SPRAWOZDANIE VIII

TEORIA WSPÓŁBIEŻNOŚCI

Asynchroniczne wykonanie zadań w puli wątków przy użyciu wzorców Executor i Future

DAWID BIAŁKA

DATA LABORATORIUM 24.11.2020

DATA ODDANIA 01.12.2020

Zadania:

1. Proszę zaimplementować przy użyciu Executor i Future program wykonujący obliczanie zbioru Mandelbrota w puli wątków. Jako podstawę implementacji proszę wykorzystać kod w Javie.
2. Proszę przetestować szybkość działania programu w zależności od implementacji Executora i jego parametrów (np. liczba wątków w puli). Czas obliczeń można zwiększać manipulując parametrami problemu, np. liczbą iteracji (MAX_ITER).

Koncepcja

Dzielimy wykonanie poszczególnych obliczeń dla danych pikseli pomiędzy wątki. Każdy wątek ma za zadanie obliczyć wartości pikseli dla swojego poziomego paska (cała szerokość planszy i wyznaczona wysokość zależna od liczby wątków, każdy dostaje taką samą wysokość oprócz ostatniego, chyba, że wysokość dzieli się przez liczbę wątków bez reszty). Po ukończeniu obliczeń każdy wątek zwraca swój kawałek obrazka, po czym wszystkie te kawałki są łączone w jedną całość i wyświetlane (dla testowania szybkości działania programu wyświetlanie zostaje wyłączone). Sprawdzamy szybkość działania programu w zależności od implementacji Executora i liczby wątków.

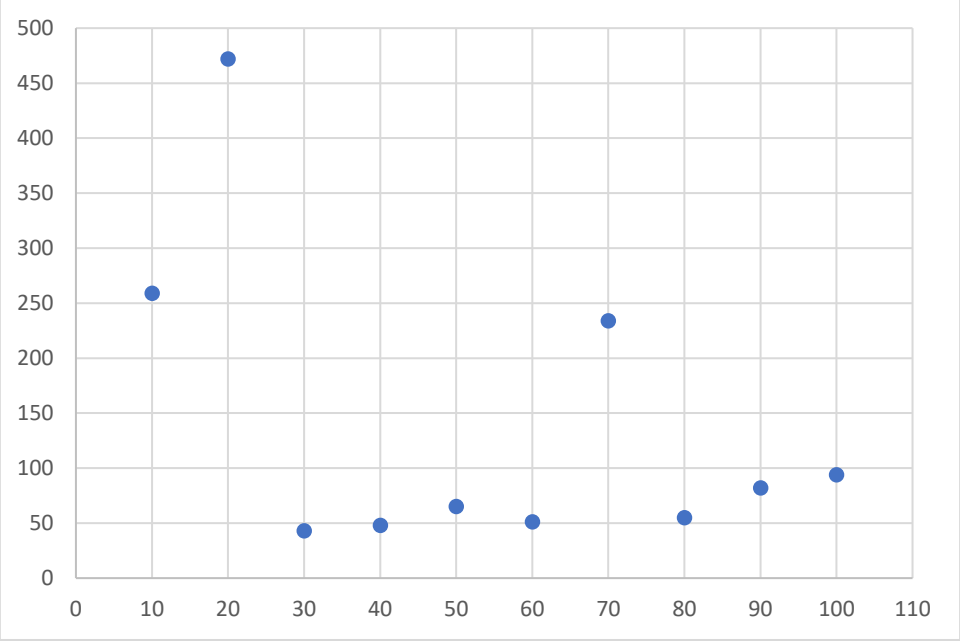
Implementacja i wyniki

```
public class Main{
    public static void main(String[] args){
        int width = 800;
        int height = 600;
        //MandelbrotDrawer mandelbrotDrawer = new MandelbrotDrawer(width, height);
        int numOfThreads = Integer.parseInt(args[0]) * 10;
        ExecutorService pool = Executors.newFixedThreadPool(numOfThreads);
        //ExecutorService pool = Executors.newCachedThreadPool();
        //ExecutorService pool = Executors.newWorkStealingPool();
        //ExecutorService pool = Executors.newSingleThreadExecutor();

        List<Future<BufferedImage>> futureList = new ArrayList<>();
        Callable<BufferedImage> callable;

        for(int i=1; i<=numOfThreads; i++) {
            if(height % numOfThreads != 0 && i == numOfThreads) {
                callable = new MandelbrotCallable(height/numOfThreads*i - height/numOfThreads,
                                                  height,800,150);
            } else {
                callable = new MandelbrotCallable(height/numOfThreads*i - height/numOfThreads,
                                                  height/numOfThreads*i,800,150);
            }

            Future<BufferedImage> future = pool.submit(callable);
            futureList.add(future);
        }

        int i = 1;
        long start = System.nanoTime();
        try {
            for(Future<BufferedImage> future : futureList) {
                //mandelbrotDrawer.addImage(future.get(), height/numOfThreads*i - height/numOfThreads);
                BufferedImage im = future.get();
                im.getGraphics();
                i++;
            }
        } catch (ExecutionException | InterruptedException e) {
            e.printStackTrace();
        }

        long end = System.nanoTime();
        long timeElapsed = end - start;

        try {
            FileWriter myWriter = new FileWriter("results.txt", true);
            myWriter.write(String.valueOf(timeElapsed/1000000) + "\n");
            myWriter.close();
        } catch (IOException e) {
            System.out.println("An error occurred.");
            e.printStackTrace();
        }

        System.exit(1);
    }
}
```

```java
public class MandelbrotCallable implements Callable {

   private final int MAX_ITER = 6000;
   private double zx, zy, cX, cY, tmp;
   private final int heightStart;
   private final int heightEnd;
   private final int width;
   private final double ZOOM;
   private BufferedImage I;

   public MandelbrotCallable(int heightStart, int heightEnd, int width, int ZOOM) {
      this.heightStart = heightStart;
      this.heightEnd = heightEnd;
      this.width = width;
      this.ZOOM = ZOOM;
      I = new BufferedImage(width, heightEnd-heightStart, BufferedImage.TYPE_INT_RGB);
   }

   @Override
   public BufferedImage call(){
      for (int y = heightStart; y < heightEnd; y++) {
         for (int x = 0; x < width; x++) {

            zx = zy = 0;
            cX = (x - 400) / ZOOM;
            cY = (y - 300) / ZOOM;
            int iter = MAX_ITER;
            while (zx * zx + zy * zy < 4 && iter > 0) {
               tmp = zx * zx - zy * zy + cX;
               zy = 2.0 * zx * zy + cY;
               zx = tmp;
               iter--;
            }
            I.setRGB(x, y-heightStart, iter | (iter << 8));
         }
      }
      return I;
   }

}
```

