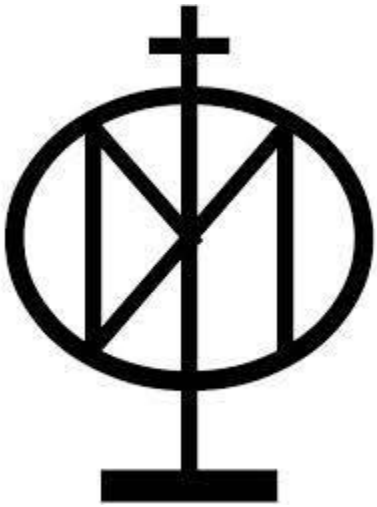
# Изобразяване на фрактал

Проект по Разпределени софтуерни архитектури



Изготвил: Денис Дуев, 61911

Проверил:.....

*/ас. Христо Христов/*

# Съдържание:

## 1. Условие на задачата

Нека разгледаме формулата:

$$F(Z) = Z^2 * e^{Z^2} + C$$

Вашата задача е да напишете програма за визуализиране на множеството на Манделброт, определено от формула (7). Програмата трябва да използва паралелни процеси (нишки) за да разпредели работата по търсенето на точките от множеството на Манделброт на повече от един процесор. Програмата трябва да осигурява и генерирането на изображение (например **.png**), показващо така намереното множество.

Изискванията към програмата са следните:

(о) Програмата да позволява (разбира от) команден параметър, който задава големината на генерираното изображение, като широчина и височина в брой пиксели. Той има вида: „**-s 640x480**“ (или **„-size“**); При не-въведен от потребителя команден параметър, за големина на изображението, програмата подразбира - широчина (width) **640px** и височина (height) **480px**;

(о) Команден параметър, който да задава частта от комплексната равнина, в която ще търсим визуализация на множеството на Манделброт: **„-r -2.0:2.0:-1.0:1.0“** (или **„-rect“**). Стойността на параметъра се интерпретира както следва: a∈[−2.0,2.0],b∈[−1.0,1.0] . При не въведен от потребителя параметър програмата приема че е зададена стойност по подразбиране: **„-2.0:2.0:-2.0:2.0“.**

(о) Друг команден параметър, който задава максималния брой нишки (паралелни процеси) на които разделяме работата по генерирането на изображението: **„-t 3“** (или **„-tasks“**); При не-въведен от потребителя команден параметър за брой нишки – програмата подразбира **1** нишка;

(о) Команден параметър указващ името на генерираното изображение: **„-o zad20.png“** (или **„- output“**). Съответно програмата записва генерираното изображение в този файл. Ако този параметър Зад. 20, 3/5 (1.2) е изпуснат (не е зададен от потребителя), се избира име по подразбиране: „**zad20.png**“;

(о) Програмата извежда подходящи съобщения на различните етапи от работата си, както и времето отделено за завършване на всички изчисления по визуализирането на точките от множеството на Манделброт (пресмятане на множеството на Манделброт); Примери за подходящи съобщения:

**„Thread- started.“,**

**„Thread- stopped.“,**

**„Thread- execution time was (millis): “,**

**„Threads used in current run: “,**

**„Total execution time for current run (millis):** “ и т.н.;

(о) Да се осигури възможност за **„quiet“** режим на работа на програмата, при който се извежда само времето през което програмата е работила (без „подходящите“ съобщения от предходната точка). Параметърът за тази цел нека да е **„-q“** (или **„-quiet“**); Тихият режим не отменя записването на изображението във изходният файл;

## 2. Преглед на алгоритмите за решение на задачата

Нека сега прегледаме какво можем да използваме за да решим задачата:
Ще започнем от това как се представят комплексните числа:

$$Z = a + bi, i = \sqrt{-1}$$

Комплексните числа могат да се разполагат в равнина като точки:
a – реалната част – кординатната точка на абсцисата
b – имагинерна част – координатната точка по ординатата

Ще използваме тези точки и тази координтна система за да озобрацим фрактала. Съответно ще трябва да проверяваме дали точката е част от фрактала – част от Mandelbrot set или не. В зависимост от това, можем да оцветим тази точка в различни цветове и да получим графика на фрактала.

