Laboratorium VIII - Teoria współbieżności

Weronika Szybińska, 12.12.2022

Treść 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).

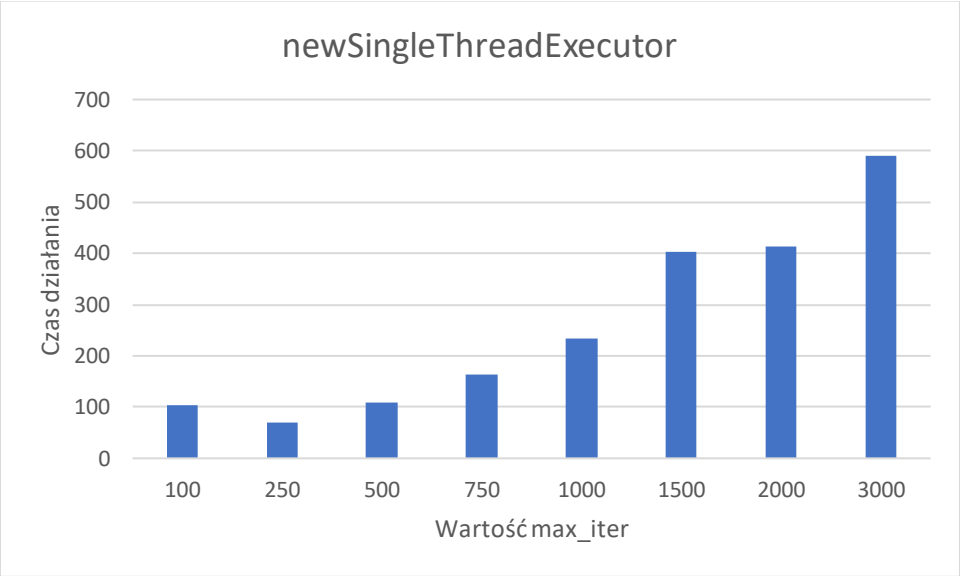
Rozwiązanie zadania:

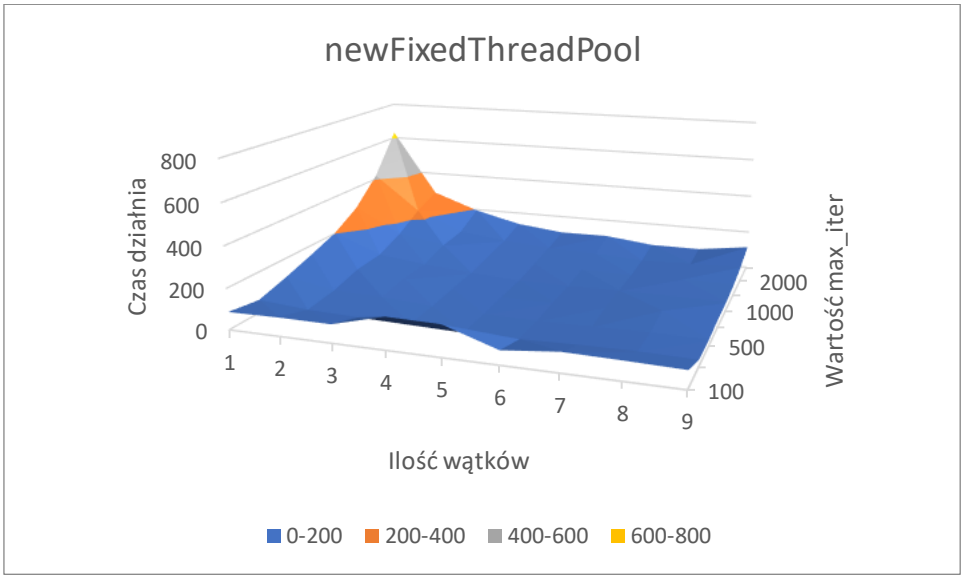
1. Do rozwiązania zadania pierwszego wykorzystana implementacja kodu obliczającego zbiór Mandelbrota. Główne obliczenia przeniesione zostały do nowej klasy MandelbrotCalculations aby możliwe było odpalenie ich w osobnym wątku. Klasa ta w konstruktorze dostaje zmienne: max_iter, która określa maksymalną ilość iteracji w trakcie obliczeń, width która określa szerokość tworzonego obrazu oraz height która określa obecną wysokość obrazu na której są obecnie wykonywane obliczenia. Zwraca ona tablice wartości rgb obrazu, który tworzymy dla danej szerokości i wysokości. Zwracane dane opakowywane są w obiekt Future. W klasie głównej Mandelbrot wywoływana jest motoda call() klasy MandelbrotCalculations za pomocą obiektu ExecutorService i metody sumbit(). Następnie na podstawie danych znajdujących się w tablicy futures (zawierających obiekty Future z tablicami zawierającymi wartości rgb), utworzony zostaje obraz. Poniżej przedstawiona jest implementacja programu.

