Lab 08

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

Adrian Madej 26.11.2023

1. Treści zadań

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. Rozwiązywanie zadań

Zadanie 1

Koncepcja

Przykładowy kod Mandelbrot wzbogacamy o wielowątkowość. Zauważmy, że wewnętrzna pętla obliczającą kolejne piksele obrazu wynikowego jest niezależna od wartości poprzednich, co pozwala rozdzielić ją między wątki tak, aby każdy obliczał jeden wiersz obrazu.

Implementacja

Klasa MandelbrotWorker reprezentuje pracę jednego wątku, który przelicza wartości zbioru Mandelbrota dla każdego piksela w określonym wierszu i zapisuje odpowiednie kolory na obrazie.

```
class MandelbrotWorker implements Callable<Void> {
    private final double ZOOM = 150;

    private final int iterations;
    private final BufferedImage img;
    private final int width;
    private final int height;

    public MandelbrotWorker(int iterations, int width, int
height, BufferedImage img) {
        this.iterations = iterations;
        this.width = width;
        this.height = height;
        this.img = img;
    }

    @Override
    public Void call() {
        double cY = (height - 300) / ZOOM;
        for (int x = 0; x < width; x++) {
            double zy = 0;
            double zx = 0;
            double cX = (x - 400) / ZOOM;
            int iter = iterations;
            while (zx * zx + zy * zy < 4 && iter > 0) {
                double tmp = zx * zx - zy * zy + cX;
                zy = 2.0 * zx * zy + cY;
                zx = tmp;
                iter--;
            }
            img.setRGB(x, height, iter | (iter << 8));
        }
        return null;
    }
}
```

Klasa Mandelbrot reprezentuje główną logikę programu obsługuje ona tworzenie obrazu zbioru Mandelbrota, korzystając z puli wątków do równoczesnego obliczania wartości dla różnych wierszy obrazu. Następnie wyświetla ten obraz w oknie aplikacji.

