Laboratorium Nr 8
"Asynchroniczne wykonanie zadań w puli wątków przy użyciu wzorców Executor i Future"
Radosław Kopeć
24.11.2020

I. Zadanie 1

1. Treść zadania

1. Proszę zaimplementować przy użyciu Executor i Future program wykonujący obliczanie zbioru Mandelbrota w puli wątków.
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).

2. Koncepcja rozwiązania

Krok 1: Utworzenie klasy UnitWork implementującej interfejs Callable, będzie ona reprezentować jedno zadanie z puli wątków i będzie obliczała zbiór Mandelbrota dla przypisanego do niej punktu układu współrzędnych. Będzie zwracać trójkę (punkt x, punkt y, wynik obliczeń). Będzie jej potrzebna maksymalna liczba operacji.

Krok 2: Utworzenie klasy CalcualteMandelbrot która będzie uruchamiała wszystkie zadania w puli wątków. Jako swoje atrybuty będzie przyjmowała rozmiar puli wątków, oraz maksymalna liczbę iteracji. Metoda "calcualte" tej klasy zwróci nam pomiar czasu dla określonych parametrów.

Krok 3: Utworzenie klasy MeasureExecutorService w której będzie znajdowała się pętla główna programu mierząca czasy dla konkretnych parametrów. Będzie ona rysowała wykresy.

3. **Implementacja:**

Klasa UnitWork:

```
public class UnitWork implements Callable<Three<Integer>> {
    private final int iterations;
    private final int x;
    private final int y;

    public UnitWork(int iterations, int x, int y) {
        this.iterations = iterations;
        this.x = x;
        this.y = y;
    }
```

```
    @Override
    public Three<Integer> call() throws Exception {
        double zx = 0;
        double zy = 0;
        double tmp = 0;
        double ZOOM = 150;
        double cX = (x - 400) / ZOOM;
        double cY = (y - 300) / ZOOM;
        int iter = iterations;
        while (zx * zx + zy * zy < 4 && iter > 0) {
            tmp = zx * zx - zy * zy + cX;
            zy = 2.0 * zx * zy + cY;
            zx = tmp;
            iter--;
        }
        return new Three<Integer>(x,y, z: iter | (iter << 8));
    }
```

Klasa CalculateMandelbrot:

```java
public class CalculateMandelbrot {

    private final int poolSize;
    private final int iterations;
    private final int width;
    private final int height;

    public CalculateMandelbrot(int poolSize, int maxIterations, int width, int height) {
        this.poolSize = poolSize;
        this.iterations = maxIterations;
        this.width = width;
        this.height = height;
    }
```

```java
public long calculate(){
    ExecutorService executorService = Executors.newFixedThreadPool(poolSize);
    List<Future<Three<Integer>>> result;
    List<UnitWork> tasks = new ArrayList<>();

    for(int i=0;i<width;i++){
        for(int j=0;j<height;j++){
            tasks.add(new UnitWork(iterations,i,j));
        }
    }

    long startTime = System.nanoTime();

    try {
        result = executorService.invokeAll(tasks);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    return System.nanoTime() - startTime;
}
```

Klasa Three:

```java
public class Three<T1> {

    T1 x;
    T1 y;
    T1 z;

    public Three(T1 x, T1 y, T1 z) {
        this.x = x;
        this.y = y;
        this.z = z;
```

Klasa MeasureExecutorService:

