

Курсов проект

по

Разпределени софтуерни архитектури

летен семестър, учебна година 2019/2020

**Тема 17**

**Изобразяване на фрактал по формулата**

Ръководители : проф. Васил Цунижев, ас. Христо Христов

Изготвил: Иван Ивов Чучулски

Факултетен номер : 62167

Специалност : Софтуерно инженерство

Дата : Проверил :……………………………..

1. **Анализ**
   1. **същност на проблема**

Фракталите са математически обекти, които представляват множество от точки върху комплексната равнина. Те са детайлно изследвани за пръв път в средата на 20-ти век от математика Беноа Манделброт. Фракталите намират приложение в моделирането на структурата на природни обекти и взаимодействията между тях, фрактална графика за компютърни изображения, анализ и компресия на сигнали, метод за създаване на картини и архитектурен дизайн и други.

Целта на проекта е да се създаде програма, която да визуализира фрактал, който е зададен чрез рекурентна формула **.**

Реализирано е решение на Java програми, което използва асинхронна паралелна обработка на escape-time алгоритъм за пресмятане и визуализиране на множеството на Манделброт. Направена е декомпозиция на данните SPMD, като е използвана средна грануларност при формиране на заданията – броят на заданията е равен на броя на редовете на изображението. Имплементирани са два начина за балансиране на изпълнението : статично и динамично.

* 1. **преглед на функционалността на приложения, които пресмятат множеството на Манделброт (функционален анализ)**

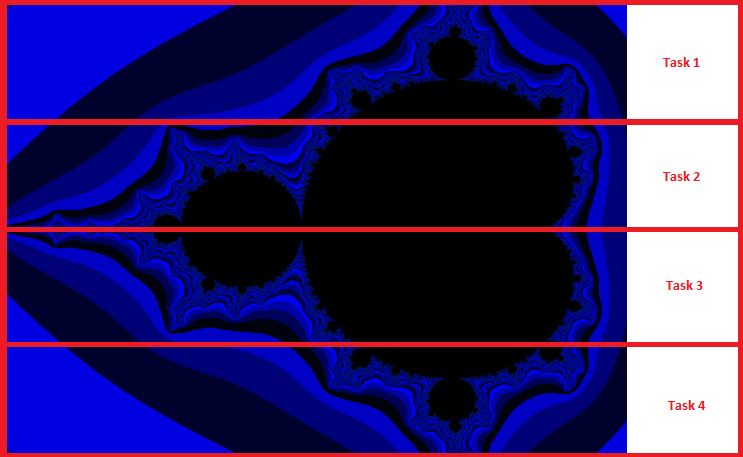
Източник [1] представя математическа дефиниция на множеството на Манделброт и escape-time алгоритъма за пресмятането и оцветяването на множеството. За да определим дали точка от комплексната равнина принадлежи на множеството на Манделброт, трябва да пресметнем границата на редицата, чиито членове се получават при прилагане на зададената рекурентна формула започвайки от точката . Ако границата на редицата от комплексни числа не клони към безкрайност при пресмятане на безбройно много членове на тази редица, то точката принадлежи на множеството. Разбира се, безкрайността не може да бъде точно моделирана в компютърните изчисления, затова идеята на escape-time алгоритъмът е да използваме големината на модула на комплексното число за определяне дали точката принадлежи на множеството. Задаваме число, което обозначава максималния брой итерации, които правим на всяка точка и пресмятаме броя на итерациите, за които модулът на числото остава по-малък от дадена стойност. Отношението на максималния брой итерации и броя итерации за дадена точка можем да използваме за задаване на цвят на точката при генериране на изображението. В [1] е представена реализация на последователна програма, която използва escape-time алгоритъма на езика C#.

Източник [2] и източник [3] представят два начина за имплементиране на паралелна реализация на escape-time алгоритъма на езика C. Първо е направена декомпозиция по данни SPMD и статично балансиране на изпълнението. Създаването и управлението на паралелните процеси става по модела Master-Slaves. Master частта разделя изображението на блокове, които се състоят от реда и колони. След това създава изпълняващите процеси и ги пуска. Първата нишка обработва първия такъв блок, втората работи върху втория и така нататък, докато всички не приключат.

След това източник [2] дава идеята за прилагане на различни схеми за статично балансиране на изпълнението. За статичното е на принципа “round robin”, т.е. нишките се редуват в разпределянето на задачите. Представени са и два начина за правене на динамично балансиране – централизирано и децентрализирано. Първия е Master-slaves модел : заданията се разпределят по време на изпълнение от един процес, наречен Master на множеството обработващите процеси наречени slaves. Между slave процесите няма комуникация и обмен на данни. При втория начин всички задания първоначално се разпределят между обработващите процеси и в хода на изпълнение те могат да дават задания помежду си, с цел да се уеднакви натоварването.

