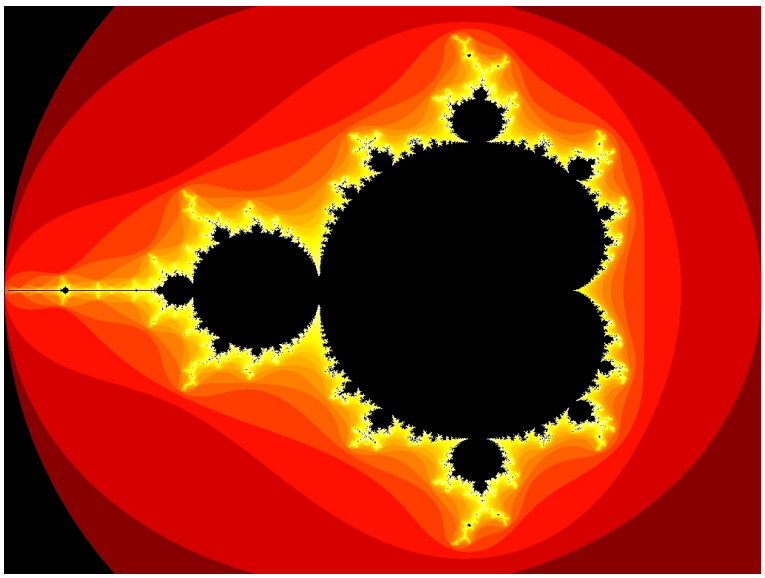
Teoria współbieżności

Andrzej Ratajczak

Laboratorium 8

1. Wstęp

Celem ćwiczenia było napisać program generujący fraktal – Zbiór Mandelbrota. Po stworzeniu programu należało dokonać pomiarów czasu obliczania fraktala przy pomocy różnych strategii dzielenia pracy na wątki ExectuorService'u.



Rysunek 1. Fraktal – Zbiór Mandelbrota

2. Realizacja ćwiczenia

Do generowania fraktala wykorzystałem gotowy szielet zaproponowany w instrukcji ćwiczenia. Naniosłem zmiany tak aby można było testować czasy przy użyciu różnych strategii Executor Service'u.

```
class Mandelbrot extends JFrame {

    private ExecutorService es;

    private final double ZOOM = 150;
    private final int MAX_ITER = 570;
    private BufferedImage I;
    private double zx, zy, cX, cY, tmp;

    public Mandelbrot(ExecutorService es, int MAX_ITER) {
        super("Mandelbrot Set");
        long start = System.nanoTime();
        this.es = es;
        setBounds(100, 100, 800, 600);
        Future[] fs = new Future[getHeight()];
        setResizable(false);
```

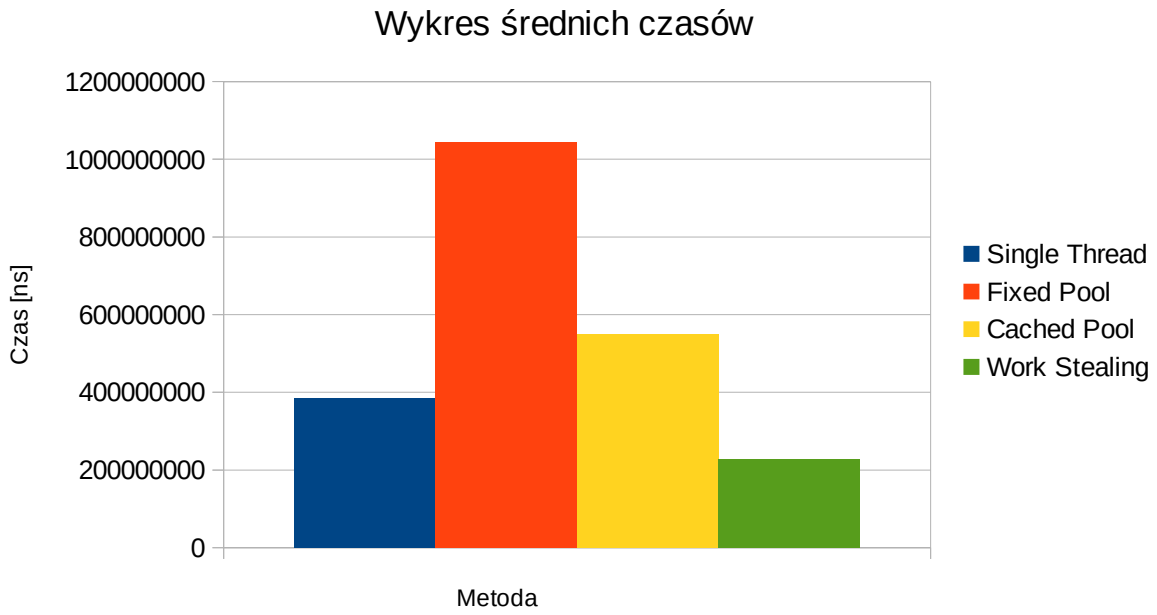
```
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        I = new BufferedImage(getWidth(), getHeight(), BufferedImage.TYPE_INT_RGB);
        for (int y = 0; y < getHeight(); y++) {
            final int j = y;
            fs[y] = es.submit(new Runnable() {
                  @Override
                  public void run() {
                      for (int x = 0; x < getWidth(); x++) {
                          zx = zy = 0;
                          cX = (x - 400) / ZOOM;
                          cY = (j - 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--;
                          }
                          I.setRGB(x, j, iter | (iter << 8));
                      }
                  }
              }
            );
        }
        for(int i = 0; i < getHeight(); i++) {
            try {
                fs[i].get();
            } catch (Exception e) { e.printStackTrace(); }
        }
        long end = System.nanoTime();
        System.out.println(end - start);
    }

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

    public static void main(String[] args) {
//        ExecutorService es = Executors.newSingleThreadExecutor();
        ExecutorService es = Executors.newFixedThreadPool(4);
//        ExecutorService es = Executors.newCachedThreadPool();
//        ExecutorService es = Executors.newWorkStealingPool();
        for(int i = 400; i <= 4000; i+=400)
            new Mandelbrot(es, i).setVisible(true);
    }
}
```

3. Wyniki

Program przetestowałem czterema różnymi strategiami. Wyniki zebrałem w poniższej tabeli oraz zaprezentowałem na diagramie. Czas podany w nanosekundach.

Metoda	Single Thread	Fixed Thread Pool	Cached Thread Pool	Work Stealing Pool
1 próba	125823430	81284210	117028479	79994117
2 próba	175710899	366064133	57176491	68316172
3 próba	262540934	78951649	96054365	450634206
4 próba	304685685	125418477	934168421	106004443
5 próba	361971306	1888549786	118218741	119370236
6 próba	442268074	2113023324	161459280	763078213

7 próba	456426674	222665836	1529330427	144454524
8 próba	535255255	1688883621	165034752	148421684
9 próba	569357234	194815755	198070885	216434044
10 próba	610562012	3668523136	2123451628	190423861
Średni czas	384460150,3	1042817992,7	549999346,9	228713150



Ciekawym zjawiskiem jest, że operacja wykonaywana współbieżnie w Fixed Thread Pool lub Cached Thread Pool wykonwała się wolniej od pojedynczego wątku. Powodem może być niestabilność systemu (inne procesy uruchomiły się w trakcie tych testów i zabużyły wyniki) oraz potrzeba zarządzania więszką pulą wątków.