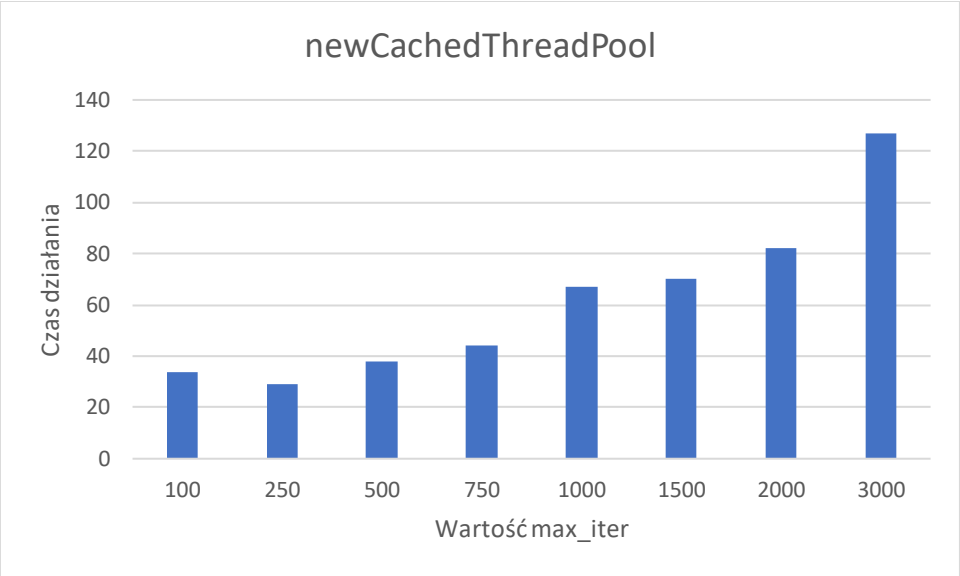
```java
class MandelbrotCalculations implements Callable<Integer[]> {

    private final double ZOOM = 150;
    private final int max_iter;
    private final int width;
    private final int height;

    public MandelbrotCalculations(int max_iter, int width, int height){
        this.max_iter = max_iter;
        this.width = width;
        this.height = height;
    }

    @Override
    public Integer[] call(){
        Integer[] x_rgb = new Integer[width];
        for (int x = 0; x < width; x++) {
            double zy;
            double zx = zy = 0;
            double cX = (x - 400) / ZOOM;
            double cY = (height - 300) / ZOOM;
            int iter = 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--;
            }
            x_rgb[x] = iter | (iter << 8);
        }
        return x_rgb;
    }
}
```