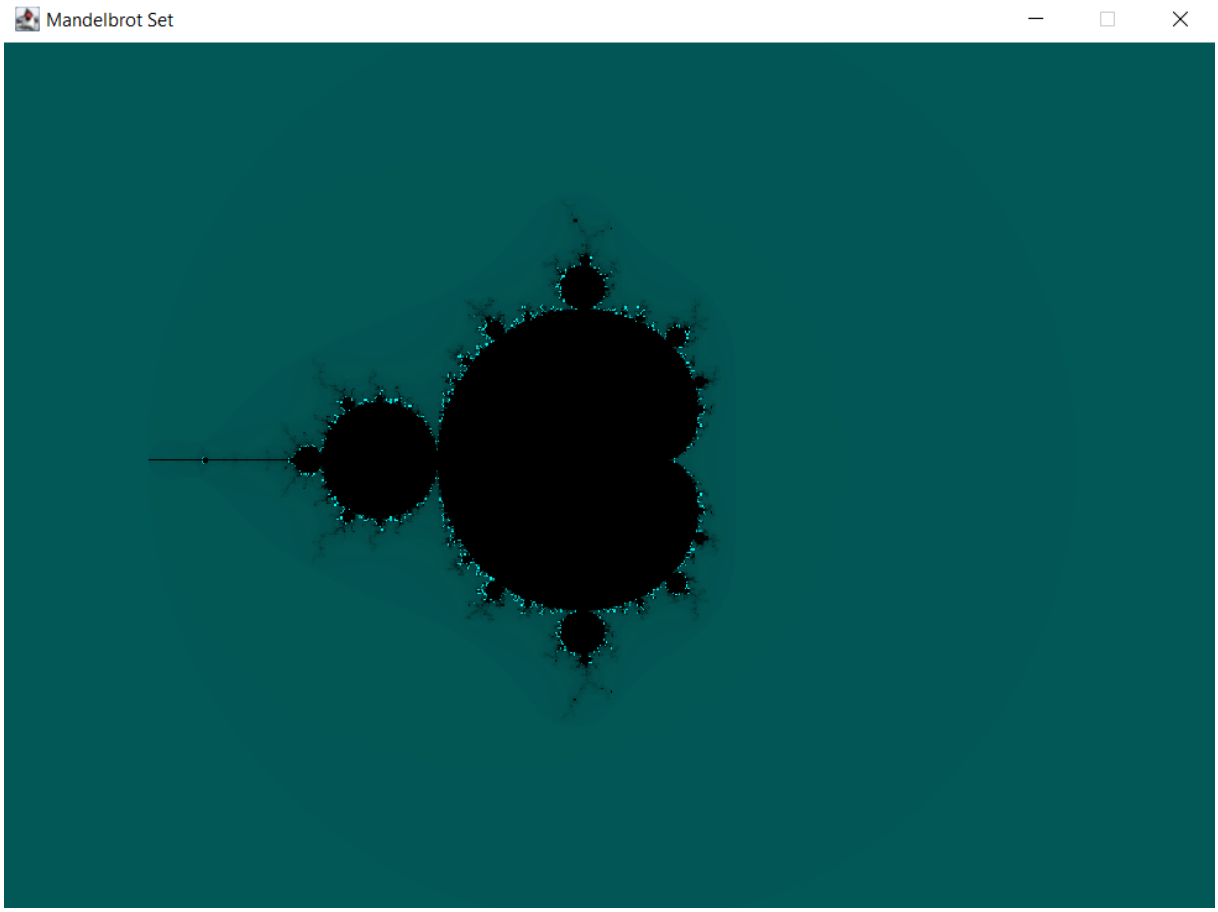
```
     class Mandelbrot extends JFrame {
    private final int maxIter;
    private final BufferedImage img;
    private final int width;
    private final int height;

    public Mandelbrot(int max_iter) {
        super("Mandelbrot Set");
        setBounds(100, 100, 800, 600);
        setResizable(false);
        setDefaultCloseOperation(EXIT_ON_CLOSE);

        this.maxIter = max_iter;
        this.width = getWidth();
        this.height = getHeight();
        this.img = new BufferedImage(getWidth(), height,
BufferedImage.TYPE_INT_RGB);
    }

    public void run(ExecutorService executorService) {
        LinkedList<Future<Void>> futures = new LinkedList<>();
        for (int y = 0; y < height; y++) {
            MandelbrotWorker task = new
MandelbrotWorker(maxIter, width, y, img);
            futures.add(executorService.submit(task));
        }

        for (Future<Void> f : futures) {
            try {
                f.get();
            } catch (InterruptedException | ExecutionException
e) {
                e.printStackTrace();
            }
        }
    }

    public void display() {
        setVisible(true);
    }

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

Klasa Main inicjalizuje obiekt Mandelbrot, tworzy pulę wątków, uruchamia obliczenia zbioru Mandelbrota za pomocą metody run(executor), a następnie wyświetla wynik w oknie aplikacji za pomocą display(). Program używa puli wątków do równoczesnego obliczania wartości zbioru Mandelbrota dla różnych wierszy obrazu.

```
class Main {
    public static void main(String[] args) {
        Mandelbrot mandelbrot = new Mandelbrot(600);
        ExecutorService executor =
Executors.newFixedThreadPool(6);
        mandelbrot.run(executor);
        mandelbrot.display();
    }
}
```

Wyniki

W wyniku wywołania program utworzył nam się obraz:



Program wygenerował prawidłowy obraz zbioru Mandelbrota. Rozdzielanie danych między wątkami zajmuje więcej czasu niż faktyczne obliczenia, co może sugerować, że podział zadania na wątki może być całkowicie zbędny.

Zadanie 2

Koncepcja

Do pomiarów zostanie wykorzystany kod z zadania pierwszego. Testy przeprowadzone zostaną w oparciu o różne implementacje Executora:

- newSingleThreadExecutor
- newFixedThreadPool
- newCachedThreadPool
- newWorkStealingPool

Dodatkowo będzimy także manipulować poszczególnymi parametrami takimi jak liczba wątków w puli, czy też liczbą iteracji (MAX_ITER).

Implementacja

Do klasy Mandelbrot dodajemy pole timeOfExecution, które będzie mierzyło czas pracy od rozpoczęcia obliczeń do zakończenia dla wszystkich wierszy obrazy

