Проект по Разпределени софтуерни архитектури

**Изобразяване на фрактал**

Изготвил: Виктор Христов, СИ 3, №62151

Дата: 15.06.2020

Съдържание

[1. Увод 2](#_Toc43066272)

[1.1. Цел и предназначение на проектираното приложение 2](#_Toc43066273)

[1.2. Преглед на решения на задачата на Манделброт 2](#_Toc43066274)

[1.3. Извод от разгледаните решения на задачата на Манделброт 7](#_Toc43066275)

[2. Проектиране 9](#_Toc43066276)

[2.1. Диаграми и дизайн на програмата 9](#_Toc43066277)

[2.2. Подробно описание на изпълнението на програмата 11](#_Toc43066278)

[3. Тестване 13](#_Toc43066279)

[3.1. Тестова архитектура 13](#_Toc43066280)

[3.2. Тестови параметри 13](#_Toc43066281)

[3.3. Тестване с променлива размер на изображението 13](#_Toc43066282)

[3.4. Тестване с променлива сектор на изображението 13](#_Toc43066283)

[3.5. Тестване с променлива коефициент на грануларност 13](#_Toc43066284)

[4. Списък източници 13](#_Toc43066285)

[Parallel Implementation and Analysis of Mandelbrot Set Construction, 13](#_Toc43066286)

[Parallel Fractal Image Generation - A Study of Generating Sequential Data with Parallel Algorithms, 13](#_Toc43066287)

[Parallel Mandelbrot in Julia, C++, and OpenCL, 13](#_Toc43066288)

[Tutorial: Develop an Application with Intel® Threading Building Blocks - Overview, 13](#_Toc43066289)

# Увод

# Цел и предназначение на проектираното приложение

Целта на този проект е да се създаде програма, която генерира изображение на множество на Манделброт с функция **Zn+1=C∗cos(Zn)** за комплексно число C чрез реализиране на паралелен алгоритъм.

Множеството на Манделброт е вид фрактал. Това са фигури, които рекурсивно съдържат себеподобни фигури по своите граници. Всеки фрактал се изобразява чрез множество от комплекси числа. Комплексните числа се пресмятат чрез използване на комплексна равнина. Това е координатна система, при която реалната част на числото се представя чрез абцисната ос, а имагинерната - чрез ординатната. След като се знае стойността на комплексното число, то тогава може да се определи дали то принадлежи към конкретния фрактал. Прилага се следната система:

1. Z0=0
2. Zn+1=C∗cos(Zn)

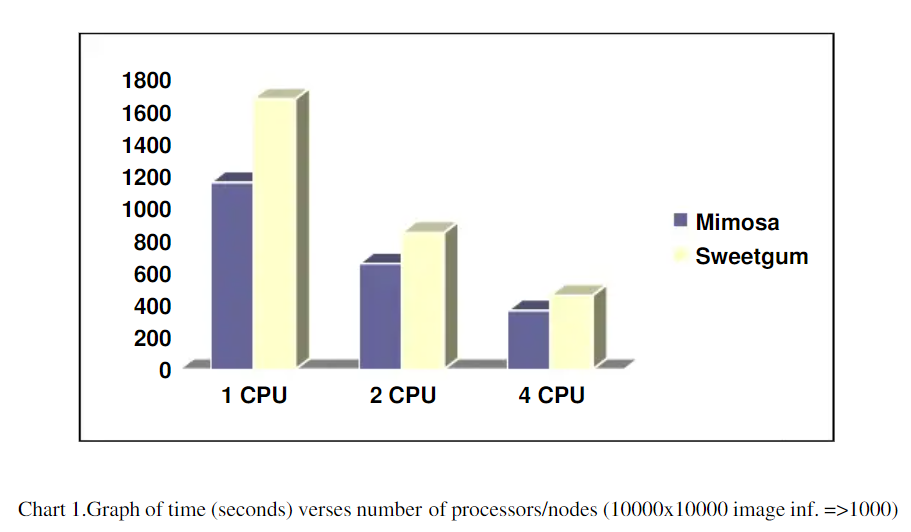
Ако след избран от програмиста брой итерации на функцията не може да се определи дали комплексното число клони към безкрайност, то тогава се приема, че принадлежи към това множество на Манделброт.

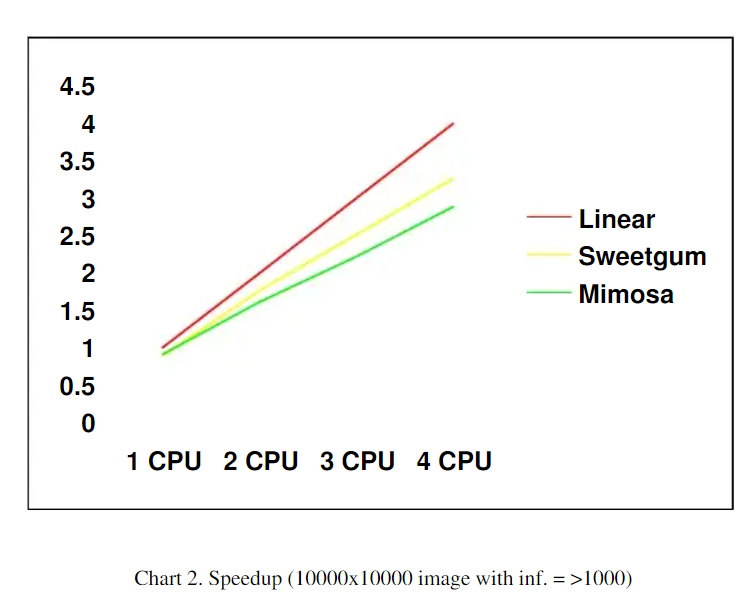
За да се генерира пълното изображение, ще е нужно да се изчисли всяка точка от зададената комплексна равнина, независимо дали е част от множеството на Манделброт или не. Единствено след като е приключило изчислението ще може да се запази генерираното изображение във визуален формат.

