Laboratorium 8 - Active Objectasynchroniczne wykonanie zadań w puli wątków przy użyciu wzorców Executor i Future

Piotr Karamon

02.12.2024r.

Treści zadań

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`).

Implementacja

Obliczanie obrazu zbioru Mandelbrota jest zadaniem, które można łatwo zrównoleglić. Każdy piksel obrazu jest obliczany niezależnie od innych pikseli. W związku z tym, dzielimy zadanie wygenerowania całego obrazu, na wiele zadań które polegają na wygenerowaniu wiersza obrazu. Wykonaniem takiego pojedynczego zadania zajmuję się klasa `Worker`.

```
class Worker implements Runnable {
    private final double ZOOM = 150;

    private final int[] row;
    private final int row_num;
    private final int max_iter;

    public Worker(int[] row, int row_num, int max_iter) {
        this.row = row;
        this.row_num = row_num;
        this.max_iter = max_iter;
    }

    @Override
    public void run() {
        double zx, zy, cX, cY, tmp;

        for (int col = 0; col < row.length; col++) {
            zx = zy = 0;
            cX = (col - 400) / ZOOM;
            cY = (row_num - 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--;
            }
            row[col] = iter | (iter << 8);
        }
    }
```

```
}
```

Wyświetlenie zbioru

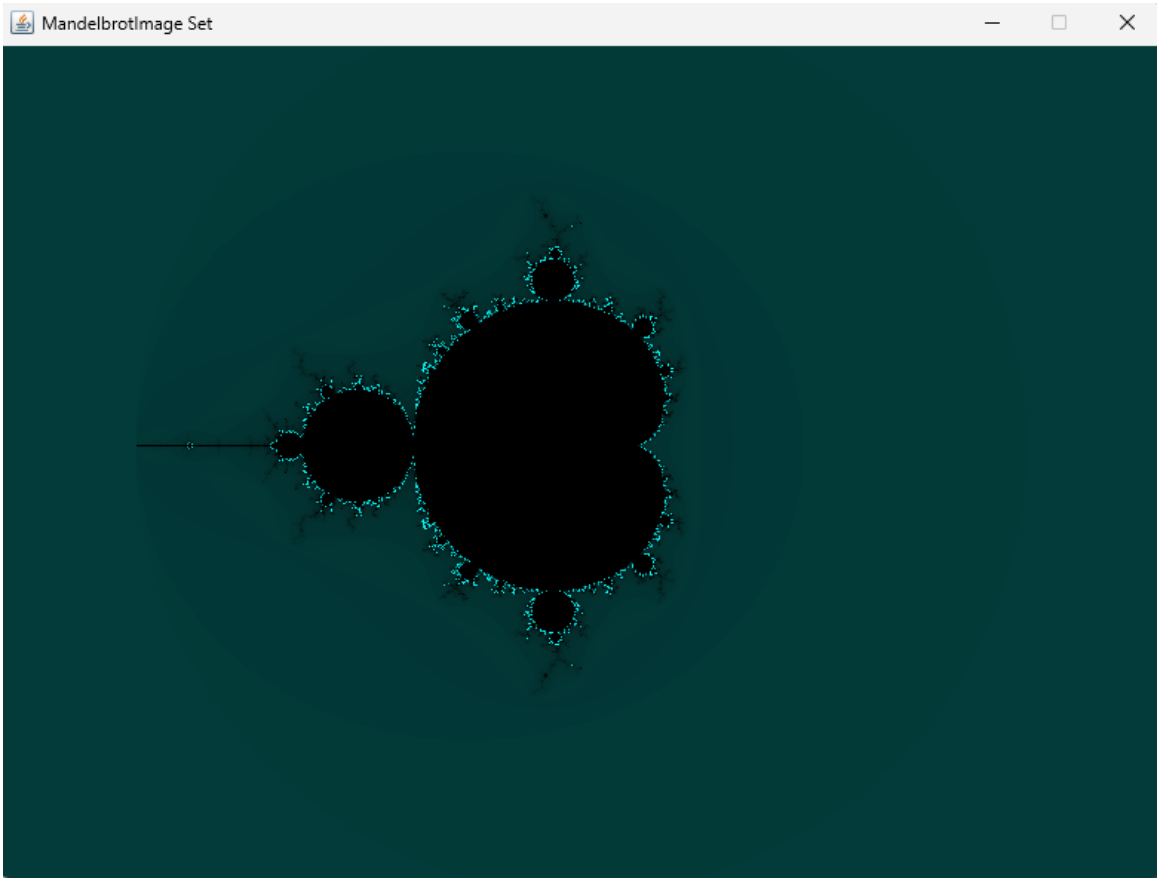
Aby wyświetlić zbiór używamy biblioteki awt. Klasa odpowiedzialna na wyświetlenie wygenerowanego obrazu:

```
class MandelbrotImage extends JFrame {
    private BufferedImage I;

    public MandelbrotImage() {
        super("MandelbrotImage Set");
        setResizable(false);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
    }


    public void display(int[][] image) {
        setBounds(100, 100, image[0].length, image.length);
        I = new BufferedImage(getWidth(), getHeight(), BufferedImage.TYPE_INT_RGB);
        for (int row = 0; row < getHeight(); row++) {
            for (int col = 0; col < getWidth(); col++) {
                I.setRGB(col, row, image[row][col]);
            }
        }
        setVisible(true);
    }


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

W funkcji `main`, używamy dowolnego executora i wybieramy rozmiar naszego obrazu oraz maksymalną ilość iteracji. W celu zaczekania na zakończenie wszystkich zadań używamy metody `submit` na executorze, która zwraca `Future`, a następnie wywołujemy metodę `get` na każdym z `Future`.

