Софийски университет „Св. Климент Охридски“
Факултет по математика и информатика

КУРСОВ ПРОЕКТ

по

Разпределени софтуерни архитектури

спец. Софтуерно инженерство, 3 курс, летен семестър,

академична година 2022/2023

MandelWorld: Паралелен тест на Манделброт при статично и динамично балансиране. Грануларност и адаптивност към L1 D-Cache

Изготвил: Стефан Велев

Ф. No.: 62537

Ръководители:
проф. д-р Васил Георгиев
ас. Христо Христов

04.06.2023
гр. София

СЪДЪРЖАНИЕ

Цел на курсовия проект и въведение в задачата

Целта на курсовия проект е да се изследва паралелизма при провеждането на тест на Манделброт. Ще се разгледа как статичното и динамичното балансиране оказват влияние на полученото ускорение и ще бъде направена съпоставка между двете. Ще бъде отговорено на въпроси, касаещи се до грануларността, т.е. как са съпоставени отделните задачи спрямо големината им и разпределението им по броя нишки. Ще бъде изследвана темата с адаптивността към L1 D-Cache, т.е. дали може да се намали толкова размера на отделното задание, така че то да се помести изцяло в големината на L1 D-Cache. Ако това бъде постигнато, ще бъдат разгледани, ако има такива, случаи на суперлинейни аномалии, които ще бъдат видими най-ясно в тестовия план и графиките към съответните тестови случаи.

Последователността на секциите в курсовия проект ще следва процеса по разработка на софтуер в рамките на софтуерното инженерство.

В 1. Анализ ще бъдат проучени и представени четири източника, които са максимално подобни по задача с поставената като цел. Ще бъдат анализирани, с цел да се изведат добрите и не толкова добрите практики, които да се следват и съответно избягват от автора в последващите етапи. Заедно с функционалния анализ, ще присъства и технологичен обзор, който ще изследва използваните технологии, езици, средства, които ще бъдат инструментът, спомагащ за решаването на проблема. Секцията ще завърши със сравнителна таблица между източниците, която цели да обобщи най-важната извлечена информация от обзора и едновременно да представлява сравнение с визията на автора по отношение на задачата.

В 2. Проектиране ще бъдат разгледани същинските въпроси по реализацията на проблема. Функционалностите, които ще притежава готовия продукт, ще бъдат представени чрез т.нар. „ръководство за потребителя“ („user guide“). В него ще бъдат описани изискуемите входни параметри, които може да бъдат подавани при стартиране на програмата, и точният начин те да бъдат представени. Нефункционалните характеристики ще бъдат въведени под формата на „справочно ръководство“ („reference manual“), което ще разгледа въпроси, свързани с конкретно избрания начин на работа и имплементация. Тук ще бъдат разгледани и сравнени двата вида балансиране с цел да се покаже на практика кой вариант е за предпочитане и какви са му силните и слабите страни. За двата случая ще бъде представена UML диаграма на последователността (UML Sequence Diagram), с помощта на която ще се онагледят последователността на действията, които се реализират по време на стартиране на програмата, както и от кои инстанции на представените класове се извършват. По този начин ходът на отделните съобщения ще бъде

очертан ясно. По отношение на използваните средства, ще бъдат описани всички зависимости под формата на библиотеки, които се използват за работа на софтуера.

В 3. Тестване. Настройка. Внедряване ще бъде представен тестов план, който ще се състои от множество тестови случаи, целящи да изследват това как ще се държи програмата по време на изпълнение при подадени различни стойности на входните параметри. Това ще бъде направено, за да може да се пресметне ускорението (speedup) в различните случаи. С помощта на графики ще бъде онагледен резултатът в някои от случаите, който ще послужи за формулирането на изводи по отношение на разглежданите проблеми. В частта за внедряването ще бъдат демонстрирани изображения, резултат от програмата.

Курсовият проект ще завърши със секция за Източници ("References"), които ще бъдат използвани при извършване на предварителния анализ, както и по отношение на самото проектиране.

Множеството на Манделброт се представя под формата на фрактал. По същество фракталът е модел, който се повтаря вечно и всяка част от фрактала, независимо колко е увеличен или намален мащабът, изглежда много подобно на цялото изображение.

Множеството на Манделброт се задава с реда $z_{n+1} = z_n^2 + c$, където z и c са комплексни числа. Изчислението на конкретна точка става именно чрез задаването на нейните координати в комплексната равнина под формата на фиксираната константа c. За всяка нейна стойност получаваме различни редове от горния вид за които има два варианта:

(i) Точката c принадлежи на множеството на Манделброт M:

$$c \in M \iff |z_n| \leq 2 \text{ за } \forall\, n \geq 0$$

Това означава, че редът има крайна граница, която в компютърните системи може да бъде проверена чрез не преминаването на границата до известно фиксирано място, тъй като във физическия свят винаги съществува някакво ограничение на ресурси. В този случай точката c се изобразява в черен пиксел на генерираното изображение.

(ii) Точката c не принадлежи на множеството на Манделброт M:

$$c \notin M \iff \exists\, n \geq 0 : |z_n| > 2$$

Това само за конкретния случа означава, че редът има безкрайна граница, т.е. клони към безкрайност. Математически установено е, че когато от дадено място нататък, стойността на реда надхвърли границата 2, то той ще започне да нараства бързо и няма да бъде сходящ. Броят итерации, необходими да бъдат направени до момента

на достигане и преминаване през числото 2, ни е необходим, за да определим цвета на пиксела за дадената точка в генерираното изображение.

Казахме, че в света на технологиите няма как да представим безкрайността. Тогава възниква въпросът какъв да бъде максималният брой итерации, които да се правят при изчисленията за всяка точка. Истината е, че различният брой итерации ни дава различно ниво на детайлност на множеството на Манделброт. Ясно е, че при зададен по-голям брой на възможните итерации, изчислителната сложност на задачата ще расте. Тъй като обаче алгоритъмът е напълно асинхронен, това няма да е съпроводено с много по-големи изчаквания в края, стига, разбира се, конкретната реализация да е добре балансирана между различните работници. В крайна сметка, всичко се свежда до нашия избор относно това колко детайлно искаме да бъде генерираното изображение в резултат на изпълнение на програмата и колко време искаме да чакаме за това то да бъде получено.

1. Анализ

1.1 „Parallel Implementation and Analysis of Mandelbrot Set Construction“ [1]

1.1.1 Функционален анализ

В този източник задачата е сходна на поставената. Цели се да се проектира последователен и паралелен алгоритъм на теста на Манделброт. Статията започва с описание на проблема, където е въведена основната информация, необходима на читателя да навлезе в задачата.

След кратък коментар на последователното решение, започва разглеждането на паралелния алгоритъм. Коментира се по проблема с балансирането и как стандартното разделение на по-едри редове и колони няма да е ефективно, тъй като черните области от изображението биха отнели много повече време за изчисление от по-светлите. Това е така, защото черният пиксел означава, че в изчислението на реда от множеството на Манделброт са извървяни всички последователни членове до достигане на максимума, който задаваме като входен параметър или по време на имплементация. Комуникацията между отделните процеси (нишки) е сведена до минимум – само преди (за да им бъде дадена задачата и поредния номер) и след (за да предадат свършената работа, която да бъде обединена на едно място накрая) направата на изчисленията.

Резултати са получени от две тестови машини (*Mimosa* и *Sweetgum*), чиито механизъм и технология ще бъдат разгледани по-подробно в секцията за технологичен анализ. От графиката на полученото ускорение забелязваме, че то много се доближава до линейно. На представената фигура се акцентира върху хардуерния паралелизъм на машините, тъй като по абсцисата се вижда, че мерната единица са самият брой процесори (CPU). Забелязваме, че *Sweetgum* се представя по-добре в моментите, когато сме излезли от ситуацията на последователен алгоритъм. За да разберем защо това е така, ще трябва да разгледаме нефункционалните аспекти, стоящи зад двете тестови машини.

1.1.2 Технологичен анализ

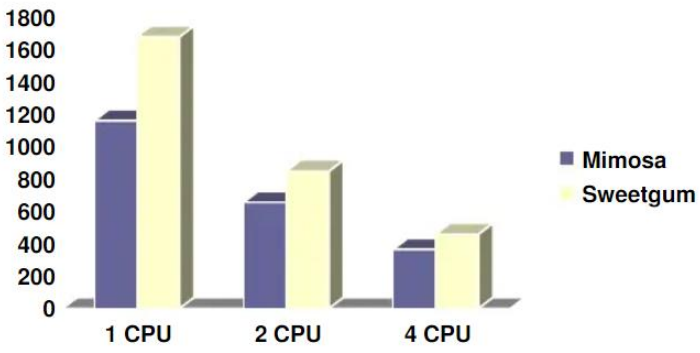
В цитирания източник представения механизъм за комуникация, който се използва, е MPI общодостъпно предаване. Това е постигнато с помощта на езика C++ и интерфейсът за обмен на съобщения (MPI). С помощта на пакетна обработка чрез специална система, задачите се предават на две паралелни машини – *Mimosa* и *Sweetgum*. Характерно за тях е, че *Mimosa* представлява разпределена система – мултикомпютър (distributed memory system), докато *Sweetgum* е система с обща памет – мултипроцесор (shared memory system). В случая и двете машини работят под Linux среда.

Казахме, че използваният механизъм за комуникация е чрез обмен на съобщения. В такъв случай характерния модел на *Mimosa* според разширената таксономия на Flynn-Johnson би бил DMMP (distributed memory, message passing). Това е чистият вариант на мултикомпютърна архитектура. Предимство на механизма за обмяна на съобщения е, че нямаме странични ефекти като състезателен достъп (race condition) или взаимна блокировка (deadlock), тъй като получаващият съобщението процес не може да

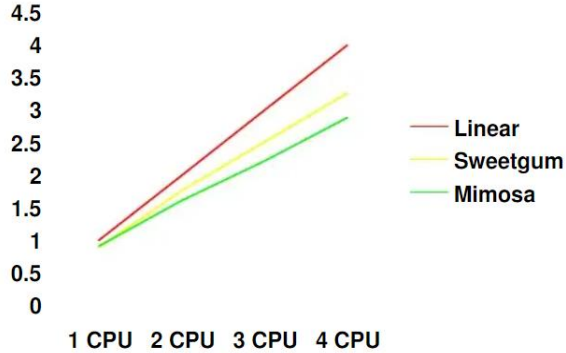
извърши четене преди да бъде изпратено и получено то. По този начин самата размяна на съобщения играе ролята на ключалка, от която иначе бихме имали нужда, ако използвахме механизма на общите променливи.

Споменахме, че *Sweetgum* е система с обща памет. Тъй като механизма за размяна на съобщения остава, вероятният случай, в който попадаме тук според разширената таксономия на Flynn-Johnson е GMMP (general memory, message passing). В този случай имаме обща памет на една машина, но въпреки това използваме механизма за обмяна на съобщения. Това се прави, защото казахме, че последният не води до споменатите странични ефекти. Този механизъм е характерен за функционални езици като Erlang и др.

След като изяснихме въпросите по архитектурата на машините, можем да направим изводи по отношение на получените резултати за ускорението. В стълбчестата диаграма за отминалото време за изчисление на алгоритъма забелязваме, че когато той е паралелен, разликата между времената на двете машини е осезаема. *Mimosa* се представя много по-добре от *Sweetgum*. С увеличаването на хардуерния паралелизъм обаче, може да видим как тази разлика намалява. От графиката за ускорението, в крайна сметка, може да видим, че *Sweetgum* се представя като цяло по-добре от *Mimosa*, т.е. системата с обща памет при обмен на съобщения се представя по-добре от другата система – тип мултикомпютър. Това е така, защото в този случай се приближаваме повече до чисто паралелния подход SPMD (single program, multiple data), докато при втория подход все пак оставаме повече зависими от допълнителна външна среда middleware (междинна платформа). Все пак остава под въпрос коректността на графиките, защото след направата на някои изчисления от показаните времена на първата графика и представените ускорения на втората графика, не се открива точната зависимост. Въпреки това, и от графиката на времената може да се заключи чрез груби сметки на ускорението, че *Sweetgum* е системата, която води до по-голямо ускорение.



Фигура 1: *Стълбчеста диаграма на изминалото време спрямо броя процесори (10000x10000)*



Фигура 2: *Графика на ускорението спрямо броя процесори (10000x10000)*

1.2 „Parallel generation of a Mandelbrot Set“ [2]

1.2.1 Функционален анализ

В този източник задачата е да се генерира множеството на Манделброт чрез паралелен алгоритъм. Изчислението е напълно асинхронно, защото за пресмятането на дадена задача, не е необходима информация за останалите. Статията започва с кратко въведение, което разяснява изследвания проблем. Споменава се целта на статията и се предлагат за разглеждане две различни паралелни схеми, които водят до различни по производителност резултати.

Във въведението авторите ни запознават в още повече детайли с проблема. След това се споменава механизмът, който се използва за паралелната комуникация, а именно MPI (интерфейс за обмен на съобщения). За неговата реализация ни е нужна мрежа от компютри, които с помощта на middleware (междинна платформа) да осъществяват този обмен. Дискутирани са основните команди за размяната на съобщения, сред които са стандартните MPI_Send и MPI_Recv, които са блокиращи и следователно спират нормалния поток на алгоритъма, за да изчакат завършване на операцията. Това е причината механизмът за обмяна на съобщения да няма допълнителни странични ефекти, за разлика от този на общите променливи в мултипроцесорите.

Статията продължава с въвеждане на последователния алгоритъм за изчисление на теста на Манделброт. Естествено, това означава, че започвайки от някой ъгъл последователно ще се изчисляват отделните области (задачи), т.е. една след друга. Целта обаче е да се разгледат паралелни решения на този проблем. Именно затова са предложени два варианта на решение – статично балансиране (Static Load Balancing) и динамично балансиране (Dynamic Load Balancing). При статичното балансира всяка нишка (в случая процес) има предварително определени области от множеството, които трябва да изчисли. При стартиране на програмата, те са ясно определени, в зависимост от подадените параметри, и това става с помощта на задаване на индекси, чрез които всяка нишка знае кои задачи трябва да изпълни и коя нова да поеме след като приключи с предишната. При модела на динамичното балансиране няма ясно зададени начални правила, а се решава по време на изпълнение на програмата коя нишка с коя от задачите да продължи. За целта е нужно да се използва задължително или ключалка, която да е общия механизъм за комуникация, или отделен за това процес, който да има задачата да ръководи изпълнението на останалите като им поставя нови задачи след като са готови. Последното се отклонява до известна степен от чистата паралелна програма (SPMD), защото за оркестриращия процес трябва да бъде направен нов код, което не е характерно за паралелизма по данни като цяло. Очаква се този начин на работа да е по-бавен, тъй като това време за решаване и поставяне на нови задачи на нишките отнема време и увеличава общото време за решение.

Забелязваме, че доста често при модела на статично циклично балансиране грануларността е едра, т.е. на всеки един процес се дават максимално големи области от теста, без да се разцепват на още допълнителни части, които да се редуват по процесите. Това е проблем, тъй като едрата грануларност води до проблеми с балансирането при силно небалансирани проблеми, какъвто е и множеството на Манделброт.

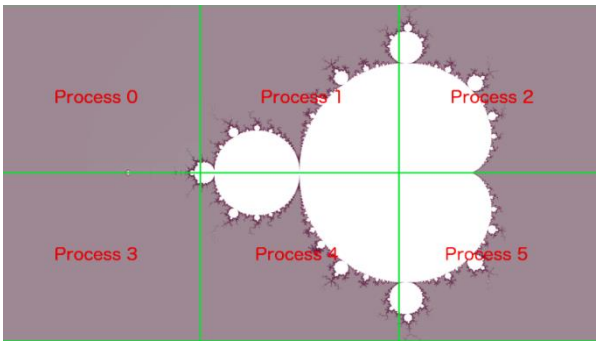
В анализа на получените резултати можем да направим извод, че при тестове с размер на матрицата 12800x12800 и брой итерации 1000, ускорението при статично циклично балансиране не надминава 4. При динамично централизирано балансиране обаче, при стартирани 16 на брой процеси, ускорението е около 14. Тук ясно се вижда проблема, че при максимално едра грануларност, статичното циклично балансиране няма как да даде по-добри резултати. Това може да се подобри и дори да надмине динамичното, ако задачите бяха с по-малък размер и по-разнообразни, като това може да се постигне с увеличаване на коефициента на грануларност.

Правят се и тестове с по-малка матрица – 1600x1600. При тях отново динамичното централизирано балансиране води до по-добри резултати. Тук обаче не е съобразен един от основните проблеми при проектирането на паралелни алгоритми, а именно по-ниската изчислителна сложност. Когато задачата става прекалено бърза за изчисление от процесите, няма как статичното циклично балансиране да даде добри резултати. Решението е да се увеличи сложността на задачата, като се увеличи областта, резолюцията, максималният брой итерации и др.

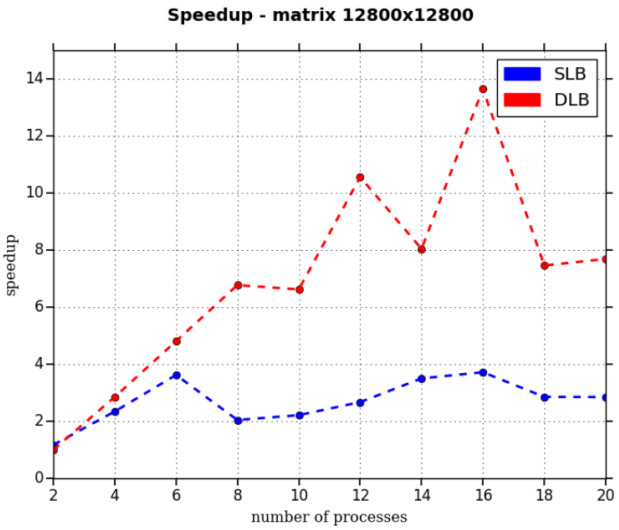
1.2.2 Технологичен анализ

В цитирания източник отново представеният механизъм за комуникация, който се използва, е MPI общодостъпно предаване. Приложените архитектурни решения са SPMD за статичното циклично балансиране и моделът Master/Slave, който се доближава до концепцията за MPMD и се използва в случая за динамичното централизирано балансиране.

Изчислителните платформи за пресмятането на задачите са реализирани на две различни инфраструктури: платформа A (клъстер от 11 Intel dualcore 32 bit процесора с 2 GB RAM) и платформа B (клъстер от 13 Intel quadcore 64 bit процесора с 2 GB RAM). Още на този етап, идеята за клъстер може да ни говори за механизма за обмяна на съобщения, тъй като клъстерите стоят в основата на изграждането на различните видове мултикомпютри.



Фигура 3: *Използвана максимално едра грануларност при реализация на статично циклично балансиране*



Фигура 4: *Графика на ускорението при статично и динамично балансиране спрямо броя процесори (12800x12800; 1000 итерации)*

1.3 „Efficient Generation of Mandelbrot Set using Message Passing Interface“ [3]

1.3.1 Функционален анализ

В този източник е разгледан отново проблемът с паралелната имплементация на множеството на Манделброт. Статията започва с кратко въведение в проблема. Следващата секция продължава с предварителен анализ, в който авторите проверяват първо най-важното – дали алгоритъмът може да бъде имплементиран паралелно, както и какво се очаква да бъде теоретично полученото ускорение. Всичко това се прави, с цел да се провери смисълът на поставения проблем, както и да се анализират възможните ползи като резултат от задачата.