# Преглед на решения на задачата на Манделброт

В тази подточка ще се разгледат три решения на задачата за изобразяване на множество на Манделброт. Всяко от тях обяснява какво представлява задачата, как чрез комплекси числа се изобразява на координатна система и как се пресмята дали едно число е част от множеството. Поради това при този анализ ще се обърне внимание само на по-значителните части от статиите, главно какво алгоритми прилагат решенията.

1. Parallel Implementation and Analysis of Mandelbrot Set Construction [[1]](#_Parallel_Implementation_and)

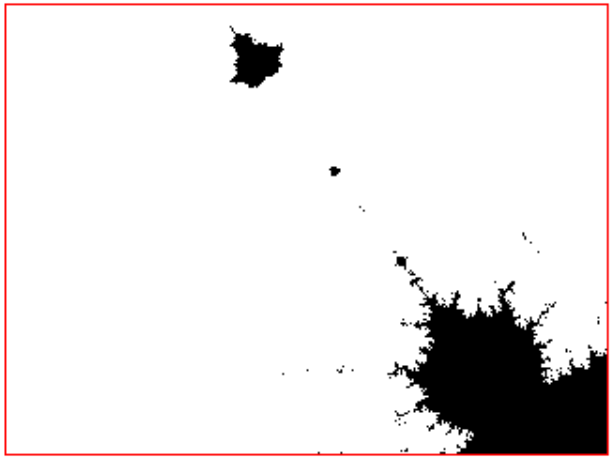
* Авторът обяснява „escape time“ алгоритъма, чрез който се извършва оцветяването на всеки пиксел от изображението. Алгоритъмът се състои от многократно изчисляване на функцията Zn+1=Zn2 + C. Константата C се определя спрямо координатите на пиксела, който в момента се пресмята, а броят максимални итерации е предварително зададен от програмиста. При всяка итерация се извършва проверка дали е удовлетворено условието за излизане.
  + Ако е изпълнено това условие, то се приема, че пиксела е извън множеството на Манделброт и се оцветява спрямо броя на итерации, които се били нужни, за да се изпълни условието.
  + Ако всички итерации приключат без да се изпълни условието, то се приема, че пиксела принадлежи на множеството и се оцветява в един конкретен цвят, като общоприетия вариант е черен цвят.
* Авторът обръща внимание на проблема, че при разделянето на изображението на големи задания се получа лошо балансиране на работата на процесите, и предлага своето решение.
* Идеята на неговия паралелен алгоритъм е всеки процес да изчислява случайно избрани пиксели от изображението. Инициализират се два вектора, които представляват интервала на координатите X и Y от комплексната равнина. На случаен принцип се генерира число, спрямо което стойностите на векторите се разбъркват. След това векторите се използват, за да се генерира изображението.
* Комуникацията между процесите е минимизирана. Преди да се извършат изчисленията, процесите получават информация за векторите, след което използват тази информация, за да пресметнат заданията, които са им разпределени. Накрая главния процес събира всичките задания и сглобява пълното изображение.

Фигура 1: Графика на тестови резултати представящ време за изпълнение спрямо брой процеси

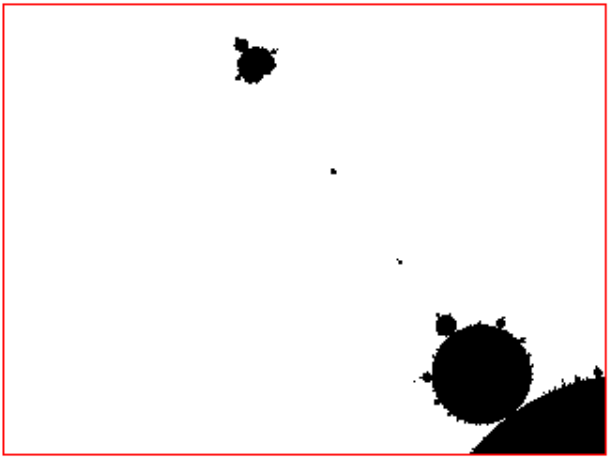
Фигура 2: Графика на тестови резултати представящ ускорение спрямо брой процеси

