Prácticas Concurrencia y Distribución (22/23)

Arno Formella, Alba Nogueira Rodríguez, David Ruano Ordás

semana 13 marzo – 17 marzo

Práctica 4: Gestión de hilos para tareas concurrentes

Objetivos: Estudio de dividir tareas concurrentes: dividir tareas realizar trabajo concurrente en una matriz.

1. Este problema sigue estudiando hilos concurrentes independientes. En particular, dada una matriz grande, cada hilo debe realizar un cálculo en una subregión de esta matriz. Para obtener algo visual (facilita la depuración) usamos imágenes.

   El cálculo puede ser un filtro numérico (una operación típica en el procesamiento de imágenes), que reemplaza cada elemento o píxel de una imagen por un nuevo valor que se calcula a partir del valor actual del elemento(píxel y/o de la vecinidad del píxel.

   El problema explora el rendimiento en función del número de hilos y del tipo de distribución de datos.

   Para ello, sigue los siguientes pasos:

2. **Configuración del problema:** crea una clase de hilo con el método `run` vacío `Worker` que extiende de `Thread` (o implementa `Runnable`). También crea una clase principal (por ejemplo, `Problem4`) cuyo método principal `main` crea un *array* (o `ArrayList`) del objeto `Worker`.

   Crea una tercera clase, la clase `Matrix`, que representa una matriz bidimensional. Finalmente, implementa una estrategia para asignar bloques de la matriz a diferentes hilos.

3. **Desarrollo de la clase** `Worker`: Realiza dos operaciones sobre la imagen:

   - La primera operación es una reducción simple de colores, como en el ejemplo de código abajo. También puedes realizar otras operaciones simples, como la conversión a tonos de gris, o la conversión a negativo.
   - La segunda operación a realizar se llama filtro mediano. Es una operación más costosa y necesita acceso a más elementos de la matriz en cada operación.

Dada una matriz, el filtro mediano reemplaza cada elemento de la matriz (i, j) por el promedio calculado con los vecinos en una área cuadrática alrededor del píxel y el propio píxel (ten cuidado cuando te encuentras en los bordes de la matriz). La ecuación para este filtro, J, es:

$$J(i,j) = \frac{1}{(2f+1)^2} \sum_{k=-f}^{f} \sum_{l=-f}^{f} M(i+k, j+l)$$

donde M es la imagen original y f es el tamaño del filtro (experimenta con diferentes tamaños de filtros, $f = 1$ sería una pequeña vecinidad de 3×3 píxeles).

Para el tratamiento de los bordes de la matriz se puede asumir que si $M(i+k, j+l)$ resulta en un elemento fuera de la matriz podemos reflejar las coordenadas en el borde para simplificar el cálculo. Por ejemplo, se asume que $M(-2,-1) = M(2,1)$, e igual a lo largo de los demás bordes (y esquinas!).

4. **Desarrollo de la clase** `Matrix`: esta clase representa el objeto matriz. La clase debe implementar un método con una estrategia particular para distribuir la matriz por los hilos. La estrategia se puede basar en la división de la matriz por filas, por columnas, o por bloques.

   Observa que puedes mantener las imágenes, tanto de entrada como de salida, en variables compartidas, ya que la entrada solo se lee, y en la salida cada hilo escribe en una región diferente (por eso no se producen conflictos de escritura entre ellos).

5. **Desarrollo de la clase principal:** esta clase principal es simplemente responsable de configurar el problema y crear el objeto `Matrix` que ejecutará el filtro dentro de una subregión de la imagen.

6. **Estudio del comportamiento:** ejecuta tu programa para crear diferentes gráficos (plots) de matriz (suficientemente grande) y número de hilos. A partir de los resultados de estos gráficos, ¿notas una diferencia en el rendimiento entre las diferentes estrategias? Si es así, ¿cuál podría ser la causa?

Ayuda: el siguiente código contiene las piezas para leer una imagen desde un fichero y muestra el acceso a los píxeles/elementos.

```
/*
 * (c) copyright 2017-2023 Universidade de Vigo. All rights reserved.
 * formella@uvigo.es, http://formella.webs.uvigo.es
 */

/**
  \file
  \brief working with an image
*/

  // These are the packages we need for the example.
import java.awt.image.*;
import java.io.File;
import javax.imageio.*;

class Image {
  static BufferedImage img;

  public static void main(
    String[] args
  ) {
    System.out.println("starting...");
    if(args.length!=1) {
      System.out.println("please, provide image");
      System.exit(1);
    }
    try {
      File file_in=new File(args[0]);
      img=ImageIO.read(file_in);
      System.out.println("type: "+img.getType());
        // Prepare output image with the same size and type as the input image.
      final int width=img.getWidth();
      final int height=img.getHeight();
      BufferedImage out_img=new BufferedImage(width,height,img.getType());

        // We just make a simple color reduction by setting
        // some lowest bits of each color channel to 0.
      for(int i=0; i<width; i++) {
        for(int j=0; j<height; j++) {
          out_img.setRGB(i,j,img.getRGB(i,j)&0xF0F0F0F0);
        }
      }
        // Write the image always in png-format with a fixed name.
      File file_out=new File("my_image.png");
      ImageIO.write(out_img,"png",file_out);
    }
    catch(Exception E) {
        // Here, we just exit the program, something more useful should
        // be implemented...
      System.exit(1);
    }
      // As always: out last line.
    System.out.println("exiting...");
  }
```