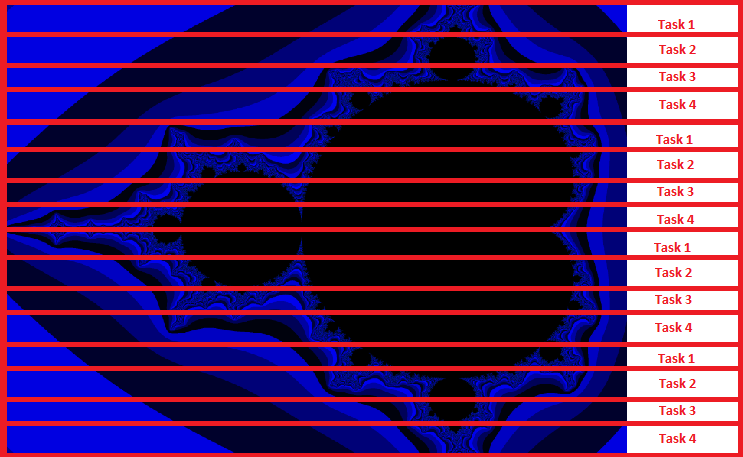
В източник [3] е представено решение, като грануларността е средна и е използвано динамично балансиране с централен процес, т.е. топология звезда. Изображението е разделено на задания, като заданията представляват хоризонталните редове на изображението. Master частта създава масив, която да представлява заданията, след това създава обработващите процеси и ги пуска да работят. Те трябва да достъпват масива със заданията, за да вземат задание, след това го да го обработят и отново да потърсят задание, докато няма повече задания. Master частта изчаква всички Slaves да приключат и след това програма приключва. Обозначено е, че достъпването на масива със заданията трябва да не наруши структурата от данни и следователно обработващите процеси трябва да осигурят синхронизация помежду си, когато взимат задание.

Източник [4] представлява имплементация на първата идея от източник [2] и [3] на езика Java,т.е. Master-Slaves програма, при която е направена SPMD декомпозиция по данните на блокове с големина реда и колони, и е използвано статично балансиране на изпълнението.



Фигура 1. Декомпозиция на данните на блокове и статично балансиране на изпълнението, източник [3]

Също така е представена втора имплементация на Master-Slaves програма, при която е направена SPMD декомпозиция по данните и използвана по-фина грануларност, като са задавани различен брой редове, които да се съдържат в даден блок. Балансирането на изпълнението отново е статично.



Фигура 2. Декомпозиция на данните по редове и статично балансиране на изпълнението, източник [3]

Следната сравнителна таблица показва характеристиките на паралелните решения, разгледани в източниците. Всички те представят асинхронна паралелна имплементация на escape-time алгоритъма по модела Master-Slaves.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Образец | Декомпозиция | Грануларност | Балансиране | Коментар |
| Пример 1 от [2] и [3], Пример 1 от [4] | По данни, SPMD | Едра, блокове, които се състоят от по (височина на изображението)/(брой нишки) реда и (ширина на изображението) колони | Статично, всяка нишка обработва блок, който съответства на номера ѝ. Нагледно показано на Фигура 1. | При използването на едра грануларност и статичното балансиране е възможно да се получи сериозен дисбаланс в товара на различните нишки. Тези, които получат области със много точки от фрактала ще се забавят и от там цялата паралелна част. |
| Пример 2 от [2] | По данни, SPMD | Средна, редовете на изображението са оформени като индивидуални задания. | Динамично с централен процес, звездовидна топология. Заданията са в масив, който се съдържа в централният процес. | Средната грануларност и динамичното разпределение на заданията гарантира равномерно разпределяне на товара между нишките. Остава въпрос за оптимизиране на структурата от данни със заданията и начина по който се прави синхронизацията между обработващите процеси. |
| Пример 2 от [4] | По данни, SPMD | Различна грануларност, от едра към средно-едра. Коефициентът на грануларност е вариран между 8, 16 и 32. Изображението се разделя на блокове, която имат по (коефициент на грануларност \* брой нишки) реда и (ширина на изображението) колони. Броят на блоковете е (коефициент на грануларност \* брой нишки). | Статично разпределение. Нишка с индекс t, обработва блок с индекс b, ако  b % (брой нишки ) == t.  Нагледно показано на Фигура 2. | По-добър вариант от Пример 1 от [4], тъй като по-фината грануларност и „цикличното“ разпределение би „разбило“ секторите, в които има повече точки от множеството на няколко по-малки парчета, които да се обработят от няколко нишки. |

* 1. **нефункционален анализ**
     1. **технологии, използвани в решението**

За реализирането на двете решения е използван езика **Java** версия 8. Стандартната библиотека на езика предлага удобна и проста абстракция на нишка, класа **Thread**, която използваме за имплементиране на изчислителните процеси. От пакета за паралелно програмиране използваме и класа **ConcurrentLinkedQueue**, който служи за опашката от задания. За генерирането на изображението на фрактала е използван класа **BufferedImage**. За оцветяването е използван метод, който превръща модела HSB(Hue, Saturation, Brightness) в RGB(Red, Green, Blue).

За обработването на аргументите от командния ред е използвана библиотеката **Аpache commons-cli-1.4**. С нейна помощ се прави разделянето на опциите и взимането на техните стойности, които се подават при стартиране на програмата. За извършване на изчисленията с комплексни числа е използвана библиотеката **Apache commons-math3-3.5**. От нея са използвани класа, който представя комплексно число **Complex**. В този клас статичните методи за косинус **cos()** и експоненциална функция **exp()**служат за пресмятането на математическите функции, които участват във рекурентната формула на фрактала.

Изборът на Java за език на реализацията дава на програмата доста широка преносимост, тъй като има множество имплементации на JVM(Java Virtual Machine) на повечето съвременни хардуерни и софтуерни платформи. Също така това гарантира безпроблемното използване на паралелна обработка на всяка от тези платформи, като ние можем да използваме API-то предложено от Java, а не специфичното за съответната платформа. Разбира се, това означава допълнително ниво на абстракция, предоставено от Java, което ще доведе до по-ниско ускорение в сравнение с използването на механизмите за паралелна обработка, които например са директно предоставени от ядрото на дадена операционна система.

* + 1. **инфраструктурни изисквания**

За решението е използват софтуерния модел Master-Slaves. Изборът на Java за език за реализиране на решението и използване на предоставените от него абстракции за паралелна обработка може да гарантира сравнително еднаква производителност на различните платформи, които поддържат JVM.