* Авторът е показал следните графики като резултат от проведените тестове. Полученото ускорение е близо до линейното, но не може да се направи точен извод, защото е тествано единствено до 4 процеса. Още от изпълнение с 3 процеса започва да се спада ефективността, което подсказва че при тестване с повече от 4 процеса резултатите няма да са задоволителни.
* Под въпрос е колко е правилно да се използва случайно-генерирано число при разбъркване на стойностите на векторите. Случайността призовава единствено допълнителен хаос и несигурност в програмата. В повечето случаи ще се получи разбъркване на координатите по такъв начин, че всеки процес да обработва равномерно-балансирани задания. Но в някои случаи разбъркванията ще доведат до концентриране на точките от множеството в малък брой задания и няма да реши проблема на едрата грануларност. Този проблем щеше да си проличи, ако авторът беше показал подробно тестване на алгоритъма.
* Авторът описва своя алгоритъм на високо ниво и без показване на примерен код, поради което възникват въпроси относно реализацията на паралелния алгоритъм. Неговият подход решава проблема за балансирането на работата в повечето случаи, но не се обяснява как точно заданията се разпределят между процесите след като те получат информацията от векторите.

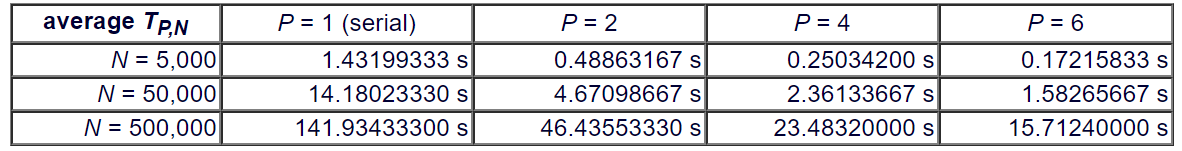
1. Parallel Fractal Image Generation - A Study of Generating Sequential Data with Parallel Algorithms [[2]](#_Parallel_Fractal_Image)

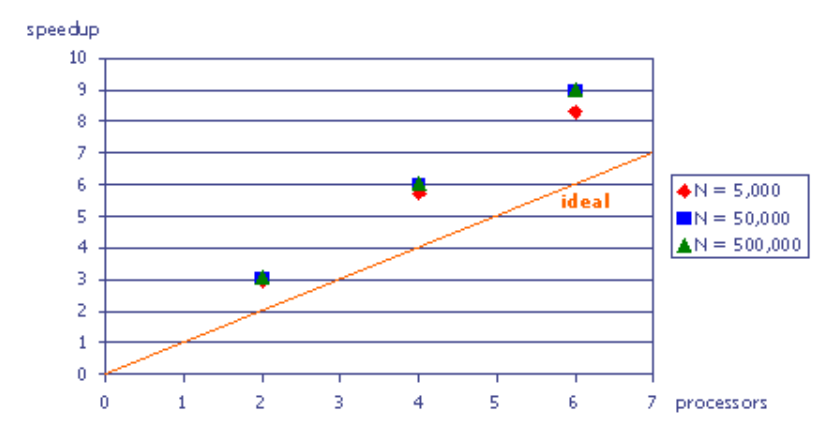
* Авторът на тази статия подчертава важността от броя на итерациите при изчисляване дали точка принадлежи на множеството на Манделброт. Някои точки се нуждаят от огромен брой итерации, за да се изпълни условието на „escape time“ алгоритъма, споменат в първия източник.

Фигура 3: Сектор от множеството на Манделброт след изчисление с 150 итерации



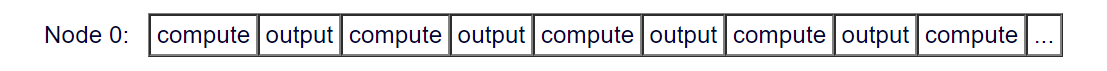
Фигура 4: Сектор от множеството на Манделброт след изчисление с 1500000 итерации

* Колкото повече итерации се извършват, толкова по-правилно ще е крайното изображение. Но от друга гледна точка, колкото повече итерации се извършват, толкова по-бавно ще се генерира изображението и приключи изпълнението на програмата. Поради това е важно да има компромис между точността и бързодействието на алгоритъма.
* Относно алгоритъма за паралелна обработка, авторът също разглежда проблема за балансиране на данните и показва, че при едра грануларност малък брой процеси ще вършат по-голямата част от работата. Като решение на проблема предлага вместо отделни сегменти всеки процес да изчислява определени редове от изображението. Балансирането на данните се извършва статично. Първия процес изчислява 1ви, 5ти, 9ти... ред от изображението, втория процес изчислява 2ри, 6ти, 10ти... ред и аналогично за останалите процеси.
* Авторът отбелязва, че това не перфектно решение. Причината е, че някои редове изискват повече изчисления от други, но този подход е достатъчно ефективен за повечето видове фрактали.
* Самото изчисление не изисква комуникация между процесите, защото всеки точка от изображението е независима една от друга. Комуникация е нужна единствено при сглобяване на крайния резултат. В тази имплементация финалният резултат се принтира на конзолата като ASCII изображение. Това довежда до нужда от синхронизация между процесите, защото те изчисляват алтерниращи редове, но трябва всеки процес да си изведе редовете в конзолата в правилна подредба.
* Авторът решава този проблем чрез имплементиране на „Polling“ синхронизация. Чрез този алгоритъм извеждането на завършеното изображение се разделя на сегменти с еднакъв размер. Главния процес изпраща съобщение към останалите процеси, чрез което иска да получи първите изчислени редове на всеки процес. След това главния процес изчислява първия свой определен ред и ако при завършване на изчислението е получил първите редове на всички процеси, то тогава целият сегмент се принтира и се изтрива от паметта. В противен случай главният процес продължава да изчислява следващите свои редове докато не получи отговор от всички други процеси. След извеждането на първия сегмент, алгоритъмът се повтаря за следващите сегменти докато не се изведе цялото изображение.
* Авторът използва този алгоритъм, защото си е поставил ограничение, че никой процес не трябва да съдържа информация за цялото завършено изображение. Ако нямаше това ограничение, то можеше главния процес да съдържа буфер, в който останалите процеси да записват своите редове и да се премахне напълно нуждата от синхронизация.
* За проверка на алгоритъма се изпълняват тестове върху изображение с размер 160x120. Получени са следните резултати:

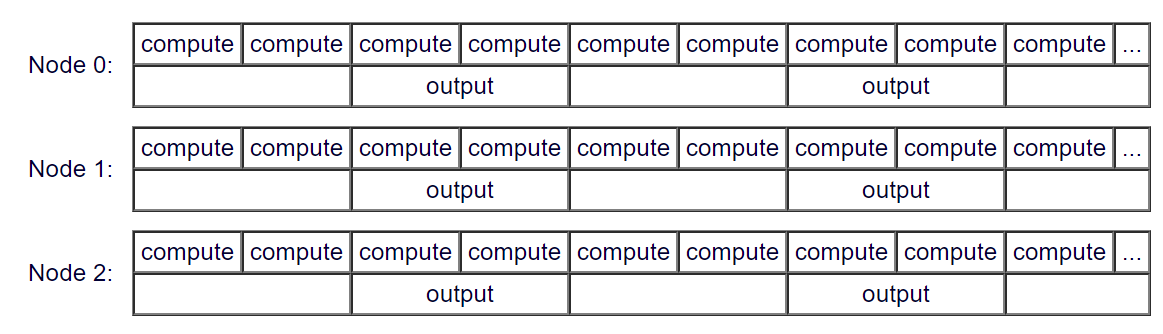
Фигура 5: Таблица с тестови резултати, където N – брой итерации, P – брой паралелни процеси и TP,N – времето за изпълнение на теста

Фигура 6: Графика на тестови резултати представящ ускорение спрямо брой процеси при тестване на различен брой итерации

* Както се вижда, авторът получава суперлинейна аномалия в своите резултати. Това е аномалия, която възниква когато ускорението на някой тест се изчисли да е по-голямо от броя на използваните процеси в теста. Тази аномалия предупреждава, че при последователното изпълнение на програмата е възникнало значително закъснение, което липсва при паралелно изпълнение.



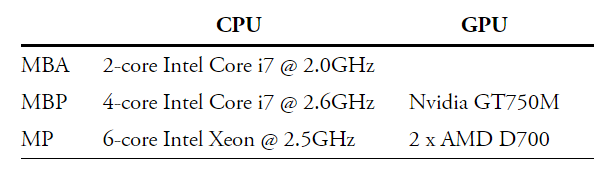
Фигура 7: Визуализиране на работата при последователно изпълнение на програмата

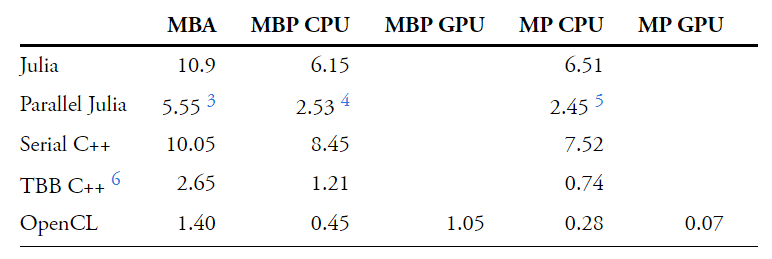


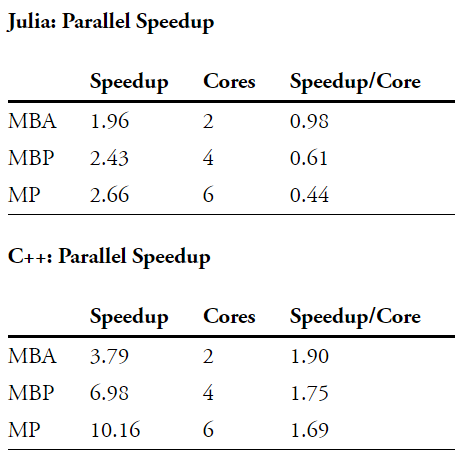
Фигура 8: Визуализиране на работата при паралелно изпълнение на програмата

* Авторът открива и описва къде лежи проблема, който поражда тази аномалия. При изпълнението на програмата с единствен процес, изчисляването и принтирането на редовете става напълно последователно. Когато се изпълнява програмата с повече от един процес, то изчисленията стават паралелно. Уникалното е, че изпращането на съответните редове към главния процес става във фонов режим докато процесите изчисляват следващите редове. Така се достига допълнителен слой паралелизъм, който спестява времето нужно за изпращане на редовете. Докато в последователното изпълнение един процес отговаря за всяко принтиране, което допълнително забавя програмата и води до по-силно проявено ускорение при сравнение с резултатите при паралелно изпълнение.