Дали алгоритъмът може да се декомпозира по данни може да бъде проверено чрез използването на условията на Бернщайн (Bernstein’s conditions). Те гласят, че две задачи могат да бъдат паралелизирани, ако са изпълнени три условия, свързани с техните входове и изходи. Това са именно, че входът на първата задача не трябва да има никаква зависимост с изхода на втората задача, входът на втората задача не трябва да има никаква зависимост с изхода на първата задача, както и че двата изхода на двете задачи не трябва да бъдат еднакви. В случая на теста на Манделброт отделните един до друг пиксели от изображението нямат пряка зависимост един от друг и може да се стигне до заключението, както са го направили и авторите на статията, че генерирането на множеството на Манделброт удовлетворява условията на Бернщайн и следователно може да бъде разработен паралелно.

Втората стъпка е изчислението на теоретичното ускорение. За целта се използва законът на Амдал (Amdahl’s Law), който ни дава формула, по която можем да изчислим теоретичното ускорение S_p. То зависи обратно пропорционално на сбора от времето, прекарано в последователната част от кода, и времето, прекарано в паралелната част, разделено на броя задачи. В резултат авторите са обобщили получените резултати в таблица, от която се вижда, че има смисъл от паралелния тест на Манделброт.

В статията се представени три реализации на теста на Манделброт. Те са разгледани в детайли без да се прави явна разлика между статично и динамично балансиране. В статията е споменато, че и всеки един от трите подхода ще изпраща данни обратно до основната нишка и че следователно се използва архитектурата Master/Slave. Една от причините затова е операцията по писане във файл, която е непредсказуема и не е добър подход да се прави от всяка една нишка, тъй като това може да доведе до проблеми с конкурентността. Въпреки това, в различните подходи можем да забележим елементи, които ни доближават от една страна до оприличаване със статично циклично балансиране, а от друга – до динамично балансиране.

Първият подход (Naive: Row Based Partition Scheme) се доближава до статичното циклично балансиране. Изображението се разделя на толкова равни части (редове), колкото са на брой нишките. Това означава, че тук работим с възможно най-едрата грануларност – g = 1. Разбира се, при такъв тип проблеми това най-често резултира до проблеми с балансирането, защото проблеми като разглеждания се характеризират с небалансираността си. Главната нишка стартира всички останали нишки и продължава работа само по своята част, докато за останалите има предварително зададени задачи,

които да извършат. Когато те са готови, изпращат масив със своите получени данни, който обаче се копира от главната нишка, защото финалният голям масив с всички изчислени задачи бива използван за генериране на изображението на Манделброт.

Втората реализация (First Come First Served: Row Based Partition Scheme) може да бъде обвързана с подхода на динамичното централизирано балансиране. Това е така, защото главната нишка (master) има за задача да следи за свободни от работа нишки и да им изпраща нови задачи. Разликата вече тук е, че нямаме предсказуемост преди стартиране на програмата коя нишка, кога ще е готова, колко задачи ще извърши и т.н. Естествено, това решение разчита много на процесите по комуникация и обмяна на съобщения. Характерно за него е, че главната нишка не изчислява никакви редове сама по себе си, а се използва само за делегиране на задачи между нишките. В това решение на проблема си личи ясно изразеното забавяне, което ще бъде забелязано именно поради тази постоянна размяна на съобщения и комуникация, особено ако броя на нишките нарастне значително или пък всички приключват по едно и също време.

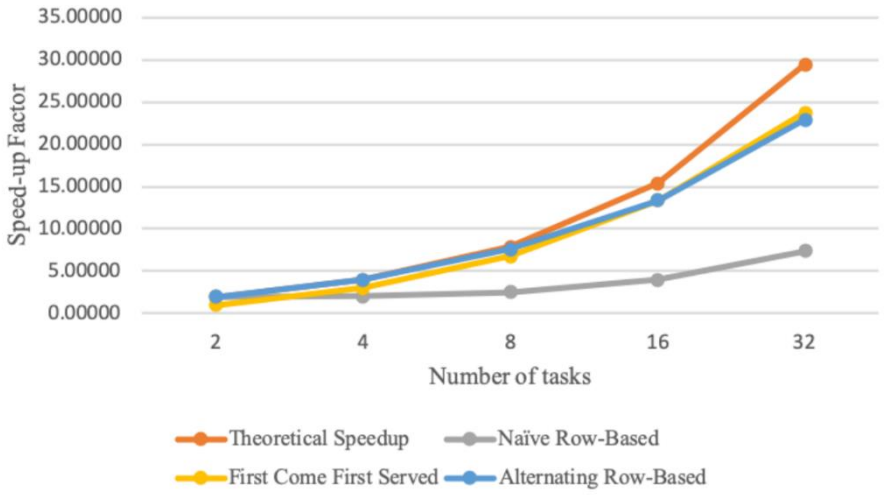
Третата реализация (Alternating: Row Based Partition Scheme) е много подобна на първата, т.е. имаме повече сходство със статичното циклично балансиране, но за разлика от първия подход, тук грануларността е по-средна. Това се прави с цел по-лесното балансиране и приключване на работата по едно и също време. В това обаче се забелязва проблем, тъй като се използва механизмът за обмяна на съобщения, а не общи променливи. Този проблем не е видим, когато задачите са сравнително небалансирани, но в този подход се забелязва, когато главната нишка получава по едно и също време готовите резултати от нишките. Това е тясното място на този подход, което се получава от комбинирането на статично циклично балансиране с подход за обмяна на съобщения с Master/Slave архитектура.

От получените от авторите резултати могат да бъдат направени някои заключения. От графиката се вижда, че подходът, който дава най-малкото по стойност ускорение е първият, т.е. при статично циклично балансиране с максимално едра грануларност. Това е очаквано, тъй като тестът на Манделброт е силно небалансиран проблем. Подходът с динамично централизирано балансиране дава добри резултати, но там процесите по размяна на съобщения отнемат изключително много време, когато говорим за паралелен алгоритъм. Най-добрата представена възможност е третата, която е статично циклично балансиране със средна грануларност. Проблемите, за които казахме, че тя има, свързани с едновременната готовност за предаване на резултати към главната нишка, могат да бъдат решени чрез използването на механизма на общите променливи, тъй като тогава споделеният ресурс ще се намира на едно място на една машина и няма да има нужда от допълнителен процес, който да има специално грижата да координира, приема и обработва готовите резултати.

1.3.2 Технологичен анализ

За тестването на всички подходи в статията се използва интерфейсът за обмяна на съобщения (MPI) на програмния език C. За всеки един от тестовете подадения размер на изображенията е 8000x8000 с 2000 итерации. Всички тестове са проведени на *MonARCH (Monash Advanced Research Computing Hybrid) HPC/HTC* клъстер, което до голяма степен обяснява и използвания механизъм за обмяна на съобщения. Всички тестове са проведени само на един вид процесори – Intel Xeon Gold 61150, за да се

избегне проблемът с липсата на наличност на някои видове ядра в даден момент от време. Всеки от процесорите разполага с 36 логически ядра, но тъй като хипертредингът (hyperthreading) бива изключен, се разчита само на 18-те физически ядра. Наличната използвана оперативна памет RAM е 32 GB за всички тестове.



Фигура 5: *Графика на полученото ускорение при трите използвани подхода (8000x8000; 2000 итерации)*

1.4 „MPI vs OpenMP: A case study on parallel generation of Mandelbrot set“ [4]

1.4.1 Функционален анализ

В този източник се прави сравнение между две технологии, зад които стоят всъщност механизмите за обмяна на съобщения (Message Passing Interface) и на общите променливи (Open Multi-Processing). Разликата между тях, за която авторът споменава във въведението е, че при използването на общи променливи задачите са реализирани чрез използването на нишки в рамките на една операционна система и процес, докато при предаването на съобщения задачите са реализирани чрез използването на различни процеси на операционната система. Подчертава се, че проблемът за теста на Манделброт е избран, защото има изцяло паралелно решение, в което липсва нуждата от синхронизация, с изключение на частта накрая, в която се събират частичните изчисления от всяка нишка, за да може да се конструира финално решение.

Статията продължава със секции, в които са разгледани в детайли и като код, последователният и паралелният алгоритъм за решаването на проблема. Очаквано, при механизма за обмяна на съобщения, се разчита на модела „Master/Slave“, който се използва при динамично централизирано балансиране, тъй като тук имаме постоянно изпращане и получаване на съобщения директно между процесите в някаква среда. Съществува процес (master), който отговаря за разпределението на задачите между останалите процеси (slaves) и ги групира накрая в общ масив с изчисления. В допълнение се казва, че този процес (в случая цял процесор) ще извършва и част от изчисленията. Както и авторът очаква от думите му в началото на статията, този подход няма да дава добри резултати, изразяващи се във високо ускорение (speedup).

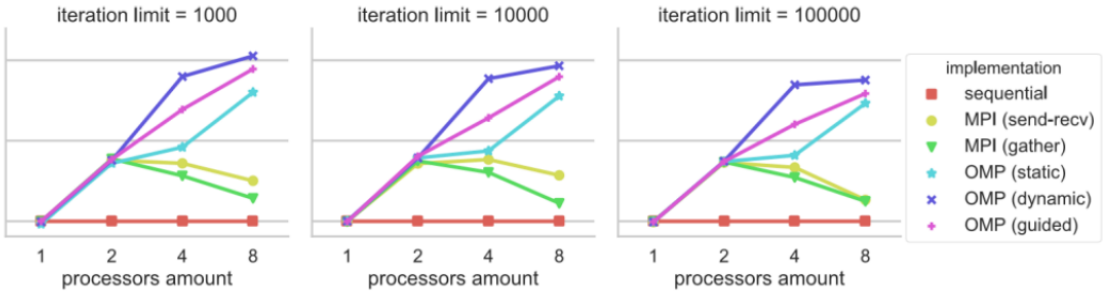
Друга отличителна черта на статията е, че авторът използва в много от случите функции на ниско ниво, вместо да използва готовите такива. Това се прави с цел постигане на бързодействие, защото често готовите решения разчитат на зависимости от други процеси, което внася допълнително ниво на забавяне в алгоритъм, който се стреми да демонстрира висока скорост. Пример за това са функциите за измерване на времето на изпълнение на програмата, които авторът сам си имплементира, вместо да използва готовите от OpenMP и MPI. Друго негово наблюдение е, че функциите от ниско ниво MPI_Send и MPI_Recv постигат по-добри резултати, отколкото варианта с функцията от по-високо ниво MPI_gather.

Очаквано, по отношение на получените резултати, се достига до извода, че механизмът на общите променливи, който е форма на паралелния подход SPMD, дава по-добро ускорение и по-малко време за изпълнение. Това се вижда и на редицата графики, които авторът ни е предоставил. При механизма за обмяна на съобщения (MPI) се забелязва, че с увеличаването на броя процесори, ускорението спада. Това е така, защото във всички случаи, една и съща задача се разпада на все по-малки задачи за изчисление от процесите, които са в комуникация след всяка една, изпълнена от тях. Тъй като обаче процесите се увеличават, а задачите стават все повече и по-малки, достига се до там, че времето за изчисление на задачата става толкова, колкото е времето за изпращане на готовата задача до главния процес (master). Целият този страничен процес по комуникация няма как да не окаже влияние в получените резултати.

1.4.2 Технологичен анализ

И в този източник се използва езикът за програмиране C++ и директивите на OpenMP и MPI. Архитектурата е централизирана с обща памет. Реализирани са много различни опити с максимален брой итерации 100, 1000, 10000 и 100000 и с брой процесори 1, 2, 4 и 8. Тези променливи се подават чрез глобални променливи на средата. Всяка комбинация е повторена три пъти, като се взима средноаритметичното, което считам, че не е винаги коректно, защото върху скоростта на някои от опитите може да оказва влияние фоновото натоварване от други процеси в компютърната система. Приложението решение в рамките на текущия проект е да се използва минимумът от времената на трите теста.

Използваната резолюция и размер на множеството на Манделброт за изчисление е 1024x1024. Машината, която се използва за изчисление, е компютърен модел *HP Notebook – 15-db0069wm* с процесор AMD Ryzen™ 5 2500U Quad-Core и RAM 8 GB DDR4-2400 SDRAM. В заключението, авторът споменава, че получените резултати са част от направено учебно проучване и че те могат да послужат за бъдещи големи проучвания и анализи на проблема и получените резултати.



Фигура 6: *Графики на полученото ускорение при различен максимален брой итерации спрямо броя процеси (1024x1024)*

1.5 Сравнителна таблица

Образец	Макс. паралелизъм	Тип балансиране	Модел на декомпозиция	Поделяне на матрицата	Грануларност	Макс. брой итерации	Размер на матрицата	Начин на обмен
[1]	4 (хардуерен)	статично циклично	декомпозиция по данни	по редове, по колони, рандомизирано	едра, средна	1000	10000 x 10000	обмен на съобщения (MPI)
[2]	20	статично циклично, динамично централизирано	декомпозиция по данни, декомпозиция по управление	по редове, по колони, по блокове	максимално едра, средна	10000	100 x 100, 1600x1600, 1920x1080, 12800x12800	обмен на съобщения (MPI)
[3]	18 (36 логически с хипертрединг)	статично циклично, динамично централизирано	декомпозиция по данни, декомпозиция по управление	по редове, по колони, по блокове	едра, средна	2000	8000 X 8000	обмен на съобщения (MPI)
[4]	4 (8 логически с хипертрединг)	статично циклично, динамично централизирано	декомпозиция по данни, декомпозиция по управление	по редове	едра	100, 1000, 10000, 100000	1024 X 1024	обмен на съобщения (MPI), общи променливи (OpenMP)
MandelWorld	16 (32 логически с хипертрединг)	статично циклично, динамично централизирано	декомпозиция по данни, декомпозиция по управление	по редове, по колони	едра, средна	1024	3840 X 2160 (4K)	общи променливи (shared variables)

2. Проектиране

В рамките на проекта са реализирани два вида балансиране: статично циклично и динамично централизирано, с оглед на това да се направи сравнение между двете. При варианта със статично циклично балансиране е осъществена декомпозиция на матрицата по редове и по колони, за да се провери влиянието на тежестта на задачите в двата случая. При варианта с динамично централизирано балансиране са реализирани две имплементации – една без използването на библиотеката „ExecutorService“ и една с нея. По този начин може да се оцени свръхтовара, който и от анализа в предишната точка, се предполага, че ще се генерира.

2.1 Реализация със статично циклично балансиране

2.1.1 Функционално проектиране

2.1.1.1 Модел на програмата

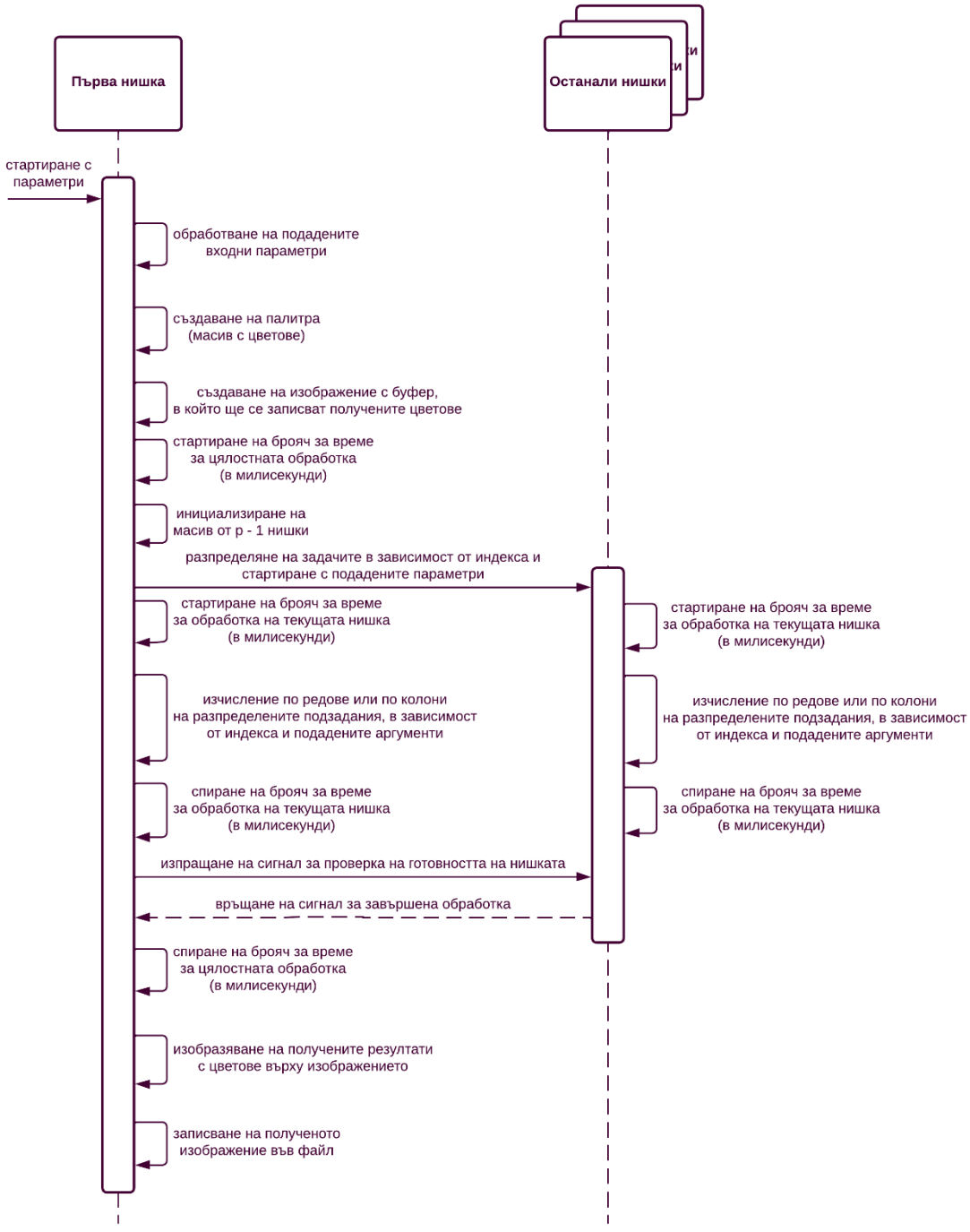
Програмата ще реализира декомпозиция по данни. Това е типичният случай за една паралелна програма. От анализа на източниците видяхме, че проблемът с теста на Манделброт може да бъде подложен на паралелна обработка, тъй като отделните нишки не биха си пречили една на друга. Отделните задачи ще бъдат сформирани от разделянето на общ масив на няколко парчета, в зависимост от използваната грануларност.

