
**Софийски университет “св. Климент Охридски”**
**Факултет по математика и информатика**

# Курсов проект

**по Разпределени софтуерни архитектури**

На тема: “Изследване на ускорението на теста на Манделброт при използване на различни грануларности, адаптивност към L1d-cache и използването на статично циклично и динамично централизирано балансиране”

Изготвил:
Светлин Попиванов
Софтуерно инженерство
Трети курс, ФН: 62275

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

# Съдържание

# 1.Въведение

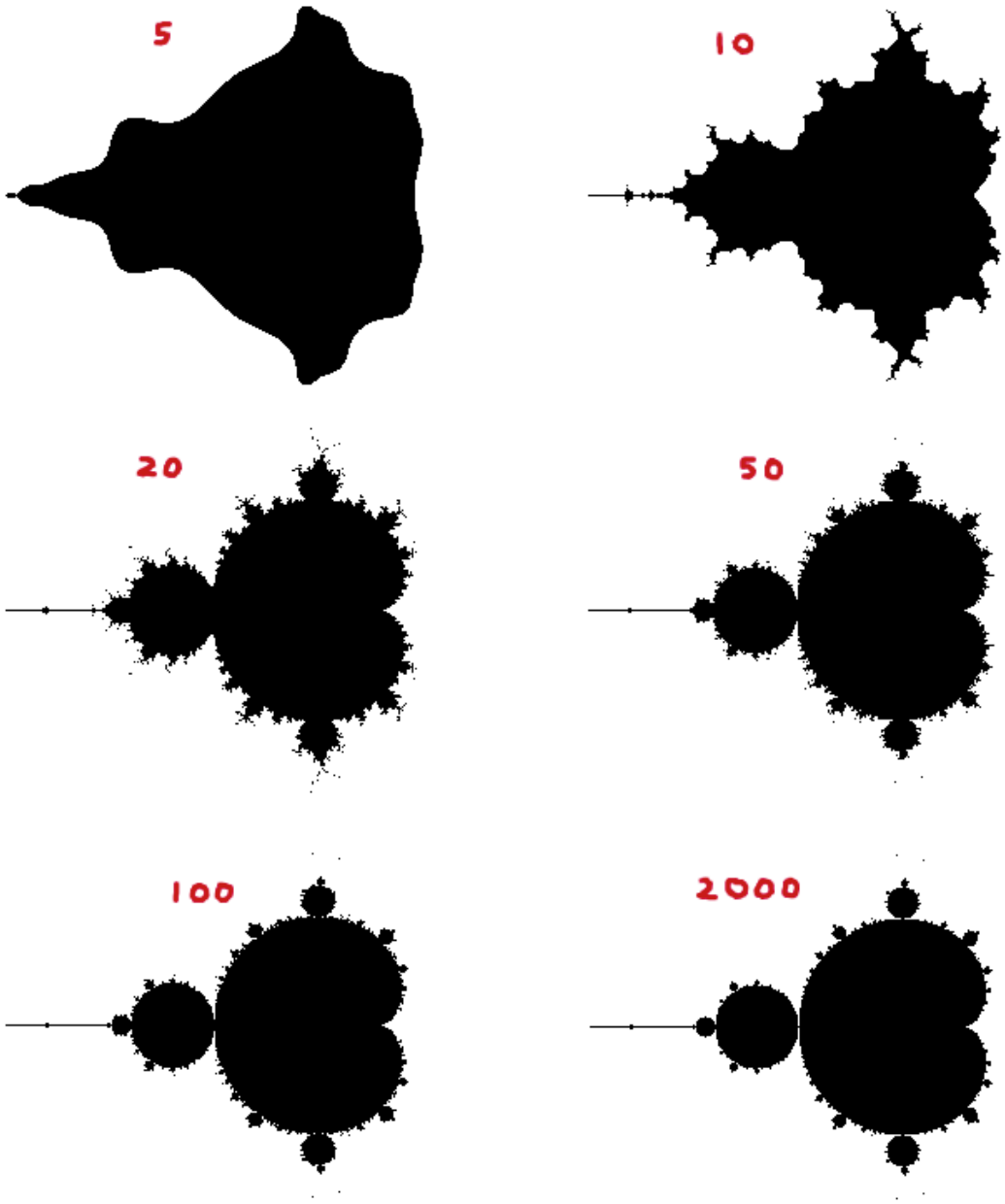
## 1.1. Уводни думи

В рамките на настоящия проект се разглеждат редица въпроси относно паралелизма на една програма. За тази цел, се използва множеството на Манделброт. Той е изключително удобен начин, чрез който да се покаже какво е паралелизъм на една програма и как се постига той. Преди да преминем към анализа на тези въпроси, първо ще се запознаем с това какво всъщност представлява множеството на Манделброт.

## 1.2. Множество на Манделброт

Множеството на Манделброт е фрактал, който се задава с реда $z_{n+1} = z_n^2 + c$, където $z$ и $c$ са комплексни числа. За всяка стойност на $c$ получаваме различни редове от горния вид. Тогава имаме две възможности. За дадено $c$, редът $z_n$ клони към безкрайност или има крайна граница. В случая когато редът клони към безкрайност (тоест съществува число $L$, за което от даден момент натам ще е изпълнено неравенството $z_n > L$), казваме че точката $c$ не принадлежи на множеството на Манделброт. В обратния случай, тоест когато редът има крайна граница, казваме че точката $c$ принадлежи на множеството. Графично, точките от множеството на Манделброт се представят като се оцветят съответните пиксели на екрана с черен цвят. За останалите ни е нужна още една информация.

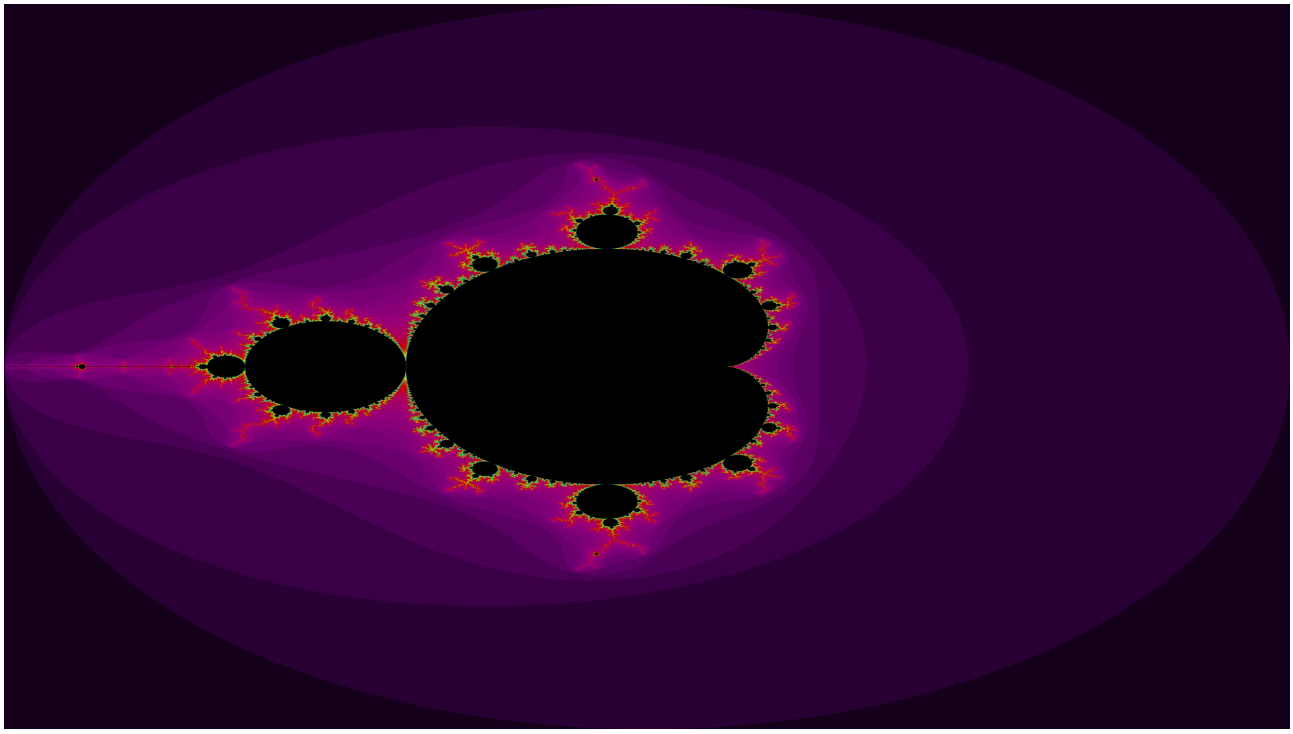
Тъй като тестваме точка за сходимост в безкрайността, програмата, която използваме, ще се върти в безкраен цикъл. Разбира се, това е нежелателно и от тук идва въпросът: Колко максимум итерации са нужни, за да сме сигурни, че $z_n$ клони към безкрайност? 10, 100, 1000? Трябва ни и такова число $L$, за което ако $|z_n| > L$ можем със сигурност да твърдим, че $z_n$ клони към безкрайност. И наистина съществува такова $L$ и то е 2. Това означава, че ако $|z_n| > 2$, то ще е валидно да твърдим, че $z_n$ клони към безкрайност. За по-подробни сведения относно това защо точно 2 е най-подходящо може да се посети източник [4], посочен в края на този проект, откъдето е и взета тази информация. Въпросът с итерациите няма ясен отговор, защото различният брой итерации ни дава различен детайл на множеството на Манделброт. Това е така тъй като самото множество е фрактал, тоест има “безкраен детайл”. Следователно максималният брой итерации зависи от това колко детайлно искаме да е генерираното изображение като резултат от изпълнението на програмата.




