Teoria Współbieżności - Sprawozdanie 8

Maciej Brzeżawski

Listopad 2024

1 Treść zadania

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

- Proszę zaimplementować przy użyciu Executor i Future program wykonujący obliczanie zbioru Mandelbrota w puli wątków. Jako podstawę implementacji proszę wykorzystać kod w Javie.
- 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).

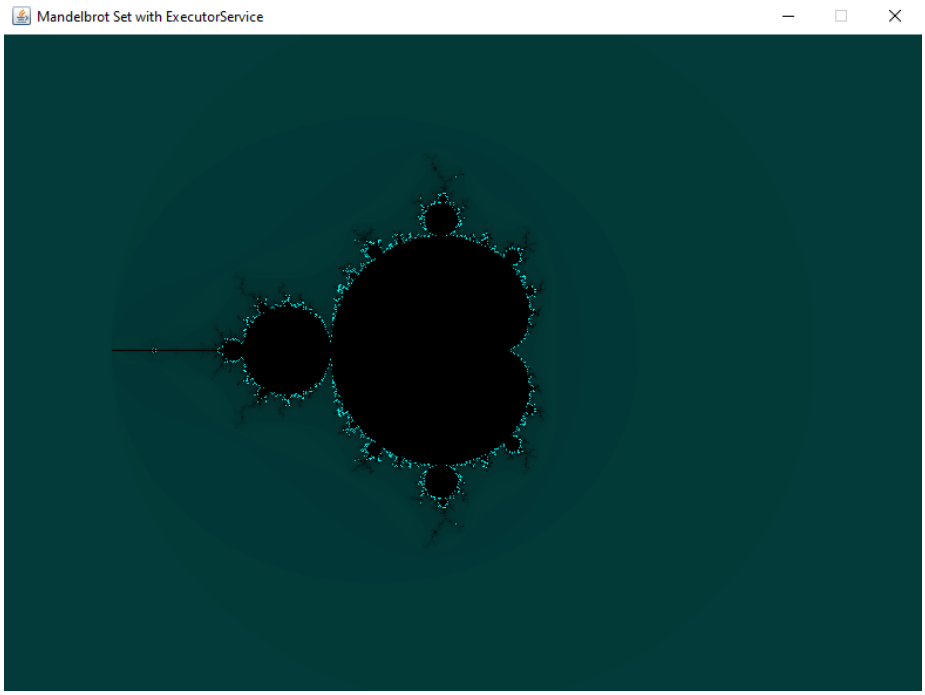
2 Implementacja programu obliczającego zbiór Mandelbrota w puli wątków

```
import java.awt.Graphics;
import java.awt.image.BufferedImage;
import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.*;

import javax.swing.JFrame;

public class Mandelbrot extends JFrame {
    private final int MAX_ITER = 570; // Maksymalna liczba iteracji
    private final double ZOOM = 150;  // Zoom obrazu
    private final int WIDTH = 800;   // Szerokość obrazu
    private final int HEIGHT = 600;  // Wysokość obrazu
    private BufferedImage image;

    public Mandelbrot() {
        super("Mandelbrot Set with ExecutorService");
        setBounds(100, 100, WIDTH, HEIGHT);
        setResizable(false);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        image = new BufferedImage(WIDTH, HEIGHT, BufferedImage.TYPE_INT_RGB);

        // Tworzymy ExecutorService z pulą wątków
        int numThreads = Runtime.getRuntime().availableProcessors();
        ExecutorService executor = Executors.newFixedThreadPool(numThreads);

        // Lista zadań Callable
        List<Future<Void>> tasks = new ArrayList<>();

        // Tworzymy zadania dla każdego wiersza
        for (int y = 0; y < HEIGHT; y++) {
            final int row = y; // Zmienna finalna dla lambdy
            tasks.add(executor.submit(() -> {
                computeRow(row);
                return null; // Callable wymaga zwracania wartości
            }));
        }

        // Oczekiwanie na zakończenie wszystkich zadań
        for (Future<Void> task : tasks) {
            try {
                task.get(); // Blokuje do czasu zakończenia danego zadania
```

```
            } catch (InterruptedException | ExecutionException e) {
                e.printStackTrace();
            }
        }

        // Zamykamy ExecutorService
        executor.shutdown();
    }

    // Metoda obliczająca piksele dla danego wiersza
    private void computeRow(int y) {
        double zx, zy, cX, cY, tmp;
        for (int x = 0; x < WIDTH; x++) {
            zx = zy = 0;
            cX = (x - WIDTH / 2.0) / ZOOM;
            cY = (y - HEIGHT / 2.0) / 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--;
            }
            image.setRGB(x, y, iter | (iter << 8));
        }
    }

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

    public static void main(String[] args) {
        long startTime = System.currentTimeMillis();

        Mandelbrot mandelbrot = new Mandelbrot();
        mandelbrot.setVisible(true);

        long endTime = System.currentTimeMillis();
        System.out.println("Execution Time: " + (endTime - startTime) + "ms");
    }
}
```