Въпросът който обаче стои пред нас е как да разберем дали точката е част от множеството на Манделброд (Mandelbrot set)?

Ще използваме следната рекурсивна формула:

$$Z_n = Z_{n-1} + C$$

Където в нашата равнина $C = a + bi$

От горните редове, получаваме:

$$Z_1 = Z_0^2 + C, \text{като за } Z_0 = (0,0) \rightarrow Z_1 = C$$
$$Z_2 = Z_1^2 + C = C^2 + C$$
$$Z_3 = Z_2^2 + C = \cdots$$

Съответно определяме точките по следните правила:

- точките, които са далеч от (извън) множеството на Манделброт се придвижват сравнително „бързо“ (клонят „бързо“) към безкрайността;
- точките, които са близко до (но, все още извън) множеството на Манделброт се придвижват сравнително „бавно“ (клонят „бавно“) към безкрайността;
- точките, които са от множеството на Манделброт, никога не клонят към безкрайността;

## 3. Реализация на алгоритмите

В задачата като входни данни получаваме размерите на изображението, което трябва да се генерира (или ако не е подадено се взимат стойности по подразбиране). Тези размери определят големината на „равнината“ която разглеждаме. Съответно върху нашата равнина ще има width x height брой точки.
За всяка една от тези точки трябва да проверим дали е част от множеството на Манделброт. Ще използваме рекурсивния алгоритъм който дефинарахме по нагоре:

```
private static int zCheck(Complex c) {

    Complex zPrevious = new Complex(0.0, 0.0);
```

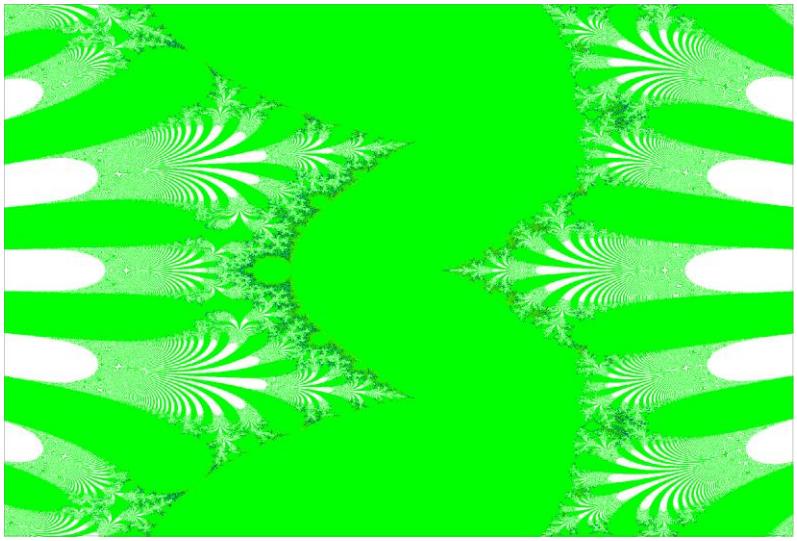
```
        Complex zIterable;

        int steps = 0;
        Double realPartOfZ;

        for(int i = 0; i < Main.dimWidth; i++) {

            zIterable = zIterate(zPrevious, c);
            zPrevious = zIterable;

            realPartOfZ = zPrevious.getReal();

            if (realPartOfZ.isInfinite() || realPartOfZ.isNaN()) {
                steps = i;
                break;
            }
        }

        return steps;
    }
```

Показаният метод приема комплекно число итерира през него чрез *zIterate* и накрая връща броя стъпки, които му е отнело, или това колко бързо се „придвижва“ точката. Самият метод *zIterate* извършва изчисленията зададени от условието на задачата:

$$F(Z) = Z^2 * e^{Z^2} + C$$

```
private static Complex zIterate(Complex z, Complex c) {
    return (z.multiply(z).multiply(E.pow(z.multiply(z)))).add(c);
}
```