```
class Mandelbrot extends JFrame {
    private final int maxIter;
    private final BufferedImage img;
    private final int width;
    private final int height;
    private long timeOfExecution;

    public Mandelbrot(int max_iter) {
        super("Mandelbrot Set");
        setBounds(100, 100, 800, 600);
        setResizable(false);
        setDefaultCloseOperation(EXIT_ON_CLOSE);

        this.maxIter = max_iter;
        this.width = getWidth();
        this.height = getHeight();
        this.img = new BufferedImage(getWidth(), height,
BufferedImage.TYPE_INT_RGB);
    }
```

```
    public void run(ExecutorService executorService) {
        long start = System.currentTimeMillis();

        LinkedList<Future<Void>> futures = new LinkedList<>();

        for (int y = 0; y < height; y++) {
            MandelbrotWorker task = new
MandelbrotWorker(maxIter, width, y, img);
            futures.add(executorService.submit(task));
        }

        for (Future<Void> f : futures) {
            try {
                f.get();
            } catch (InterruptedException | ExecutionException
e) {
                e.printStackTrace();
            }
        }
        timeOfExecution = System.currentTimeMillis() - start;
    }

    public void display() {
        setVisible(true);
    }

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

    public long getTimeOfExecution() {
        return timeOfExecution;
    }
}
```

Klasa MandelbrotWorker pozostaje bez zmian tzn. jest identyczna jak w zadaniu 1.

Modyfikujemy klasę Main, która jest odpowiedzialna za przeprowadzenie testów wydajności dla różnych implementacji ExecutorService przy użyciu różnych kombinacji liczby iteracji i liczby wątków dla generowania zbioru Mandelbrota. Wyniki testów są zapisywane do plików CSV.

```
class Main {
    private static final List<Integer> maxItersList =
List.of(100, 500, 1000, 5000);
    private static final List<Integer> threadCountList =
List.of(2, 4, 8, 16);
```

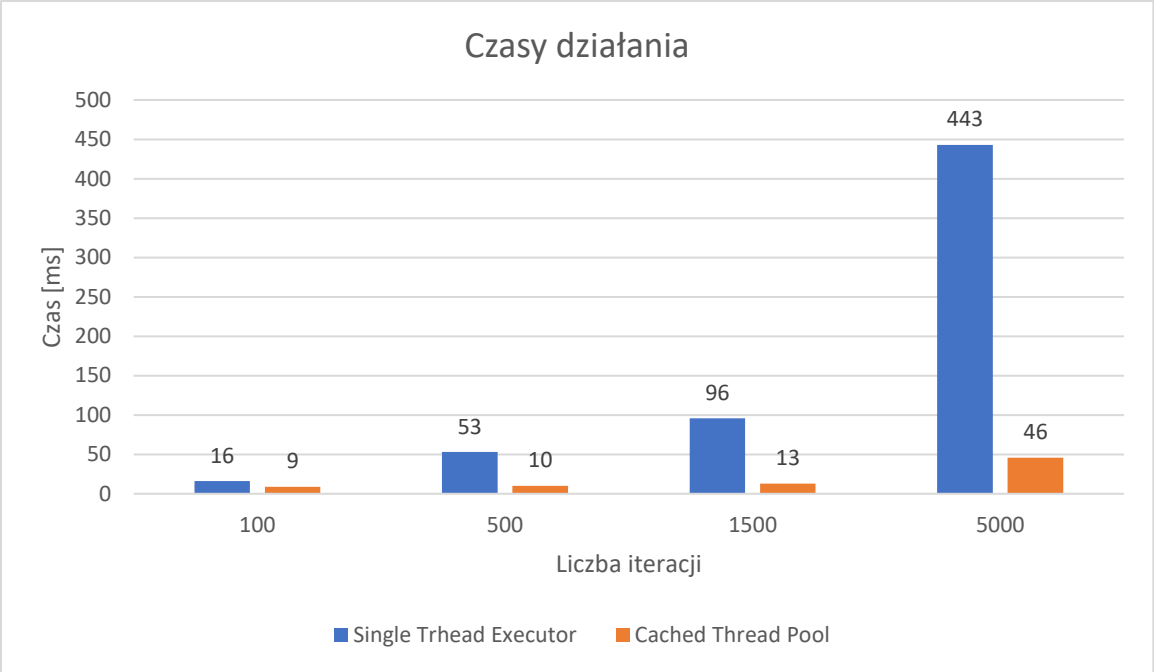
```
    public static void saveResults(String fname, List<String>
data) {
        Path out = Paths.get(fname + ".csv");
        try {
            Files.write(out, data, Charset.defaultCharset());
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    public static void newFixedThreadPoolTest() {
        LinkedList<String> results = new LinkedList<>();
        for (Integer t : threadCountList) {
            for (Integer i : maxItersList) {
                Mandelbrot mandelbrot = new Mandelbrot(i);
                ExecutorService executor =
Executors.newFixedThreadPool(t);
                mandelbrot.run(executor);
                executor.shutdownNow();
                long time = mandelbrot.getTimeOfExecution();
                results.add(t + "," + i + "," + time);
            }
        }