**Фигура 1.** Пример от източник [4] за различното ниво на детайлност, отговарящо на съответния максимален брой итерации

Максималният брой итерации помага и в друго отношение. Точките, които не принадлежат на множеството на Манделброт могат да бъдат оцветени в различни цветове, в зависимост от това колко итерации са отнели до спирането на цикленето (с други думи, точката е нарушила неравенството $|z_n| \leq 2$). По този начин те се представят графично върху генерираното изображение.

В рамките на този проект са използвани 1024 максимален брой итерации. Една от причините за този избор е именно постигането на сравнително добро ниво на детайлност на изображението на Манделброт, генерирано от програмата. С този брой итерации се получава следното изображение:



**Фигура 2.** Генерираното от програмата, реализираща статично балансиране, изображение на множеството на Манделброт в границите (−2;2) за реалната ос и (−2;2) за имагинерната ос на комплексната равнина.

# 2. Функционален анализ

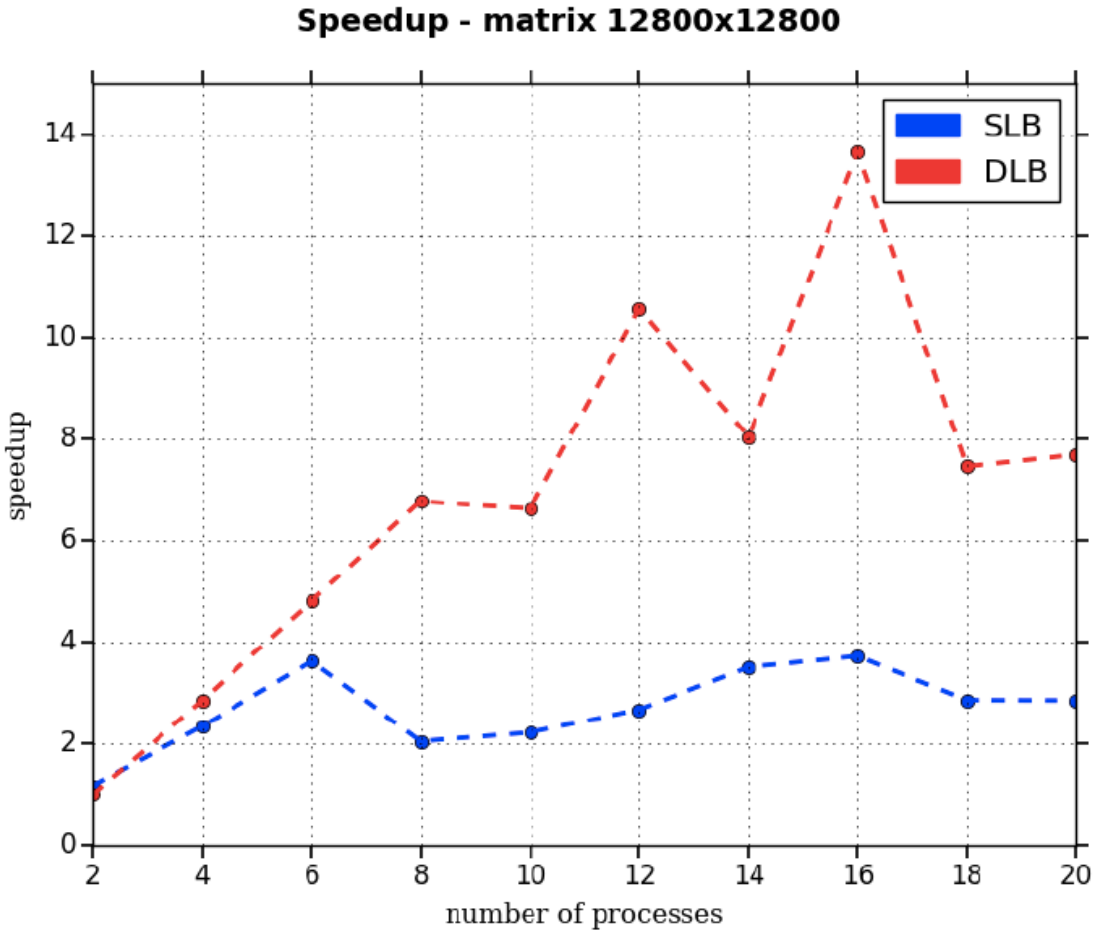
## 2.1. Анализ на първи източник

В рамките на този източник е разгледана подробно разликата между статично циклично и динамично централизирано балансиране. Но са получени резултати, които показват, че статичното се представя значително по-зле от динамичното. В рамките на настоящия проект тази тема също ще бъде разгледана, но получените резултати са коренно различни от тези. Посочена е и сравнително подробна информация за тестовите машини, върху които са изпълнени самите тестове. За повече информация относно машините може да се прегледа точка 3.1.

Статичното балансиране, което е реализирано в рамките на тази статия е циклично и с максимално едра грануларност. Тоест 1 нишка ще изпълни 1 задача (1 кубче с размер PxQ) и нишките последователно ще вземат задачите. Максимално едрата грануларност не е най-оптималната, защото ще има много голям дисбаланс между изчислителните сложности на отделните нишки и това заключение може да се онагледи с резултатие, които са получени, а именно статичното балансиране не надминава 4 при наличието на матрица за изчисление с размерност 12800x12800 дори след паралелност 6 [Фигура 3]. Въпреки това, в програмата си, главната нишката създава останалите нишки, изпраща им задачите и след това изпълнява своята част от областта. Накрая, тя изчаква останалите нишките да приключат. Това е много добро решение особено при наличието на средна или фина (но не много фина) грануларност, тъй като останалите нишките ще имат по-малко на брой задачи, които да изпълнят, и изчислителните им сложности ще се уравновесят. За получените резултати е използвана декомпозиция по

блокове, тоест подматрици от първоначалната матрица, обхващаща цялата изследвана област. Тъй като грануларността на решението е 1, то тези подматрици напълно ще разбиват цялата подматрица (ще има толкова подматрици, колкото са нишките).

При динамичното централизирано балансиране са използвани различни грануларности и моделът Master-Slave чрез обмен на съобщения (message passing). За получените резултати, представени на Фигура 3 по-долу е използвана средна грануларност. Използването на моделът Master-Slave означава, че master нишката стартира slave нишките, разпределя първоначалните задачи, след което следи за завършване на някоя от нишките, за да й даде нова задача и така до изчерпване на всички задачи. В това отношение master нишката няма своя задача за изпълнение, нейната работа е да следи останалите slave нишки и да разпределя задачите. Задачите отново са разделени на кубчета. Получените резултати се онагледяват чрез следната графика:




**Фигура 3.** Резултати от програмата с използването на статично (в синьо) и динамично (в червено) балансиране, чийто пускания са върху платформа B.