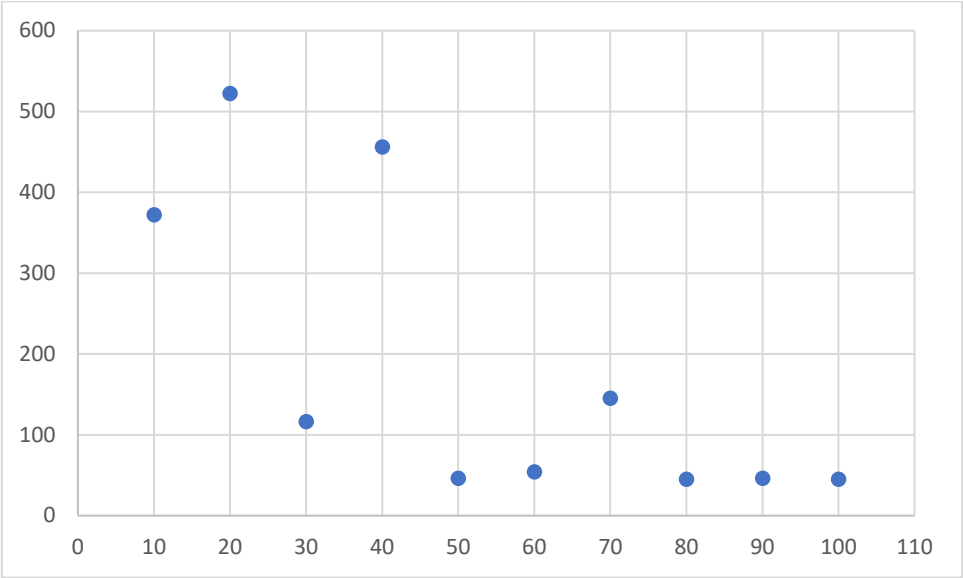
```java
public class MandelbrotDrawer extends JFrame {
    private final int MAX_ITER = 5700;
    private final double ZOOM = 150;
    private BufferedImage combined;
    private double zx, zy, cX, cY, tmp;

    public MandelbrotDrawer(int width, int height){
        super("Mandelbrot Set");
        setBounds(100, 100, width, height); //800x600
        setResizable(false);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setVisible(true);
        combined = new BufferedImage(getWidth(), getHeight(), BufferedImage.TYPE_INT_RGB);
    }

    public void addImage(BufferedImage imageToAdd, int height) {
        Graphics g = combined.getGraphics();
        g.drawImage(imageToAdd, 0, height, null);
        this.repaint();
    }


    @Override
    public void paint(Graphics g) {
        g.drawImage(combined, 0, 0, this);
    }
}
```