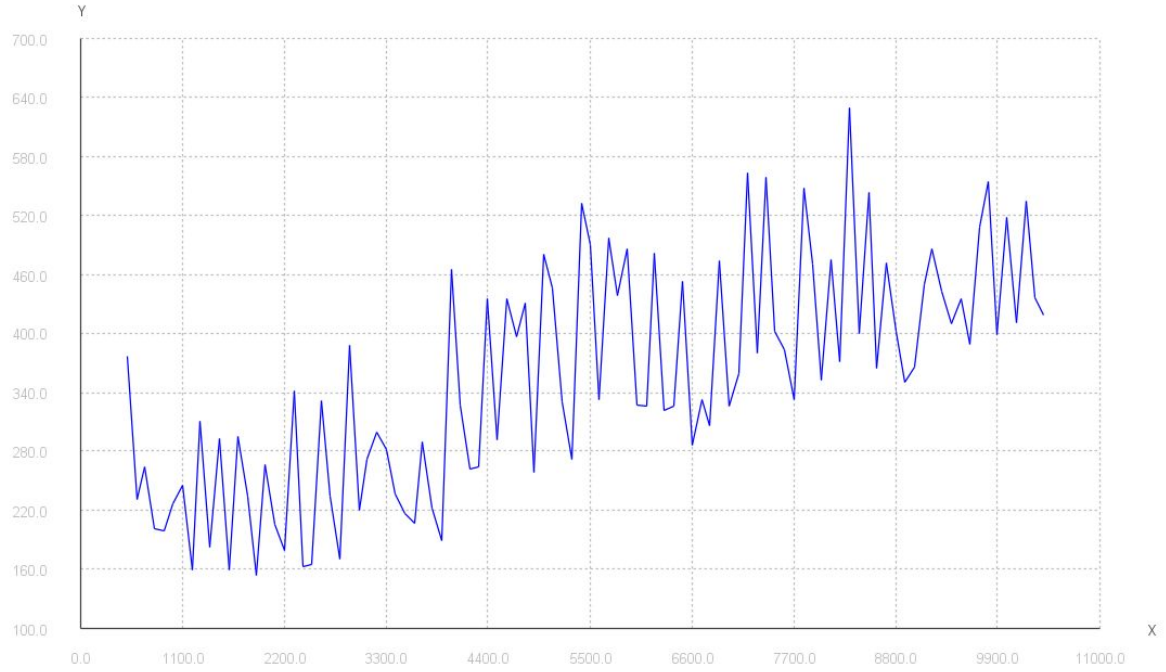
```java
public class MeasureExecutorService {

    public static void main(String[] args) {
        int width = 800;
        int height = 600;

        int iterStep = 100;
        int iterStartIteration = 500;
        int iterSteps = 100;
        int iterThreadPool = 10;
        double[] iterResult = new double[iterSteps];
        double[] iterX = new double[iterSteps];

        int threadPoolStep = 1;
        int threadPoolStart = 1;
        int threadPoolSteps = 100;
        int threadPoolMaxIter = 10000;
        double[] threadResult = new double[100];
        double[] threadX = new double[iterSteps];
```

```
        for(int i=0;i<iterSteps;i++){
            iterX[i] = iterStartIteration;
            CalculateMandelbrot calculateMandelbrot = new CalculateMandelbrot(iterThreadPool,iterStartIteration,width,height);
            iterResult[i] = (double) (calculateMandelbrot.calculate()/1000000);
            iterStartIteration += iterStep;
            System.out.println("Iteration progress: " + (double) (i + 1)/iterSteps);
        }

        for(int i=0;i<threadPoolSteps;i++){
            threadX[i] = threadPoolStart;
            CalculateMandelbrot calculateMandelbrot = new CalculateMandelbrot(threadPoolStart,threadPoolMaxIter,width,height);
            threadResult[i] = (double) (calculateMandelbrot.calculate()/1000000);
            threadPoolStart += threadPoolStep;
            System.out.println("Thread progress: " + (double) (i + 1)/threadPoolSteps);
        }

        draw2DPlot(iterX,iterResult, name: "", plotName: "The time in function of max iterations");
        draw2DPlot(threadX,threadResult, name: "", plotName: "The time in function of the thread pool size ");
    }
```

```
private static void draw2DPlot(double[] arguments, double[] values, String name, String plotName){

    Plot2DPanel plot = new Plot2DPanel();
    plot.addLinePlot(name, arguments, values);
    JFrame frame = new JFrame(plotName);
    frame.setContentPane(plot);
    frame.setVisible(true);
}
```