При статично циклично балансиране всички нишки имат много подобна (почти еднаква) функция. Главната нишка стартира останалите и започва работа по своята част, докато останалите нишки също от самото начало знаят кои области от проблема

са предназначени за тях и отново започват работа. Причината да се следва този модел е, че множеството задачи са аналогични една на друга и няма нужда от отделен процес, който да прави синхронизация и да раздава задачи. Предварителното разпределение на задачи се осъществява на база индекса на нишката още в самото начало.

Характерно за този случай е, че главната нишка прави изчисления също като останалите нишки. Именно затова, моделът няма как да бъде отъждествен с Master/Slave модела, при който задачата на основната нишка е изцяло различна.

2.1.1.2 Диаграма на последователностите



Фигура 7: *Диаграма на последователностите за реализация със статично циклично балансиране*

Диаграмата на последователностите (UML Sequence Diagram) показва последователността на хода на програмата при статично циклично балансиране. В началото, програмата започва с обработка на входните параметри според предварително дефинирания формат. След това се създава палитра, която според броя извършени итерации, ще определя цвета на съответния пиксел. Прави се отмерване на текущо време, с което ще се оцени изминалото време за работа на цялата програма. След инициализирането на масив с p – 1 нишки се преминава към разпределяне на задачите на всяка нишка и тяхното стартиране. Характерно за статичното циклично балансиране е, че след това и първата нишка започва със своите изчисления, т.е. тя има сходни задачи с останалите нишки. Всяка нишка отмерва времето, което ѝ е нужно за изчисление на задачите си и изпраща информация към основната нишка. Това на практика може да се случи с обща променлива в паметта. Накрая се измерва цялостното време, необходимо за извършване на изчисленията. Полученото изображение като резултат се записва в отделен външен файл. Характерно за горната диаграма е, че нишките от дясната страна са p – 1 на брой, но поради ограничение на мястото и прегледност, е изобразена само една на брой.

2.1.1.3 Описание на командните параметри

Параметър	Описание	Стойност по подразбиране
-s или **-size**	Задава се **размерът на изображението**, което ще се генерира, като широчина и височина в брой пиксели	3840 x 2160 (4K)
-r или **-rect**	Задават се **границите на областта** от комплексната равнина, в която желаем да визуализираме множеството на Манделброт. Задава се чрез подаване на четири числа, разделени със знака „:“, които представляват съответно x-координати и y-координати на избраната област	-0.6386:-0.5986: 0.4486:0.4686 $z = a + i * b$ $a, b \in \mathbb{R}$ $a \in [-0.6386;\ -0.5986]$ $b \in [0.4486;\ 0.4686]$
-o или **-out**	Задаване **име на изображението**, което се получава в резултат на изпълнение на програмата	MandelWorld.png
-t или **-threads**	Задават се **броя нишки**, които ще работят паралелно	1
-g или **-gran**	Задава се **грануларността** (броя подзадания, които всяка нишка трябва да изпълни)	1 (възможно най-едрата)
-c или **-cols**	Задава се дали декомпозицията да е **по колони**	false
-q или **-quiet**	Задава се дали да не се визуализира **информация** за времената на отделните нишки	false
-h или **-help**	Задава се дали да се визуализира **информация** за това какви **параметри** поддържа програмата и начин на използване	false

Когато програмата не е в тих режим, за всяка нишка се визуализират следните съобщения:

Thread_01 released.

Thread_01 ready. Execution time was … ms.

Общото време за работа на програмата се визуализира винаги и е в следния формат:

Total execution time is … ms.

Програмата се стартира чрез shell скрипт по следния начин:

./executeMandelWorld <list-of-arguments>

<list-of-arguments> представлява множеството на входните параметри, които се подават при стартиране на програмата. Ако някои от тях липсват, тогава програмата се стартира с аргументите по подразбиране, описани в горната таблица.

Примерно стартиране на програмата е:

./executeMandelWorld -t10 -s2160x2160 -c

При горното стартиране на програмата, входните параметри са три на брой. Първият от тях е свързан с това, че програмата трябва да използва 10 на брой нишки. Вторият аргумент указва размера на изображението, което ще се генерира, в случая с размери 2160x2160px. Последният параметър говори за това, че декомпозицията, която ще се изпълни, ще бъде по колони.

2.1.2 Технологично проектиране

2.1.2.1 Използван език и външни библиотеки

За реализирането на приложението е използван езикът за програмиране *Java*. Начинът на стартиране на програмите е през терминал чрез *bash* скрипт.

За изпълнението на програмата са използвани две външни библиотеки под формата на jar файлове. Едната библиотека улеснява изчисленията с комплексни числа, тъй като видяхме, че множеството на Манделброт се дефинира за комплексната равнина. Това е библиотеката: *commons-math3-3.6.1.jar* (The Apache Commons Mathematics Library).

За обработката на аргументите, които се подават при вход на програмата, е използвана библиотеката *commons-cli-1.4.jar* (Apache Commons CLI).

2.1.2.2 Тестови среди

Изследванията са проведени на следните две машинни архитектури:

RMI тестова машина

Architecture: x86_64
CPU op-modes(s): 32-bit, 64-bit
Byte Order: Little Endian
CPU(s): 32 (16 физически)

Socket(s): 2
Core(s) per socket: 8
NUMA node(s): 2
Model name: Intel(R) Xeon(R) CP E5-2660 0 @ 2.20GHz
L1d cache: 32KB
L1i cache: 32KB
L2 cache: 256KB
L3 cache: 20480KB

Личен лаптоп

Processor: AMD Ryzen 7 3700U with Radeon Vega Mobile Gfx 2.30 GHz
CPU(s): 4 (8 логически с хипертрединг)
CPU op-modes(s): 32-bit, 64-bit
Base Clock: 2.3GHz
L1 cache: 384KB
L2 cache: 2MB
L3 cache: 4MB
Installed RAM: 16 GB

2.1.2.3 Особености на програмата и работа с нишките

Създаването на нишки става последователно чрез задаването на идентификационен номер (id) на нишка и поредица от параметри, които определят работата на нишката:

```
workers = new Thread[threads];

for (int j = 1; j < threads; j++) {

WorkerStatic r = new WorkerStatic(j, quiet,maxIterations,
                width,height,taskPixWidth, taskPixHeight, rows, cols,
                tasks, byCols, threads);

Thread t = new Thread(r);

t.start();

workers[j] = t;

}
```

След разпределянето и стартирането на отделните нишки, главната нишка също преминава в режим на изчисление, както и останалите нишки. Това става с извикването директно на метода run(), който е подобен на start() с тази разлика, че вече не се извиква нова нишка, а вече текущият ход на програмата продължава директно в нишката:

```
new WorkerStatic(0, quiet, maxIterations, width, height, taskPixWidth,

                taskPixHeight, rows, cols, tasks, byCols, threads).run();
```

След като главната нишка приключи със своите задачи, тя изчаква последователно всяка една от останалите нишки да приключи. Ако дадена нишка не е готова, главната нишка се приспива, докато не получи сигнал от нишката, че вече е приключила. След това се преминава към следващата нишка и така докрая. Това става чрез метода join():

```
                for (int j = 1; j < threads; j++) {
                        try {
                            workers[j].join();
                        } catch (Exception e) {
                            e.printStackTrace();
                        }
                }
```

2.2 Реализация с динамично централизирано балансиране

2.2.1 Функционално проектиране

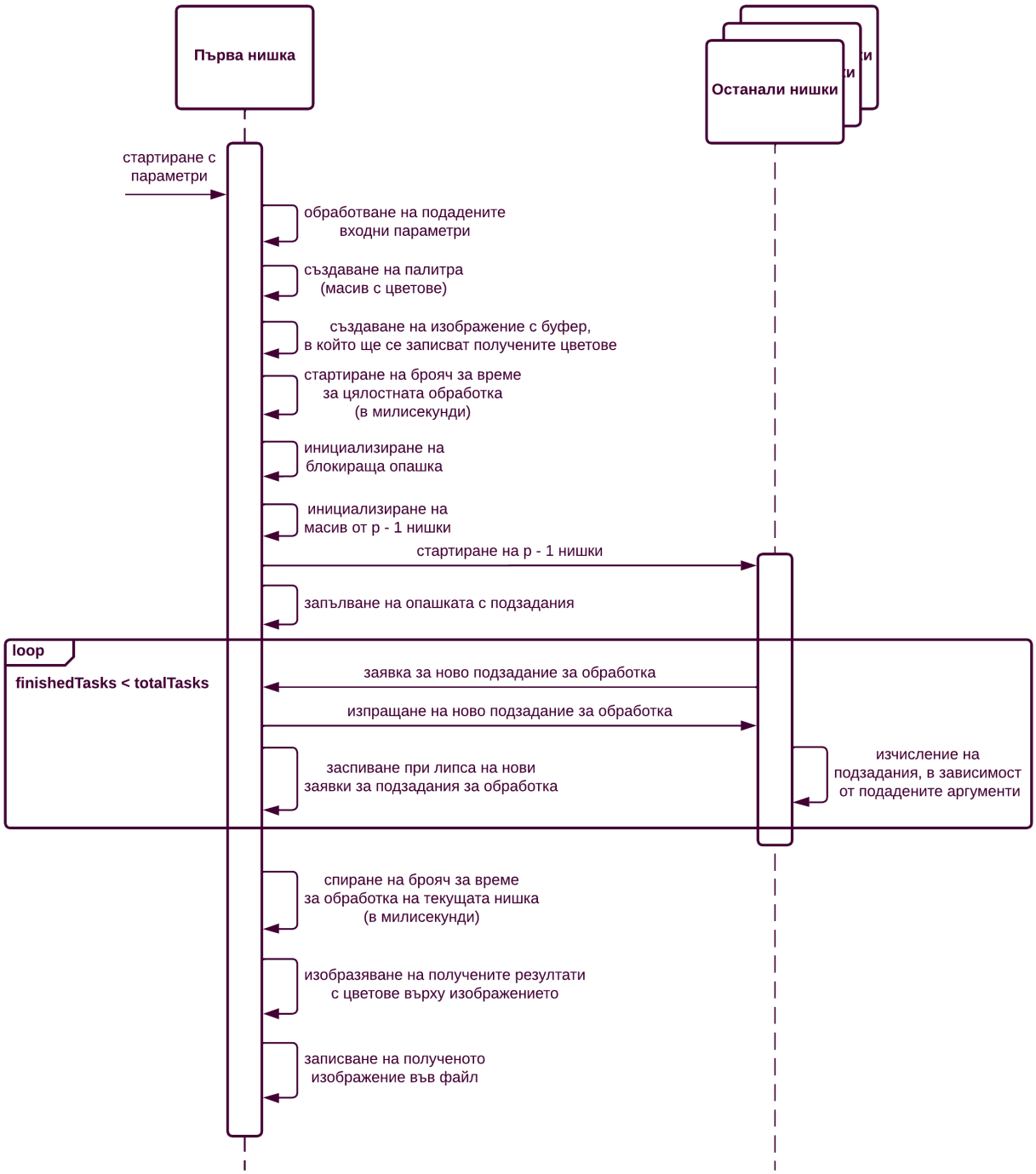
2.2.1.1 Модел на програмата

Моделът на програмата е ясно изразена йерархична Master/Slave архитектура, при която имаме декомпозиция по управление. Характерно за този случай е, че винаги имаме една главна (master) нишка, която не извършва същата работа като останалите нишки, а отговаря само за координирането и синхронизирането на отделните задачи. В началото *master* нишката раздава последователно задачите на останалите нишки, които след като приключат, получават нови задачи от нея. Целият този процес по координация и комуникация с *master* нишката води до известно забавяне в паралелния алгоритъм.

При този модел имаме декомпозиция по код, а не по данни. Това е така, защото главната нишка има различни задачи, а именно да разпределя заданията измежду останалите p – 1 на брой нишки (slaves). Може да се счита, че това вероятно подобрява производителността, тъй като не съществуват нишки, които да остават за дълго време без работа, при условие, че има още задачи, което може да стане при статичното циклично балансиране. От друга страна обаче, процесите по раздаване на нови задачи от главната нишка отнемат време, особено ако много нишки приключат в един и същ момент. Това води до формиране на тясно място (bottleneck), тъй като логическата топология е централизирана („звезда“), която има своите недостатъци по отношение на натовареността на централния възел (концентратора).

При динамичното централизирано балансиране отделните нишки не получават своите задачи предварително въз основа на своя индекс. Задачите им се назначават динамично по време на изпълнение на програмата. Това става чрез т.нар. „thread pool“, който позволява рециклиране на нишките и тяхната повторна работа след приключване на дадено задание. За да проверим твърдението в [4], настоящият проект ще реализира динамично централизирано балансиране по два начина – без използването на готовата вътрешна библиотека java.util.concurrent.Executors и вариант с използването ѝ. Ще реализираме двата начина с цел да проверим закъснението, което готовите функции може да ни генерират, както беше коментирано по-рано.

2.2.1.2 Диаграма на последователностите



Фигура 8: *Диаграма на последователностите за реализация с динамично централизирано балансиране*

Диаграмата на последователностите (UML Sequence Diagram) показва стъпките, извършвани от програмата, по време на динамично централизирано балансиране. Чрез цикъл са изобразени постоянните процеси по комуникация между главната нишка и останалите нишки. Забелязваме, че процесите, които стоят в началото и в края на програмата са аналогични на тези при статично циклично балансиране.

2.2.1.3 Описание на командните параметри

2.2.1.3.1 Входни параметри при реализация без библиотеката „ExecutorService“

Параметър	Описание	Стойност по подразбиране
-s или **-size**	Задава се **размерът на изображението**, което ще се генерира, като широчина и височина в брой пиксели	3840 x 2160 (4K)
-r или **-rect**	Задават се **границите на областта** от комплексната равнина, в която желаем да визуализираме множеството на Манделброт. Задава се чрез подаване на четири числа, разделени със знака „:“, които представляват съответно x-координати и y-координати на избраната област	-0.6386:-0.5986: 0.4486:0.4686 $z = a + i * b$ $a, b \in \mathbb{R}$ $a \in [-0.6386; -0.5986]$ $b \in [0.4486; 0.4686]$
-o или **-out**	Задаване **име на изображението**, което се получава в резултат на изпълнение на програмата	MandelWorldDynamic CustomThreadPool.png
-t или **-threads**	Задават се **броя нишки**, които ще работят паралелно	2
-g или **-gran**	Задава се **грануларността** (броя подзадания, които всяка нишка трябва да изпълни)	1 (възможно най-едрата)
-c или **-cols**	Задава се дали декомпозицията да е **по колони**	false
-q или **-quiet**	Задава се дали да не се визуализира информация за **броя обработени подзадания** на отделните нишки	false
-h или **-help**	Задава се дали да се визуализира **информация** за това какви **параметри** поддържа програмата и начин на използване	false

Когато програмата не е в тих режим, за всяка нишка се визуализират броят задачи, извършени от нишката. Например:

3 tasks done by this thread.

Общото време за работа на програмата се визуализира винаги и е в следния формат:

Total execution time is … ms.

Програмата се стартира чрез bash скрипт по следния начин:

./executeMandelWorld <list-of-arguments>

<list-of-arguments> представлява множеството на входните параметри, които се подават при стартиране на програмата. Ако някои от тях липсват, тогава програмата се стартира с аргументите по подразбиране, описани в горната таблица.

Примерно стартиране на програмата е:

./executeMandelWorld -t 16 -s 2160x2160

При горното стартиране на програмата входните параметри са два на брой. Първият от тях е свързан с това, че програмата трябва да използва 16 на брой нишки. Вторият аргумент указва размерът на изображението, което ще се генерира, в случая с размери 2160x2160px.

2.2.1.3.2 Входни параметри при реализация с библиотеката „ExecutorService“

Параметър	Описание	Стойност по подразбиране
-s или **-size**	Задава се **размерът на изображението**, което ще се генерира, като широчина и височина в брой пиксели	3840 x 2160 (4K)
-r или **-rect**	Задават се **границите на областта** от комплексната равнина, в която желаем да визуализираме множеството на Манделброт. Задава се чрез подаване на четири числа, разделени със знака „:“, които представляват съответно x-координати и y-координати на избраната област	-0.6386:-0.5986: 0.4486:0.4686 $z = a + i * b$ $a, b \in \mathbb{R}$ $a \in [-0.6386; -0.5986]$ $b \in [0.4486; 0.4686]$
-o или **-out**	Задаване **име на изображението**, което се получава в резултат на изпълнение на програмата	MandelWorldDynamic.png
-t или **-threads**	Задават се **броят нишки**, които ще работят паралелно	2
-g или **-gran**	Задава се **грануларността** (условно, броя подзадания, които всяка нишка трябва да изпълни)	1 (възможно най-едрата)
-h или **-help**	Задава се дали да се визуализира **информация** за това какви **параметри** поддържа програмата и начин на използване	false

Общото време за работа на програмата се визуализира винаги и е в следния формат:

Total execution time is … ms.

Програмата се стартира чрез bash скрипт по следния начин:

./executeMandelWorld <list-of-arguments>

<*list-of-arguments*> представлява множеството на входните параметри, които се подават при стартиране на програмата. Ако някои от тях липсват, тогава програмата се стартира с аргументите по подразбиране, описани в горната таблица.

Примерно стартиране на програмата е:

./executeMandelWorld -t 10 -s 2160x2160

При горното стартиране на програмата входните параметри са два на брой. Първият от тях е свързан с това, че програмата трябва да използва 10 на брой нишки. Вторият аргумент указва размерът на изображението, което ще се генерира, в случая с размери 2160x2160px.

2.2.2 Технологично проектиране

2.2.2.1 Използван език и външни библиотеки

За реализирането на приложението е използван езикът за програмиране *Java*. Начинът на стартиране на програмите е през терминал чрез *bash* скрипт.

За изпълнението на програмата са използвани две външни библиотеки под формата на jar файлове. Едната библиотека улеснява изчисленията с комплексни числа, тъй като видяхме, че множеството на Манделброт се дефинира за комплексната равнина. Това е библиотеката: *commons-math3-3.6.1.jar* (The Apache Commons Mathematics Library).

За обработката на аргументите, които се подават при вход на програмата, е използвана библиотеката *commons-cli-1.4.jar* (Apache Commons CLI).

Вътрешната библиотека *java.util.concurrent.Executors* се използва в едната от реализациите. За другата имплементация се разчита на собственоръчно написана блокираща опашка.

2.2.2.2 Тестови среди

Изследванията са проведени на следните две машинни архитектури:

RMI тестова машина

Architecture: x86_64
CPU op-modes(s): 32-bit, 64-bit
Byte Order: Little Endian
CPU(s): 32 (16 физически)
Socket(s): 2
Core(s) per socket: 8
NUMA node(s): 2
Model name: Intel(R) Xeon(R) CP E5-2660 0 @ 2.20GHz
L1d cache: 32KB
L1i cache: 32KB
L2 cache: 256KB
L3 cache: 20480KB

Личен лаптоп

Processor: AMD Ryzen 7 3700U with Radeon Vega Mobile Gfx 2.30 GHz
CPU(s): 8 (4 физически)
CPU op-modes(s): 32-bit, 64-bit
Base Clock: 2.3GHz
L1 cache: 384KB
L2 cache: 2MB
L3 cache: 4MB
Installed RAM: 16 GB

2.2.2.3 Особености на програмата и работа с нишките

2.2.2.3.1 Особености при реализация без библиотеката „ExecutorService“

В тази имплементация не се разчита на библиотеката *java.util.concurrent.Executors*, за да може се оцени липсата на свръхтовар, породен от употребата на библиотеки. За целта се използва Java обект от клас ArrayBlockingQueue<Runnable>. Тази колекция от тип „блокираща опашка“ поддържа операции по изчакване, ако се опитаме да добавим елемент, а тя е вече пълна.