След това намерената точка се оценява колко бързо расте и спрямо това се оцветява (задава ѝ се цвят) върху равнината, която след това ще се превърне в готовото изображение:

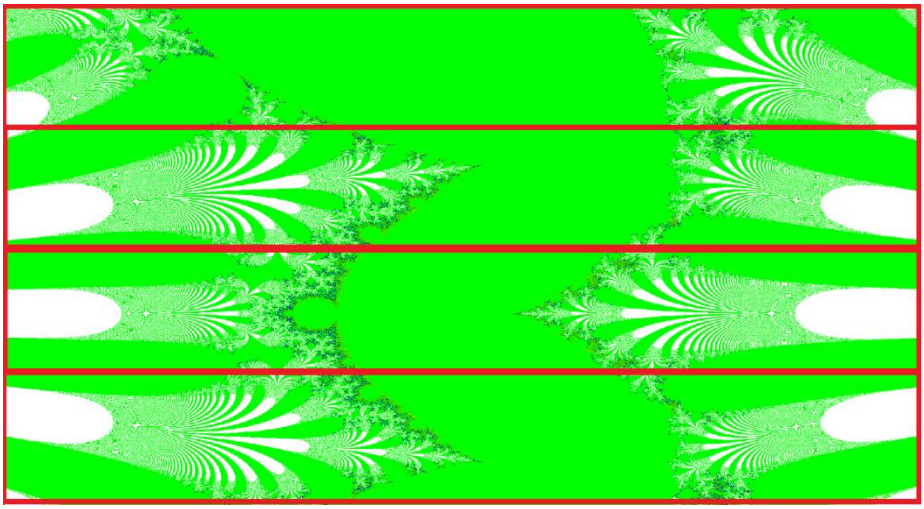


Как обаче е решен проблема с паралелизирането на изпълнението на задачата?
Има няколко неща които бяха взети предвид:
- след подаване на началните параметри се знае колко голяма ще бъде равнината -> следователно се знае колко точки има.
- изчислението на една точка, не е много времеотнемащо

Имаше два варианта:

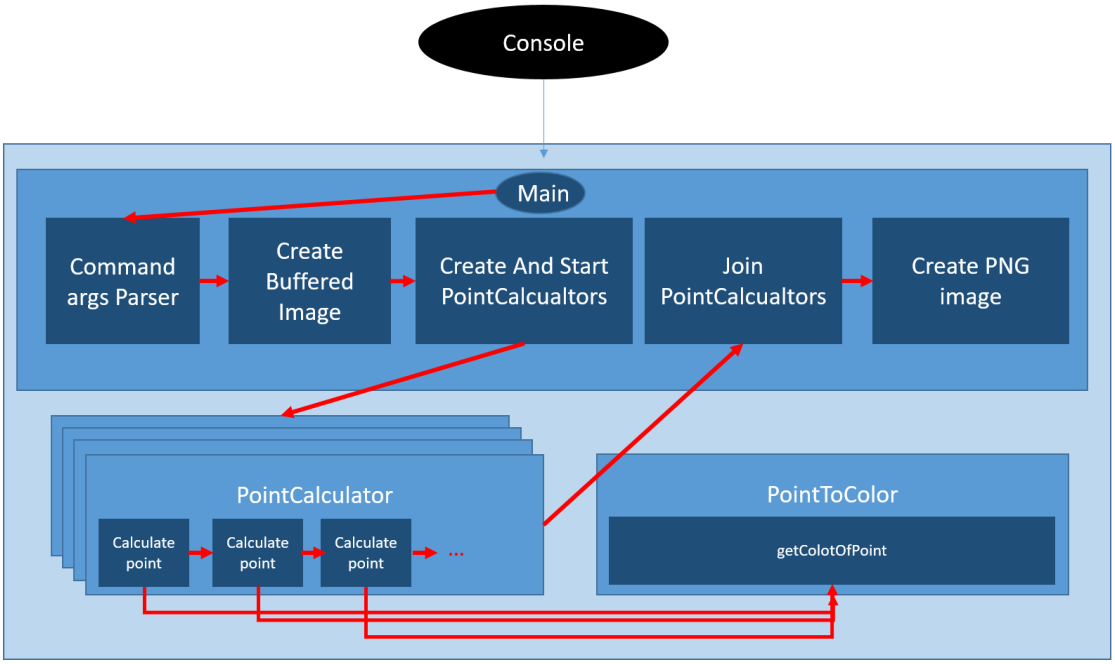
| Паралелизация за всяко пресмятане | Паралезация “на парче” |
|---|---|
| Този вариант идва пръв на ум, но след малко замисляни или опит, се вижда, че отнема повече време и ресурси, самото създаване на нишките, които ще извършват изчисленията. Губят се ценни ресурси при постоянното създаване на нови. | При вторият вариант взимаме предвид факта, че знаем колко е голямо исканото изображение, както и колко нишки ще го обработват. Съответно най-простото нещо е да разделим изображението на равни части за всяка нишка и накрая да изчакаме тяхното завършване. |