Implementacja newFixedThreadPool



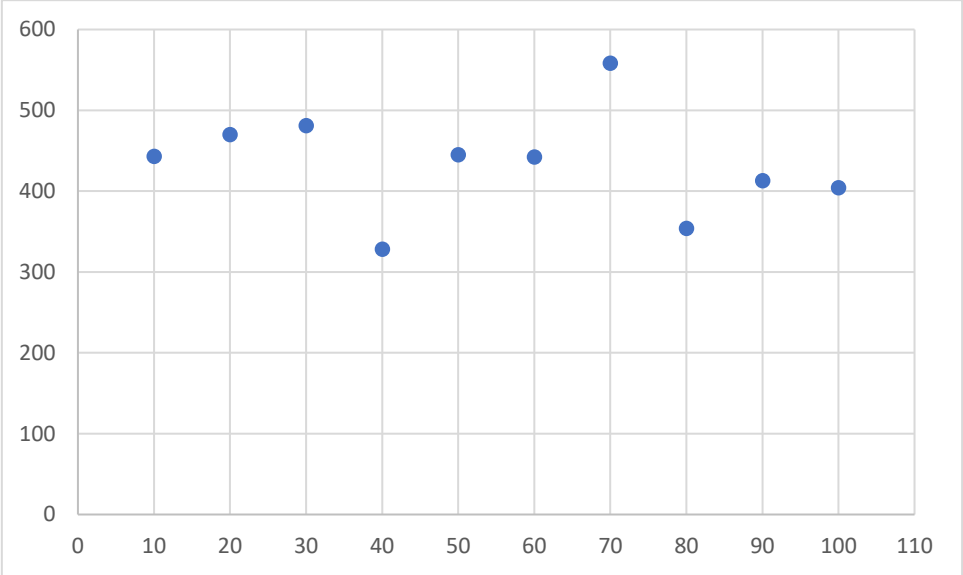
Wykres zależności czasu działania programu od liczby wątków

Implementacja newCachedThreadPool



Wykres zależności czasu działania programu od liczby zadań

Implementacja newWorkStealingPool



Wykres zależności czasu działania programu od liczby zadań

Implementacja newSingleThreadExecutor

Licz.wątków	Czas
1	959
1	941
1	919
1	924
1	918
1	924
1	1011
1	992
1	909
1	910

Wnioski

Dla implementacji newFixedThreadPool wraz ze wzrostem liczby wątków czas wykonania spada, ale powyżej 30 zadań czas wykonania nie zmienia się za wiele Powyżej 80 zadań czas zaczyna rosnąć. Tutaj liczba zadań jest równa liczbie utworzonych wątków. Zbyt duża liczba wątków powoduje spadek wydajności.

Dla implementacji newCachedThreadPool jest podobna tendencja ale nie widzimy żadnego wzrostu czasu wykonania dla dużych liczb zadań. Jest to spowodowane tym, że ta implementacja w przeciwieństwie do newFixedThreadPool nie tworzy dokładnie tyle wątków ile jest zadań tylko na bieżąco dostosowuje liczbę wątków. Tworzy tyle ile potrzeba, ale będzie wykorzystywał poprzednie wątki jeśli te będą dostępne. Ten rodzaj implementacji zwiększa wydajność programów, które wykonują dużo krótkich asynchronicznych zadań, tak jak w naszym przypadku.

Dla implementacji newWorkStealingPool uzyskujemy bardzo podobne czasy bez względu na liczbę zadań. Wykorzystuje wszystkie dostępne procesory dla JVM jako swój „target parallelism level”, dlatego zmiana liczby zadań nie powoduje zmiany czasu trwania programu.

Dla implementacji newSingleThreadExecutor mamy jeden wątek i zadania są wykonywane po kolei. Widzimy, że czas działania dla tej implementacji jest najgorszy, co jest oczywiście zgodne z intuicją.

Bibliografia

Z. Weiss, T. Gruźlewski, Programowanie współbieżne i rozproszone. WNT, Warszawa 1993.

http://home.agh.edu.pl/~funika/tw/lab8/

https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Executor.html

https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Executors.html#newFixedThreadPool-int-

https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Executors.html#newCachedThreadPool--

https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Executors.html#newSingleThreadExecutor--

https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Executors.html#newWorkStealingPool-int-

http://rosettacode.org/wiki/Mandelbrot_set#Java

https://stackoverflow.com/questions/25169635/usage-of-generic-in-callable-and-future-interface