Главната нишка (master) има за задача да запълни блокиращата опашка с отделните подзадания. След всяка приета нова задача от останалите нишки, броят от извършени задачи от дадената нишка се увеличава с единица, с цел накрая да бъде изведен общия брой задачи, изпълнени от всяка нишка:

```
public void run() {
        this.thread = Thread.currentThread();
        while(!isStopped()) {
            try{
                Runnable runnable = (Runnable)taskQueue.take();
                ++countTasks;
                runnable.run();
            } catch(Exception e) { }
        }
    }
```

След това основната нишка изчаква, докато всички задачи са изпълнени, преди да предизвика прекъсване на останалите нишки, реферирайки към тях с метода interrupt():

```
while(!this.taskQueue.isEmpty()) {
            try {
                Thread.sleep(20);
            } catch (InterruptedException e) { e.printStackTrace(); }
}
```

2.2.2.3.2 Особености при реализация с библиотеката „ExecutorService“

В тази имплементация се разчита на библиотеката *java.util.concurrent.Executors*, за да може се оцени евентуалното наличие на свръхтовар, породен от употребата на библиотеки.

Разчита се на т.нар. „FixedThreadPool“, за който е характерно, че се състои от фиксиран брой нишки, като ако в опашката има повече задачи, отколкото налични нишки, задачите не се обработват, докато не се освободи нишка. Първоначално, главната нишка (master) отговаря за създаването на p – 1 останали нишки (slaves). Впоследствие, те биват инициализирани и стартирани с помощта на библиотеката:

```
for (int i = 0; i < rows; i++) {
      pool.execute(tasks[i]);
}
```

Характерно за метода execute(Runnable command) на *Executor.java* обаче е, че според документацията, дадената команда се изпълнява „в определено време в бъдещето“. Това създава неопределеност, когато се разчита на тази имплементация, която в този смисъл не е толкова надеждна, когато става въпрос за ускорението, целящо да се постигне в една паралелна програма. В това отношение, наша собствена реализация на тези методи би била по-добрият избор, тъй като ще имаме повече възможности за управление и контрол над програмата.

Накрая се преминава към организирано изключване, което се състои в привършване на вече започнатите задачи и спиране на приемането на нови задачи. Това се реализира чрез метода shutdown(). Изчакването от страна на главната нишка по отношение на вече приетите задачи се осъществява чрез метода awaitTermination(long timeout, TimeUnit unit):

```
pool.shutdown();
try {
      pool.awaitTermination(Long.MAX_VALUE, TimeUnit.NANOSECONDS);
} catch (InterruptedException e) {
      e.printStackTrace();
}
```

Отново, в неговата документация може да се забележи непредвиденост по отношение на времето на приключване и как то се определя.

След това програмата завършва с обичайните дейности, които бяха характерни и при реализацията със статично циклично балансиране, а именно засичане на време на приключване и изчисляване на общо време за изпълнение, генериране на изображение върху канава и записване във файл.

3. Тестване. Настройка. Внедряване

Всички тестови случаи са проведени при 1024 максимален брой итерации като параметър. Направени са опити за увеличаването му, с цел постигане на по-голяма изчислителна мощност и евентуално по-добро ускорение, но тестовете показаха доста сходни резултати.

3.1 Тестване и настройка на входните параметри

3.1.1 Резултати при реализация със статично циклично балансиране

3.1.1.1 Тестов план

В представения тестов план под формата на таблица са използвани следните означения на колоните:
p – паралелизъм (броят нишки)
g – грануларност
$\mathbf{T_p^{(i)}}$ – време за изпълнение при паралелизъм p, получено на i-тия тест
$\mathbf{S_p}$ – ускорение при паралелизъм p
$\mathbf{E_p}$ – ефективност при паралелизъм p

Тестова машина:	rmi.yaht.net
Област:	-0.6386 : -0.5986 : 0.4486 : 0.4686
Балансиране:	статично цилично балансиране
Макс. брой итерации:	1024
Декомпозиция:	по редове
Размер:	3840 x 2160 (4K)

#	p	g	T_p (1)	T_p (2)	T_p (3)	T_p=min($T_p^{(i)}$)	$S_p = T_1/T_p$	$E_p = S_p/p$
1	1	1	***15861***	***15841***	***15884***	**15841**	1	1
2	2	1	*12406*	*12415*	*12874*	**12406**	1.2768822	0.63844108
3	4	1	*10061*	*9984*	*10120*	**9984**	1.5866386	0.39665966
4	8	1	*6247*	*6299*	*6332*	**6247**	2.5357772	0.31697215
5	12	1	*4695*	*4581*	*4636*	**4581**	3.4579786	0.28816488
6	16	1	*3777*	*3804*	*4080*	**3777**	4.1940694	0.26212934
7	20	1	*3345*	*3581*	*3330*	**3330**	4.7570571	0.23785285
8	24	1	*3154*	*2846*	*3030*	**2846**	5.5660576	0.23191907
9	28	1	*2743*	*2827*	*2763*	**2743**	5.7750638	0.20625228
10	32	1	*2533*	*2565*	*2501*	**2501**	6.3338665	0.19793333
11	1	4	***15866***	***15873***	***15862***	**15862**	1	1
12	2	4	*9766*	*9762*	*9702*	**9702**	1.6349206	0.81746032
13	4	4	*5663*	*5757*	*5814*	**5663**	2.8009889	0.70024722
14	8	4	*3357*	*3364*	*3333*	**3333**	4.7590759	0.59488449
15	12	4	*2486*	*2432*	*2453*	**2432**	6.5222039	0.543517
16	16	4	*1936*	*1983*	*2001*	**1936**	8.1931818	0.51207386

17	20	4	*1858*	*1683*	*1717*	**1683**	9.4248366	0.47124183
18	24	4	*1689*	*1792*	*1712*	**1689**	9.3913558	0.39130649
19	28	4	*1650*	*1671*	*1674*	**1650**	9.6133333	0.34333333
20	32	4	*1570*	*1551*	*1626*	**1551**	10.22695	0.3195922
21	1	10	***15849***	***15901***	***15896***	**15849**	1	1
22	2	10	*9118*	*9155*	*9066*	**9066**	1.74818	0.87409001
23	4	10	*4975*	*5159*	*5218*	**4975**	3.1857286	0.79643216
24	8	10	*2724*	*2701*	*2773*	**2701**	5.8678267	0.73347834
25	12	10	*2030*	*1990*	*1997*	**1990**	7.9643216	0.66369347
26	16	10	*1671*	*1612*	*1622*	**1612**	9.8318859	0.61449287
27	20	10	*1763*	*1738*	*1599*	**1599**	9.9118199	0.49559099
28	24	10	*1635*	*1625*	*1593*	**1593**	9.9491525	0.41454802
29	28	10	*1491*	*1471*	*1504*	**1471**	10.774303	0.38479654
30	32	10	*1442*	*1382*	*1397*	**1382**	11.468162	0.35838007

Тестова машина:	rmi.yaht.net
Област:	-0.6386 : -0.5986 : 0.4486 : 0.4686
Балансиране:	статично цилично балансиране
Макс. брой итерации:	1024
Декомпозиция:	по колони
Размер:	3840 x 2160 (4K)

#	p	g	T_p (1)	T_p (2)	T_p (3)	T_p=min($T_p^{(i)}$)	$S_p = T_1/T_p$	$E_p = S_p/p$
1	1	1	***15649***	***15688***	***15666***	**15649**	1	1
2	2	1	*9862*	*9798*	*9723*	**9723**	1.6094827	0.80474133
3	4	1	*7285*	*7393*	*7334*	**7285**	2.1481126	0.53702814
4	8	1	*5046*	*5083*	*5036*	**5036**	3.1074265	0.38842832
5	12	1	*4121*	*4192*	*4118*	**4118**	3.8001457	0.31667881
6	16	1	*3319*	*3313*	*3268*	**3268**	4.7885557	0.29928473
7	20	1	*3118*	*2938*	*2776*	**2776**	5.6372478	0.28186239
8	24	1	*2576*	*2648*	*2626*	**2576**	6.0749224	0.25312177
9	28	1	*2334*	*2329*	*2321*	**2321**	6.7423524	0.2407983
10	32	1	*2095*	*2228*	*2199*	**2095**	7.4696897	0.2334278
11	1	4	***15635***	***15668***	***15674***	**15635**	1	1
12	2	4	*8781*	*8766*	*8767*	**8766**	1.7835957	0.89179786
13	4	4	*5366*	*5361*	*5344*	**5344**	2.9257111	0.73142777
14	8	4	*3080*	*2994*	*3063*	**2994**	5.2221109	0.65276386
15	12	4	*2175*	*2194*	*2177*	**2175**	7.1885057	0.59904215
16	16	4	*2016*	*2067*	*1923*	**1923**	8.1305252	0.50815783
17	20	4	*1841*	*1784*	*1850*	**1784**	8.7640135	0.43820067
18	24	4	*1732*	*1746*	*1770*	**1732**	9.0271363	0.37613068
19	28	4	*1622*	*1621*	*1577*	**1577**	9.9143944	0.35408551
20	32	4	*1529*	*1481*	*1455*	**1455**	10.745704	0.33580326

21	1	10	***15702***	***15637***	***15653***	**15637**	1	1
22	2	10	*8825*	*8784*	*8619*	**8619**	1.8142476	0.9071238
23	4	10	*4671*	*4659*	*4591*	**4591**	3.4060118	0.85150294
24	8	10	*2522*	*2577*	*2514*	**2514**	6.2199682	0.77749602
25	12	10	*1919*	*1893*	*1857*	**1857**	8.4205708	0.70171423
26	16	10	*1809*	*1586*	*1907*	**1586**	9.8593947	0.61621217
27	20	10	*1653*	*1698*	*1794*	**1653**	9.4597701	0.47298851
28	24	10	*1582*	*1508*	*1627*	**1508**	10.369363	0.43205681
29	28	10	*1444*	*1467*	*1459*	**1444**	10.828947	0.38674812
30	32	10	*1564*	*1411*	*1443*	**1411**	11.082211	0.3463191

Тестова машина:	rmi.yaht.net
Област:	-1.5 : 0.7 : -1 : 1
Балансиране:	статично цилично балансиране
Макс. брой итерации:	1024
Декомпозиция:	по редове
Размер:	3840 x 2160 (4K)

#	p	g	T_p (1)	T_p (2)	T_p (3)	T_p=min($T_p^{(i)}$)	$S_p = T_1/T_p$	$E_p = S_p/p$
1	1	1	***54514***	***54678***	***54552***	**54514**	1	1
2	2	1	*28546*	*28265*	*28399*	**28265**	1.928675	0.96433752
3	4	1	*23329*	*23233*	*23378*	**23233**	2.3464038	0.58660096
4	8	1	*14700*	*14356*	*14626*	**14356**	3.7972973	0.47466216
5	12	1	*10162*	*10215*	*10164*	**10162**	5.3644952	0.44704126
6	16	1	*7988*	*7950*	*7976*	**7950**	6.8571069	0.42856918
7	20	1	*6587*	*6790*	*6903*	**6587**	8.2759982	0.41379991
8	24	1	*6190*	*6180*	*6135*	**6135**	8.8857376	0.37023907
9	28	1	*6061*	*5610*	*5420*	**5420**	10.057934	0.35921191
10	32	1	*5519*	*5425*	*5149*	**5149**	10.587299	0.33085308
11	1	4	***54401***	***54540***	***54523***	**54401**	1	1
12	2	4	*28077*	*28414*	*28279*	**28077**	1.9375646	0.96878228
13	4	4	*16058*	*15479*	*15361*	**15361**	3.5415012	0.8853753
14	8	4	*8291*	*8296*	*8314*	**8291**	6.5614522	0.82018152
15	12	4	*6249*	*5864*	*5963*	**5864**	9.2771146	0.77309288
16	16	4	*4759*	*4844*	*4835*	**4759**	11.431183	0.71444894
17	20	4	*5039*	*5035*	*5177*	**5035**	10.804568	0.5402284
18	24	4	*4682*	*4797*	*4700*	**4682**	11.61918	0.48413249
19	28	4	*4562*	*4517*	*4568*	**4517**	12.043613	0.43012904
20	32	4	*4360*	*4294*	*4387*	**4294**	12.669073	0.39590854
21	1	10	***54492***	***54564***	***54451***	**54451**	1	1
22	2	10	*28486*	*28458*	*28453*	**28453**	1.9137174	0.95685868
23	4	10	*15104*	*15033*	*14970*	**14970**	3.6373413	0.90933534
24	8	10	*7958*	*7887*	*7962*	**7887**	6.9038925	0.86298656

25	12	10	*5661*	*5580*	*5638*	**5580**	9.7582437	0.81318698
26	16	10	*4830*	*4714*	*4668*	**4668**	11.664739	0.72904617
27	20	10	*5156*	*5157*	*5231*	**5156**	10.560706	0.5280353
28	24	10	*4575*	*4776*	*4509*	**4509**	12.07607	0.50316959
29	28	10	*4518*	*4487*	*4485*	**4485**	12.140691	0.43359611
30	32	10	*4347*	*4273*	*4319*	**4273**	12.743038	0.39821993

Тестова машина:	rmi.yaht.net
Област:	-1.5 : 0.7 : -1 : 1
Балансиране:	статично цилично балансиране
Макс. брой итерации:	1024
Декомпозиция:	по колони
Размер:	3840 x 2160 (4K)

#	p	g	T_p (1)	T_p (2)	T_p (3)	T_p=min($T_p^{(i)}$)	$S_p = T_1/T_p$	$E_p = S_p/p$
1	1	1	***54296***	***54345***	***54404***	**54296**	1	1
2	2	1	*35485*	*35440*	*35667*	**35440**	1.5320542	0.76602709
3	4	1	*27551*	*27206*	*27250*	**27206**	1.9957362	0.49893406
4	8	1	*14173*	*13886*	*14049*	**13886**	3.9101253	0.48876566
5	12	1	*11168*	*11210*	*11124*	**11124**	4.8809781	0.40674817
6	16	1	*8159*	*8216*	*8143*	**8143**	6.6678128	0.4167383
7	20	1	*7203*	*7441*	*7350*	**7203**	7.5379703	0.37689851
8	24	1	*6810*	*6506*	*6350*	**6350**	8.5505512	0.35627297
9	28	1	*5968*	*5561*	*6720*	**5561**	9.7637116	0.34870398
10	32	1	*5351*	*4925*	*5899*	**4925**	11.024569	0.34451777
11	1	4	***54423***	***54629***	***54452***	**54423**	1	1
12	2	4	*29130*	*29149*	*28746*	**28746**	1.8932373	0.94661866
13	4	4	*15463*	*14991*	*15430*	**14991**	3.6303782	0.90759456
14	8	4	*7972*	*8845*	*7886*	**7886**	6.9012173	0.86265217
15	12	4	*5541*	*5561*	*5545*	**5541**	9.8218733	0.81848944
16	16	4	*4553*	*4810*	*4588*	**4553**	11.953218	0.7470761
17	20	4	*4977*	*5082*	*4943*	**4943**	11.010115	0.55050577
18	24	4	*4667*	*4706*	*4624*	**4624**	11.76968	0.49040333
19	28	4	*4431*	*4476*	*4393*	**4393**	12.388573	0.44244903
20	32	4	*4275*	*4311*	*4291*	**4275**	12.730526	0.39782895
21	1	10	***54293***	***54370***	***54388***	**54293**	1	1
22	2	10	*27891*	*28299*	*28156*	**27891**	1.9466136	0.9733068
23	4	10	*14665*	*14677*	*14447*	**14447**	3.7580813	0.93952032
24	8	10	*7541*	*7710*	*7764*	**7541**	7.1997083	0.89996353
25	12	10	*5940*	*5635*	*5564*	**5564**	9.757908	0.813159
26	16	10	*4514*	*4594*	*4463*	**4463**	12.165136	0.76032097
27	20	10	*5023*	*5063*	*5108*	**5023**	10.808879	0.54044396
28	24	10	*4339*	*4712*	*4557*	**4339**	12.512791	0.52136629

29	28	10	*4467*	*4423*	*4522*	**4423**	12.275153	0.43839831
30	32	10	*4339*	*4310*	*4345*	**4310**	12.596984	0.39365574

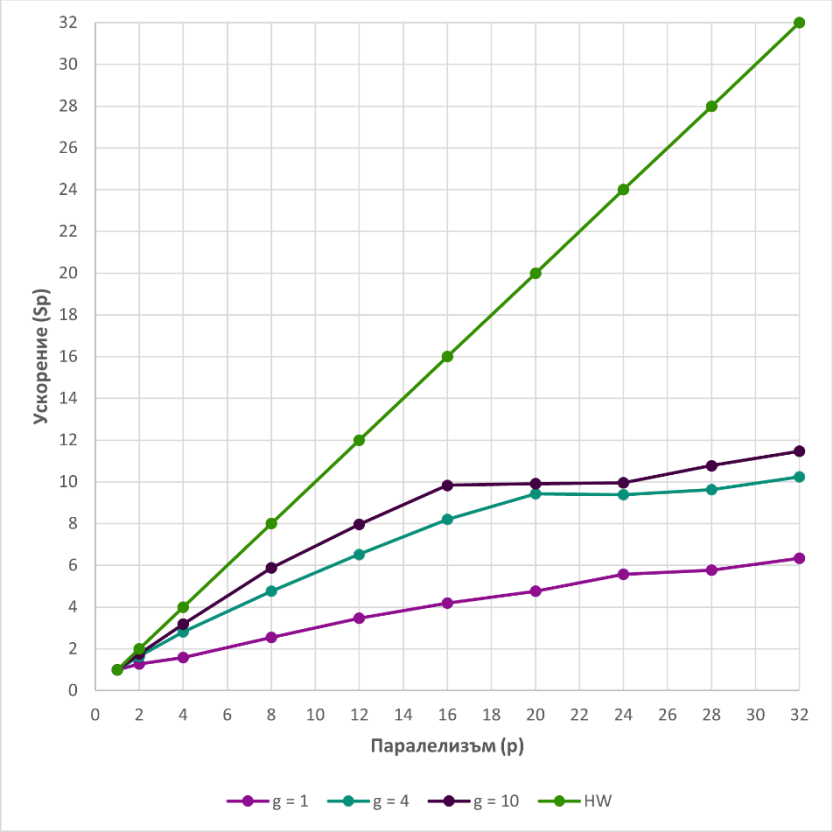
Тестова машина:	PC
Област:	-0.6386 : -0.5986 : 0.4486 : 0.4686
Балансиране:	статично цилично балансиране
Макс. брой итерации:	1024
Декомпозиция:	по колони
Размер:	3840 x 2160 (4K)