        saveResults("newFixedThreadPool", results);
    }

    public static void newSingleThreadExecutorTest() {
        LinkedList<String> results = new LinkedList<>();
        for (Integer i : maxItersList) {
            Mandelbrot mandelbrot = new Mandelbrot(i);
            ExecutorService executor =
Executors.newSingleThreadExecutor();
            mandelbrot.run(executor);
            executor.shutdownNow();
            long time = mandelbrot.getTimeOfExecution();
            results.add(i + "," + time);
        }

        saveResults("newSingleThreadExecutor", results);
    }

    public static void newCachedThreadPoolTest() {
        LinkedList<String> results = new LinkedList<>();
        for (Integer i : maxItersList) {
            Mandelbrot mandelbrot = new Mandelbrot(i);
            ExecutorService executor =
Executors.newCachedThreadPool();
            mandelbrot.run(executor);
            executor.shutdownNow();
```

```
            long time = mandelbrot.getTimeOfExecution();
            results.add(i + "," + time);
        }

        saveResults("newCachedThreadPool", results);
    }

    public static void newWorkStealingPoolTest() {
        LinkedList<String> results = new LinkedList<>();
        for (Integer t : threadCountList) {
            for (Integer i : maxItersList) {
                Mandelbrot mandelbrot = new Mandelbrot(i);
                ExecutorService executor =
Executors.newWorkStealingPool(t);
                mandelbrot.run(executor);
                executor.shutdownNow();
                long time = mandelbrot.getTimeOfExecution();
                results.add(t + "," + i + "," + time);
            }
        }