На графиката виждаме много немонотонни аномалии, които са нежелателни, когато изследваме даден алгоритъм за паралелност. Те могат да се дължат на няколко причини. Една от тях е фоновото натоварване. Не става ясно от статията дали такова е имало, но променливите резултати, които правят

правата на ускорението нелинейна могат да бъдат обяснени с наличието му. Друга причина (това се отнася за статичното балансиране) е едрата грануларност. Използването ѝ в този случай е неоптимално, тъй като изчислителните сложности на заданията на всяка нишка ще се различават една от друга с немалък коефициент. Затова за предпочитане е използване на средна грануларност или достатъчно фина грануларност, която да балансира изчислителните сложности на задачите на отделните нишки.

**Изводи:** Статията предоставя една много добра обща представа за това какво трябва да се направи, за да се паралелизира алгоритъма за генериране на множеството на Манделброт. Въпреки това, някои решения относно проектирането на самия паралелен алгоритъм, предприети в рамките на статията, не са оптимални и не са за предпочитане. От тази статия могат да се вземат заключения за това колко важен е изборът на такава грануларност, която да балансира до максимална степен изчислителната сложност на различните задачи, които всяка нишка взима. Целта е те максимално да се балансират. От друга страна, решението главната нишката да изпълнява задания е добро, тъй като тя също ще изпълнява задачи вместо просто да разпределя задачите и да чака да свършат другите нишките. Това решение ще е валидно, когато проектираме нашата паралелна програма със статично балансиране, защото то няма как да се приложи при динамичното централизирано решение.

## 2.2. Анализ на втори източник

Този източник е много по-обстоен от този, който разгледахме горе. Първо се преминава през обяснение на теста на Манделброт, след което се представя подробен анализ защо този тест може да се паралелизира и накрая преди преминаването към разглеждане на това как са извършили балансирането на задачите, се показва теоретичното ускорение, изчислено с помощта на закона на Амдал (Amdahl’s law). Тук е използван MPI интерфейса за осъществяване на комуникацията между отделните нишки, но се споменава, че използваният език за програмиране е C. Тоест още в началото, можем да си изградим представа за това какво да очакваме. Именно самият факт, че реализацията на представените алгоритми са били написани на C, говори, че очакваме сравнително добро ускорение, тъй като езикът е на достатъчно ниско ниво, чрез което по-лесно да се управлява паметта (и по-конкретно употребата на кешовете, особено L1-d кеша).

В тази статия са представени 3 реализации на теста на Манделброт, като нито една от тях не е специфицирана като използваща статично или динамично балансиране. Въпреки това и трите реализации са подробно обяснени и балансирането може да се определи без особена трудност. Трябва да се отбележи, че и трите реализации използват обмен на съобщения, чиито проблеми ще разгледаме малко по-надолу. Също така, генерирането на изображението се извършва в главната нишка чак след приключването на последната нишка, което си има своите предимства пред директно писане във файл директно от работещите нишки. Това се дължи на факта, че операцията по писане във файл е непредсказуема поради това, че не знаем вътрешната реализация на самата операция. Тоест ние не можем да

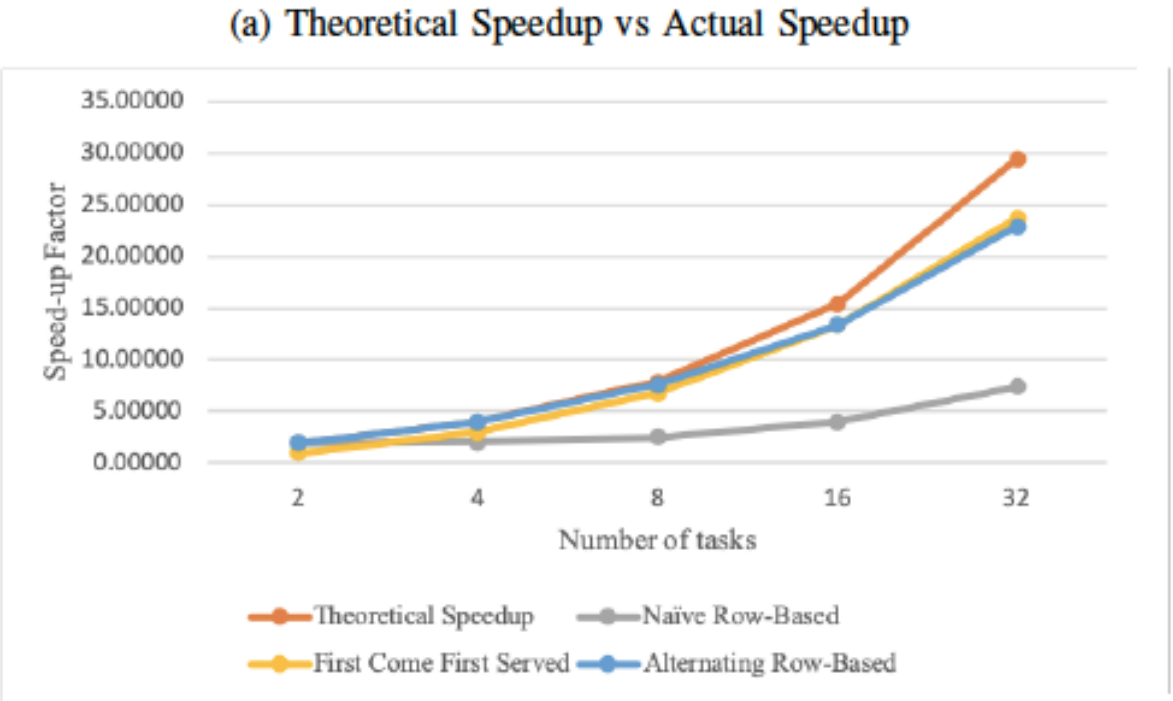
планираме времето за изпълнение, което не е добър подход при проектирането на паралелни алгоритми.

Първата реализация (наречена Naive: Row Based Partition Scheme) използва статично циклично балансиране. Цялото изображение се разделя на P реда, където P е броят нишки. Това, разбира се, означава, че в тази реализация е използвана най-едрата грануларност g = 1, която както видяхме от предния източник не е добър подход за постигането на добра паралелност на нашия алгоритъм. Главната нишката стартира всички нишки и започва работа по своята част, след което изчаква останалите нишки да приключат своята работа. Те от своя страна изпращат масив със своите изчисления, който бива копиран от главната нишка в своя масив (той ще се използва за генериране на изображението на множеството на Манделброт).

Втората реализация (наречена First Come First Served : Row Based Partition Scheme) използва динамично централизирано балансиране, тоест Master-Slave модела. Не става ясно грануларността каква е, но от картинката, представена за тази реализация може да се заключи, че използваната грануларност със сиурност не е най-едрата. Тук master нишката играе ролята на сървър, който обслужва клиенти (клиентите в този случай са аналог на slave нишките). Той първоначално разпределя задачите, след което чака някоя нишка да завърши своята работа, за да може да й възложи работа. Нужно е да се отбележи, че когато дадена нишка свърши своята работа, тя предава своя масив с извършените изчисления на master нишката, която го копира в своя. Цялата тази комуникация се осъществява посредством обмен на съобщения.

Третата реализация (наречена Alternating: Row Based Partition Scheme) използва статично циклично балансиране. Тази реализация се гради върху основите на първата, но този път с по-голяма гранулярност с цел да се балансира работата на всяка отделна нишка. Тук проблем, който се появява, е употребата на обмен на съобщения, тъй като това създава bottleneck при наличието на голям брой процеси. Това се дължи на реализацията, в която след като дадена нишка приключи своята работа, тя изпраща своя резултат на главната нишката и тя го копира в своя масив. Тоест тук, когато работата между отделните нишки е балансирана, се получава така, че по сравнително еднакво време главната нишката получава накуп всичките резултати. Това е рецепта за забавяне, тъй като тя ще трябва да копира дълго време. Този проблем не се появява в първата реализация, защото там работата не бе балансирана. Ето защо, употребата на обмен на съобщения при статично циклично балансиране не е добра идея, защото е предпоставка за bottleneck, който значително наранява нашата крива на ускорението при големи стойности на броя нишки.

За осъществяването на тестовете, авторите са използвали две машини поради това, че на всяка една от тях хипертрединга е бил изключен. Това обяснява и използването на обмена на съобщения в рамките на своите 3 реализации. Двете машини разполагат с 36 логически ядра, но поради липсата на хипертрединг те реално имат 18, което обяснява и решението да се използват две машини. Тук изрично е казана липсата на фоново натоварване на двете машини.