* + 1. **обосновка на избраното решение**

Първоначалната беше обмислена реализация на програмата като Пример 1 от източник [2] и Пример 1 от [3]. При този метод обаче има няколко проблема като на първо място поради едрата грануларност е възможно някои нишки да получат блок, който се състои главно от точки в множеството, като това означава по-тежки сметки и така тяхното изпълнение ще се забави. Тъй като балансирането е статично, други нишки, които са получили по-лек блок и приключили работата си по-бързо няма как да помогнат на тези, които се бавят.

За решението на тези проблеми първо е използването на по-фина грануларност: средна грануларност, при която заданията няма да бъдат блокове от по няколко реда, а всеки хоризонтален ред от изображението представлява отделно задание.

Също така може да се направи различно статично балансиране, при което нишките няма да обработват последователни редове, а вместо това нишка обработва първо реда, който съответства на нейния индекс, после „прескача“ „брой на нишките“ редове напред, пак обработва един ред и пак „прескача“ докато не се изчерпат редовете.

След това е имплементирана опция на програмата, при която се извършва динамично балансиране при разпределяне на заданията между обработващите процеси. По този начин нишки, на които се паднат по-лесни задания и те успяват да ги обработят бързо, могат да помогнат с обработването на по-интензивни задания, които при статично балансиране биха могли да се паднат на изцяло на една нишка.

1. **Проектиране**
   1. **описание на реализирания алгоритъм**

Имплементиран е асинхронен паралелен алгоритъм, софтуерният модел е **Master**-**Slaves** направена е декомпозиция по данни SPMD. Всеки ред от изображението е отделно задание, например при размер 1920 на 1440 се обособяват 1440 задание от по 1 пиксел на височина и 1920 пиксела широчина. При бройка на обработващите процеси между 1 и 24, отношението е , което означава средна грануларност. Имплементирани са два начина за балансиране на изпълнението: статично и динамично с централен процес и опашка, от която обработващите процеси взимат заданията.

**Main** класът е в ролята на **Master**. Първоначално се обработват аргументите от командния ред. С тях се задават големината на изображението, крайните точки на комплексната равнина, името на изходния файл, максимален брой итерации на точка, начина на балансиране на изпълнението и дали програмата да работи в „тих режим“, т.е. дали да изписва съобщения за започване и завършване на обработващите процеси или просто времето, за което се е изпълнила програмата. След това се създава обекта от класа **BufferedImage**, който представлява изображението и се запълва с бял цвят. Следва проверка за вида на балансирането, което да бъде използвано.

Общото при двата начина на балансиране е алгоритъмът за изчисляване на точките от множеството на Манделброт и той е базиран на escape-time алгоритъма. При него първо за всеки пиксел от заданието се намира съответстващата му точка в комплексната равнина. За тази комплексна точка в метода **calculateNumberOfSteps()** се изчисляват членовете на редицата, определени от рекурентната формула **.** Ако членовете на тази редица не нарастват към безкрайност, означава че точката принадлежи на фрактала. Когато това е изпълнено, то се извършват максималният брой итерации пъти пресмятания(по подразбиране 500) по формулата и заради това, задания с много точки от множеството отнемат много изчислителен ресурс. В противен случай, изчисленията завършват и се връща достигнатия брой итерации. На база на тази бройка обработващите процеси определят цвета на точката чрез метода **getHSBToRGBColor()** и след това изпълняват метода **drawPixel()**,който записва резултата в изображението. Тъй като изображението е споделен ресурс във всички нишки, то този метод е **synchronized**. Това означава, че кодът в него е критична секция и само една нишка може да го изпълнява в даден момент от време.

Ако е избрано статично балансиране, то Main класът създава обработващите процеси, които са от обекти от класа **StaticSlaveThread**. При създаването на всяка нишка ѝ се задава индекс и референция към обекта за изображението **BufferedImage**. **Master**-ът ги пуска да работят като извиква техния **run()** метод. В него всяка нишка започва да обработва реда с номер равен на индекса си и след това отива на реда с номер (сегашен номер на ред + брой на нишки) и така отново докато не надхвърли броя на редовете. Илюстративно това е показано на Фигура 2, но разликата е, че там нишка обработва по няколко реда един след друг. **Master**-ът изчаква всички нишки да приключат работата си, като извиква метода им **join().**

