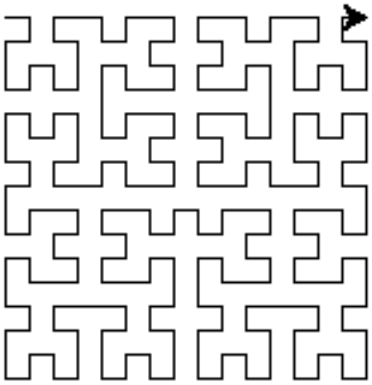

Figure 3.1: Resultado versión Lineal

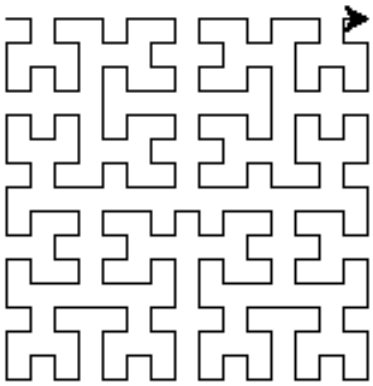

Figure 3.2: Resultado versión paralela

References

[1] 3Blue1Brown. Video: Curva de hilbert. https://www.youtube.com/watch?v=3s7h2MHQtxc.

[2] EcuRed. Curva de hilbert. https://www.ecured.cu/Curva_de_Hilbert#Aplicaciones.

[3] HPC Educationl. Video: Librería openmp. https://www.youtube.com/watch?v=sS3jq1Liv_U.