**Фигура 4.** Резултати от пускането на горните три реализации върху гореспоменатите машини.

**Изводи:** Както виждаме от графиката, употребата на статично циклично балансиране с по-фина грануларност е доста добро, въпреки наличието на bottleneck след 16 нишки, който се отразява върху ефективността на програмата, в следствие от употребата на обмен на съобщения. Тази програма може да бъде подобрена, като се премахне обменът на съобщения и се вгради решение, в което всяка нишка пише в ресурс, намиращ се в главната нишка. То няма да пречи на цялостното време за работа, защото, както в началото на статията е показано, никой вход на дадена нишка не зависи от друг вход или изход на друга нишка, а същото важи и за изхода на тази нишка. Това означава, че всяка нишка ще работи върху своя собствена част от общия ресурс и тя няма да се “бие” с други за него. Друг извод е разликата между най-едрата грануларност и достатъчно фина грануларност, която да балансира работата на отделните нишки. Поради тази причина трябва да се избягва едрата грануларност и да има стремеж към средна или фина (но не прекалено фина) грануларност.

## 2.3. Анализ на трети източник

Този източник дава насоки за реализацията на Mandelbrot на езика Java. В книгата също така, подробно и последователно са разгледани теми като балансиране на товара (load balancing) и как се осъществява като код на езика Java.

Друга полезна информация е това, че библиотеката Graphics2D, която в този проект е използвана за генериране на изображението на множеството на Манделброт няма добра производителност. Това означава, че самият процес на генериране на изображението трябва да бъде отделен от паралелната обработка. Това е причината, поради която генерирането на

изображението на множеството на Манделброт в съответната област се извършва чак след като всички нишки приключат своята работа.

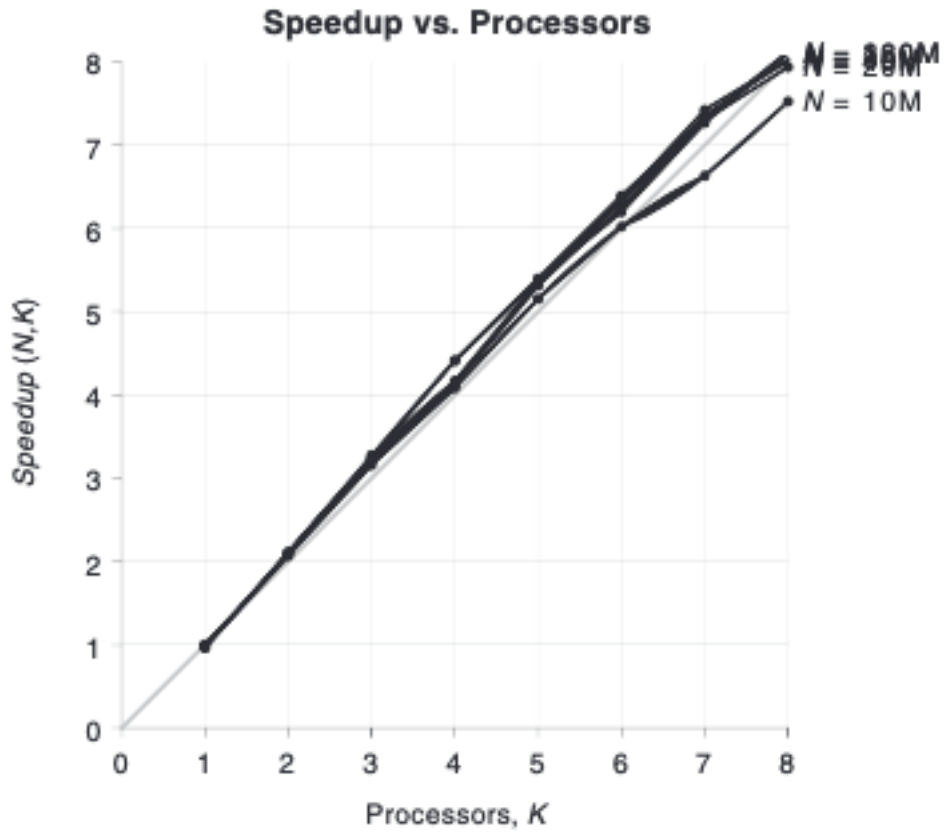
В книгата са посочени 3 различни решения, които се целят да балансират изчислителните сложности на нишките. И в трите реализации е използван параметърът N – общият броят итерации. Балансирането се осъществява на ниво for цикъл.

Първата реализация авторът нарича fixed schedule. При нея общият брой итерации N се разделя на K на брой равни участъци, които след това се поделят измежду наличните нишки, които са K на брой. След това, когато всяка нишка извиква своят run() метод, тя го извиква със съответното начало и край, по които for цикъла да обхожда. Тоест тук заданията се определят от това от коя итерация се започва и до коя се завършва. По това отношение, представената реализация използва максимално едра грануларност и статично циклично балансиране, макар и в по-необикновена форма.

Втората реализация авторът нарича dynamic schedule. При нея общият брой итерации N се разделя на число k, което може да се специфицира през командния ред. Това е размера на участъка, което по подразбиране е 1, тоест 1 итерация. Тук, master нишката дава последователно задания на останалите и след като дадена slave нишка приключи, тя й дава следващата за изпълнение задача. Тоест, това е динамично централизирано балансиране с използването на променлива грануларност в зависимост от размера на участъците.

Третата реализация, която е използвана за получаването на графиката на ускорението, показана по-долу, авторът нарича guided schedule. При нея общият брой итерации N се разделя на участъци, които първоначално са големи, но последователно стават все по-малки и по-малки. Тук може да се специфицира колко да е минимиалния размер на тези участъци. Алгоритъмът, по който се определя колко голямо да е следващото задание е следният: размерът на заданието е половината от оставащите итерации, разделен на броят нишки. Например, ако разполагаме с 1 нишка, 100 итерации и минимален размер 1, размерите ще са 100/2/1 = 50, (100-50)/2/1 = 25, 12, 6, 3, 1, 1. Колкото повече са нишките, толкова повече участъци има и толкова по-малки са те. Тоест това решение отново използва динамично балансиране с променлива грануларност (най-вече фина).

Резултатите, получени при използването на 10, 20 и 320 милиона итерации и третата реализация се онагледяват със следната фигура:




**Фигура 5.** Графика на ускорението, получено в източник [3]

**Изводи:** Въпреки, че реализацията на такива видове балансирания е труден процес, могат да се вземат поуки от техните резултати. Най-важната, от които е използването на достатъчно фина грануларност, с която да се балансират различните подзадания на нишките. Това е извод, който вече изведохме и от останалите източници, тоест този параметър на паралелността е от голямо значение за постигането на добри резултати.

## 2.4. Сравнителна таблица

| Образец | Максимален паралелизъм | Декомпозиция | Тип балансиране | Грануларност | Максимален брой итерации | Размер на изображението |
|---|---|---|---|---|---|---|
| **[1]** | 20 | По блокове | Статично циклично и динамично централизирано | Максимално едра и средна | 10000 | От 100x100 px до 12800x12800 px |

| Образец | Максимален паралелизъм | Декомпозиция | Тип балансиране | Грануларност | Максимален брой итерации | Размер на изображението |
|---|---|---|---|---|---|---|
| **[2]** | 32 | По редове, по блокове от редове | Статично циклично и динамично централизирано | Максимално едра и фина | 2000 | 8000x8000 px |
| **[3]** | 8 | По блокове | Статично циклично и динамично централизирано | Фина | 1000 | От 3200x3200 px |
| **Mandelbrot.java** | 16 | По редове и по колони | Статично циклично | Едра и средна | 1024 | 3840x2160 px |
| **DynamicMandelbrot.java** | 16 | По редове | Динамично централизирано | Едра и средна | 1024 | 3840x2160px |

# 3. Технологичен анализ

Нека сега представим характеристиките на тестовите среди, които са били използвани за постигането на резултатите, показани в точка 2.

## 3.1. Първи източник

Платформа A – Клъстер от 11 Intel dualcore 32 bit процесора с 2 GiB RAM и i686 архитектура

Платформа B – Клъстер от 13 Intel quadcore 64 bit процесора с 2 GiB RAM и x86_64 архитектура

За съжаление, тъй като не са специфицирани точните Intel чипове, няма как да се извлече информация за кешовете. Тази информация липсва и в самият източник.