#	p	g	T_p (1)	T_p (2)	T_p (3)	T_p=min($T_p^{(i)}$)	$S_p = T_1/T_p$	$E_p = S_p/p$
1	1	1	***10599***	***11715***	***11071***	**10599**	1	1
2	2	1	*8080*	*9302*	*9805*	**8080**	1.3117574	0.65587871
3	4	1	*5399*	*5414*	*7127*	**5399**	1.9631413	0.49078533
4	6	1	*5496*	*6132*	*6545*	**5496**	1.9284934	0.32141557
5	8	1	*5344*	*6082*	*5793*	**5344**	1.9833458	0.24791823
6	1	4	***9808***	***11010***	***10705***	**9808**	1	1
7	2	4	*7210*	*7538*	*7667*	**7210**	1.3603329	0.68016644
8	4	4	*5662*	*6371*	*6030*	**5662**	1.7322501	0.43306252
9	6	4	*5149*	*5576*	*5332*	**5149**	1.9048359	0.31747265
10	8	4	*4235*	*5015*	*5655*	**4235**	2.3159386	0.28949233
11	1	10	***10062***	***10566***	***10184***	**10062**	1	1
12	2	10	*6498*	*5993*	*5903*	**5903**	1.704557	0.8522785
13	4	10	*4176*	*4210*	*4083*	**4083**	2.4643644	0.61609111
14	6	10	*3914*	*4417*	*4904*	**3914**	2.5707716	0.42846193
15	8	10	*4229*	*4815*	*4891*	**4229**	2.3792859	0.29741074
16	1	32	***10240***	***10410***	***10326***	**10240**	1	1
17	2	32	*6688*	*7133*	*7389*	**6688**	1.5311005	0.76555024
18	4	32	*5497*	*5779*	*5503*	**5497**	1.8628343	0.46570857
19	6	32	*4656*	*5111*	*5239*	**4656**	2.1993127	0.36655212
20	8	32	*4082*	*4617*	*4712*	**4082**	2.5085742	0.31357178
21	1	64	***10403***	***11006***	***10766***	**10403**	1	1
22	2	64	*5755*	*6870*	*7908*	**5755**	1.8076455	0.90382276
23	4	64	*5399*	*5658*	*5165*	**5165**	2.0141336	0.5035334
24	6	64	*4638*	*5160*	*4995*	**4638**	2.2429927	0.37383211
25	8	64	*3746*	*4551*	*4041*	**3746**	2.7770956	0.34713695

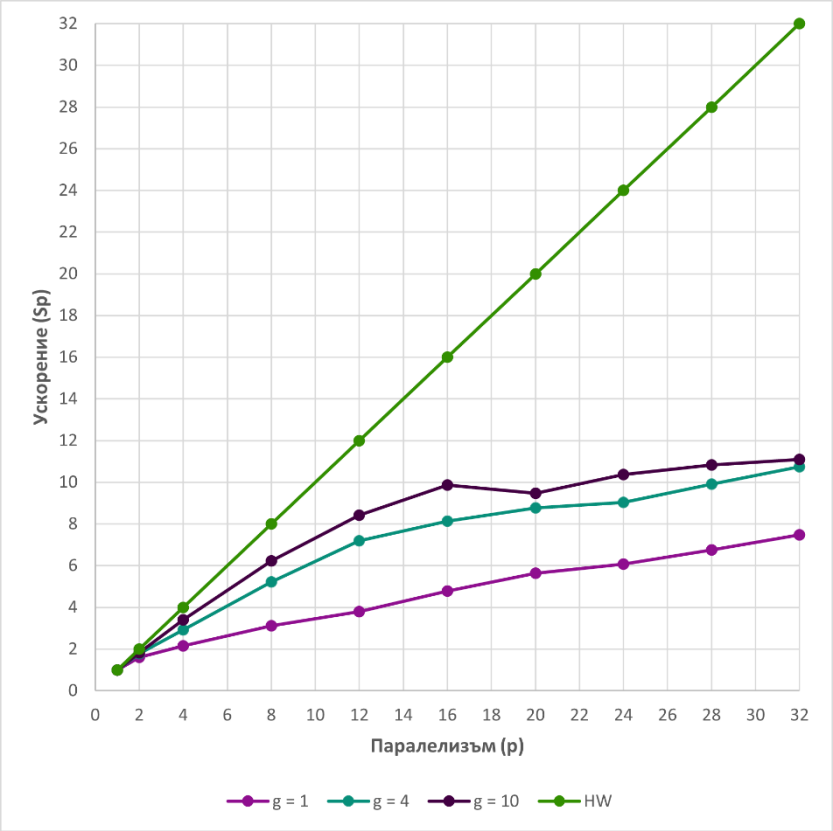
Тестова машина:	PC
Област:	-1.5 : 0.7 : -1 : 1
Балансиране:	статично цилично балансиране
Макс. брой итерации:	1024
Декомпозиция:	по колони
Размер:	3840 x 2160 (4K)

#	p	g	T_p (1)	T_p (2)	T_p (3)	T_p=min($T_p^{(i)}$)	$S_p = T_1/T_p$	$E_p = S_p/p$
1	1	32	***35570***	***35346***	***38106***	**35346**	1	1
2	2	32	*17423*	*18404*	*19166*	**17423**	2.0286977	1.01434885
3	4	32	*14866*	*17740*	*20658*	**14866**	2.3776403	0.59441006
4	6	32	*16101*	*16664*	*14675*	**14675**	2.408586	0.40143101
5	8	32	*13863*	*15295*	*10535*	**10535**	3.355102	0.41938776

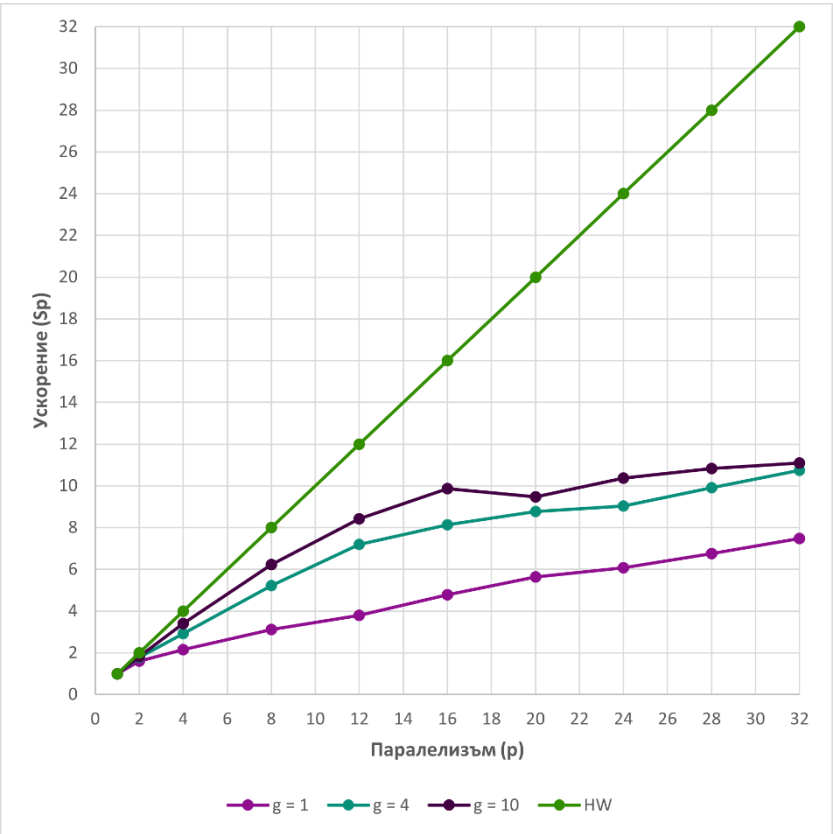
3.1.1.2 Графики



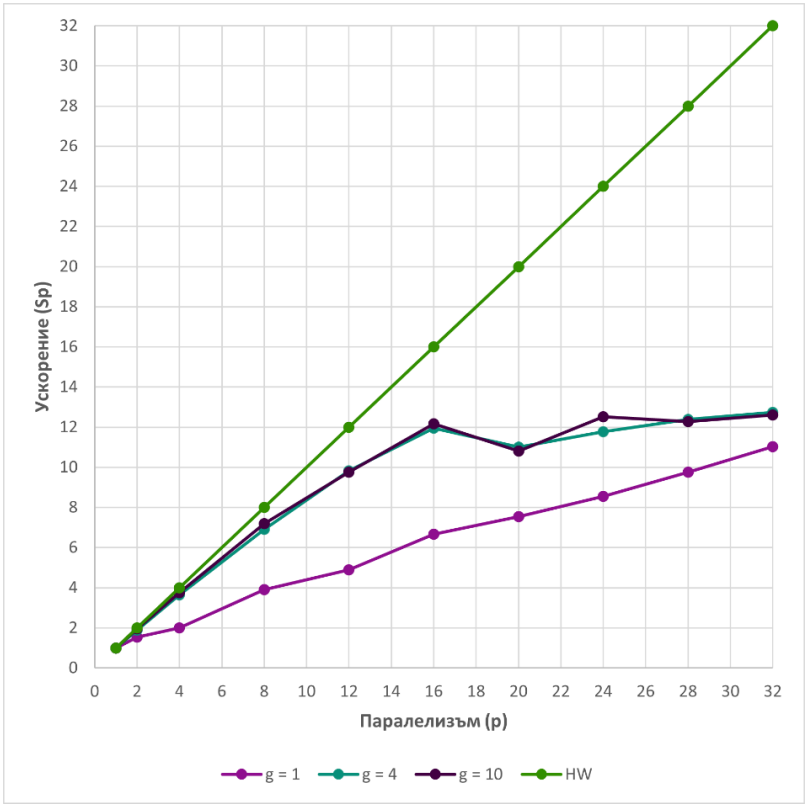
Фигура 9: *Графика на ускорението при статично циклично балансиране, 1024 итерации, по редове, област: -0.6386:-0.5986:0.4486:0.4686, размер на генерирано изображение: 4K*



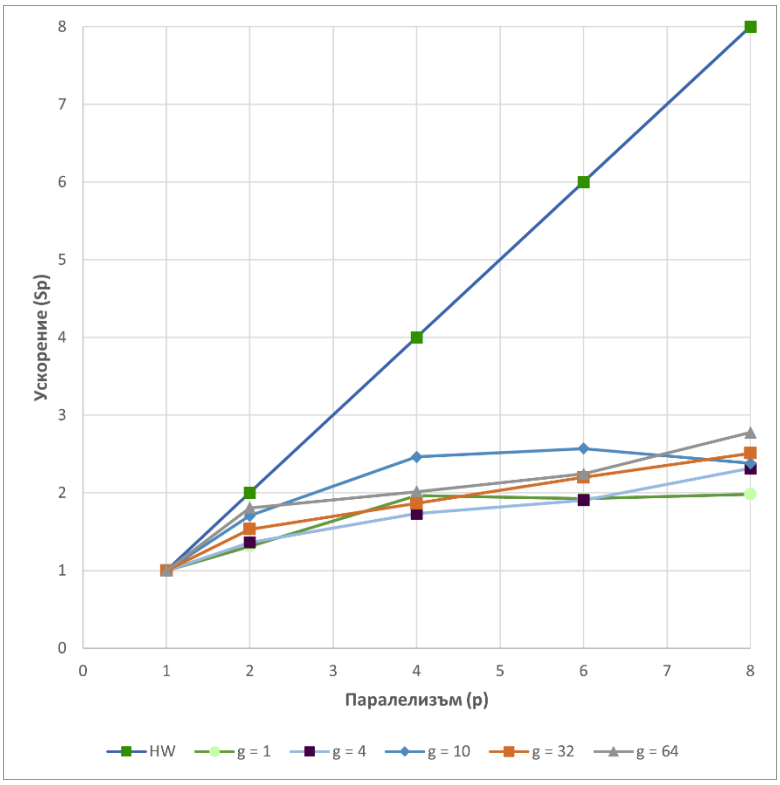
Фигура 10: *Графика на ускорението при статично циклично балансиране, 1024 итерации, по колони, област: -0.6386:-0.5986:0.4486:0.4686, размер на генерирано изображение: 4K*



Фигура 11: *Графика на ускорението при статично циклично балансиране, 1024 итерации, по редове, област: -1.5:0.7:-1:1, размер на генерирано изображение: 4K*



Фигура 12: *Графика на ускорението при статично циклично балансиране, 1024 итерации, по колони, област: -1.5:0.7:-1:1, размер на генерирано изображение: 4K*



Фигура 13: *Графика на ускорението на четириядрена машина при статично циклично балансиране, 1024 итерации, по колони, област: -0.6386:-0.5986:0.4486:0.4686, размер на генерирано изображение: 4K*

3.1.1.3 Изводи

От кривите на ускорението се забелязва, че при избраната малка област от теста на Манделброт, декомпозицията по редове води до малко по-високи резултати за ускорението, отколкото декомпозицията по колони. Въпреки това, и двете са много сходни. Отчасти, това се дължи на сложността на избраната област и спецификата на множеството на Манделброт. Забелязваме, че максималното получено ускорение на практика не надминава 12.

Наблюдава се и че при най-едрата грануларност, се постига двойно по-малко ускорение. Това е очаквано, защото едрата грануларност може да доведе до проблеми с балансирането, особено в небалансирани проблеми, какъвто е тестът на Манделброт. При него няма как да не настъпи ситуация, в която дадена нишка да има по-сложна задача, която да ѝ отнеме повече време за обработка, в сравнение с тази на някоя друга нишка. За да се постигне желаното ускорение е много важно във всеки един момент всички или поне повечето нишки да са ангажирани със задачи, за да може последователната част на алгоритъма да бъде сведена до минимум.

Причината за по-добрите резултати при нарастване на коефициента на грануларност е, че в този случай броят подзадания се увеличава и има по-голям шанс отделните нишки да получат по няколко по-лесни и по-трудни задачи за изчисление.

При тестовите случаи проведени на четириядрена машина се забелязва случай на близка стойност до хардуерния паралелизъм. Това е суперлинейна аномалия, която може да се дължи на няколко причини. От една страна, няма как да изключим възможността за статистическа грешка, която може да се дължи и на самото измерване. От друга страна, възможно е и ситуация, в която при избраната грануларност и паралелизъм, размерът на отделните подзадания да е станал толкова малък, че да се е вместил цялостно в L1 кеша за данни. В такива случаи говорим за асоциативност на L1 D-Cache. Това би създало по-малко натоварване и по-рядко обръщане към операционната система, което би могло да доведе до тази немонотонна, спрямо останалите резултати, аномалия.

3.1.2 Резултати при реализация с динамично централизирано балансиране без използване на библиотеката „ExecutorService“

3.1.2.1 Тестов план

В представения тестов план под формата на таблица са използвани следните означения на колоните:
p – паралелизъм (броят нишки)
g – грануларност
$\mathbf{T_p^{(i)}}$ – време за изпълнение при паралелизъм p, получено на i-тия тест
$\mathbf{S_p}$ – ускорение при паралелизъм p
$\mathbf{E_p}$ – ефективност при паралелизъм p

Тестова машина:	rmi.yaht.net
Област:	-0.6386 : -0.5986 : 0.4486 : 0.4686
Балансиране:	динамично централизирано балансиране (без библиотека "ExecutorService")
Макс. брой итерации:	1024
Декомпозиция:	по редове
Размер:	3840 x 2160 (4K)

#	p	g	T_p (1)	T_p (2)	T_p (3)	T_p=min($T_p^{(i)}$)	$S_p = T_1/T_p$	$E_p = S_p/p$
1	1	1	***15861***	***15841***	***15884***	**15841**	1	1
2	2	1	*15802*	*15840*	*16165*	**15802**	1.002468	0.50123402
3	4	1	*10123*	*10101*	*10167*	**10101**	1.5682606	0.39206514
4	8	1	*6189*	*6146*	*6339*	**6146**	2.5774487	0.32218109
5	12	1	*4734*	*4601*	*4670*	**4601**	3.4429472	0.28691227
6	16	1	*3840*	*3803*	*3769*	**3769**	4.2029716	0.26268573
7	20	1	*3236*	*3290*	*3314*	**3236**	4.895241	0.24476205
8	24	1	*3030*	*3056*	*2888*	**2888**	5.4851108	0.22854628
9	28	1	*2729*	*2624*	*2730*	**2624**	6.0369665	0.21560595
10	32	1	*2560*	*2561*	*2563*	**2560**	6.1878906	0.19337158
11	1	4	***15866***	***15873***	***15862***	**15862**	1	1
12	2	4	*15778*	*16665*	*15847*	**15778**	1.0053239	0.50266193
13	4	4	*5690*	*5648*	*5625*	**5625**	2.8199111	0.70497778
14	8	4	*2680*	*2656*	*2657*	**2656**	5.9721386	0.74651732
15	12	4	*1835*	*1872*	*1860*	**1835**	8.6441417	0.72034514
16	16	4	*1570*	*1577*	*1624*	**1570**	10.103185	0.63144904
17	20	4	*1744*	*1665*	*1521*	**1521**	10.428665	0.52143327
18	24	4	*1478*	*1497*	*1596*	**1478**	10.73207	0.4471696
19	28	4	*1381*	*1441*	*1468*	**1381**	11.48588	0.41020999
20	32	4	*1380*	*1385*	*1373*	**1373**	11.552804	0.36102513
21	1	10	***15849***	***15901***	***15896***	**15849**	1	1
22	2	10	*16158*	*15819*	*15760*	**15760**	1.0056472	0.5028236
23	4	10	*5684*	*5701*	*5704*	**5684**	2.7883533	0.69708832
24	8	10	*2626*	*2642*	*2653*	**2626**	6.0354151	0.75442688
25	12	10	*1834*	*1845*	*1843*	**1834**	8.6417666	0.72014722
26	16	10	*1596*	*1533*	*1501*	**1501**	10.558961	0.65993504
27	20	10	*1424*	*1580*	*1504*	**1424**	11.129916	0.55649579
28	24	10	*1421*	*1442*	*1438*	**1421**	11.153413	0.46472555
29	28	10	*1381*	*1387*	*1404*	**1381**	11.476466	0.4098738
30	32	10	*1335*	*1351*	*1365*	**1335**	11.87191	0.37099719

Тестова машина:	rmi.yaht.net
Област:	-0.6386 : -0.5986 : 0.4486 : 0.4686
Балансиране:	динамично централизирано балансиране (без библиотека "ExecutorService")
Макс. брой итерации:	1024
Декомпозиция:	по колони
Размер:	3840 x 2160 (4K)