```
public static void main(String[] args) {
    ExecutorService executor = Executors.newFixedThreadPool(4);
    int[][] image = run(executor, 570, 600, 800);
    new MandelbrotImage().display(image);


}

private static int[][] run(ExecutorService executor, int max_iter, int height, int width)
↪   {
    int[][] image = new int[height][width];
    var futures = IntStream.range(0, height)
            .mapToObj(row -> new Worker(image[row], row, max_iter))
            .map(executor::submit)
            .toList();

    executor.shutdown();
    for (var future : futures) {
        try {
            future.get();
        } catch (Exception e) {
            throw new RuntimeException("Failed to get future", e);
        }
```

```
    }

    return image;
}
```



Rysunek 1: Wynik programu

Porównanie w zależności od implementacji Executora

Metoda `run` pozwala nam na przetestowanie różnych implementacji executorów. Parametr `MAX_ITER` ustawiamy na 200000, aby zwiększyć czas obliczeń. Kod został uruchomiony na komputerze ośmiordzeniowym.

Kod testujący:

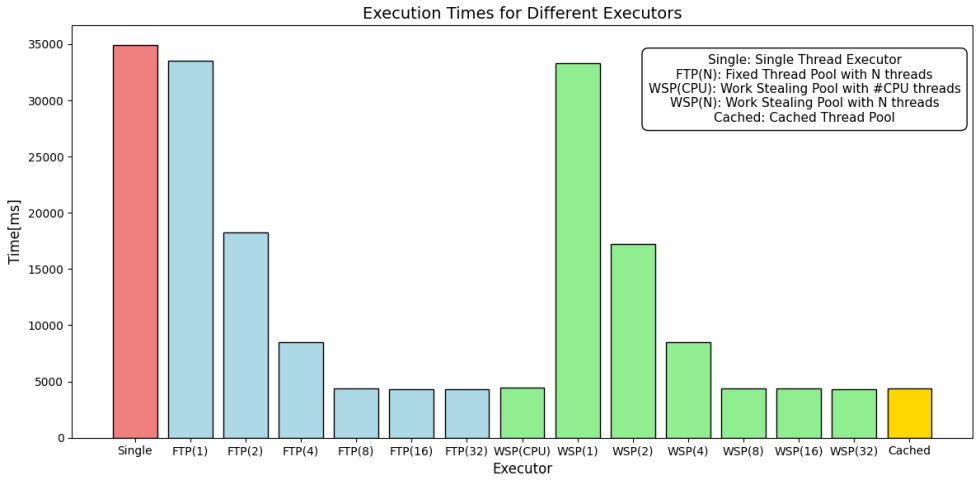
```
private record ExecutorWithDesc(ExecutorService executor, String desc) {
}
public static void main(String[] args) {
        var executors = List.of(
                new ExecutorWithDesc(Executors.newSingleThreadExecutor(),
                        "Single thread"),
                new ExecutorWithDesc(Executors.newFixedThreadPool(1),
                        "Fixed thread pool (1 threads)"),
                new ExecutorWithDesc(Executors.newFixedThreadPool(2),
                        "Fixed thread pool (2 threads)"),
                new ExecutorWithDesc(Executors.newFixedThreadPool(4),
                        "Fixed thread pool (4 threads)"),
                new ExecutorWithDesc(Executors.newFixedThreadPool(8),
                        "Fixed thread pool (8 threads)"),
                new ExecutorWithDesc(Executors.newFixedThreadPool(16),
                        "Fixed thread pool (16 threads)"),
                new ExecutorWithDesc(Executors.newFixedThreadPool(32),
```