        saveResults("newWorkStealingPool", results);
    }

    public static void main(String[] args) {
        newFixedThreadPoolTest();
        newSingleThreadExecutorTest();
        newCachedThreadPoolTest();
        newWorkStealingPoolTest();
    }
}
```

Wyniki

W wyniku wywołania programu dostajemy następujące dane:

Liczba iteracji	Single Thread Executor [ms]	Cached Thread Pool [ms]
100	16	9
500	53	10
1000	96	13
5000	443	46



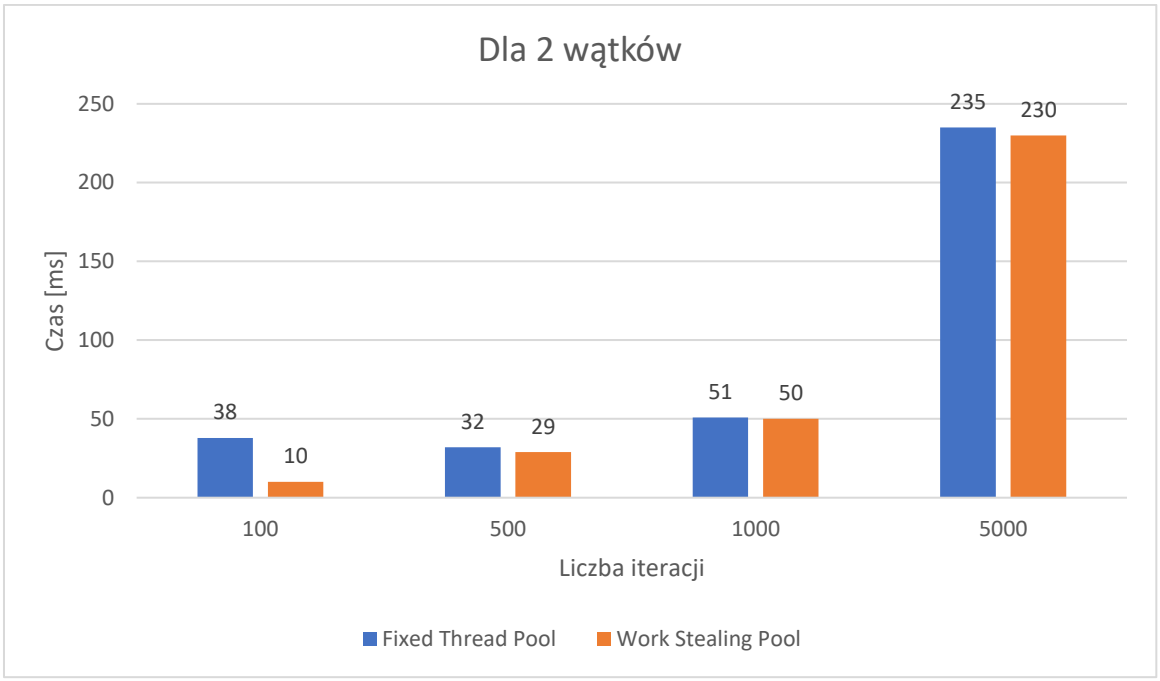
Jak widzimy czas rośnie w zależności od liczby iteracji – im większy parametr MAX_ITERS tym więcej czasu zajmują obliczenia.

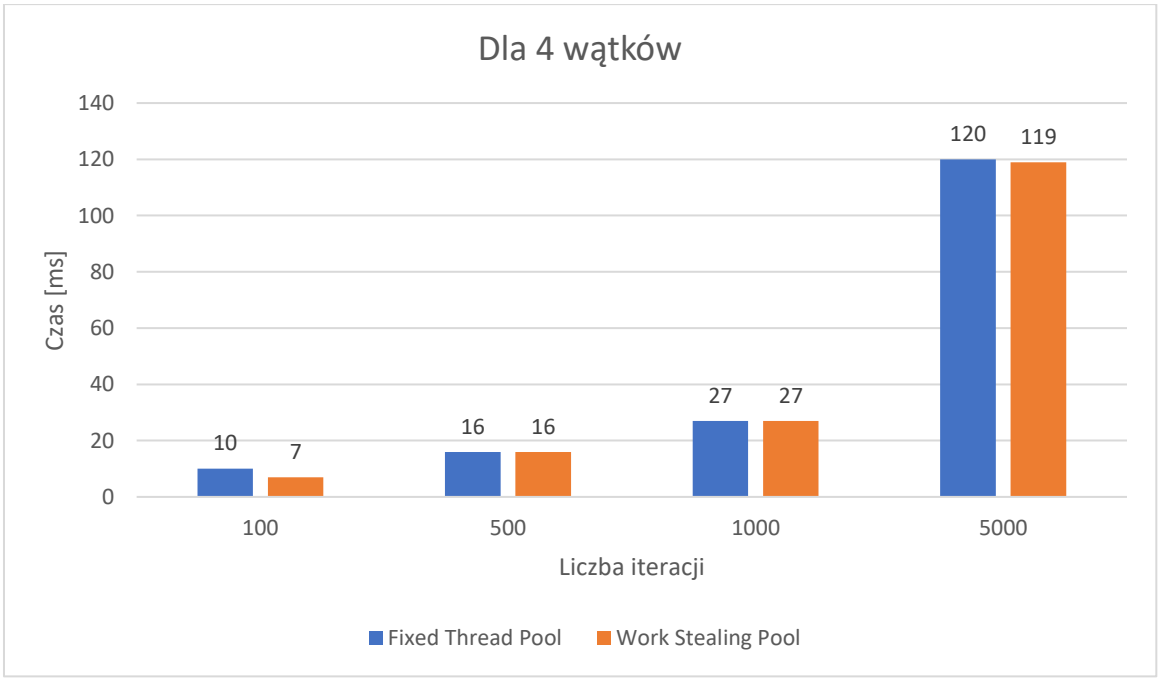
Single Thread Executor do obliczeń używa tylko jednego wątku, co może być problematyczne w momecie gdy mamy do wykonania dużą liczbę operacji. Jak widzimy jego działanie dla 5000 operacji jest niemalże 10 krotnie dłuższe od Cached Thread Poola.

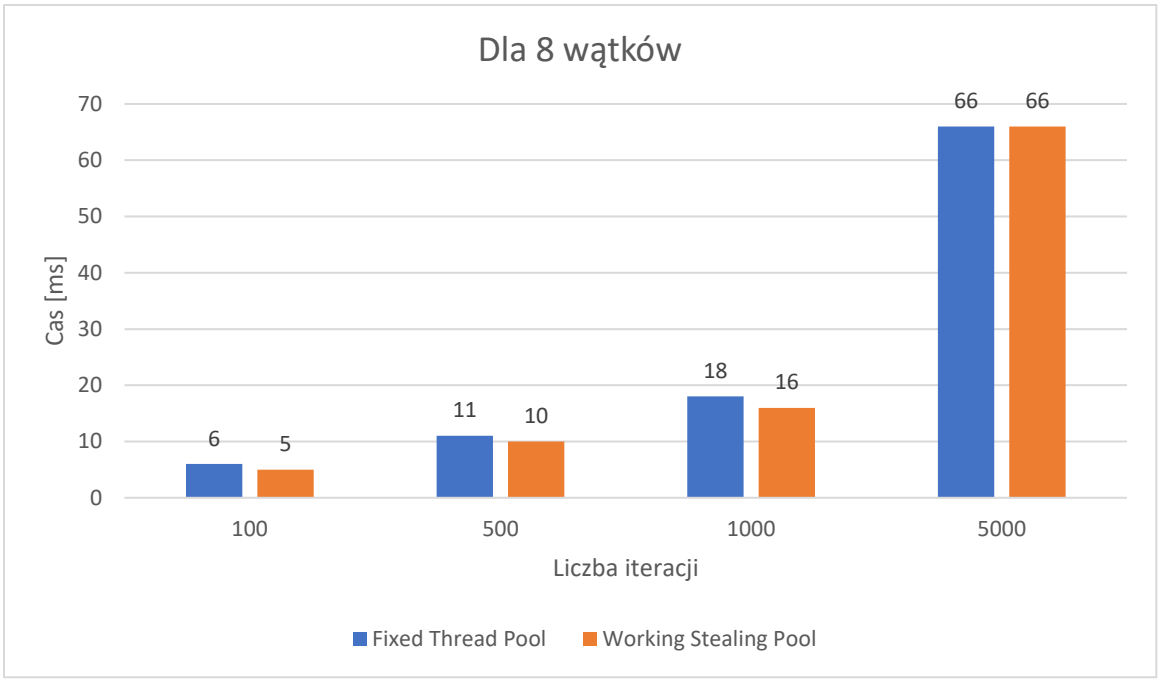