3 Przykładowe wywołanie programu

Czas wykonania w milisekundach dla MAX_ITER = 570:
Execution Time: 523ms



Rysunek 1: Obraz zbioru Mandelbrota w oknie graficznym

4 Test szybkości działania programu w zależności od implementacji Executora i jego parametrów

```
public class Mandelbrot extends JFrame {
    private int MAX_ITER;           // Liczba iteracji (dynamiczna)
    private double ZOOM = 150;      // Zoom obrazu
    private final int WIDTH = 800;  // Szerokość obrazu
    private final int HEIGHT = 600; // Wysokość obrazu
    private BufferedImage image;

    public Mandelbrot(int maxIter) {
        super("Mandelbrot Set with Dynamic MAX_ITER");
        this.MAX_ITER = maxIter;
        setBounds(100, 100, WIDTH, HEIGHT);
        setResizable(false);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        image = new BufferedImage(WIDTH, HEIGHT, BufferedImage.TYPE_INT_RGB);
    }

    public void generate(int numThreads, ExecutorService executor) {
        List<Future<Void>> tasks = new ArrayList<>();

        // Tworzymy zadania dla każdego wiersza
        for (int y = 0; y < HEIGHT; y++) {
            final int row = y;
            tasks.add(executor.submit(() -> {
                computeRow(row);
                return null;
            }));
        }

        // Czekamy na zakończenie wszystkich zadań
        for (Future<Void> task : tasks) {
            try {
                task.get();
            } catch (InterruptedException | ExecutionException e) {
                e.printStackTrace();
            }
        }

        executor.shutdown();
    }

    private void computeRow(int y) {
```

```
    double zx, zy, cX, cY, tmp;
    for (int x = 0; x < WIDTH; x++) {
        zx = zy = 0;
        cX = (x - WIDTH / 2.0) / ZOOM;
        cY = (y - HEIGHT / 2.0) / 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--;
        }
        image.setRGB(x, y, iter | (iter << 8));
    }
}

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

public static void main(String[] args) {
    int[] threadCounts = {1, 2, 4, 8, 16};  // Liczba wątków
    int[] maxIterValues = {500, 1000, 2000, 4000}; // Różne wartości MAX_ITER

    System.out.println("Testing Mandelbrot generation with different
    thread counts and MAX_ITER values...");

    for (int maxIter : maxIterValues) {
        System.out.println("\nTesting with MAX_ITER = " + maxIter);
        for (int numThreads : threadCounts) {
            System.out.println("Threads: " + numThreads);
            ExecutorService executor = Executors.newFixedThreadPool(numThreads);
            Mandelbrot mandelbrot = new Mandelbrot(maxIter);

            long startTime = System.currentTimeMillis();
            mandelbrot.generate(numThreads, executor);
            long endTime = System.currentTimeMillis();

            System.out.println("Threads: " + numThreads + ", MAX_ITER: " +
            maxIter + ", Time: " + (endTime - startTime) + "ms");
        }
    }

    System.out.println("\nTesting with CachedThreadPool for MAX_ITER variations...");
    for (int maxIter : maxIterValues) {
```

```
            ExecutorService cachedExecutor = Executors.newCachedThreadPool();
            Mandelbrot mandelbrot = new Mandelbrot(maxIter);

            long startTime = System.currentTimeMillis();
            mandelbrot.generate(Runtime.getRuntime().availableProcessors(),
            cachedExecutor);
            long endTime = System.currentTimeMillis();

            System.out.println("CachedThreadPool, MAX_ITER: " + maxIter +
            ", Time: " + (endTime - startTime) + "ms");
        }

        System.out.println("Testing complete.");
    }
}
```