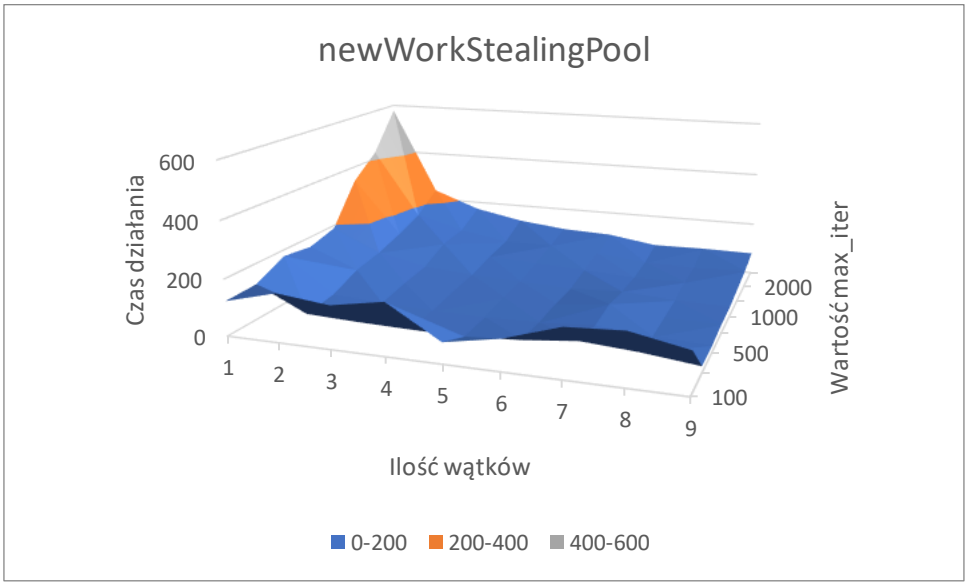
```java
public class Mandelbrot extends JFrame {

    private final int[] MAX_ITER = {100,250,500,750,1000,1500,2000,3000};
    private final BufferedImage I;

    public Mandelbrot() throws ExecutionException, InterruptedException {

        super( title: "Mandelbrot Set");
        setBounds( x: 100,  y: 100,  width: 800,  height: 600);
        setResizable(false);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        I = new BufferedImage(getWidth(), getHeight(), BufferedImage.TYPE_INT_RGB);
        ExecutorService executorService = Executors.newFixedThreadPool( nThreads: 9);
        Future<Integer[]>[] futures = new Future[getHeight()];

        for (int y = 0; y < getHeight(); y++) {
            var task = new MandelbrotCalculations(MAX_ITER[7], getWidth(), y, I);
            futures[y] = (executorService.submit(task));
        }

        for( int i=0; i < futures.length; i++){
            Integer[] element = futures[i].get();
            for(int j=0; j < element.length; j++){
                I.setRGB(j, i, element[j]);
            }
        }
    }

    @Override
    public void paint(Graphics g) {
        g.drawImage(I,  x: 0,  y: 0,  observer: this);
    }

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

2. W zadaniu drugim zmierzone zostały czasy działania programu w zależności od maksymalnej ilości iteracji, ilości używanych wątków oraz implementacji Exectura. Pomiary zostały przedstawione na wykresach poniżej.









Wnioski:

Użycie ExecutorService do wywołania metody obliczeniowej znacząco zmniejsza czas działania programu. Dodatkowo dzięki zastosowaniu obiektów Future, mamy pewność że wszystkie wątki wykonają się i zwrócą wartości przed zakończeniem głównego wątku. W przypadku użycia implementacji newSingleThreadExecutor, można zauważyć że wraz ze wzrostem wartości maksymalnych iteracji zwiększa się czas działania programu. Dzieje się tak również w przypadku newCachedThreadPool. W przypadku implementacji newFixedThreadPool oraz newWorkStealingPool wraz ze wzrostem ilości użytych wątków czas działania programu maleje. W przypadku użycia jednego wątku najlepiej wypada newCachedThreadPool, którego czas jest kilkukrotnie niższy niż innych implementacji. Mimo tego newFixedThreadPool oraz newWorkStealingPoo dają możliwość odpalenia większej ilości wątków co znacznie obniżają czas działania dla dużych wartości max_iter. Podsumowując w przypadku tego zadania najlepsze wyniki uzyskała implementacja newWorkStealingPool, która dla max_iter równego 3000 i 9 wątków okazała się najszybsza.

Bibliografia:

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

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