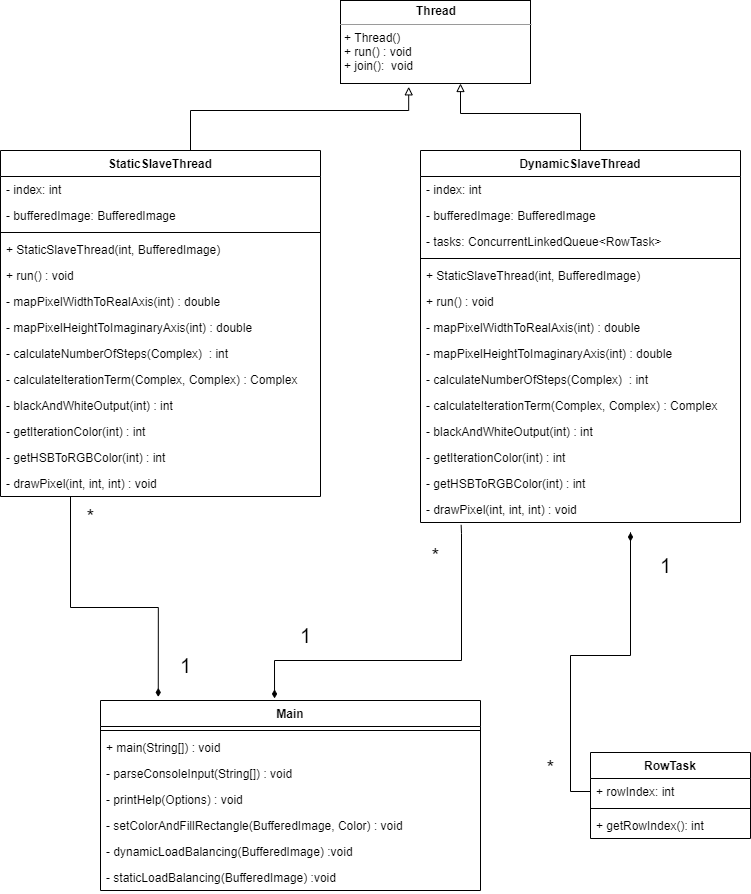
При избрано динамично балансиране Main класът първо създава опашката от задания, които са обекти от класа **RowTask**. Този клас е създаден единствено с цел по-ясно дефиниране на абстракцията задание. Той има една член-данна и тя е номера на реда, който представлява това задание. Самата опашка е обект от класа **ConcurrentLinkedQueue**. След това **Main** класът създава **DynamicSlaveThread** обектите, които са **Slave** процесите, като им се задава индекс, референция към обекта за изображението **BufferedImage** и референция към обекта за опашката от задания. **Master**-ът пуска нишките да работят като извиква **run()** метода на **DynamicSlaveThread.** В този метод всяка нишка взема задание от опашката със задания, обработва го и записва резултата в изображението и проверява за ново задание в опашката. Ако няма такова, то нишката приключва изпълнението си. **Master** класа изчаква всички нишки да приключат работата си, като извиква метода им **join().**

Изборът на структура от данни за опашката от задания първоначално беше интерфейса **Queue** и класа **LinkedList**, който е една от имплементациите му във вид на свързан списък. За да се осигури правилното паралелно достъпване на опашката беше използван механизма със **synchronized** метод, който позволява само на една нишка да изпълнява кода, който вади елемент от опашката. При увеличаване на броя на нишките, честата синхронизация при вземане на задание забавяше цялата паралелна част на програмата. Затова имплементацията на опашката със задание беше сменена на **ConcurrentLinkedQueue**. Този клас е от пакета на **java.util.concurrent**, в който има варианти на основните структури от данни, които са пригодени за използване от повече от една нишка, като се гарантира целостта на данните и правилния резултат[6].

След тази промяна динамичното балансиране показа равни по ускорение резултати на статичното, а при тестване на по-небалансирани участъци от фрактала полученото ускорение беше по-високо. Това се дължи на факта, че при изполването на **synchronized** метод в класа **DynamicSlaveThread** за достъпване на опашката синхронизацията е „по-груба“, тъй като целият код на метода **poll()** на опашката е маркиран като критична секция. При използване на **ConcurrentLinkedQueue** класа само отделни части на метода са критични секции, по-точно само промените при референциите към възлите на опашката, които са от тип **AtomicReference.** Така синхронизацията е върху по-малък брой инструкции и по-късата критична секция означава по-голяма пропускливост и това води до по-бързо изпълнение дори и при голям брой нишки[5], [6].

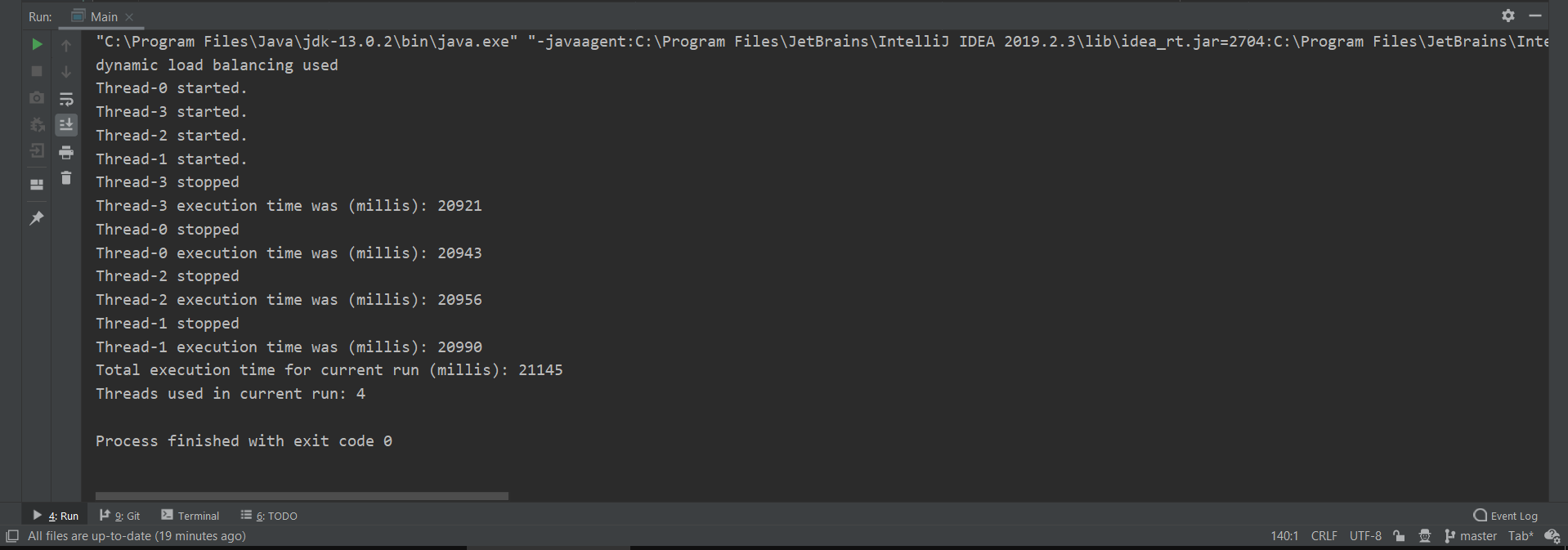
Main класа изчаква обработващите процеси да приключат работата си и записва изображението във файл.