## 3.2. Втори източник

Intel Xeon Gold 6150 процесор (36 логически ядра, тоест 18 физически)

L1d кеш – 576 KiB
L2 кеш – 18 MiB
L3 кеш – 24.75 MiB

## 3.3. Трети източник

Sun Microsystems eight-processor SMP parallel computer with four UltraSPARC-IV dual-core CPU chips

L1d кеш – 64 KiB
L2 кеш – 16 MiB

# 4. Проектиране на реализацията със статично балансиране

## 4.1. Функционално проектиране

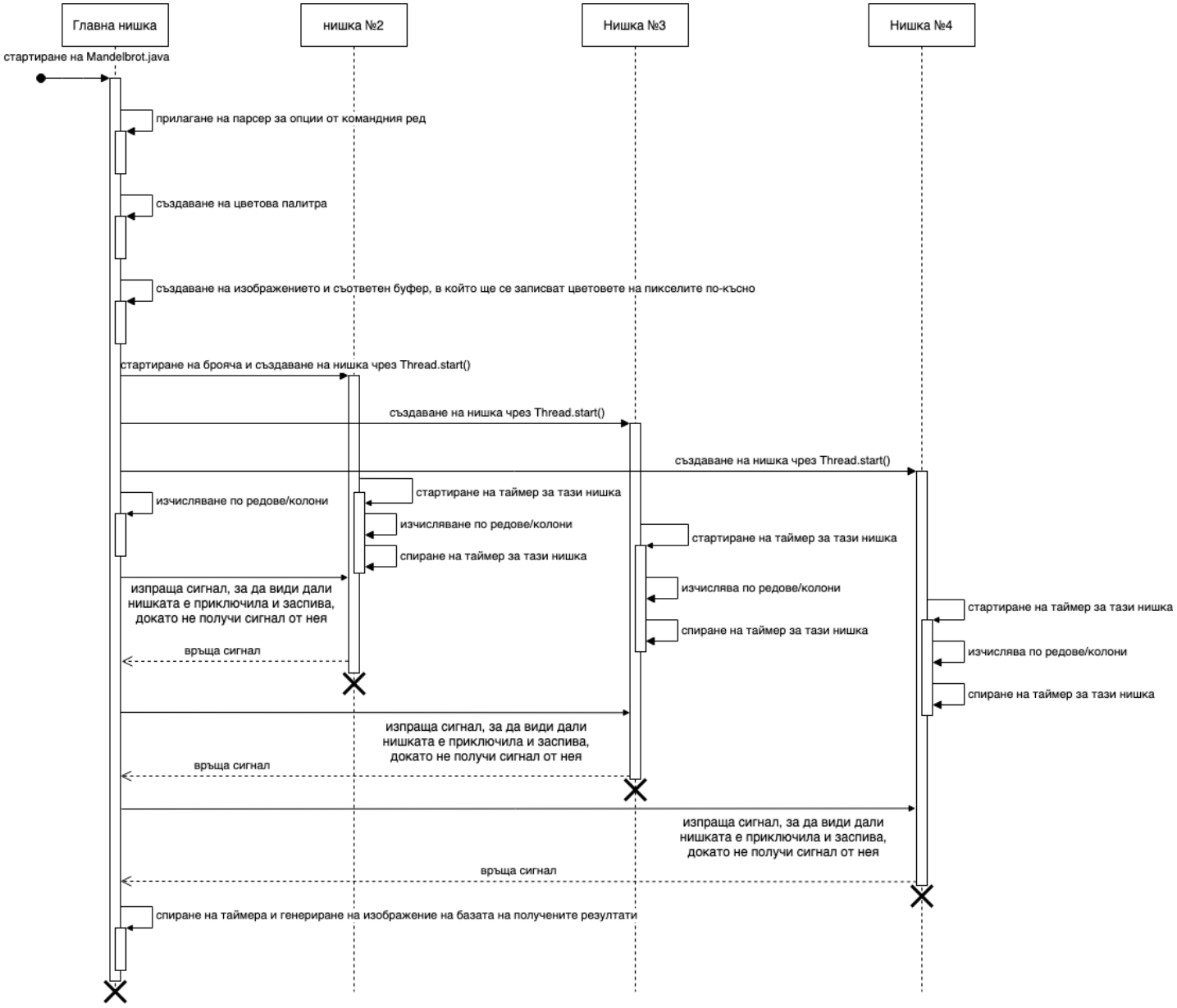
### 4.1.1. Модел на приложението

Тъй като вече обсъдихме горе, че различните нишки в паралелната обработка на изображението на множеството на Манделброт няма да си пречат една на друга, това означава, че тук ще можем да декомпозираме по данни. Тоест, можем да разделим целият масив, представляващ изследваната област, на няколко парчета в зависимост от използваната грануларност, които ще раздаваме на отделните нишки за обработка. Според Таксономията на Флин, декомпозицията по данни се класифицира като SPMD (Single Program Multiple Data) модел. Ето защо, моделът на приложението е SPMD. Главната нишка стартира останалите и започва работа по своята част, след което чака всички нишки да завършат със своите задачи. Тук всички нишки имат еднаква работа за извършване. Причината за този избор на модел на приложение са две. Първата причина е че всяка нишка трябва да провери дадена точка за това дали принадлежи в множеството на Манделброт, тоест всяка нишка ще изпълнява асинхронно един и същи тест върху различни данни. От тук следва, че ще имаме някакво множество от данни, върху което трябва да му се приложи една и съща “програма”, което обяснява причината за избора на SPMD. Втората причина е че можем да използваме главната нишката с цел да облекчим работата на останалите нишки. Това можем да го направим без някакви усложнения, поради самият факт, че тук разполагаме с асинхронни процеси. Също така, тъй като създаването на нишки в Java е тромав процес, спестяването на един такъв тромав процес ще е от полза за ускорението на програмата.

В рамките на проекта са реализирани два типа балансирания: статично циклично и динамично централизирано, като статичното е осъществено с две декомпозиции: по редове и по колони. Целта на двете декомпозиции е да провери влиянието на кешовите операции, които се случват без ние да знаем и да се сравни коя реализация постига по-добри резултати. Тази реализация след това ще бъде сравнена с динамичното централизирано балансиране, за да се оцени и свръхтовара, който централизираното решение носи със себе си.

### 4.1.2. Последователностна диаграма на процесите

Нека сега разгледаме една последователностна диаграма на процесите, с която да онагледим какво се случва в самата програма на процесорно ниво.




**Фигура 6.** Диаграма на последователностите на Mandelbrot.java (статично балансиране) при паралелност 4

На горната диаграма се проследяват събитията от пускането на програмата от командния ред до нейното завършване. Трябва да се отбележи, че изборът на 4 нишки е с цел да не се усложнят означенията върху диаграмата ненужно. Това се дължи на факта, че всяка друга нишка освен главната нишка извършва една и съща работа, а именно да изчисли своята част от изображението. Диаграмата за n процеса ще изглежда по един и същи начин, но с повече колони, отговарящи на всяка нова нишка. Също така, трябва да се забележи, че последователността на изчисление е частен случай. Тъй като всичко зависи от ядрото на операционната система по отношение на това кога ще се създаде една нишка, възможно е нишка №2 да се стартира преди нишка №1 и т.н. Възможно е също така, нишката, до която се допитва главната нишка, отдавна да е завършила работа. В този случай главната нишката не се приспива, а продължава с допитване до следващата по поредност нишка.

### 4.1.3. Описание на командните параметри

Командните параметри, реализирани в приложението са няколко. Информация за тях може да бъде намерена и от командния ред посредством опцията h (help):

| Опция | Описание |
|---|---|
| **r, rect** | Чрез нея се задават границите на областта от комплексната равнина, в която ще търсим да визуализираме множеството на Манделброт. Например: -r (-rect) -1.0:1.0:0.5:2.0, което се интерпретира като: $a \in [-1.0; 1.0], b \in [0.5; 2.0]$, където $a, b \in R$ и $z = a + i * b$. По подразбиране областта е $a \in [-2.0; 2.0], b \in [-2.0; 2.0]$. |
| **t, tasks** | Броят на нишките, които ще трябва да работят паралелно. По подразбиране, програмата стартира с 1 нишка. |
| **o, output** | Задава се името на изображението, което ще се генерира. По подразбиране, това е “Mandelbrot.png”. |
| **q, quiet** | Когато този команден параметър бъде добавен, програмата минава в тих режим, в който изписва на командния ред само и единствено общото време за изпълнение. |
| **s, size** | Размерът на изображението, което ще се генерира. По подразбиране, това е 3840x2160 (4K). |
| **h, help** | Когато този команден параметър бъде добавен, програмата извежда списък с наличните команди и тяхното значение на командния ред. След това програмата прекратява своето изпълнение. |
| **g, granularity** | С този параметър може да се посочи грануларността (или броят задания, които всяка нишка трябва да изпълни). По подразбиране, това е най-едрата грануларност $g = 1$. |
| **b, by** | С този параметър се задава моделът на декомпозиция или как трябва нишките да обхождат изображението (по редове или по колони). По подразбиране, това е по редове. Възможни стойности: rows/cols. |

Когато програмата не се намира в тих режим, тя извежда на командния ред съобщения от следния вид:

**Thread-<index> started.**
**Thread-<index> finished. Execution time: <exec time> ms.**
**Total execution time: <exec time> ms.**

В тих режим само последното съобщение се извежда на командния ред.

