Teoria Współbieżności

Sprawozdanie z laboratorium 8.

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

Joanna Bryk, grupa środa 17:50

1. Wstęp

Zadanie polegało na implementacji programu wykonującego obliczenia zbioru Mandelbrota w puli wątków. Używanymi interfejsami były *Executor* i *Future*. Następnie należało porównać czas działania implementacji *Executor*.

2. Implementacja

Postanowiłam podzielić wysokość obrazu na liczbę zadań (wątków) przechowywaną w zmiennej *NUM_OF_THREADS*. Kiedy dany wątek obliczy swoją część obrazu, dodaje ją do reszty. W tym celu stworzyłam tablice *start_values*, w której przechowuje, w którym miejscu zaczyna się obraz tworzony przez dany wątek.

```
public class Mandelbrot extends JFrame {
    BufferedImage I;

    public Mandelbrot() {
        super("Mandelbrot Set");
        setBounds(100, 100, 800, 600);
        setResizable(false);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        I = new BufferedImage(getWidth(), getHeight(),
BufferedImage.TYPE_INT_RGB);
    }

    @Override
    public void paint(Graphics g) {
        long start = System.nanoTime();
        int NUM_OF_THREADS = 8;
        ExecutorService pool =
Executors.newFixedThreadPool(NUM_OF_THREADS);
//        ExecutorService pool = Executors.newSingleThreadExecutor();
//        ExecutorService pool = Executors.newCachedThreadPool();
//        ExecutorService pool = Executors.newWorkStealingPool();


        List<Future<BufferedImage>> list_future = new ArrayList<>();
        Callable<BufferedImage> callable;
        int[] start_values = new int[NUM_OF_THREADS];
        int j = 0;
        for (int i = 0; i < getHeight(); i += getHeight() / NUM_OF_THREADS)
{
            callable = new MCallable(i, getHeight() / NUM_OF_THREADS,
```

```
getWidth());
            start_values[j] = i;
            j += 1;
            list_future.add(pool.submit(callable));
        }

        j = 0;
        for (Future<BufferedImage> f : list_future) {
            try {
                BufferedImage image = f.get();
                g.drawImage(image, 0, start_values[j], this);
                j += 1;
            } catch (ExecutionException | InterruptedException e) {
                e.printStackTrace();
            }
        }
        pool.shutdown();
        long end = System.nanoTime();

        System.out.println((end - start) / 1000000);
    }

    public static void main(String[] args) {
        new Mandelbrot().setVisible(true);
    }
}
```

Wartość MAX_ITER ustawiłam na 5000.

```
import java.awt.image.BufferedImage;
import java.util.concurrent.Callable;

public class MCallable implements Callable<BufferedImage> {
    int endHeight;
    int width;
    int startHeight;
    final int MAX_ITER = 5000;

    public MCallable(int startHeight, int height, int Width) {
        this.startHeight = startHeight;
        this.endHeight = startHeight + height;
        this.width = Width;
    }

    @Override
    public BufferedImage call() {
        BufferedImage image = new BufferedImage(this.width, this.endHeight
- startHeight, BufferedImage.TYPE_INT_RGB);
        for (int y = startHeight; y < endHeight; y++) {
            for (int x = 0; x < width; x++) {
                double zy;
                double zx = zy = 0;
                double ZOOM = 150;
                double cX = (x - 400) / ZOOM;
                double cY = (y - 300) / ZOOM;
                int iter = this.MAX_ITER;
                while (zx * zx + zy * zy < 4 && iter > 0) {
                    double tmp = zx * zx - zy * zy + cX;
                    zy = 2.0 * zx * zy + cY;
                    zx = tmp;
                    iter--;
```