* 1. **диаграма на класовете**

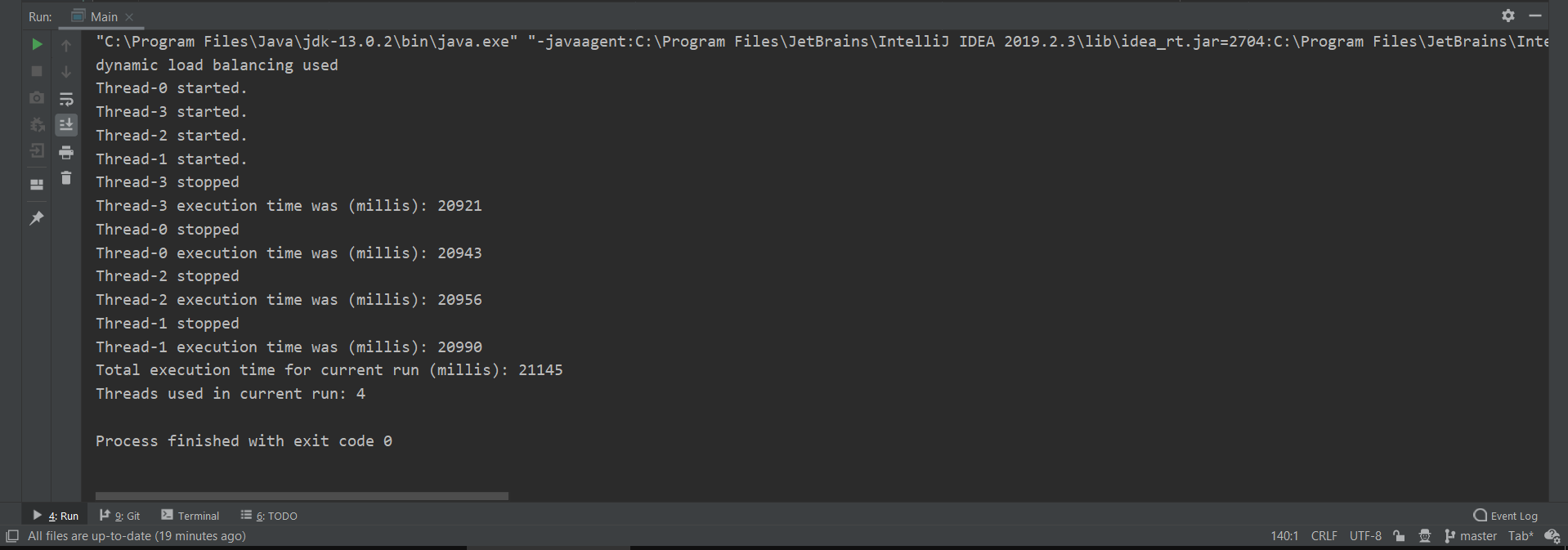


Фигура 3. Клас диаграма

* 1. **интерфейсни екрани**