#	p	g	T_p (1)	T_p (2)	T_p (3)	T_p=min($T_p^{(i)}$)	$S_p = T_1/T_p$	$E_p = S_p/p$
1	1	1	***15649***	***15688***	***15666***	**15649**	1	1
2	2	1	*15702*	*15699*	*15678*	**15678**	0.9981503	0.49907514
3	4	1	*7472*	*7305*	*7405*	**7305**	2.1422313	0.53555784
4	8	1	*6322*	*6226*	*6210*	**6210**	2.5199678	0.31499597
5	12	1	*4212*	*4229*	*4211*	**4211**	3.7162194	0.30968495
6	16	1	*3419*	*3344*	*3657*	**3344**	4.6797249	0.29248281
7	20	1	*2825*	*2780*	*2806*	**2780**	5.6291367	0.28145683
8	24	1	*2502*	*2542*	*2664*	**2502**	6.2545963	0.26060818
9	28	1	*2406*	*2283*	*2493*	**2283**	6.8545773	0.24480633
10	32	1	*2137*	*2158*	*2255*	**2137**	7.3228825	0.22884008
11	1	4	***15635***	***15668***	***15674***	**15635**	1	1
12	2	4	*15523*	*15520*	*16059*	**15520**	1.0074098	0.5037049
13	4	4	*5659*	*5617*	*5706*	**5617**	2.7835143	0.69587858
14	8	4	*2673*	*2634*	*2679*	**2634**	5.935839	0.74197988
15	12	4	*1998*	*1923*	*1963*	**1923**	8.1305252	0.67754377
16	16	4	*1602*	*1619*	*1580*	**1580**	9.8955696	0.6184731
17	20	4	*1481*	*1542*	*1564*	**1481**	10.557056	0.5278528
18	24	4	*1480*	*1416*	*1519*	**1416**	11.041667	0.46006944
19	28	4	*1484*	*1496*	*1524*	**1484**	10.535714	0.37627551
20	32	4	*1430*	*1434*	*1422*	**1422**	10.995077	0.34359617
21	1	10	***15702***	***15637***	***15653***	**15637**	1	1
22	2	10	*15861*	*16021*	*15602*	**15602**	1.0022433	0.50112165
23	4	10	*5545*	*5494*	*5524*	**5494**	2.8461959	0.71154896
24	8	10	*2637*	*2639*	*2632*	**2632**	5.9411094	0.74263868
25	12	10	*1771*	*1826*	*1776*	**1771**	8.8294749	0.73578957
26	16	10	*1583*	*1504*	*1497*	**1497**	10.445558	0.65284736
27	20	10	*1464*	*1459*	*1500*	**1459**	10.717615	0.53588074
28	24	10	*1440*	*1451*	*1421*	**1421**	11.004222	0.45850927
29	28	10	*1420*	*1410*	*1420*	**1410**	11.090071	0.39607396
30	32	10	*1378*	*1400*	*1367*	**1367**	11.438917	0.35746617

Тестова машина:	rmi.yaht.net
Област:	-1.5 : 0.7 : -1 : 1
Балансиране:	динамично централизирано балансиране (без библиотека "ExecutorService")
Макс. брой итерации:	1024
Декомпозиция:	по редове
Размер:	3840 x 2160 (4K)

#	p	g	T_p (1)	T_p (2)	T_p (3)	T_p=min($T_p^{(i)}$)	$S_p = T_1/T_p$	$E_p = S_p/p$
1	1	1	***54514***	***54678***	***54552***	**54514**	1	1
2	2	1	*55389*	*55367*	*55093*	**55093**	0.9894905	0.49474525
3	4	1	*23922*	*23201*	*23321*	**23201**	2.3496401	0.58741003
4	8	1	*14295*	*14395*	*14385*	**14295**	3.8135012	0.47668765
5	12	1	*10303*	*10384*	*10345*	**10303**	5.2910803	0.44092336
6	16	1	*8248*	*8074*	*8195*	**8074**	6.7517959	0.42198724
7	20	1	*6912*	*6773*	*7059*	**6773**	8.0487229	0.40243614
8	24	1	*5929*	*5922*	*6184*	**5922**	9.205336	0.38355567
9	28	1	*6034*	*5811*	*5601*	**5601**	9.7329048	0.34760374
10	32	1	*5473*	*5500*	*5554*	**5473**	9.9605335	0.31126667
11	1	4	***53976***	***54540***	***54523***	**53976**	1	1
12	2	4	*55080*	*55118*	*55334*	**55080**	0.9799564	0.48997821
13	4	4	*20529*	*20386*	*19968*	**19968**	2.703125	0.67578125
14	8	4	*9164*	*8893*	*8963*	**8893**	6.0694929	0.75868661
15	12	4	*6145*	*6123*	*6109*	**6109**	8.8354886	0.73629072
16	16	4	*4943*	*4829*	*4855*	**4829**	11.177469	0.69859184
17	20	4	*4660*	*4686*	*4560*	**4560**	11.836842	0.59184211
18	24	4	*4571*	*4493*	*4458*	**4458**	12.107672	0.50448632
19	28	4	*4429*	*4429*	*4508*	**4429**	12.18695	0.4352482
20	32	4	*4399*	*4467*	*4300*	**4300**	12.552558	0.39226744
21	1	10	***54492***	***54564***	***54016***	**54016**	1	1
22	2	10	*54753*	*54814*	*54953*	**54753**	0.9865396	0.49326978
23	4	10	*18725*	*18823*	*18583*	**18583**	2.9067427	0.72668568
24	8	10	*8615*	*8893*	*8644*	**8615**	6.2699942	0.78374927
25	12	10	*5843*	*5889*	*5881*	**5843**	9.2445661	0.77038051
26	16	10	*5042*	*4686*	*4682*	**4682**	11.53695	0.72105938
27	20	10	*4534*	*4460*	*4521*	**4460**	12.111211	0.60556054
28	24	10	*4446*	*4425*	*4356*	**4356**	12.400367	0.51668197
29	28	10	*4321*	*4405*	*4415*	**4321**	12.50081	0.4464575
30	32	10	*4313*	*4369*	*4297*	**4297**	12.570631	0.39283221

Тестова машина:	rmi.yaht.net
Област:	-1.5 : 0.7 : -1 : 1
Балансиране:	динамично централизирано балансиране (без библиотека "ExecutorService")
Макс. брой итерации:	1024
Декомпозиция:	по колони
Размер:	3840 x 2160 (4K)

#	p	g	T_p (1)	T_p (2)	T_p (3)	T_p=min($T_p^{(i)}$)	$S_p = T_1/T_p$	$E_p = S_p/p$
1	1	1	***54296***	***54345***	***54404***	**54296**	1	1
2	2	1	*55490*	*55360*	*55433*	**55360**	0.98078035	0.49039017
3	4	1	*27889*	*27556*	*28038*	**27556**	1.97038757	0.49259689
4	8	1	*13981*	*14176*	*14081*	**13981**	3.88355625	0.48544453
5	12	1	*11152*	*11354*	*11190*	**11152**	4.8687231	0.40572692
6	16	1	*8236*	*8181*	*8369*	**8181**	6.63684146	0.41480259
7	20	1	*7471*	*7771*	*7498*	**7471**	7.26756793	0.3633784
8	24	1	*6710*	*6191*	*6167*	**6167**	8.80428085	0.36684504
9	28	1	*6470*	*5843*	*5919*	**5843**	9.29248674	0.33187453
10	32	1	*6008*	*5581*	*5774*	**5581**	9.72872245	0.30402258
11	1	4	***54006***	***54526***	***54452***	**54006**	1	1
12	2	4	*55162*	*55285*	*55443*	**55162**	0.97904354	0.48952177
13	4	4	*20365*	*20210*	*20168*	**20168**	2.67780643	0.66945161
14	8	4	*9647*	*9698*	*9540*	**9540**	5.66100629	0.70762579
15	12	4	*6378*	*6168*	*6167*	**6167**	8.75725636	0.72977136
16	16	4	*4768*	*4909*	*4765*	**4765**	11.333893	0.70836831
17	20	4	*4992*	*4764*	*4831*	**4764**	11.336272	0.5668136
18	24	4	*4558*	*4546*	*4521*	**4521**	11.9455873	0.4977328
19	28	4	*4452*	*4425*	*4482*	**4425**	12.2047458	0.43588378
20	32	4	*4354*	*4281*	*4268*	**4268**	12.653702	0.39542819
21	1	10	***54036***	***54370***	***54388***	**54036**	1	1
22	2	10	*54991*	*54896*	*54869*	**54869**	0.98481839	0.49240919
23	4	10	*18965*	*18801*	*18823*	**18801**	2.87410244	0.71852561
24	8	10	*8624*	*8648*	*8771*	**8624**	6.26576994	0.78322124
25	12	10	*5932*	*6015*	*6029*	**5932**	9.10923803	0.75910317
26	16	10	*4737*	*4862*	*4745*	**4737**	11.4072198	0.71295123
27	20	10	*4600*	*4461*	*4404*	**4404**	12.2697548	0.61348774
28	24	10	*4453*	*4389*	*4424*	**4389**	12.3116883	0.51298701
29	28	10	*4408*	*4371*	*4347*	**4347**	12.4306418	0.44395149
30	32	10	*4337*	*4284*	*4265*	**4265**	12.6696366	0.39592614

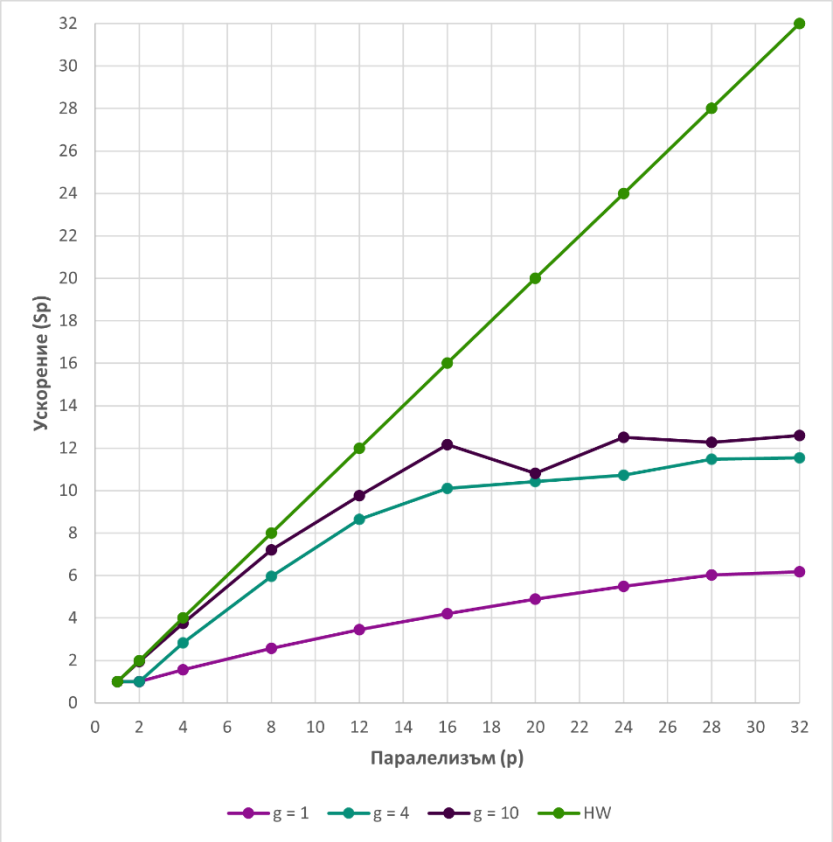
Тестова машина:	PC
Област:	-0.6386 : -0.5986 : 0.4486 : 0.4686
Балансиране:	динамично централизирано балансиране (без библиотека "ExecutorService")
Макс. брой итерации:	1024
Декомпозиция:	по редове
Размер:	3840 x 2160 (4K)

#	p	g	T_p (1)	T_p (2)	T_p (3)	T_p=min($T_p^{(i)}$)	$S_p = T_1/T_p$	$E_p = S_p/p$
1	1	1	***10599***	***11715***	***11071***	**10599**	1	1
2	2	1	*10130*	*10290*	*10212*	**10130**	1.0462981	0.52314906
3	4	1	*5385*	*5542*	*5424*	**5385**	1.9682451	0.49206128
4	6	1	*4914*	*4845*	*5142*	**4845**	2.1876161	0.36460268
5	8	1	*4396*	*4384*	*4365*	**4365**	2.4281787	0.30352234
6	1	4	***10062***	***10566***	***10184***	**10062**	1	1
7	2	4	*10157*	*10656*	*10798*	**10157**	0.9906468	0.49532342
8	4	4	*4374*	*4398*	*4634*	**4374**	2.3004115	0.57510288
9	6	4	*3959*	*3997*	*4017*	**3959**	2.5415509	0.42359182
10	8	4	*3942*	*4786*	*4744*	**3942**	2.5525114	0.31906393
11	1	10	***10062***	***10566***	***10184***	**10062**	1	1
12	2	10	*11062*	*10268*	*10805*	**10268**	0.9799377	0.48996884
13	4	10	*4370*	*4628*	*4541*	**4370**	2.3025172	0.57562929
14	6	10	*3919*	*4040*	*3922*	**3919**	2.5674917	0.42791528
15	8	10	*3277*	*3253*	*3261*	**3253**	3.0931448	0.3866431
16	1	32	***10240***	***10410***	***10326***	**10240**	1	1
17	2	32	*10293*	*10130*	*10236*	**10130**	1.0108588	0.50542942
18	4	32	*4391*	*4298*	*4701*	**4298**	2.3825035	0.59562587
19	6	32	*3879*	*4005*	*4070*	**3879**	2.6398556	0.43997594
20	8	32	*4132*	*4751*	*4882*	**4132**	2.4782188	0.30977735
21	1	64	***10403***	***11006***	***10766***	**10403**	1	1
22	2	64	*10146*	*10158*	*10836*	**10146**	1.0253302	0.51266509
23	4	64	*4429*	*4624*	*4523*	**4429**	2.3488372	0.5872093
24	6	64	*3846*	*3848*	*3873*	**3846**	2.7048882	0.4508147
25	8	64	*3270*	*3334*	*3250*	**3250**	3.2009231	0.40011538

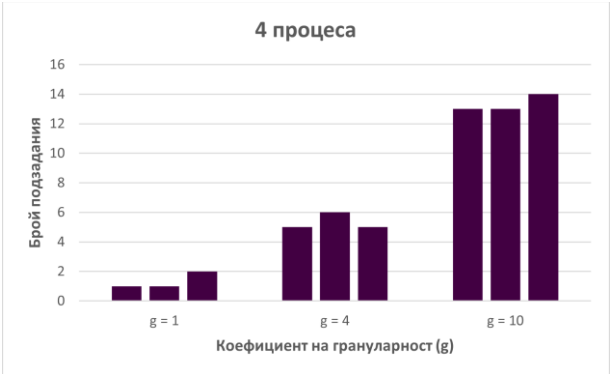
Тестова машина:	PC
Област:	-1.5 : 0.7 : -1 : 1
Балансиране:	динамично централизирано балансиране (без библиотека "ExecutorService")
Макс. брой итерации:	1024
Декомпозиция:	по колони
Размер:	3840 x 2160 (4K)

#	p	g	T_p (1)	T_p (2)	T_p (3)	T_p=min(T_p(i))	$S_p = T_1/T_p$	$E_p = S_p/p$
1	1	32	***35570***	***35346***	***38106***	**35346**	1	1
2	2	32	*36522*	*34998*	*36275*	**34998**	1.0099434	0.50497171
3	4	32	*14790*	*15746*	*15701*	**14790**	2.389858	0.5974645
4	6	32	*17102*	*18091*	*18244*	**17102**	2.0667758	0.34446264
5	8	32	*11734*	*15591*	*14877*	**11734**	3.012272	0.376534

3.1.2.2 Графики



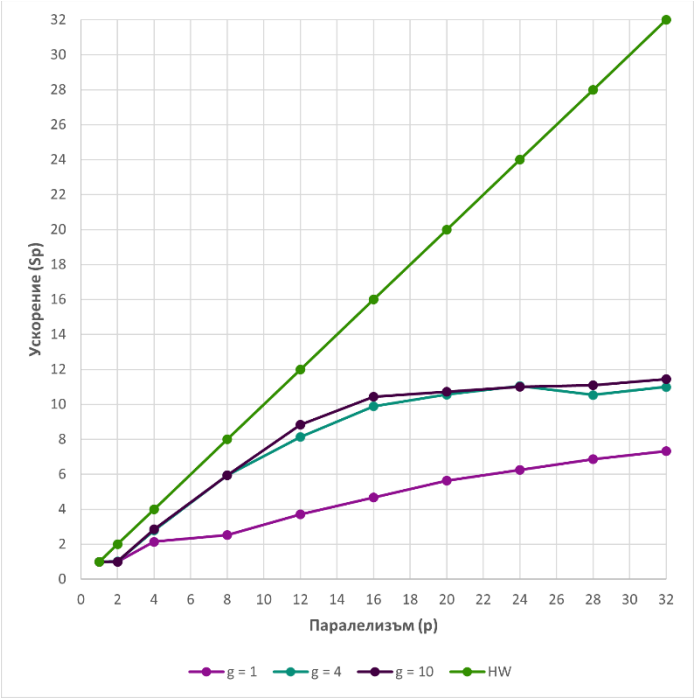
Фигура 14: *Графика на ускорението при динамично централизирано балансиране, 1024 итерации, по редове, област: -0.6386:-0.5986:0.4486:0.4686, размер на генерирано изображение: 4K*



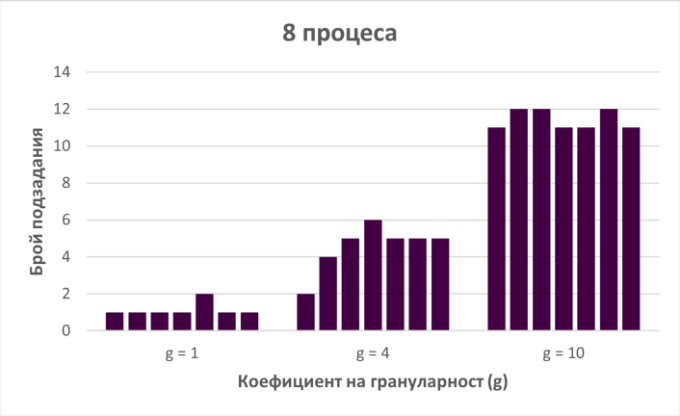
Фигура 15: *Хистограма на броя обработени подзадания от всяка нишка при паралелизъм 4, при динамично централизирано балансиране, 1024 итерации, по редове, област: -0.6386:-0.5986:0.4486:0.4686, размер на генерирано изображение: 4K*

Фигура 16: *Хистограма на броя обработени подзадания от всяка нишка при паралелизъм 16, при динамично централизирано балансиране, 1024 итерации, по редове, област: -0.6386:-0.5986:0.4486:0.4686, размер на генерирано изображение: 4K*

Фигура 17: *Хистограма на броя обработени подзадания от всяка нишка при паралелизъм 32, при динамично централизирано балансиране, 1024 итерации, по редове, област: -0.6386:-0.5986:0.4486:0.4686, размер на генерирано изображение: 4K*