1. Parallel Mandelbrot in Julia, C++, and OpenCL [[3]](#_Parallel_Mandelbrot_in)

* Тази статия е значително по-практически настроена от предните две разгледани. Голяма част от нея е програмен код и обяснения на идеята, която стои зад този код.
* Julia като програмен език не съдържа експлицитни нишки, но позволява размяна на съобщения между различни процеси работници, чрез което се синхронизира тяхното изпълнение. Използва се споделен масив, в който всичките процеси работници записват своите изчислени задания. Главната програма изчаква всеки процес да приключи изпълнението си, но останалите детайли за паралелния алгоритъм не стават ясни.
* Авторът реализира паралелен алгоритъм за решаване на задачата на Манделброт първоначално чрез езика Julia, след което транслира кода към C++ и OpenCL с цел да се сравнят резултатите в различни езици.
* При паралелизирането на алгоритъма в C++, авторът използва Threading Building Blocks, за да раздели работата на програмата на различни задания спрямо зададена грануларност. TBB е модел за паралелно програмиране, който е базиран на динамична обработка. [[4]](#_Tutorial:_Develop_an) TBB сам създава и планира процесите, които ще работят върху заданията на програмата. Когато един процес работник приключи своето задание, той получава следващото в опашката и продължава своята работа. Така всеки процес не губи време в чакане на останалите, а вместо това постоянно получава нови задания да обработва.
* Чрез TBB авторът реализира динамично разпределяне на работата, защото редът, в който ще бъдат обработени заданията не се определя преди да се стартира програмата.
* Тестовете на програмата са следните:

Фигура 9: Архитектура на тестовите машини, където MBA – MacBook Air 2012, MBP – MacBook Pro 2013, MP – Mac Pro 2013

Фигура 10: Време в секунди за изпълнение на проведените тестове на всеки алгоритъм

Фигура 11: Ускорение при изпълнение на проведените тестове на паралелния алгоритъм на Julia и C++

* На пръв поглед отново се получава суперлинейна аномалия при изпълнението на паралелния C++ алгоритъм, но благодарение на Хипернишкова технология на Intel всяко ядро има възможност да изпълнява до два процеса едновременно. Тази технология е приложима, защото алгоритъмът е реализиран чрез TBB. Така реално хардуерната граница на ускорението ще е два пъти по броят използвани ядра и аномалията се избягва.

# Извод от разгледаните решения на задачата на Манделброт

1. Използвани технологии в решенията

* Първият източник разработва задачата на C++, за да може да приложи библиотека EasyBMP. За паралелизиране на задачата е използвана серия от библиотеки Message Passing Interface, както и Portable Batch System за да се достави програмата към двете тестови машини – Mimosa и Sweetgum. Mimosa е система, базирана на разпределена памет, докато Sweetgum е базирана на споделена памет.
* Вторият източник описва реализацията на алгоритъма под формата на псевдокод и не става ясно на какъв език е написана. Споменато е, че в тази реализация също се използва Message Passing Interface, което подсказва, че програмния език е C, C++ или Fortran. Тестовете са извършени върху суперкомпютърът на университета в Монтана.
* Третия източник реализира алгоритъма на езиците Julia, C++ и фреймуърка OpenCL. C++ имплементацията допълнително използва Threading Building Blocks на Intel. Тестовете са извършени на следните три машини: MacBook Air 2012, MacBook Pro 2013 и Mac Pro 2013.