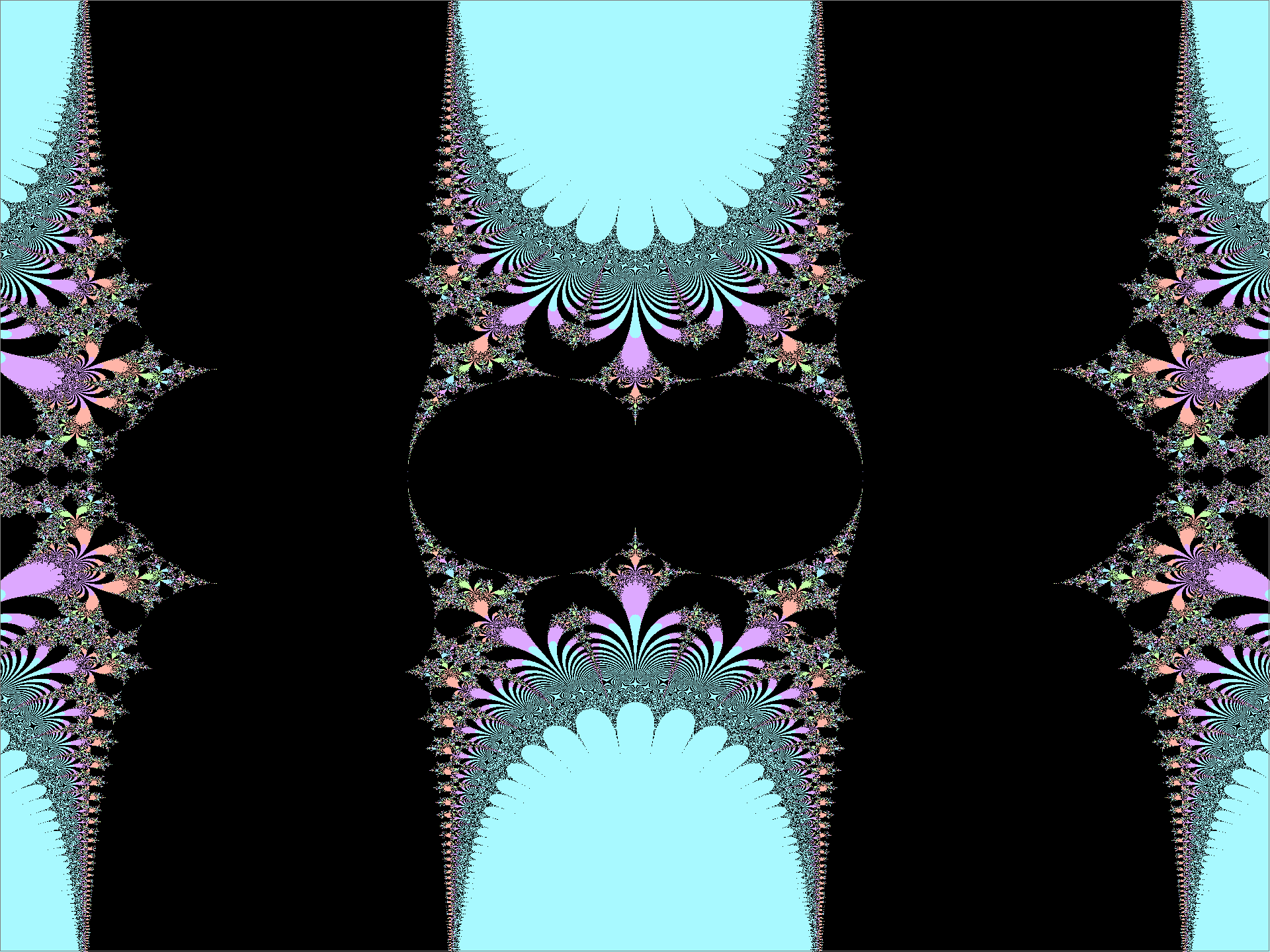


Фигура 4. Изпълнение на програмата с опцията за показване на помощ