## 4.2. Технологично проектиране

### 4.2.1. Тестова среда

Следната машина бе използвана за тестването на програмата:

**t5600.rmi.yaht.net**

```
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                32 (16 физически)
On-line CPU(s) list:   0-31
Thread(s) per core:    2
Core(s) per socket:    8
Socket(s):             2
NUMA node(s):          2
Model name:            Intel(R) Xeon(R) CPU E5-2660 0 @ 2.20GHz
L1d cache:             32K
L1i cache:             32K
L2 cache:              256K
L3 cache:              20480K
```

### 4.2.2. Използван език и външни библиотеки

За реализирането на програмата е използван езикът Java, като са използвани две външни библиотеки.

За осъществяването на репрезентация на комплексните числа и коректни операции между тях е използвана библиотеката commons-math3-3.6.1.jar (The Apache Commons Mathematics Library).

За парсването на аргументите от командния ред е използвана библиотеката commons-cli-1.4.jar (Apache Commons CLI).

### 4.2.3. Пример за стартиране на програмата

Пример за това как се стартира програмата:

```
./runMandelbrot.sh -t 12 -g 16 -by cols
```

С този ред се извиква shell скрипта runMandelbrot.sh, който извиква програмата с параметри посочените командни параметри t, g и by. По този начин програмата ще се изпълни посредством употребата на 12 нишки, грануларността ще е 16 (тоест всички задания ще са 16*12 = 192) и ще се извърши статично циклично балансиране по колони.

### 4.2.4. Стартиране и използване на нишките

Последователно създаваме нишките, като им даваме идентификационния номер (id) и булева променлива, съответстваща на това дали дадената нишка трябва да изведе съобщения за това кога е започнала и кога е приключила

(дали програмата се намира в тих режим или не). Идентификационният номер освен че уникално определя всяка нишка, се използва и за средство, чрез което всяка нишка да се ориентира коя част от изображението трябва да изчисли. След като създадем дадена нишка, тя се стартира посредством командата t.start(), извикана от главната нишка. След като главната нишка създаде (numThreads-1) нишки (където numThreads е броят на нишките, зададен от командния ред), тя процедира към изчисляването на своята част от изображението.

```
workers = new Thread[numThreads];
for (int i = 1; i < numThreads; i++) {
      Runnable r = new Runnable(i, isQuiet);
      Thread t = new Thread(r);
      t.start();
      workers[i] = t;
}

new Runnable(0, isQuiet).run();
```

След като свърши със своята работа, главната нишка изчаква останалите да свършат последователно от нишка с идентификационен номер 1 до нишка с идентификационен номер (numThreads – 1). Това се осъществява итеративно, като главната нишка първо изпраща сигнал до нишка номер 1 и ако тя още не е приключила, се приспива, докато не получи сигнал, че е приключила. След това тази процедура се повтаря и за нишка номер 2 и така до нишка номер (numThreads – 1).

```
for (int i = 1; i < numThreads; i++) {
      try {
            workers[i].join();
      } catch (Exception e) {
            System.out.println(“Unexpected exception: “ +
e.getMessage());
      }
}
```

## 5. Резултати от тестване на реализацията със статично балансиране

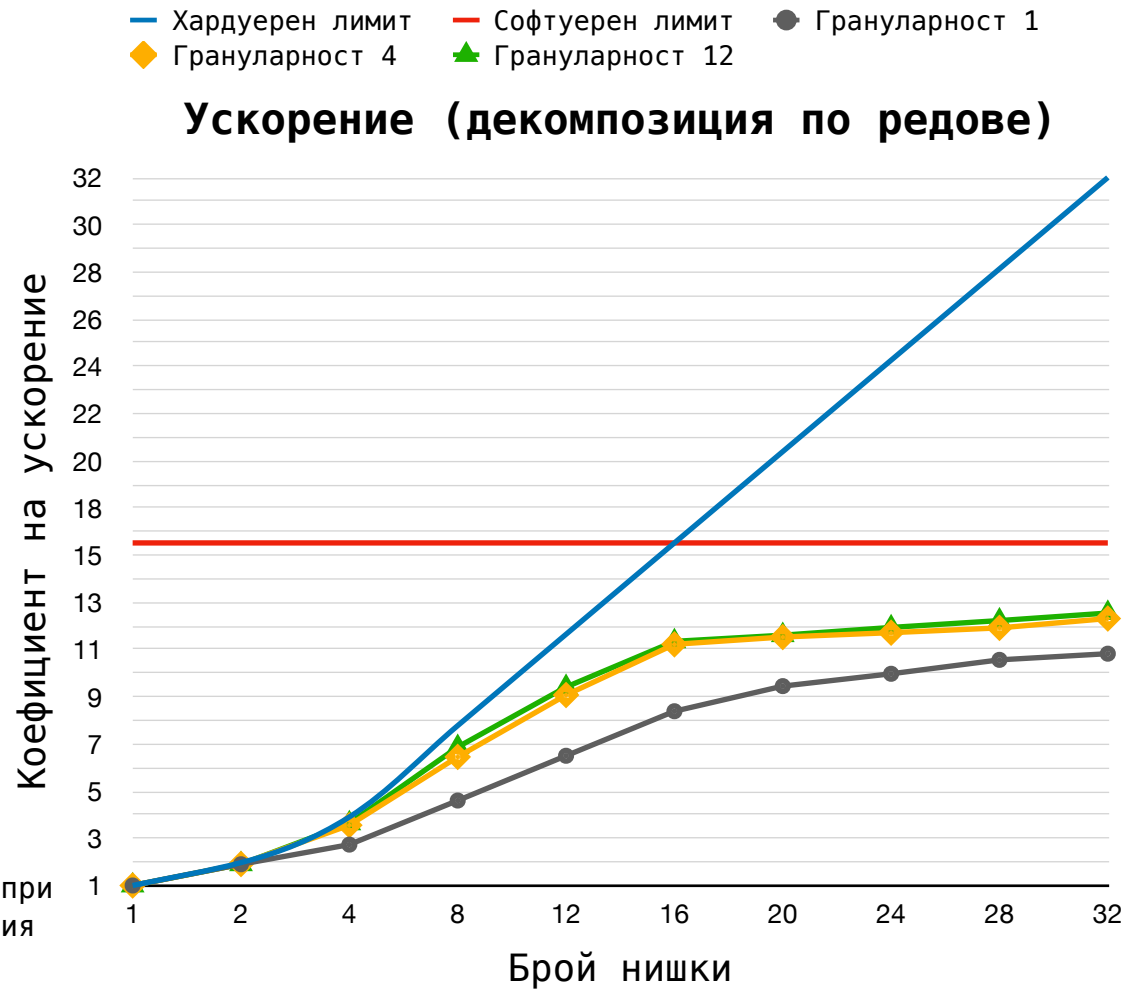
### 5.1. Статично циклично балансиране с декомпозиция по редове

В таблицата по-долу използваме следните означения:
$p$ – броят на нишките,
$g$ – грануларността,
$T_p^{(n)}$ – време за изпълнение при p нишки, получено от n-тия тест