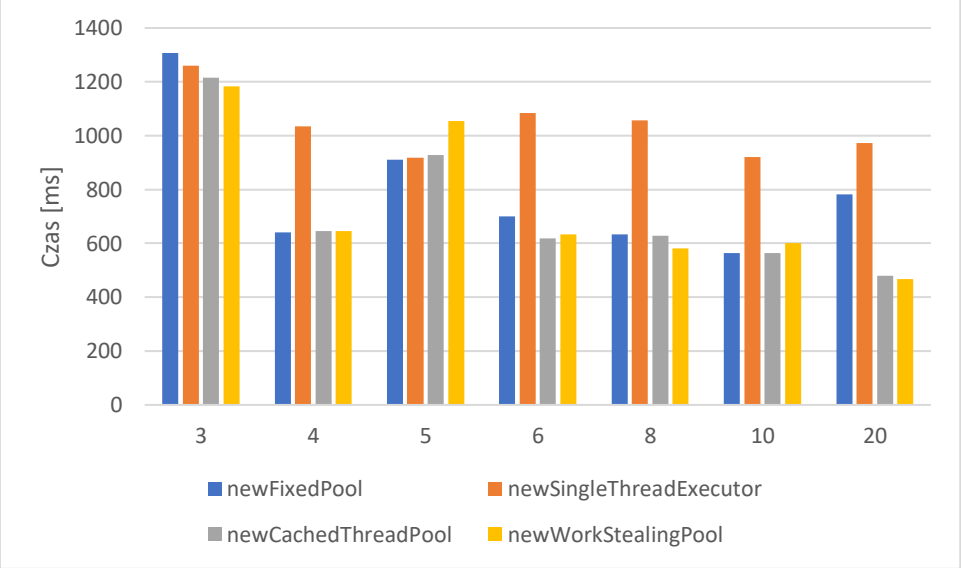
```
                }
                image.setRGB(x, y - startHeight, iter | (iter << 8));
            }
        }
        return image;
    }
}
```

W programie wykorzystałam cztery możliwe implementacje interfejsu *Executor*:

- *newSingleThreadPool* – tworzy jednowątkowy *Executor*, zadania wykonywane są po sobie, jedno zadanie w danym momencie.
- *newFixedThreadPool* – tworzy pulę stałej liczby wątków, które są wielokrotnie używane.
- *newCachedThreadPool* – tworzy pulę, która tworzy nowe wątki, jeśli takie są potrzebne. Używa także wcześniejszych wątków, jeśli są wolne. Wątki nieużywane przez 60s są usuwane.
- *newWorkStealingPool* – tworzy pulę wątków. Jeśli dany wątek skończył pracę, może „ukraść" pracę z kolejki innego wątku.

3. Wykresy

W programie zmieniałam zarówno rodzaj implementacji interfejsu *Executor* jak i liczbę wątków (na ile części należy podzielić obraz).



Wykres 1 Czas wykonania programu dla różnych implementacji i różnych wartości NUM_OF_THREADS

4. Podsumowanie

Analizując powstały wykres, stwierdzam, że najdłużej działała implementacja *newSingleThreadExecutor*. Wynik ten ma sens, biorąc pod uwagę, że w tej opcji działa tylko jeden wątek, który wykonuje wszystkie części obrazu.

Opcja *newWorkStealingPool* utrzymuje czas wykonywania na stałym poziomie, niezależnie od tego, ile wynosi zmienna *NUM_OF_THREADS*. Myślę, że ukazuje to cechy tej implementacji opisane w punkcie 2. Wyniki uzyskane dla wersji z *newCachedThreadPool* są podobne jak w przypadku *newFixedPool* z tą rożnicą, że czas programu pierwszej z tych implementacji nie wzrasta przy 20 częściach, podczas gdy w *newFixedPool* możemy zaobserwować wzrost wartości. Te trzy ostatnie implementacje mają wspólne cechy – najgorzej działały dla liczby 3 a dość dobre wyniki dla liczby 4.

5. Bibliografia

- http://rosettacode.org/wiki/Mandelbrot_set#Java
- https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Executors.html
- https://docs.oracle.com/javase/tutorial/essential/concurrency/executors.html
- https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Future.html
- https://dzone.com/articles/diving-into-java-8s-newworkstealingpools