1. Избор на език

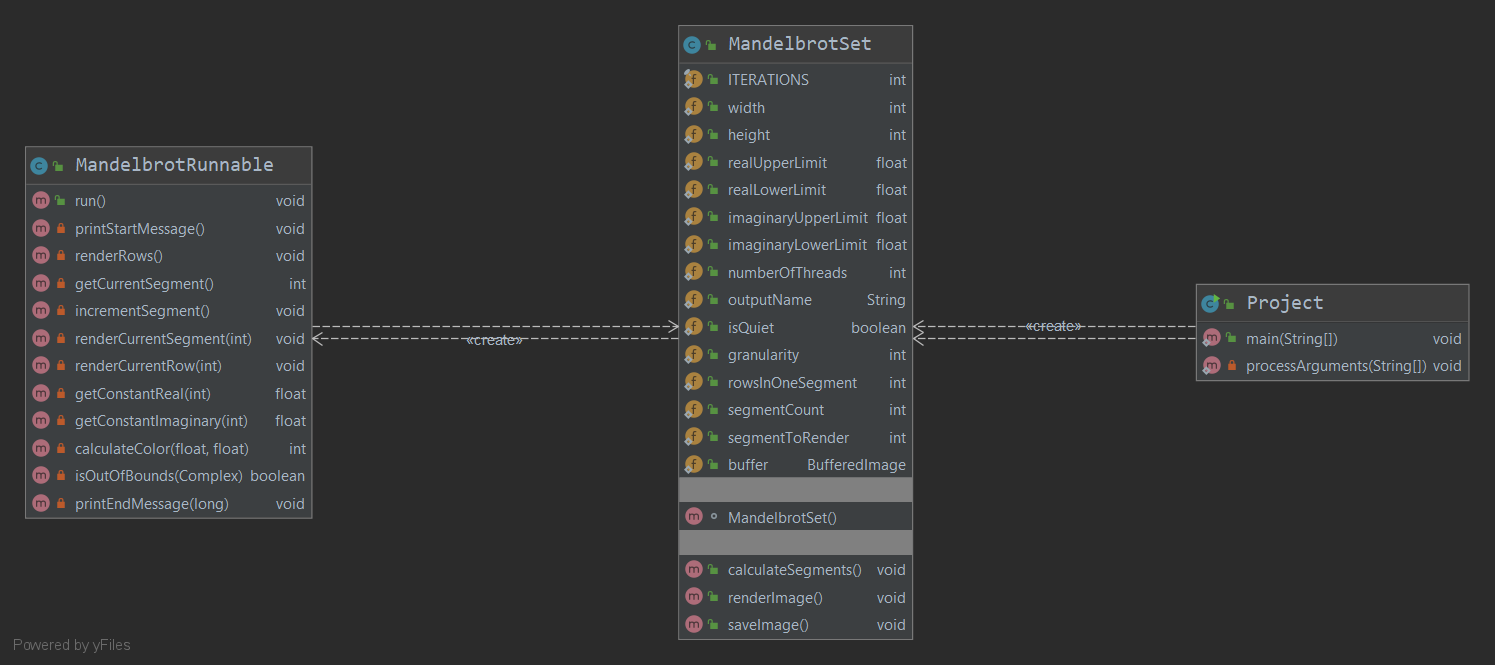
* Най-подходящия език за реализацията на този проект ще бъде обектно-ориентиран език, който позволява програмиране от високо ниво. Така разработената програма ще може лесно да се раздели на компоненти, както и да бъде по-разбираема, защото кодът на високо ниво се абстрахира от детайли, които биха евентуално довели до объркване. Също така езикът трябва да поддържа работа с паралелни процеси.
* C++ е добър избор за език, който покрива горните изисквания. Той е използван в решаване на задачата при разгледаните образци, което подсигурява, че ще е възможно да се създаде удовлетворяваща програма чрез него.
* Поради своя личен опит, като алтернатива на C++ избрах да разработя програмата на Java. Той също покрива горните изисквания и дори е език на по-високо ниво от C++ поради допълнителния слой абстракция, който възниква от виртуалната машина на Java. Това би улеснило процеса на създаване на решението.

1. Избор на софтуерна архитектура

* Задачата, която този проект се стреми да реши е генериране на изображение на фрактал чрез паралелна обработка. В този случай най-подходящо е да се ползва йерархичната архитектура Master – Slaves. Чрез нея програмата ще се раздели един главен компонент, който ще съдържа всички важни данни и един компонент Slave, който ще отговаря за пресмятане на изображението спрямо указанията на компонента Master.
* Ще се приложи модела Single Program Multiple Data, за да се декомпозира цялостното изображение на множество от задания. Процесите от компонента Slave ще изчисляват паралелно тези задания.
* Работата на всеки процес ще бъде балансирана динамично, както е подходено в третия разгледан източник. Така всеки процес ще обработва първото свободно задание и няма да има нужда от статично определяне кой процес за кое задание отговаря. В този случай ще е нужна синхронизация между работата на процесите и реализацията на паралелния алгоритъм ще се затрудни, но за сметка на това ще се получи решение, което е по-елегантно и по-разбираемо от високо ниво.

# Проектиране

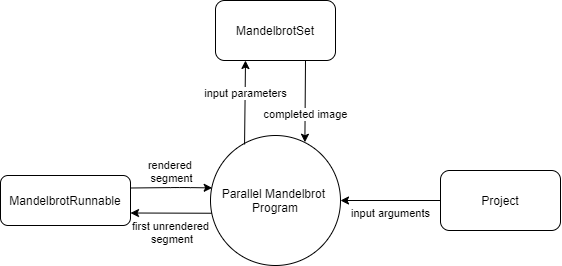
# Диаграми и дизайн на програмата

1. Клас диаграма

Фигура 12: Клас диаграма на програмата

Програмата ще се раздели на три различни компонента, които са изобразени чрез класовете в диаграмата. Накратко описано:

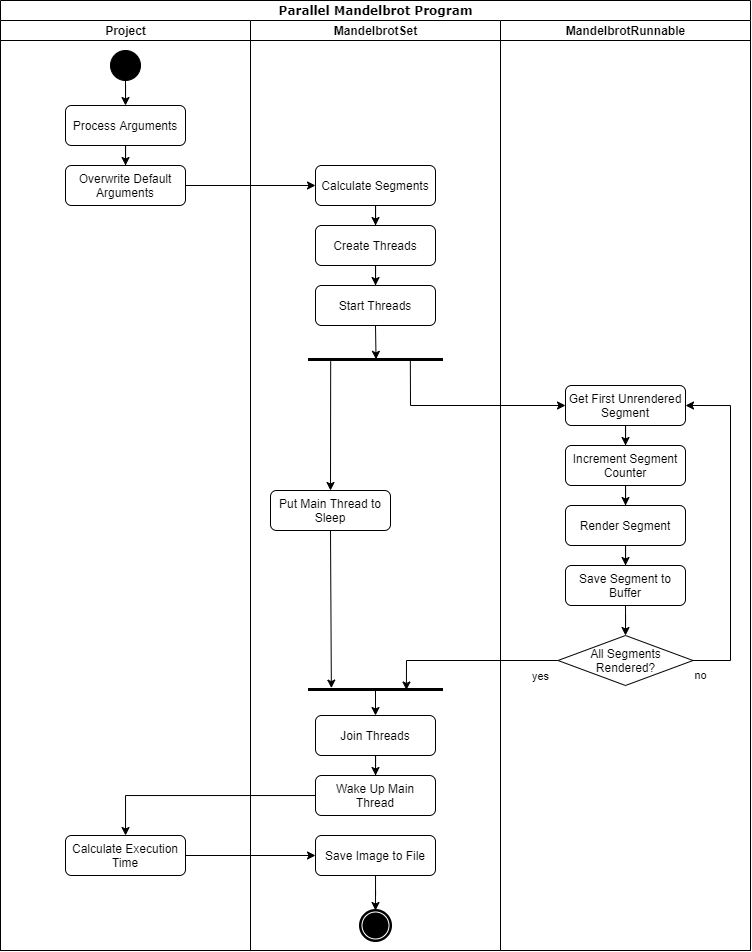
* Компонент Project показва програмата на най-абстрактно ниво. Той отговаря за обработване на входните аргументи при стартирането на програмата извикване на методите на MandelbrotSet, за да изпълни задачата.
* Компонент MandelbrotSet отговаря за запазването на всички нужни параметри на програмата. Той стартира процесите, които паралелно ще изчислят изображението, след което се приспива докато не приключат всички други процеси. Накрая той записва изображението като файл.
* Компонент MandelbrotRunnable изобразява процесите, които паралелно ще изчисляват изображението. Неговото единствено задължение е да пресметне заданията, които получава и да ги записва в общия буфер, който се намира в MandelbrotSet.

1. Контекстна диаграма на поток на данните

Фигура 13: Контекстна диаграма на поток на данните на програмата

Тази диаграма показва как данните ще се движат и променят по време на изпълнението на програмата. Спрямо компонентите:

* Компонент Project обработва входните аргументи при стартирането на програмата, след което ги записва в MandelbrotSet.
* Компонент MandelbrotSet запазва и модифицира своите параметри спрямо работата на другите компоненти. Накрая на изпълнението използва своя буфер, за да запази завършеното изображение като файл.
* Компонент MandelbrotRunnable взима номера на първото необработено задание, а след завършване на изчислението записва заданието в общия буфер на MandelbrotSet.

1. Диаграма на дейностите

Фигура 14: Диаграма на дейностите на програмата

Тази диаграма описва как протича изпълнението на програмата. Нейното обяснение съвпада със следващата подточка 2.2.

# Подробно описание на изпълнението на програмата

1. Стартиране на програмата

Програмата може да бъде стартирана с или без входни аргументи. Възможните аргументи на програмата са следните:

* Размер на изображението
* Сектор от изображението, който да се визуализира
* Брой паралелни процеси на програмата
* Име на генерирания файл
* Дали програмата да е в тих режим
* Коефициент на грануларност

Ако липсват някои входни аргументи или се въведат в невалиден формат, то програмата стартира с предварително зададените от програмиста параметри. Входен аргумент, който е в правилен формат се записва на мястото на съответния параметър по подразбиране.

1. Изчисляване на броя задания и декомпозиция на данните

След като параметрите на програмата са вече финализирани, програмата трябва да декомпозира цялостното изображение на задания. Това зависи от два параметъра: броя паралелни процеси ‚p‘ и коефициента на грануларност ‚g‘. Коефициентът g определя колко задания ще се заделят за всеки процес.

Програмата извършва паралелизъм по изчисление на редове, което означава че всяко задание ще се състои от един или повече редове. Колко реда съдържа всяко задание се определя като височината на изображението се раздели на броя задания.

* Например за p = 5 и g = 8 получаваме, че програмата ще се раздели на p \* g = 40 задания. Ако изображението има височина 2000 пиксела, то броя редове във всяко задание ще бъде 2000 / 40 = 50.

Важно е да се обърне внимание на случая, в който височината не се дели точно на броя задания. В тази неудобна ситуация остатъкът от делението ще е броя на редовете, които не се включват в никое задание и остават необработени. Разгледани са три варианта за решение на този проблем, но всеки от тях има страничен ефект, който може да доведе до разминаване между това, което програмиста очаква и това, което програмата генерира.