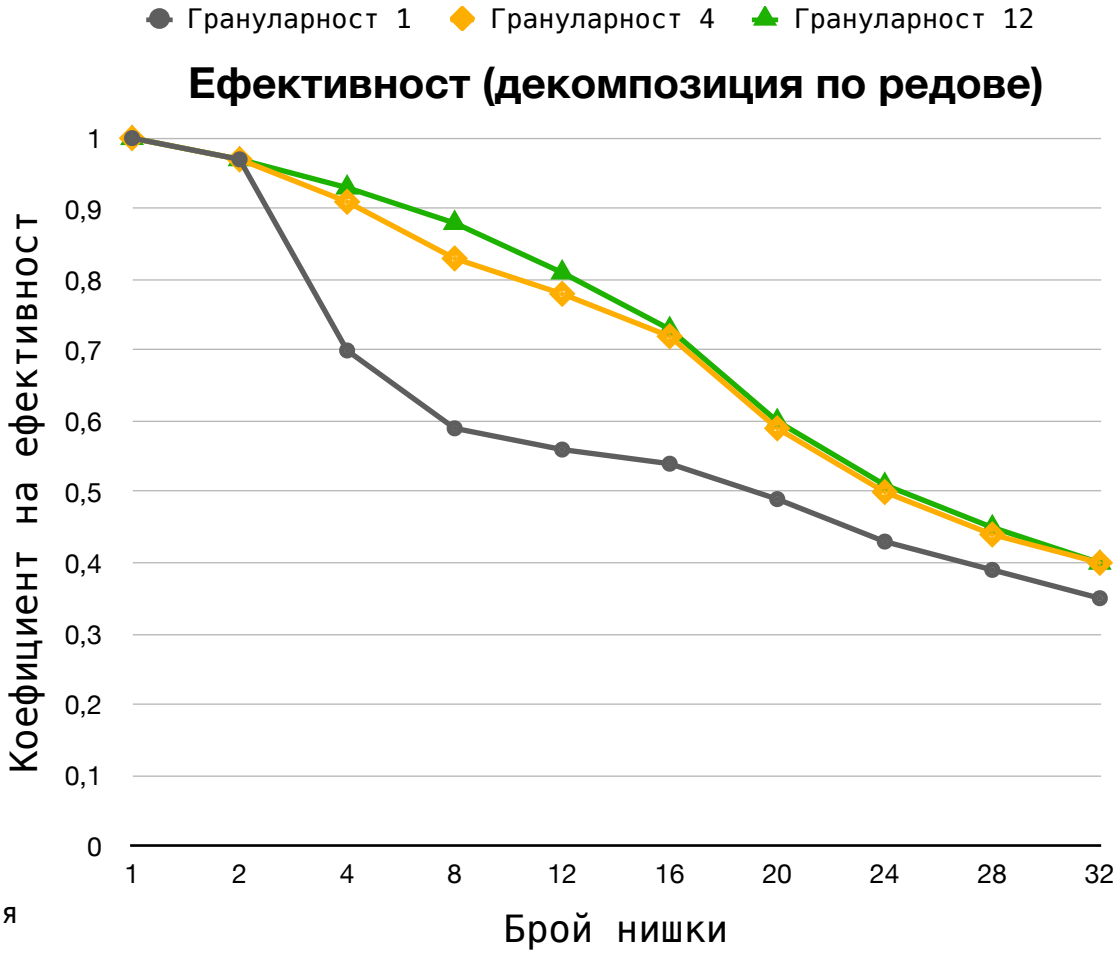
$S_p = T_1/T_p$ – ускорение
$E_p = S_p/p$ – ефективност

Единицата за време, която е използвана в таблицата е милисекунди.
Областта в комплексната равнина, в която са постигнати тези резултати е: $a \in [-1{,}67; 1]$ и $b \in [-0{,}75; 0{,}75]$, където $z = a + ib$.

| # | p | g | $T_1^{(1)}$ | $T_1^{(2)}$ | $T_1^{(3)}$ | $T_1$=min() | $T_p^{(1)}$ | $T_p^{(2)}$ | $T_p^{(3)}$ | $T_p$ = min() | $S_p = T_1/T_p$ | $E_p = S_p/p$ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 1 | 70895 | 69751 | 69411 | **69411** | 35913 | 36497 | 36226 | **35913** | **1,93** | **0,97** |
| 2 | 4 | 1 | | | | | 25401 | 24905 | 25218 | **24905** | **2,79** | **0,70** |
| 3 | 8 | 1 | | | | | 14722 | 14752 | 14700 | **14700** | **4,72** | **0,59** |
| 4 | 12 | 1 | | | | | 10392 | 10410 | 10557 | **10392** | **6,68** | **0,56** |
| 5 | 16 | 1 | | | | | 8251 | 8046 | 8097 | **8046** | **8,63** | **0,54** |
| 6 | 20 | 1 | | | | | 7147 | 7134 | 7284 | **7134** | **9,73** | **0,49** |
| 7 | 24 | 1 | | | | | 6785 | 6832 | 6757 | **6757** | **10,27** | **0,43** |
| 8 | 28 | 1 | | | | | 6421 | 6379 | 6415 | **6379** | **10,88** | **0,39** |
| 9 | 32 | 1 | | | | | 6227 | 6248 | 6256 | **6227** | **11,15** | **0,35** |
| 10 | 2 | 4 | 69542 | 70564 | 69701 | **69542** | 35873 | 36108 | 36243 | **35873** | **1,94** | **0,97** |
| 11 | 4 | 4 | | | | | 19436 | 19213 | 19130 | **19130** | **3,64** | **0,91** |
| 12 | 8 | 4 | | | | | 10721 | 10483 | 10594 | **10483** | **6,63** | **0,83** |
| 13 | 12 | 4 | | | | | 7768 | 7447 | 7762 | **7447** | **9,34** | **0,78** |
| 14 | 16 | 4 | | | | | 6016 | 6449 | 6256 | **6016** | **11,56** | **0,72** |
| 15 | 20 | 4 | | | | | 6116 | 6037 | 5855 | **5855** | **11,88** | **0,59** |
| 16 | 24 | 4 | | | | | 5956 | 5762 | 5794 | **5762** | **12,07** | **0,50** |
| 17 | 28 | 4 | | | | | 5729 | 5740 | 5662 | **5662** | **12,28** | **0,44** |
| 18 | 32 | 4 | | | | | 5607 | 5650 | 5482 | **5482** | **12,69** | **0,40** |
| 19 | 2 | 12 | 69765 | 69472 | 69915 | **69472** | 36477 | 36640 | 35915 | **35915** | **1,93** | **0,97** |
| 20 | 4 | 12 | | | | | 19341 | 18658 | 19587 | **18658** | **3,72** | **0,93** |
| 21 | 8 | 12 | | | | | 9984 | 10260 | 9838 | **9838** | **7,06** | **0,88** |
| 22 | 12 | 12 | | | | | 7275 | 7172 | 7190 | **7172** | **9,69** | **0,81** |
| 23 | 16 | 12 | | | | | 6146 | 5996 | 5942 | **5942** | **11,69** | **0,73** |
| 24 | 20 | 12 | | | | | 5921 | 5811 | 6022 | **5811** | **11,96** | **0,60** |
| 25 | 24 | 12 | | | | | 5768 | 5774 | 5648 | **5648** | **12,30** | **0,51** |
| 26 | 28 | 12 | | | | | 5591 | 5513 | 5556 | **5513** | **12,60** | **0,45** |
| 27 | 32 | 12 | | | | | 5544 | 5374 | 5471 | **5374** | **12,93** | **0,40** |