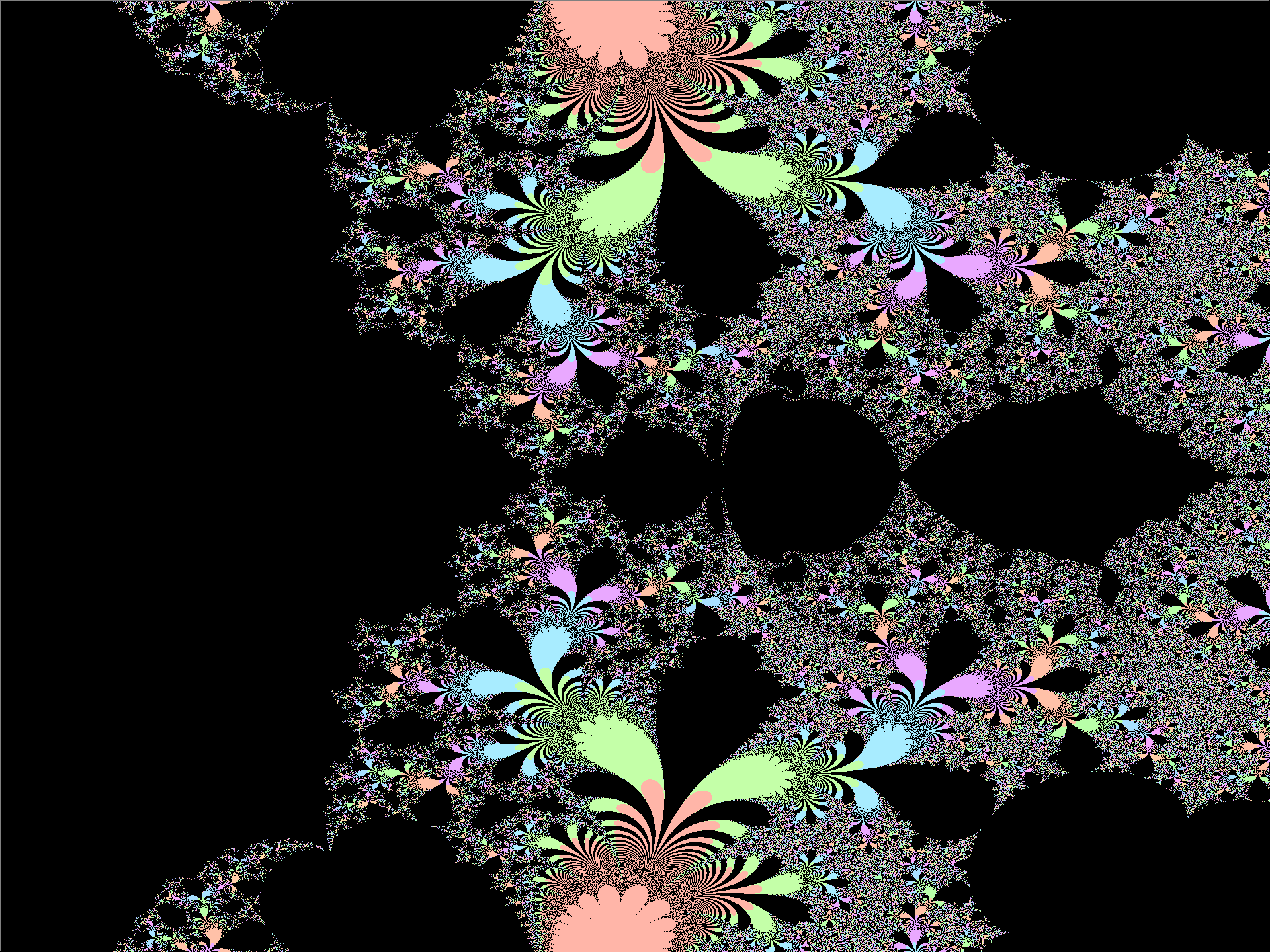
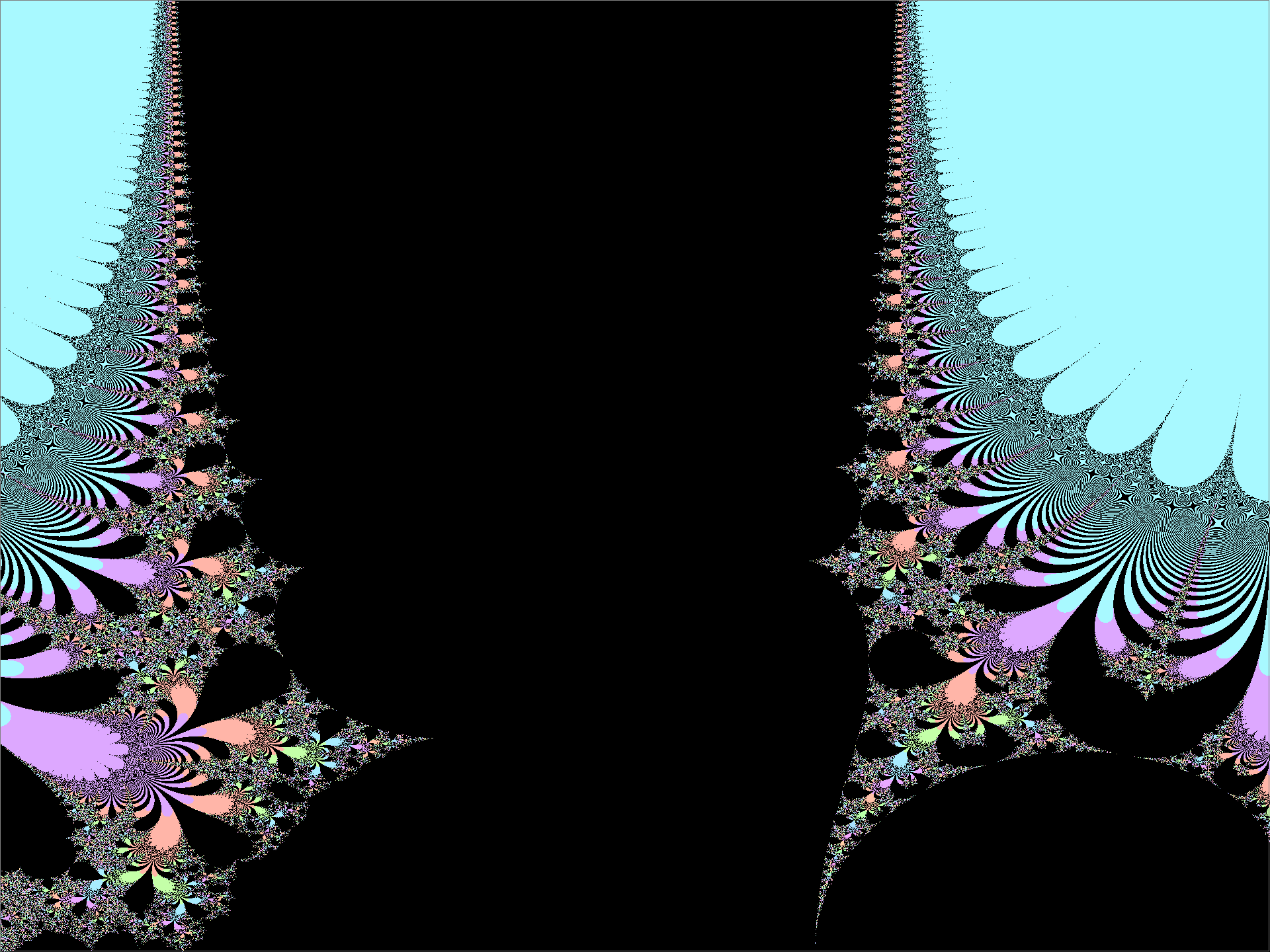
Фигура 5. Изпълнение на програмата с опцията за показване на помощ