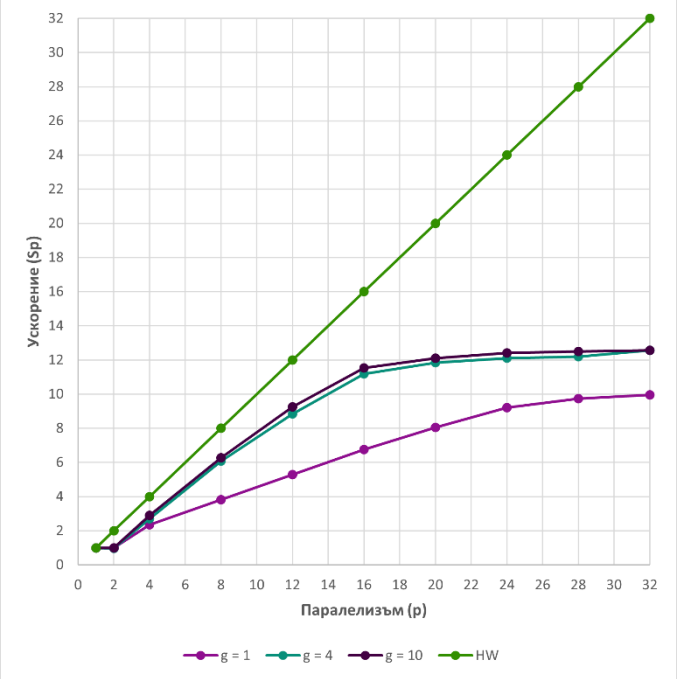
Фигура 18: *Графика на ускорението при динамично централизирано балансиране, 1024 итерации, по колони, област: -0.6386:-0.5986:0.4486:0.4686, размер на генерирано изображение: 4K*



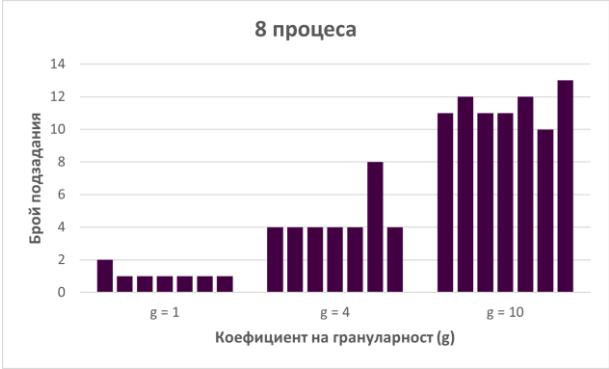
Фигура 19: *Хистограма на броя обработени подзадания от всяка нишка при паралелизъм 4, при динамично централизирано балансиране, 1024 итерации, по колони, област: -0.6386:-0.5986:0.4486:0.4686, размер на генерирано изображение: 4K*

Фигура 20: *Хистограма на броя обработени подзадания от всяка нишка при паралелизъм 16, при динамично централизирано балансиране, 1024 итерации, по колони, област: -0.6386:-0.5986:0.4486:0.4686, размер на генерирано изображение: 4K*

Фигура 21: *Хистограма на броя обработени подзадания от всяка нишка при паралелизъм 32, при динамично централизирано балансиране, 1024 итерации, по колони, област: -0.6386:-0.5986:0.4486:0.4686, размер на генерирано изображение: 4K*

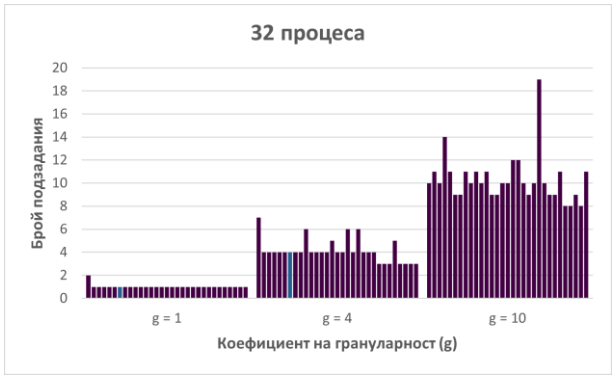


Фигура 22: *Графика на ускорението при динамично централизирано балансиране, 1024 итерации, по редове, област: -1.5:0.7:-1:1, размер на генерирано изображение: 4K*

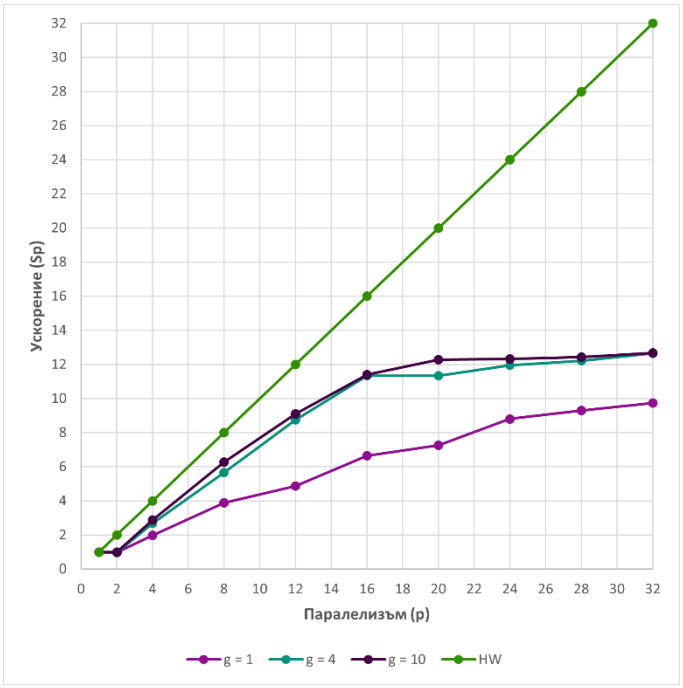


Фигура 23: *Хистограма на броя обработени подзадания от всяка нишка при паралелизъм 8, при динамично централизирано балансиране, 1024 итерации, по редове, област: -1.5:0.7:-1:1, размер на генерирано изображение: 4K*

Фигура 24: *Хистограма на броя обработени подзадания от всяка нишка при паралелизъм 16, при динамично централизирано балансиране, 1024 итерации, по редове, област: -1.5:0.7:-1:1, размер на генерирано изображение: 4K*



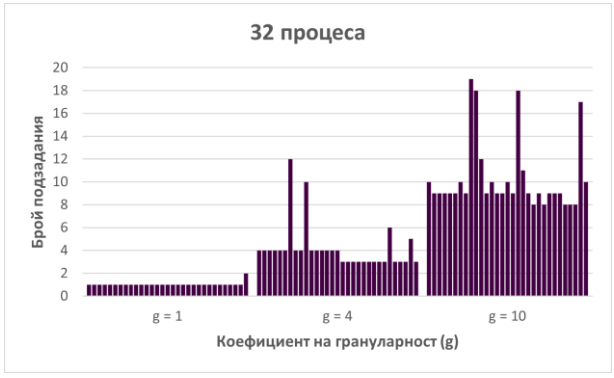
Фигура 25: *Хистограма на броя обработени подзадания от всяка нишка при паралелизъм 32, при динамично централизирано балансиране, 1024 итерации, по редове, област: -1.5:0.7:-1:1, размер на генерирано изображение: 4K*



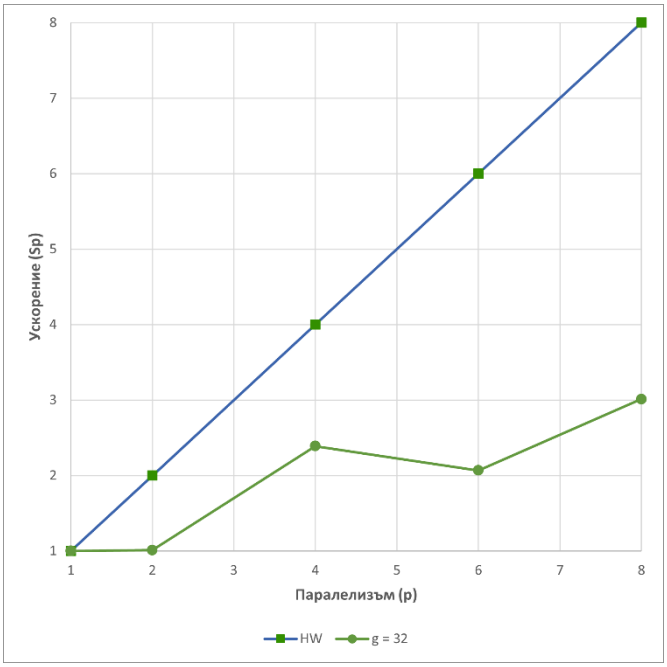
Фигура 26: *Графика на ускорението при динамично централизирано балансиране, 1024 итерации, по колони, област: -1.5:0.7:-1:1, размер на генерирано изображение: 4K*

Фигура 27: *Хистограма на броя обработени подзадания от всяка нишка при паралелизъм 4, при динамично централизирано балансиране, 1024 итерации, по колони, област: -1.5:0.7:-1:1, размер на генерирано изображение: 4K*

Фигура 28: *Хистограма на броя обработени подзадания от всяка нишка при паралелизъм 16, при динамично централизирано балансиране, 1024 итерации, по колони, област: -1.5:0.7:-1:1, размер на генерирано изображение: 4K*



Фигура 29: *Хистограма на броя обработени подзадания от всяка нишка при паралелизъм 32, при динамично централизирано балансиране, 1024 итерации, по колони, област: -1.5:0.7:-1:1, размер на генерирано изображение: 4K*



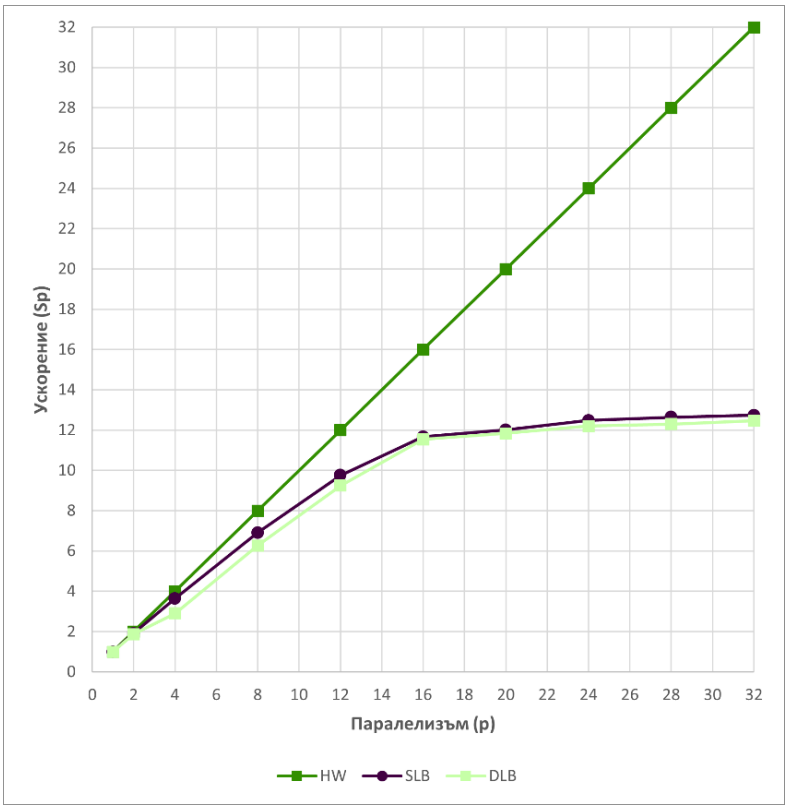
Фигура 30: *Графика на ускорението на четириядрена машина при динамично централизирано балансиране, 1024 итерации, по колони, област: -1.5:0.7:-1:1, размер на генерирано изображение: 4K*

3.1.2.3 Изводи

От кривите на ускорението се забелязва, че и при динамичното централизирано балансиране, при избраната малка област от теста на Манделброт, декомпозицията по редове води до малко по-високи резултати за ускорението, отколкото декомпозицията по колони. Въпреки това, и двете са отново много сходни. Това отново се дължи на сложността на избраната област и спецификата на теста на Манделброт.

Забелязваме, че максималното получено ускорение на практика отново не надминава 12. Въпреки, че е доста сходно с полученото ускорение при статично циклично балансиране, то все пак остава по-ниско. Причината за това е породения свръхтовар при процесите на комуникация и обмен на задачи между основната нишка (master) и работните нишки.

От хистограмите, демонстриращи броя задачи, обработени от всяка нишка, се вижда, че разпределението е сравнително равномерно. В някои от случаите се наблюдава малко по-голяма дисперсия, но това е сравнително рядко. Това е положителен факт, тъй като означава, че не се получава свръхнатоварване само върху една определена нишка, а задачите са сравнително поравно разпределени. Въпреки това, определеността и детерминистичността при статичното циклично балансиране се вижда, че остава по-добрият избор, водещ до по-високи стойности на ускорението.



Фигура 31: *Сравнение на кривите на статично циклично балансиране и динамично централизирано балансиране при коефициент на грануларност g = 10, 1024 итерации, декомпозиция по колони, област: -1.5:0.7:-1:1, размер на генерирано изображение: 4K*

3.1.3 Резултати при реализация с динамично централизирано балансиране с използване на библиотеката „ExecutorService“

3.1.3.1 Тестов план

В представения тестов план под формата на таблица са използвани следните означения на колоните:
p – паралелизъм (броят нишки)
g – грануларност
$T_p^{(i)}$ – време за изпълнение при паралелизъм p, получено на i-тия тест
S_p – ускорение при паралелизъм p
E_p – ефективност при паралелизъм p

Тестова машина:	rmi.yaht.net
Област:	-0.6386 : -0.5986 : 0.4486 : 0.4686
Балансиране:	динамично централизирано балансиране (с библиотека "ExecutorService")
Макс. брой итерации:	1024
Декомпозиция:	по редове
Размер:	3840 x 2160 (4K)

#	p	g	T_p (1)	T_p (2)	T_p (3)	T_p=min($T_p^{(i)}$)	$S_p = T_1/T_p$	$E_p = S_p/p$
1	1	1	***15861***	***15841***	***15884***	**15841**	1	1
2	2	1	*16309*	*16360*	*16399*	**16309**	0.9713042	0.48565209
3	4	1	*10491*	*10710*	*10787*	**10491**	1.5099609	0.37749023
4	8	1	*6642*	*6580*	*6546*	**6546**	2.4199511	0.30249389
5	12	1	*4945*	*5008*	*4962*	**4945**	3.2034378	0.26695315
6	16	1	*4063*	*4336*	*4042*	**4042**	3.9190995	0.24494372
7	20	1	*3763*	*3461*	*3663*	**3461**	4.5770009	0.22885004
8	24	1	*3316*	*3260*	*3332*	**3260**	4.8592025	0.20246677
9	28	1	*3043*	*3020*	*3039*	**3020**	5.2453642	0.18733444
10	32	1	*2718*	*2705*	*2706*	**2705**	5.8561922	0.18300601
11	1	4	***15866***	***15873***	***15862***	**15862**	1	1
12	2	4	*16404*	*16330*	*16462*	**16330**	0.9713411	0.48567055
13	4	4	*5734*	*5751*	*5799*	**5734**	2.7663062	0.69157656
14	8	4	*2783*	*2764*	*2726*	**2726**	5.8187821	0.72734776
15	12	4	*1903*	*1875*	*1896*	**1875**	8.4597333	0.70497778
16	16	4	*1666*	*1756*	*1522*	**1522**	10.421813	0.65136334
17	20	4	*1813*	*1699*	*1494*	**1494**	10.617135	0.53085676
18	24	4	*1656*	*1544*	*1649*	**1544**	10.273316	0.42805484
19	28	4	*1474*	*1502*	*1529*	**1474**	10.761194	0.38432836
20	32	4	*1422*	*1443*	*1424*	**1422**	11.154712	0.34858474
21	1	10	***15849***	***15901***	***15896***	**15849**	1	1
22	2	10	*16465*	*16439*	*16415*	**16415**	0.9655193	0.48275967
23	4	10	*5876*	*5821*	*5799*	**5799**	2.7330574	0.68326436

24	8	10	*2677*	*2700*	*2705*	**2677**	5.9204333	0.74005417
25	12	10	*1879*	*1882*	*1866*	**1866**	8.4935691	0.70779743
26	16	10	*1538*	*1530*	*1618*	**1530**	10.358824	0.64742647
27	20	10	*1471*	*1426*	*1457*	**1426**	11.114306	0.55571529
28	24	10	*1440*	*1423*	*1416*	**1416**	11.192797	0.46636653
29	28	10	*1412*	*1385*	*1398*	**1385**	11.443321	0.40869005
30	32	10	*1370*	*1349*	*1378*	**1349**	11.748703	0.36714696

Тестова машина:	rmi.yaht.net
Област:	-1.5 : 0.7 : -1 : 1
Балансиране:	динамично централизирано балансиране (с библиотека "ExecutorService")
Макс. брой итерации:	1024
Декомпозиция:	по редове
Размер:	3840 x 2160 (4K)

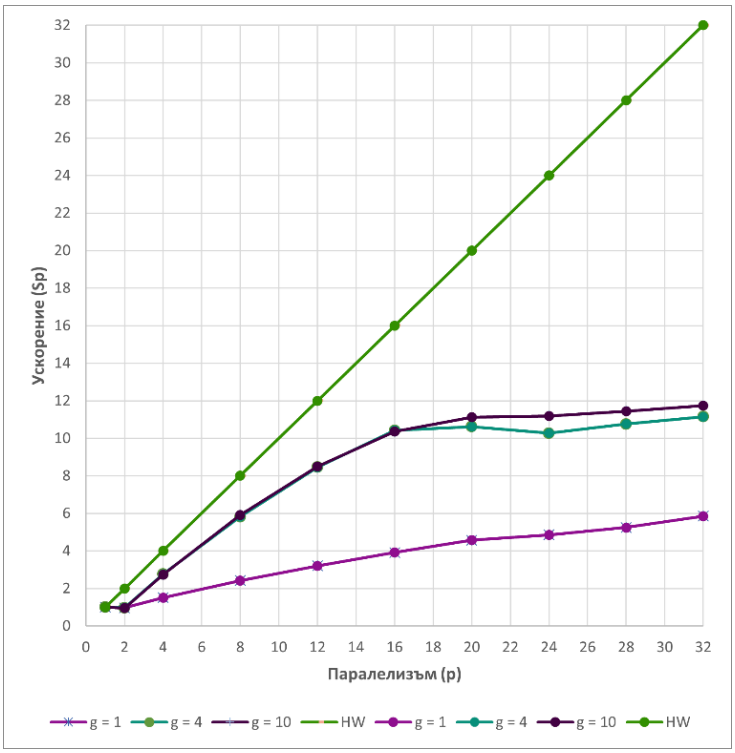
#	p	g	$T_p^{(1)}$	$T_p^{(2)}$	$T_p^{(3)}$	$T_p=\min(T_p^{(i)})$	$S_p = T_1/T_p$	$E_p = S_p/p$
1	1	1	***54514***	***54678***	***54552***	**54514**	1	1
2	2	1	*61529*	*61488*	*61488*	**61488**	0.8865795	0.44328975
3	4	1	*26279*	*26270*	*25991*	**25991**	2.0974183	0.52435458
4	8	1	*15919*	*15939*	*16151*	**15919**	3.4244613	0.42805767
5	12	1	*11344*	*11421*	*11502*	**11344**	4.805536	0.40046133
6	16	1	*8894*	*9381*	*8814*	**8814**	6.1849331	0.38655832
7	20	1	*7342*	*7244*	*7457*	**7244**	7.5254003	0.37627002
8	24	1	*6673*	*6832*	*6889*	**6673**	8.1693391	0.34038913
9	28	1	*6302*	*6287*	*6127*	**6127**	8.8973396	0.31776213
10	32	1	*5917*	*5759*	*5708*	**5708**	9.5504555	0.29845173
11	1	4	***54401***	***54540***	***54523***	**54401**	1	1
12	2	4	*61445*	*61486*	*61433*	**61433**	0.8855338	0.44276692
13	4	4	*22845*	*22621*	*22715*	**22621**	2.4048893	0.60122232
14	8	4	*10333*	*10116*	*9931*	**9931**	5.4778975	0.68473719
15	12	4	*6843*	*6782*	*6784*	**6782**	8.0213801	0.66844834
16	16	4	*5418*	*5508*	*5466*	**5418**	10.04079	0.62754937
17	20	4	*5335*	*5107*	*5210*	**5107**	10.652242	0.5326121
18	24	4	*4892*	*4917*	*5033*	**4892**	11.120401	0.46335003
19	28	4	*4858*	*4961*	*4875*	**4858**	11.19823	0.39993678
20	32	4	*4813*	*4815*	*4761*	**4761**	11.426381	0.35707441
21	1	10	***54492***	***54564***	***54451***	**54451**	1	1
22	2	10	*61996*	*62227*	*61899*	**61899**	0.879675	0.43983748
23	4	10	*21436*	*21365*	*21542*	**21365**	2.5486075	0.63715188
24	8	10	*9915*	*9878*	*10024*	**9878**	5.5123507	0.68904383
25	12	10	*6537*	*6559*	*6655*	**6537**	8.3296619	0.69413849
26	16	10	*5784*	*5275*	*5378*	**5275**	10.322464	0.64515403