Cached Thread Pool wykorzytuje wątki które zakończyły swoje działanie, lub też tworzy nowe jeśli takowe nie istenieją. Wątki które długo nie były używane zostają, usunięte z puli. Widzimy znaczną różnicę w czasie dla 5000 iteracji, co pokazuje, że zrównoleglenie operacji było jak najbardziej korzystne.

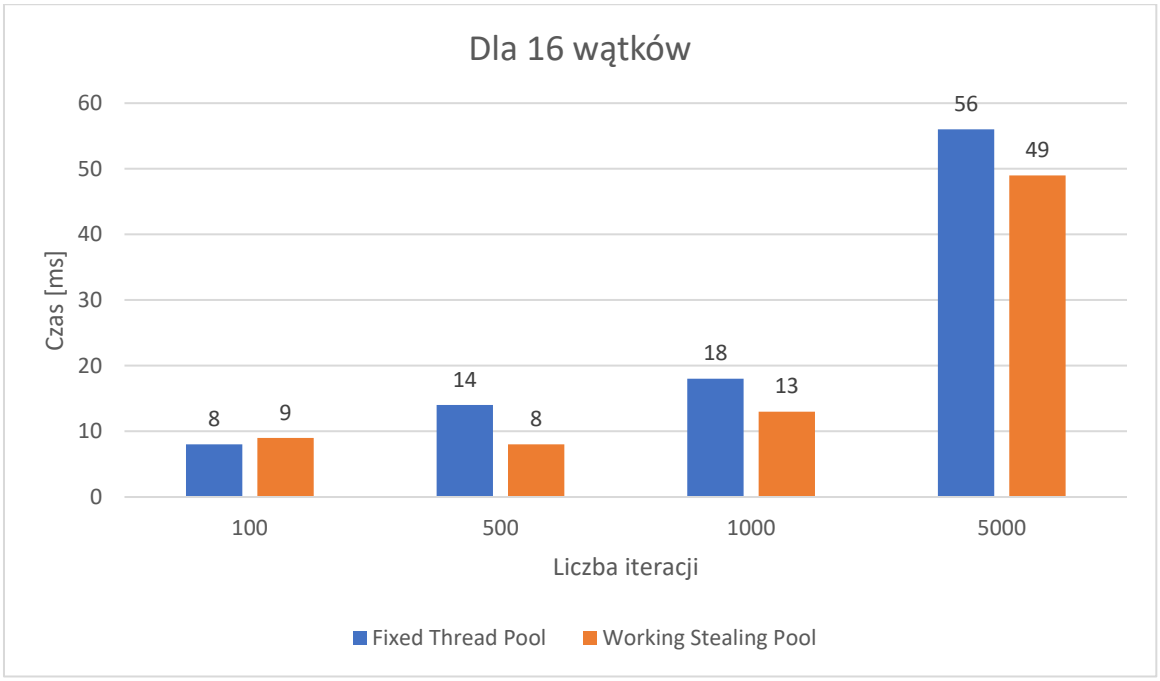
Liczba wątków	Liczba iteracji	Fixed Thread Pool [ms]	Work Stealing Pool [ms]
2	100	38	10
2	500	32	29
2	1000	51	50
2	5000	235	230
4	100	10	7
4	500	16	16
4	1000	27	27
4	5000	120	119
8	100	6	5
8	500	11	10
8	1000	18	16

8	5000	66	66
16	100	8	9
16	500	14	8
16	1000	18	13
16	5000	56	49









Fixed Thread Pool Executor wykorzystuje stałą pulę wątków, przydzielając im poszczególne zadania. Work Stealing Pool Executor dzieli zadania na mniejsze problemy (podzadania). Po takim podziale, każdy wątek oblicza dany podproblem, a następnie obliczenia zostają połączone w całość. Jak widzimy jest on szybszy od Fixed Thread Poola, przy większej liczbie iteracji, jednakże różnica nie jest zbyt wyraźna.

3. Wnioski

- Wzorzec Executor jest znacznym ułatwieniem przy pisaniu programów wielowątkowych, gdyż programista nie musi implementować puli wątków.
- Java dostarcza różnych implementacji wzorca Executor np.: FixedThreadPool, SingleThreadExecutor, CachedThreadPool i WorkStealingPool. Wybór odpowiedniego ExecutorService powinien zależeć od charakterystyki obliczeń.
- Warto zrównoleglić operacje, które nie zależą od innych obliczeń, potrafi to znacznie przyspieszyć działanie programu.
- Wzorzec Future pozwala na wykonywanie obliczeń w tle, w momencie gdy program realizuje dalsze operacje.

4. Bibliografia

(Każdy podpunkt jest hiperłączem)

- Konspekt laboratorium
- Baeldung-Future
- Baeldung-ExecutorService
- Dokumentacja Javy-Executors
- Dokumentacja Javy-Future