Wyniki:

Testing with MAX_ITER = 500

Threads	Time (ms)
1	87
2	43
4	22
8	17
16	17

Testing with MAX_ITER = 1000

Threads	Time (ms)
1	118
2	61
4	35
8	22
16	19

Testing with MAX_ITER = 2000

Threads	Time (ms)
1	231
2	120
4	63
8	36
16	30

Testing with MAX_ITER = 4000

Threads	Time (ms)
1	438
2	226
4	117
8	60
16	53

Testing with CachedThreadPool

MAX_ITER	Time (ms)
500	23
1000	26
2000	47
4000	58

Wnioski:

1. Wydajność rośnie wraz ze wzrostem liczby wątków w puli

- Dla każdej wartości `MAX_ITER` zauważalny jest znaczący spadek czasu wykonania w miarę zwiększania liczby wątków w puli.
- W przypadku testów z `MAX_ITER = 500`, czas działania zmniejszył się z **87 ms** dla jednego wątku do **17 ms** dla 8 i 16 wątków.
- Większa liczba wątków pozwala na lepsze wykorzystanie dostępnych zasobów procesora. Dalsze zwiększanie liczby wątków powyżej liczby rdzeni procesora (np. 16 wątków) nie przynosi już znacznej poprawy, co wynika z narzutu zarządzania wątkami.

2. Wpływ liczby iteracji (`MAX_ITER`)

- Zwiększanie liczby iteracji `MAX_ITER` prowadzi do proporcjonalnego wzrostu czasu obliczeń. Dla testu z `MAX_ITER = 4000`, czas wykonania dla jednego wątku wyniósł **438 ms**, podczas gdy dla `MAX_ITER = 500` było to jedynie **87 ms**.
- Efektywny podział pracy między wątki pozwala jednak utrzymać niski czas obliczeń nawet przy wysokiej wartości `MAX_ITER`, szczególnie przy użyciu większej liczby wątków.

3. Porównanie `FixedThreadPool` i `CachedThreadPool`

- `CachedThreadPool` dynamicznie dostosowuje liczbę wątków do potrzeb, co może być korzystne w przypadku zadań o mniejszym obciążeniu. Na przykład dla `MAX_ITER = 500`, `CachedThreadPool` uzyskał wynik **23 ms**, co jest nieco wolniejsze niż `FixedThreadPool` z 8 wątkami (**17 ms**).
- Jednak dla większych wartości `MAX_ITER` różnice między `CachedThreadPool` a `FixedThreadPool` stają się mniej wyraźne, ponieważ obciążenie jest na tyle wysokie, że oba podejścia wykorzystują maksymalnie dostępne zasoby.

4. Optymalna liczba wątków

- Najlepsze wyniki uzyskuje się, gdy liczba wątków w puli jest zbliżona do liczby dostępnych rdzeni procesora.
- W przypadku testów, najlepsze rezultaty osiągnięto przy użyciu 8 wątków, co może odpowiadać liczbie rdzeni w systemie.

5. Ograniczenia skalowalności

- Przy bardzo wysokiej liczbie wątków (np. 16) i stosunkowo małej liczbie iteracji, czas działania nie ulega dalszej poprawie, co wskazuje na narzut zarządzania wątkami. W takich przypadkach zwiększanie liczby wątków powyżej liczby rdzeni jest nieefektywne.

5 Bibliografia

Abraham Silberschatz, Peter B. Galvin, Greg Gagne. ***Operating System Concepts***. Wiley, 2020. ISBN: 978-1119800361.

Brian Goetz, Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes, Doug Lea. ***Java Concurrency in Practice***. Addison-Wesley Professional, 2006. ISBN: 978-0321349606.

Andrew S. Tanenbaum, Herbert Bos. ***Modern Operating Systems***. Pearson, 2015. ISBN: 978-0133591620.

Maurice Herlihy, Nir Shavit. ***The Art of Multiprocessor Programming***. Morgan Kaufmann, 2020. ISBN: 978-0124159501.

A. S. Tanenbaum, M. Van Steen. ***Distributed Systems: Principles and Paradigms***. Pearson, 2007. ISBN: 978-0132392273.

Douglas Lea. ***Concurrent Programming in Java: Design Principles and Patterns***. Addison-Wesley, 2000. ISBN: 978-0201310092.