4. Wyniki i wnioski

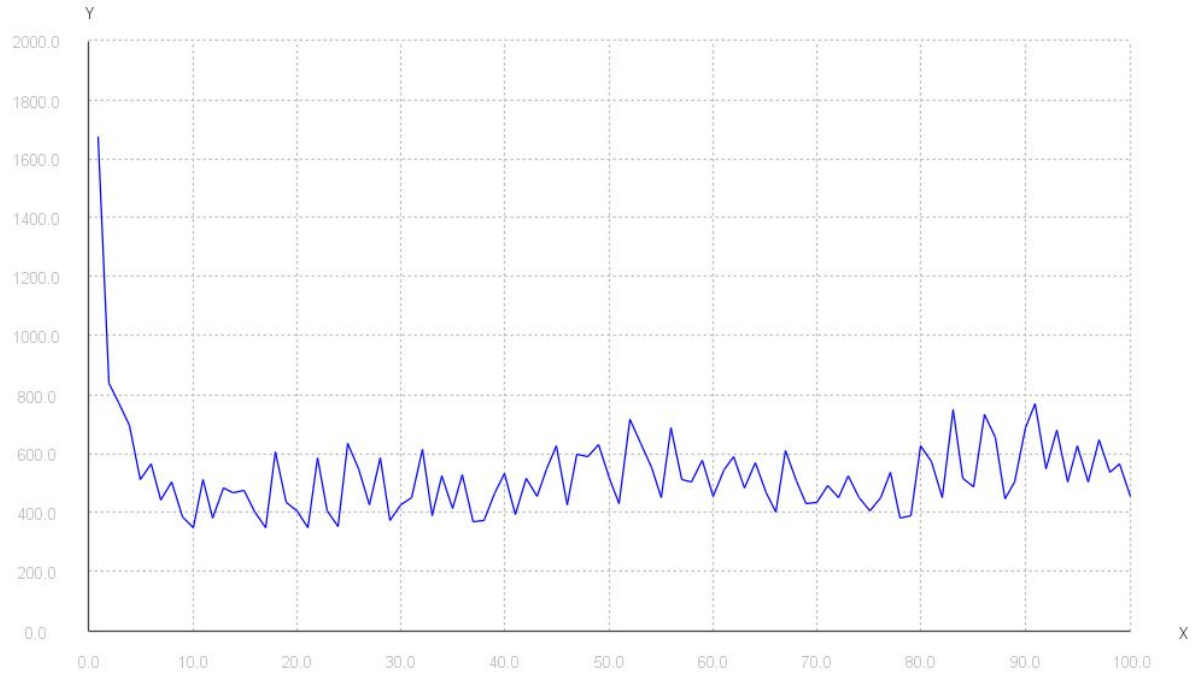
Wykres zależności czasu od liczby iteracji:



Jak widać wraz ze wzrostem liczby operacji rośnie czas wykonania całego zadania. Wyniki te oscylują ponieważ, są wykonywane współbieżnie w puli kilku wątków, a więc ich

czas nie jest zawsze ten sam (ze względu na wiele czynników podczas przetwarzania współbieżnego).

Wykres zależności czasu od liczby wątków w puli:



Na powyższym wykresie widać, że dla małej liczby wątków czas jest bardzo wysoki. Wraz ze wzrostem liczby wątków czas wykonania maleje, aż pozostaje stały. Czas wyrównuje się ponieważ liczba wykonywanych wątków jednocześnie w CPU komputera jest ograniczona i osiągnęła swój limit. Wyniki oscylują, ponieważ wykonywanie współbieżne zawsze ma w sobie jakiś czynnik losowy powodujący drobne różnice czasu (uzależnione to jest od warunków, obciążenia procesora przez inne procesy, zwalnianiem pamięci).

Główny wniosek płynący z badań: używanie ExecutorService z rozsądną pulą wątków daje duże korzyści obliczeniowe.

5. Bibliografia

http://rosettacode.org/wiki/Mandelbrot_set#Java
http://home.agh.edu.pl/~funika/tw/lab8/
https://www.baeldung.com/java-executor-service-tutorial
http://www.algorytm.org/fraktale/zbior-mandelbrota.html
https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Future.html
https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Executors.html

https://docs.oracle.com/javase/tutorial/essential/concurrency/executors.html

Kopeć Radosław