27	20	10	*5027*	*4983*	*5051*	**4983**	10.927353	0.54636765
28	24	10	*4870*	*4865*	*4887*	**4865**	11.192395	0.46634978
29	28	10	*4832*	*4843*	*4853*	**4832**	11.268833	0.40245831
30	32	10	*4694*	*4708*	*4720*	**4694**	11.600128	0.36250399

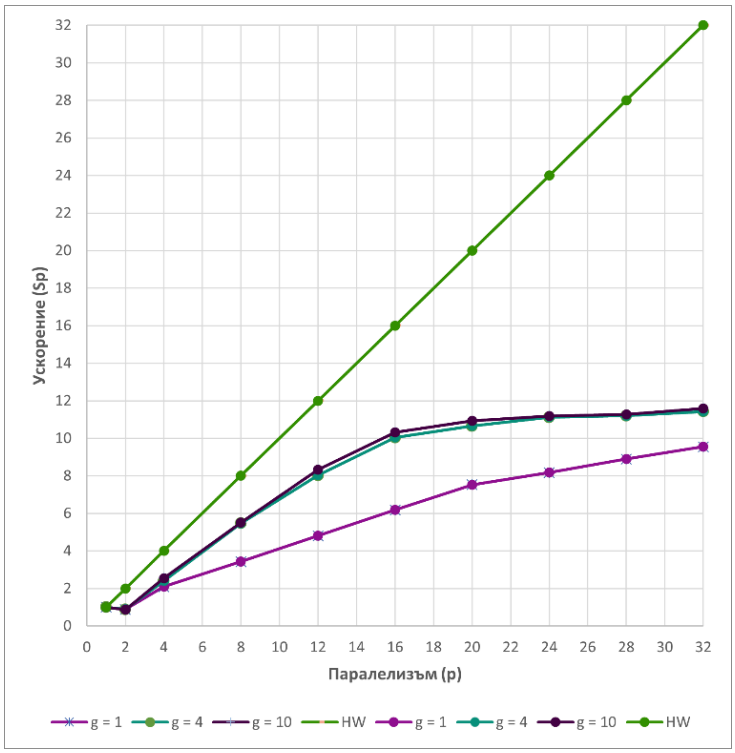
Тестова машина:	PC
Област:	-0.6386 : -0.5986 : 0.4486 : 0.4686
Балансиране:	динамично централизирано балансиране (с библиотека "ExecutorService")
Макс. брой итерации:	1024
Декомпозиция:	по редове
Размер:	3840 x 2160 (4K)

#	p	g	T_p (1)	T_p (2)	T_p (3)	T_p=min($T_p^{(i)}$)	$S_p = T_1/T_p$	$E_p = S_p/p$
1	1	1	***10599***	***11715***	***11071***	**10599**	1	1
2	2	1	*10674*	*10681*	*10779*	**10674**	0.9929736	0.49648679
3	4	1	*7295*	*7400*	*8015*	**7295**	1.452913	0.36322824
4	6	1	*6802*	*6726*	*5979*	**5979**	1.7727045	0.29545074
5	8	1	*5190*	*7032*	*5983*	**5190**	2.0421965	0.25527457
6	1	4	***10062***	***10566***	***10184***	**10062**	1	1
7	2	4	*10789*	*10995*	*10591*	**10591**	0.9500519	0.47502597
8	4	4	*4801*	*4555*	*4627*	**4555**	2.2090011	0.55225027
9	6	4	*3748*	*3897*	*5230*	**3748**	2.6846318	0.44743863
10	8	4	*4173*	*4598*	*5054*	**4173**	2.411215	0.30140187
11	1	10	***10062***	***10566***	***10184***	**10062**	1	1
12	2	10	*10604*	*10441*	*10579*	**10441**	0.9637008	0.4818504
13	4	10	*4981*	*5495*	*6075*	**4981**	2.0200763	0.50501907
14	6	10	*3914*	*3885*	*3925*	**3885**	2.5899614	0.43166023
15	8	10	*3323*	*3320*	*3778*	**3320**	3.0307229	0.37884036
16	1	32	***10240***	***10410***	***10326***	**10240**	1	1
17	2	32	*10597*	*10535*	*10896*	**10535**	0.9719981	0.48599905
18	4	32	*4393*	*4349*	*4543*	**4349**	2.3545643	0.58864107
19	6	32	*3835*	*3917*	*3850*	**3835**	2.6701434	0.4450239
20	8	32	*3318*	*4757*	*5521*	**3318**	3.0861965	0.38577456
21	1	64	***10403***	***11006***	***10766***	**10403**	1	1
22	2	64	*10382*	*10723*	*10860*	**10382**	1.0020227	0.50101137
23	4	64	*4649*	*4696*	*4911*	**4649**	2.2376855	0.55942138
24	6	64	*3907*	*3939*	*4041*	**3907**	2.6626568	0.44377613
25	8	64	*3440*	*3876*	*4532*	**3440**	3.0241279	0.37801599

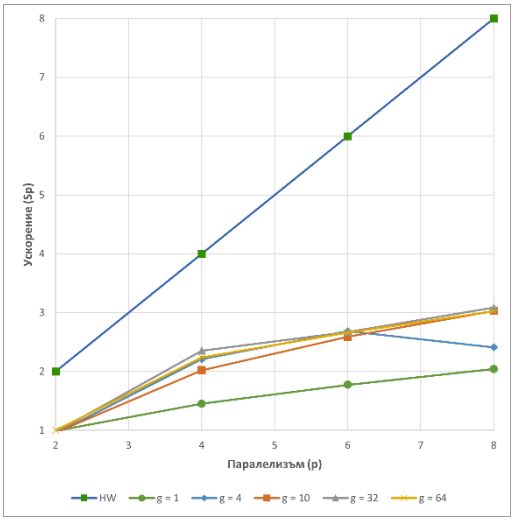
3.1.3.2 Графики



***Фигура 32**: Графика на ускорението при динамично централизирано балансиране (с библиотека „ExecutorService“), 1024 итерации, по редове, област: -0.6386:-0.5986:0.4486:0.4686, размер на генерирано изображение: 4K*



***Фигура 33:** Графика на ускорението при динамично централизирано балансиране (с библиотека „ExecutorService“), 1024 итерации, по редове, област: -1.5:0.7:-1:1, размер на генерирано изображение: 4K*

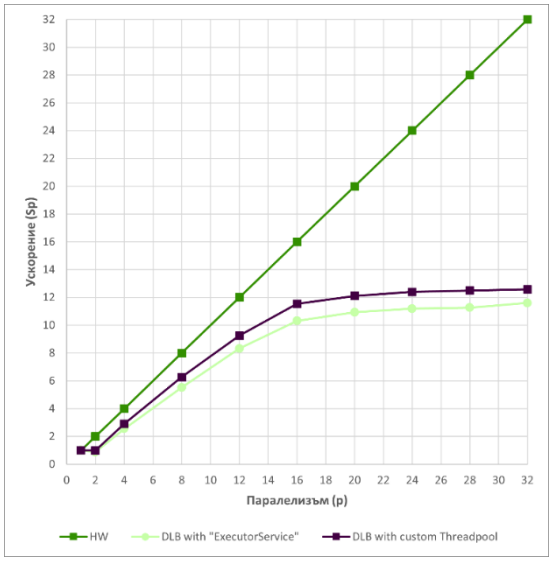


Фигура 34: *Графика на ускорението на четириядрена машина при динамично централизирано балансиране (с библиотека „ExecutorService“), 1024 итерации, по редове, област: -0.6386:-0.5986:0.4486:0.4686, размер на генерирано изображение: 4K*

3.1.3.3 Изводи

От тестовите таблици и графиките на кривите на ускорението се вижда, че динамичното централизирано балансиране, което не разчита на външната библиотека „ExecutorService“, се представя по-добре по отношение на генерираното ускорение, спрямо варианта, който зависи от библиотеката. Както казахме в частта по проектирането, външната библиотека не ни дава детерминистичността, от която се нуждаем в паралелното програмиране. Напротив, основните методи за работя с нея не конкретизират точно определен срок на работа, а посочват, че ще се изпълнят „по някое време в бъдещето“.

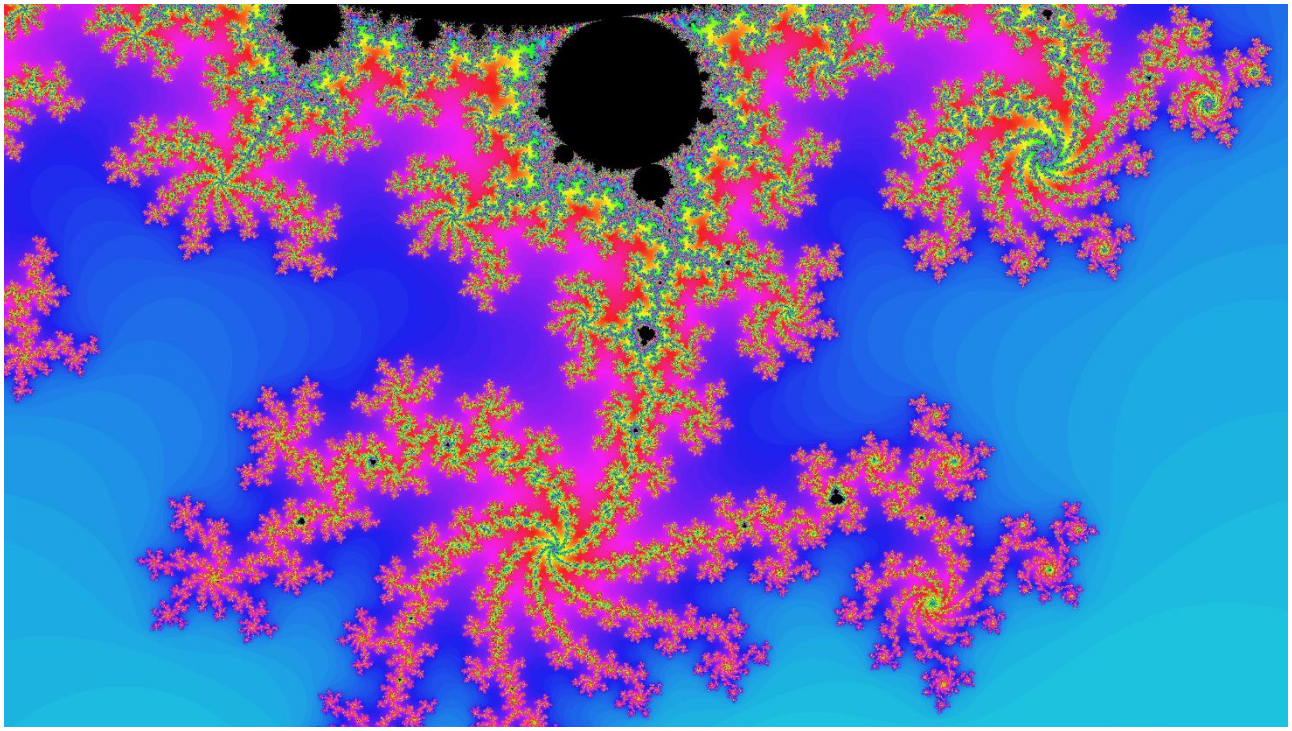
Ускорението, което се получава при реализацията, която разчита на готовата функция, в нито един момент не надминава кривата на другата реализация. Това потвърждава позицията на авторите на източник [4], че готовите функции доста често може да донесат допълнителен свръхтовар в паралелен алгоритъм.



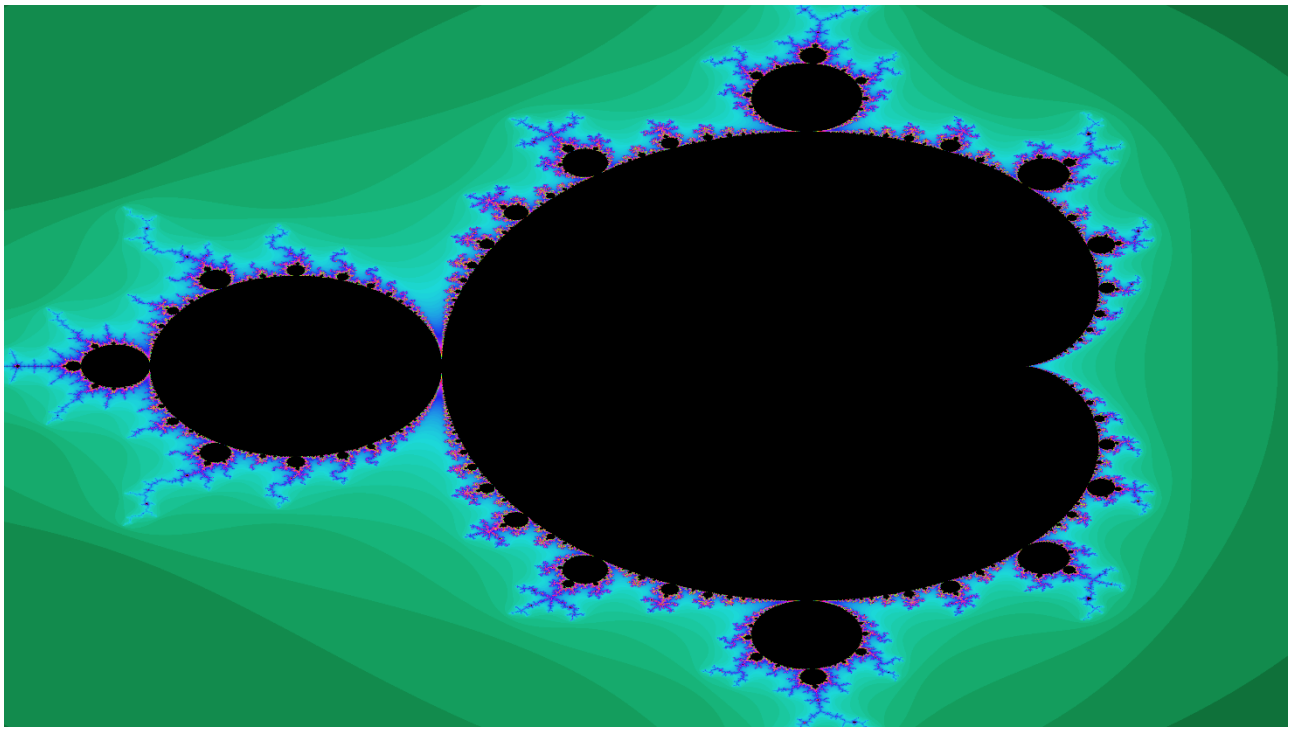
Фигура 35: *Сравнение на кривите на динамично централизирано балансиране с използване на вътрешната библиотеката „ExecutorService“ и динамично централизирано балансиране, което не разчита на библиотеката „ExecutorService“ при коефициент на грануларност g = 10, 1024 итерации, декомпозиция по редове, област:-1.5:0.7:-1:1, размер на генерирано изображение: 4K*

3.2 Внедряване

В настоящия курсов проект всички тестови случаи бяха проведени в две различни области от множеството на Манделброт. Техните точни координати могат да бъдат видени под съответните изображения, които са получени като резултати от програмата:



Фигура 36: *Резултат от изпълнението на програмата при 1024 максимален брой итерации с 4K размер в област: -0.6386:-0.5986:0.4486:0.4686*



Фигура 37: *Резултат от изпълнението на програмата при 1024 максимален брой итерации с 4K размер в област: -1.5:0.7:-1:1*

3.3 Заключения

Като извод от всички направени тестове, може да се заключи, че статичното циклично балансиране макар и с не толкова голяма преднина, винаги дава по-добри резултати от останалите две реализации на динамично централизирано балансиране.

Друга отличителна особеност в паралелното програмиране, която винаги трябва да се има предвид, е да се внимава с готовите методи и цели библиотеки. Те трябва да бъдат внимателно проучени преди да се използват в един такъв алгоритъм, защото е възможно да не са предвидени за целта, за която искаме да ги използваме. При възможност, винаги остава за предпочитане направата на собствена имплементация като най-сигурен вариант. Една от целите на настоящия курсов проект беше да докаже на практика това твърдение.

Източници

[1] **Ali**, Dia, **Dobson**, Davis, **Gäng**, Isaac, **Gourd**, Jean. *Parallel Implementation and Analysys of Mandelbrot Set Construction*. University of Southern Mississippi, 2008 [Viewed 04.06.2023]. Available from: https://www.academia.edu/1399383/Parallel_Implementation_and_Analysis_of_Mandelbrot_Set_Construction

[2] **Antonio**, Laganá, **Tracolli**, Mirco, **Pacifici**, Leonardo. *Parallel generation of a Mandelbrot set*. University of Perugia, 2016 [Viewed 04.06.2023]. Available from: http://services.chm.unipg.it/ojs/index.php/virtlcomm/article/view/112

[3] **Gamage**, Bhanuka, **Baskaran**, Vishnu. *Efficient Generation of Mandelbrot Set using Message Passing Interface*. Monah University, Malaysia, 2020 [Viewed 04.06.2023]. Available from: https://www.researchgate.net/publication/342655570_Efficient_Generation_of_Mandelbrot_Set_using_Message_Passing_Interface

[4] **Gómez**, Ernesto. *MPI vs OpenMP: A case study on parallel generation of Mandelbrot set*. University of Information Sciences, Havana, 2020 [Viewed 04.06.2023]. Available from: https://www.researchgate.net/publication/344453347_MPI_vs_OpenMP_A_case_study_on_parallel_generation_of_Mandelbrot_set