* Първото решение е да се увеличи броя на редовете, които заданията обработват. Това води до цялото изображение да се изчислява с по-малък брой задания и така получаване на по-едра грануларност, отколкото е било предназначено.
* Второто решение е да се увеличи броя на заданията, за да се покрият висящите редове. Това води до цялото изображение да се изчислява с по-голям брой задания и така получаване на по-финна грануларност, отколкото е било предназначено.
* Третото решение е да се увеличи броя на редовете на само част от заданията. Това води до изчисляване на цялото изображение със същия брой задания, но се губи свойството всяко задание да има еднакъв размер данни.

И трите решения могат да доведат до други бъдещи проблеми, главно свързани с неочаквани резултати при тестването. Но от по-голямо значение е последните редове да не бъдат оставени висящи, поради което трябва да се имплементира някое от решенията. Тази програма имплементира първото решение поради лесната му реализация. В случай на неточно делене, броят на редовете, обработени при от задание се увеличава с единица. Допълнителният ред при всяко задание е достатъчен, за да се покрият всички висящи редове.

1. Стартиране на паралелната обработка

След като завърши декомпозицията на данните, главния процес създава толкова на брой процеса, колкото е стойността на параметъра ‚p‘. Тези процеси се стартират, след което главния процес заспива докато не приключат всички други процеси. В противен случай главния процес би приключил своето изпълнение преди останалите процеси и като резултат изображението, което се запазва във файл ще бъде частично генерирано.

1. Паралелна обработка на задачата и балансиране на работата

Както бе споменато в извода от анализа, разпределянето на заданията ще става динамично като всеки процес зaема първото свободно задание. Програмата ще съдържа една споделена променлива, която представлява номера на първото свободно задание. Всеки процес ще изпълнява цикъл, в който се извършват следните действия:

1. **Прочитане на споделената променлива и запазване на номера на заданието, което се заема, в локална променлива**. Използването на локална променлива е задължително, защото номерът на заданието е нужно при определяне на кои редове трябва да се пресметнат. Споделената променлива може да се промени по всяко време като най-опасното от тях е когато процес изчислява редовете на своето задание.
2. **Проверка дали номерът на заданието е валиден.** Ако той е по-голям от общия брой задания, то процеса прекратява цикъла и завършва изпълнението си.
3. **Писане в споделената променлива чрез инкрементиране на стойността ѝ.** Така останалите процеси се уведомяват, че предното по номер задание е вече заето.
4. **Изчисляване на редовете от заданието.** Ако редовете, определени от заданието, надвишат височината на изображението, то се смятат редовете до последния допустим.
5. **Записване на изчислените редове в общ буфер.** Поради факта, че буферът е двумерна матрица и всяко задание представлява различен набор от редове, не е възможно да се получи конфликт при записване на редовете в буфера и няма нужда от синхронизация на тази стъпка.
6. **Приключване на итерацията и започване на следващата.** Така цикълът се връща към стъпка I.

При реализацията на динамично балансиране на работата е нужна синхронизация между процесите при четенето и писането в споделената променлива. Приложен е модела Parallel Random Access Machine, по-конкретно схемата Exclusive Read, Exclusive Write. Използваната споделена променлива не може да се чете или променя от повече от един процес.

Ако тази променлива е заета от един процес и други се опитат да я достъпват, то останалите процеси се приспиват докато първия процес не я освободи. Нормално това би било предпоставка за създаване на bottleneck и забавяне на изпълнението на програмата, но в този алгоритъм цялата синхронизация се извършва върху два реда код, съответно четенето и писането. Това осигурява бързото заемане и освобождаване на споделената променлива и избягва процесите да стоят значително време приспани.

1. Приключване на програмата

След приключването на всички процеси, главния процес се събужда. Той изчислява времето нужно за изпълнение на паралелната обработка като от сегашното време в UNIX формат се извади времето, в което е стартирана програмата. След това главния процес използва вече запълнения буфер, за да запази готовото изображение като файл и приключи изпълнението на програмата.

# Тестване

# Тестова архитектура

# Тестови параметри

# Тестване с променлива размер на изображението

# Тестване с променлива сектор на изображението

# Тестване с променлива коефициент на грануларност

# Списък източници

* [1] Isaac K. Gäng, David Dobson, Jean Gourd and Dia Ali,

## Parallel Implementation and Analysis of Mandelbrot Set Construction,

<https://www.academia.edu/1399383/Parallel_Implementation_and_Analysis_of_Mandelbrot_Set_Construction>

* [2] Matthias Book,

## Parallel Fractal Image Generation - A Study of Generating Sequential Data with Parallel Algorithms,

presented May 3, 2001,

<http://matthiasbook.de/papers/parallelfractals/files/parallelfractals-paper.pdf>

* [3] Distrust Simplicity,

## Parallel Mandelbrot in Julia, C++, and OpenCL,

January 9, 2015,

<http://distrustsimplicity.net/articles/mandelbrot-speed-comparison/>

* [4] Intel,

## Tutorial: Develop an Application with Intel® Threading Building Blocks - Overview,

November 9, 2018

<https://software.intel.com/content/www/us/en/develop/documentation/tbb-tutorial/top/overview.html>