```
                        "Fixed thread pool (32 threads)"),
                new ExecutorWithDesc(Executors.newWorkStealingPool(),
                        "Work stealing pool (parallelism = #CPU)"),
                new ExecutorWithDesc(Executors.newWorkStealingPool(1),
                        "Work stealing pool (parallelism = 1)"),
                new ExecutorWithDesc(Executors.newWorkStealingPool(4),
                        "Work stealing pool (parallelism = 4)"),
                new ExecutorWithDesc(Executors.newWorkStealingPool(8),
                        "Work stealing pool (parallelism = 8)"),
                new ExecutorWithDesc(Executors.newWorkStealingPool(16),
                        "Work stealing pool (parallelism = 16)"),
                new ExecutorWithDesc(Executors.newWorkStealingPool(32),
                        "Work stealing pool (parallelism = 32)"),
                new ExecutorWithDesc(Executors.newCachedThreadPool(),
                        "Cached thread pool")
        );

        System.out.println("ExecutorName,Time");
        for (var executor : executors) {
            var start = System.nanoTime();
            int[][] image = run(executor.executor, 200_000, 600, 800);
            var end = System.nanoTime();

            var timeMs = (end - start) / 1_000_000;
            System.out.println(executor.desc + "," + timeMs);
        }

}
```

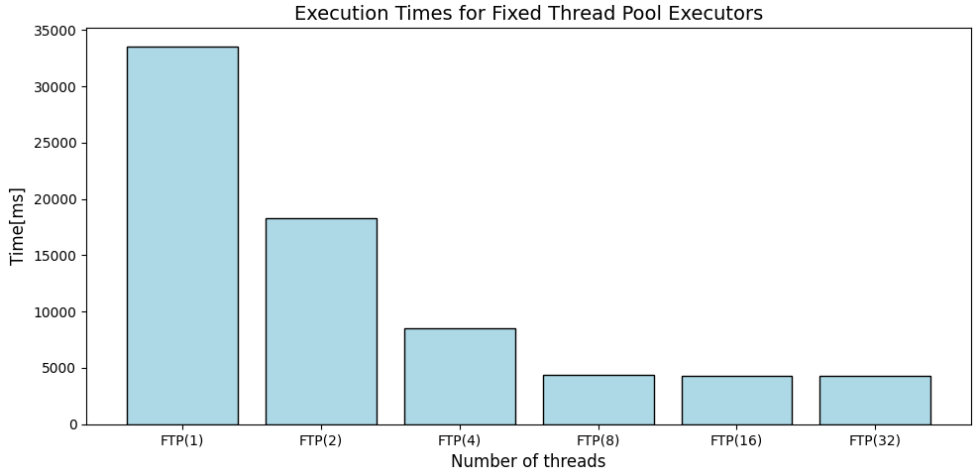
Wynik programu:

```
ExecutorName,Time
Single thread,34914
Fixed thread pool (1 threads),33507
Fixed thread pool (2 threads),18269
Fixed thread pool (4 threads),8531
Fixed thread pool (8 threads),4420
Fixed thread pool (16 threads),4331
Fixed thread pool (32 threads),4320
Work stealing pool (parallelism = #CPU),4471
Work stealing pool (parallelism = 1),33299
Work stealing pool (parallelism = 2),17205
Work stealing pool (parallelism = 4),8501
Work stealing pool (parallelism = 8),4413
Work stealing pool (parallelism = 16),4380
Work stealing pool (parallelism = 32),4325
Cached thread pool,4381
```

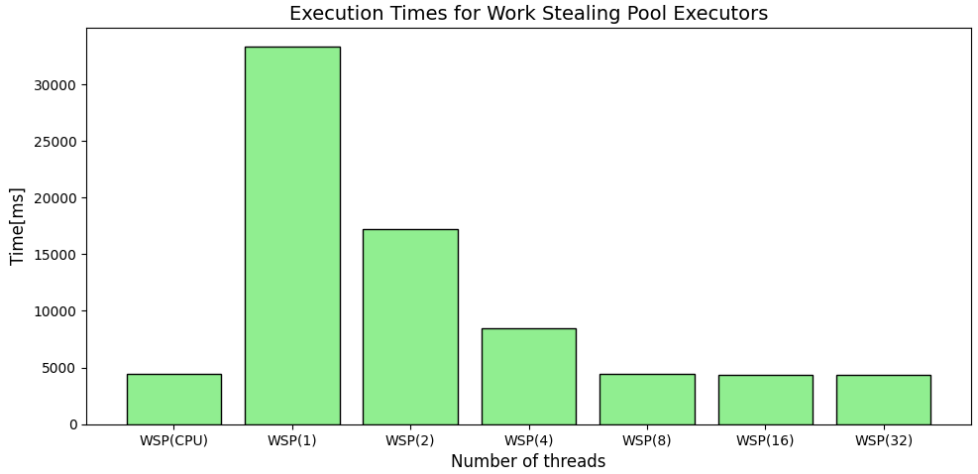
Używając tych danych możemy stworzyć wykresy porównujące te implementacje.



Rysunek 2: Czasy wykonania obliczeń w zależności od użytego executora.



Rysunek 3: Czasy wykonania obliczeń w zależności od ilości wątków dla executora o stałej liczbie wątków.



Rysunek 4: Czasy wykonania obliczeń w zależności od ilości wątków dla executora z mechanizmem "kradzieży pracy".

Wnioski

Analizując wykresy dochodzimy do następujących wniosków:

- Czas wykonania dla `SingleThreadExecutor` jest, według oczekiwań, najdłuższy. Jest to spowodowane tym, że zadania są wykonywane sekwencyjnie. Nasz program nie jest w stanie zrównoleglić obliczeń i wykorzystać wielu rdzeni procesora.
- Również wedle oczekiwań executory z pulami wątków jak `FixedThreadPoolExecutor` i `WorkStealingPool` w przypadku podania ilości wątków równej 1 "degradują" się do `SingleThreadExecutor` jeżeli chodzi o czas wykonania.
- `SingleThreadExecutor` nie jest dobry wyborem gdy mamy do wykonania mnóstwo obliczeń, które można zrównoleglić.
- Zarówno w przypadku `FixedThreadPoolExecutor` jak i `WorkStealingPoolExecutor`, czas wykonania maleje wraz ze wzrostem ilości wątków. Zwięszkenie dwukrotne ilości wątków powoduje zmniejszenie czasu wykonania o około 50%. Oczywiście do pewnego momentu(8 wątków) po czym zyski ze zwiększenia ilości wątków są w zasadzie żadne, bo użyty procesor ma jedynie 8 rdzeni. Czasy wykonania dla obu executorów są bardzo zbliżone, co oznacza, że obsługa "kradzieży pracy" nie wpływa znacząco na czas wykonania.
- `CachedThreadPoolExecutor`, charakteryzuje się brakiem stałej liczby wątków. Executor tworzy nowe wątki w miarę potrzeby, a wątki nieaktywne przez pewien czas są usuwane. W naszym przypadku, czas wykonania jest zbliżony do `WorkStealingPoolExecutor` i `FixedThreadPoolExecutor` z 8 wątkami.
- Robiąc ten eksperyment oczekiwałem, że najszybszym wyborem będzie `FixedThreadPoolExecutor` z 8 wątkami, z racji prostoty oraz liczby wątków równych liczbie rdzeni procesora. Jednakże, równie dobrze radzi sobie `WorkStealingPoolExecutor` jak i `CachedThreadPoolExecutor`. Zaletą `WorkStealingPoolExecutor` jest to, że nie musimy podawać ilości wątków, bo automatycznie dostosowuje się do liczby rdzeni procesora. Oczywiście "kradzież pracy" również jest zaletą w przypadku, gdy niektóre zadania są bardziej złożone niż inne.

Bibliografia

- Baeldung - Java Executor Service Tutorial
- Dokumentacja pakietu Executors