* 1. **фрактални изображения**

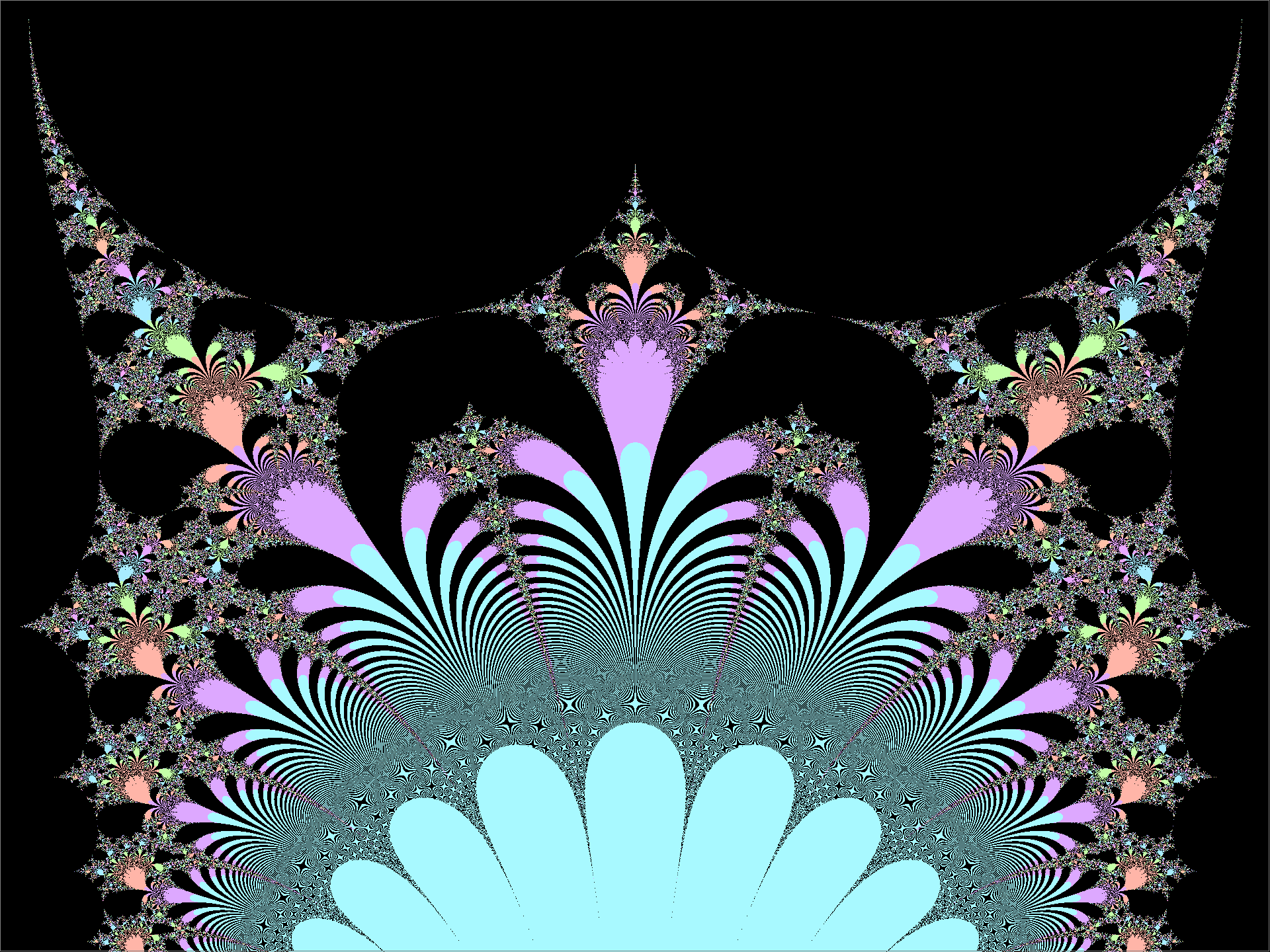


Фигура 6. фрактал, област -2.0:2.0:-2.0:2.0, размер 1920x1440

Фигура 7. област -2.0:2.0:-2.0:2.0, размер 1920x1440



Фигура 8. област -1.5:2.0:-0.25:0.25, размер 1920x1440



Фигура 9. област -0.75:0.75:0.0:1.25, размер 1920x1440

* 1. **изисквания към платформата**

Програмата може да се изпълнява на операционни платформи, които имат инсталирана JDK(Java Development Kit) от версия 8 нагоре и поддръжка на паралелна обработка. За визуализиране на изображенията, които са резултата от програмата е нужен екран с минимална резолюция 1280x960.

1. **Тестов план и тестови резултати**
2. **Анализ на получените резултати**
3. **Източници**

[1]Bastian Fredriksson, “An introduction to the Mandelbrot set”, публикуван: 01.2015г, (<https://www.kth.se/social/files/5504b42ff276543e4aa5f5a1/An_introduction_to_the_Mandelbrot_Set.pdf>)

[2]Martin Johnson, “Massey University, Parallel Computing, lecture 3 Parallel techniques, lecture 7 Asynchronous computations”, 2009г, (https://www.massey.ac.nz/~mjjohnso/notes/59735/)

[3]Douglas Thain, „University of Notre Dame, Operating System Principles, Project 3“, 2020г, (https://www3.nd.edu/~dthain/courses/cse30341/spring2020/)

[4]Thomas Uhrig, “Calculating Mandelbrot Set with Java Tasks”, публикуван: 05.11.2012г, (<https://tuhrig.de/calculating-mandelbrot-set-with-java-tasks/>)

(<https://bitbucket.org/wordless/mandelbrot/src/master/Mandelbrot/>)

[5]Brian Goetz, “Java Concurrency In Practice” 2006, издател : “Addison-Wesley”

[6]Eugen Paraschiv, “LinkedBlockingQueue vs ConcurrentLinkedQueue”, последна модификация:03.06.2020г,

(<https://www.baeldung.com/java-queue-linkedblocking-concurrentlinked>)

1. **Списък с фигури**

Фигура 1. Декомпозиция на данните на блокове и статично балансиране на изпълнението., източник [3] (<https://tuhrig.de/calculating-mandelbrot-set-with-java-tasks/>)

Фигура 2. Декомпозиция на данните по редове и статично балансиране на изпълнението, източник [3]( <https://tuhrig.de/calculating-mandelbrot-set-with-java-tasks/>)

Фигура 3. Клас диаграма

Фигура 4. Изпълнение на програмата с опцията за показване на помощ

Фигура 5. Изпълнение на програмата с опцията за показване на помощ

Фигура 6. фрактал, област -2.0:2.0:-2.0:2.0, размер 1920x1440

Фигура 7. област -2.0:2.0:-2.0:2.0, размер 1920x1440

Фигура 8. област -1.5:2.0:-0.25:0.25, размер 1920x1440

Фигура 9. област -0.75:0.75:0.0:1.25, размер 1920x1440