В проекта е реализиран вторият вариант. Пример ако имаме 4 нишки работата на всяка от тях би се разделила така:



В конкретният случай, броят редове се разделят на броят нишки. Ако числото не се дели без остатък, този остатък се разпределя по равно -> първите нишки взимат по 1 ред повече. (Пример ако нашето изображение има големина 490 и 4 нишки трябва да го генерират тогава се пада по 122.5 (което се закръгля до 122), съответно първите две нишки трябва да изпълнят с по един ред повече -> 123 + 123 + 122 +122 = 490). Така работата се разпределя „сравнително развномерно“ като не знаем кога в конкретното изчиление на дадена точка ще отнеме повече време и ресурси.

## 4. Архитектура




### Основните класове, които участват в системата:

Main класа: съдържа main метода, той парсва аргументите подадени от потребителя. Определя парчетата които всяка нишка (PointCalculator) ще се грижи да обработи, създава определените нишки и ги стартира. Създадените нишки започват обработката, а през това време основната програма ги чака. След като се увери че всички нишки са приключили се създава и крайното изображение като се ползват потребителските данни.

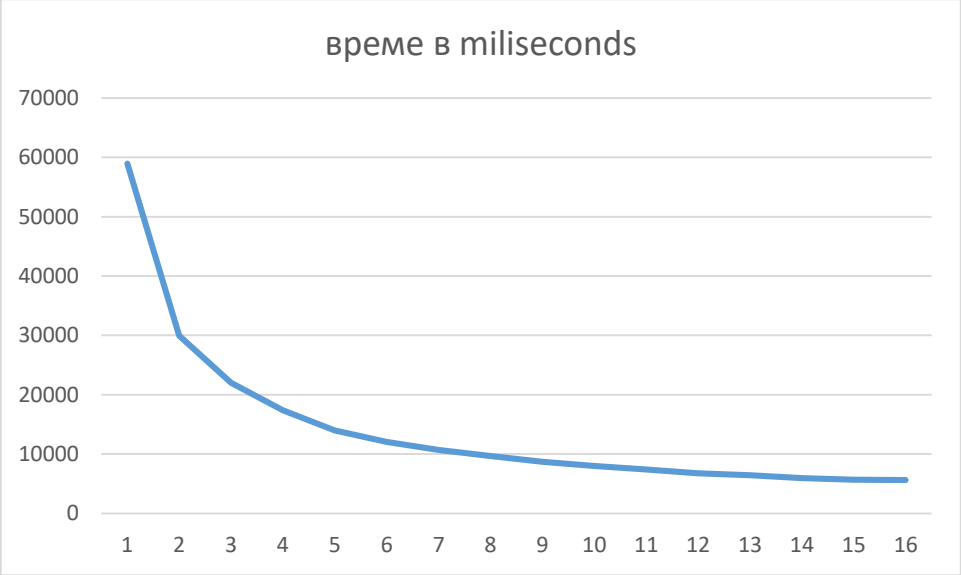
## 5. Проведени тестове и измервания

На предоставената машина за тестване (t5600.rmi.yaht.net) бяха проведени тестове. Тестовете стартират програмата с брой ядра от 1 до 16, и –q режим (за да се покаже само времето за изпълние) и дефолтни стойности за големина (640x480). Резултатите са както следва:

| брой ядра | време в miliseconds | Ускорение | Ефективност (ефикасност) |
|---|---|---|---|
| 1 | 58986 | 1 | 1 |
| 2 | 29954 | 1.96921947 | 0.984609735 |
| 3 | 22015 | 2.679354985 | 0.893118328 |
| 4 | 17388 | 3.392339545 | 0.848084886 |
| 5 | 13996 | 4.214489854 | 0.842897971 |
| 6 | 12042 | 4.898355755 | 0.816392626 |

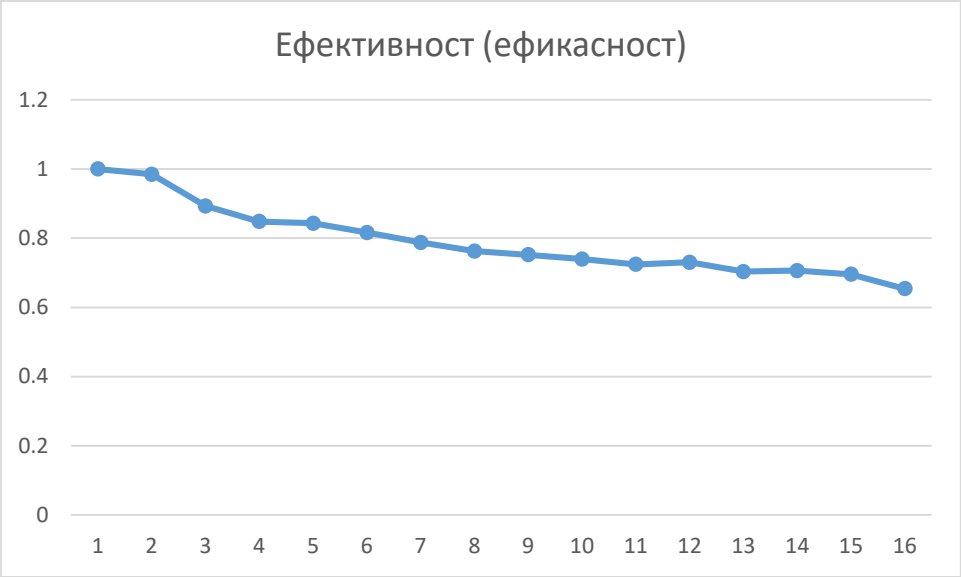
| | | | |
|---|---|---|---|
| 7 | 10699 | 5.513225535 | 0.787603648 |
| 8 | 9671 | 6.099265846 | 0.762408231 |
| 9 | 8714 | 6.769107184 | 0.75212302 |
| 10 | 7975 | 7.396363636 | 0.739636364 |
| 11 | 7404 | 7.966774716 | 0.724252247 |
| 12 | 6731 | 8.763333829 | 0.730277819 |
| 13 | 6451 | 9.143698651 | 0.703361435 |
| 14 | 5967 | 9.885369532 | 0.706097824 |
| 15 | 5655 | 10.43076923 | 0.695384615 |
| 16 | 5637 | 10.46407664 | 0.65400479 |

На диаграмата е показано времето в милисекунди при брой ядра от 1 до 16




На диаграмата е показано ускорението при брой ядра от 1 до 16

На диаграмата е показано ефективността (ефикасността) при брой ядра от 1 до 16




Изходен код на системата:
https://github.com/DenisDuev/rsa_fractal