**Фигура 7.** Ускорение при декомпозиция по редове

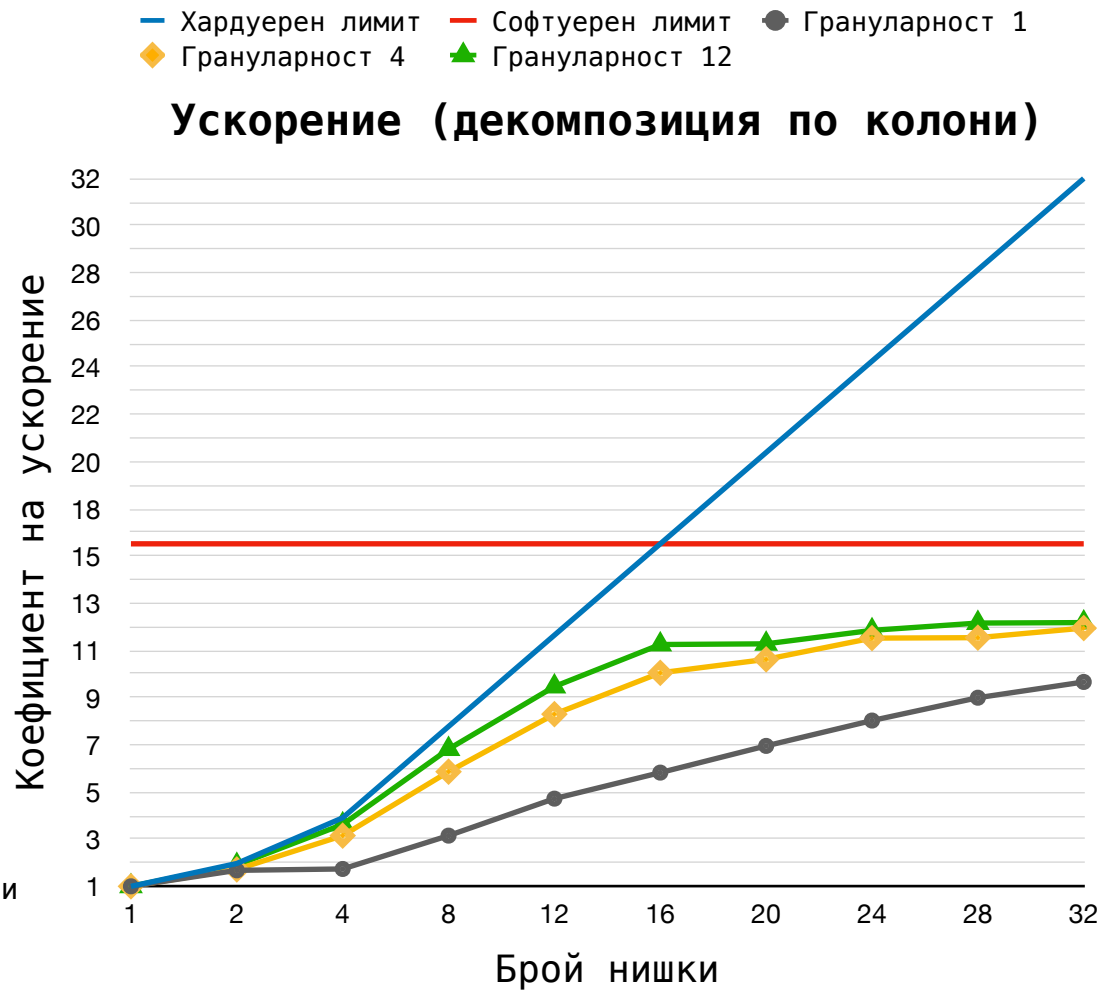



**Фигура 8.** Ефективност при декомпозиция по редове

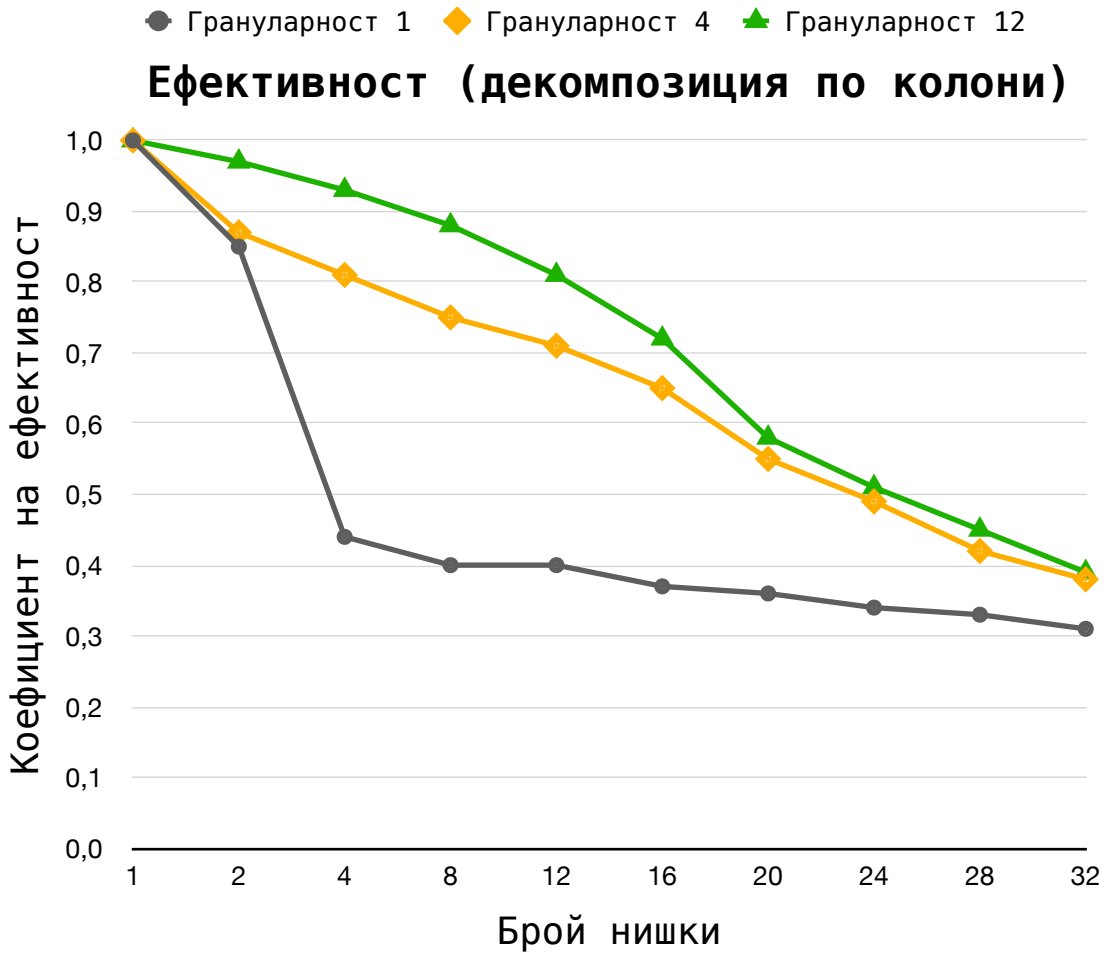
## 5.2. Статично циклично балансиране с декомпозиция по колони

Същите означения са използвани и в тази таблица, като тестовете отново са изпълнени в същата област. Единицата за време също е милисекунди.

| # | p | g | $T_1$ (1) | $T_1$ (2) | $T_1$ (3) | $T_1$=min() | $T_p$(1) | $T_p$(2) | $T_p$(3) | $T_p$ = min() | $S_p = T_1/T_p$ | $E_p = S_p/p$ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 1 | 68935 | 68508 | 68898 | **68508** | 40230 | 41170 | 41268 | **40230** | **1,70** | **0,85** |
| 2 | 4 | 1 | | | | | 38828 | 39654 | 39262 | **38828** | **1,76** | **0,44** |
| 3 | 8 | 1 | | | | | 21696 | 21246 | 21691 | **21246** | **3,22** | **0,40** |
| 4 | 12 | 1 | | | | | 14427 | 14159 | 14403 | **14159** | **4,84** | **0,40** |
| 5 | 16 | 1 | | | | | 11454 | 11523 | 11726 | **11454** | **5,98** | **0,37** |
| 6 | 20 | 1 | | | | | 9820 | 9576 | 9878 | **9576** | **7,15** | **0,36** |
| 7 | 24 | 1 | | | | | 8335 | 8450 | 8292 | **8292** | **8,26** | **0,34** |
| 8 | 28 | 1 | | | | | 7559 | 7398 | 7608 | **7398** | **9,26** | **0,33** |
| 9 | 32 | 1 | | | | | 7100 | 7166 | 6887 | **6887** | **9,95** | **0,31** |
| 10 | 2 | 4 | 72770 | 68792 | 68686 | **68686** | 39690 | 39516 | 40422 | **39516** | **1,74** | **0,87** |
| 11 | 4 | 4 | | | | | 22357 | 21308 | 22377 | **21308** | **3,22** | **0,81** |
| 12 | 8 | 4 | | | | | 11542 | 11411 | 11435 | **11411** | **6,02** | **0,75** |
| 13 | 12 | 4 | | | | | 8122 | 8112 | 8046 | **8046** | **8,54** | **0,71** |
| 14 | 16 | 4 | | | | | 6677 | 6634 | 6788 | **6634** | **10,35** | **0,65** |
| 15 | 20 | 4 | | | | | 6276 | 6357 | 6527 | **6276** | **10,94** | **0,55** |
| 16 | 24 | 4 | | | | | 5786 | 5808 | 5830 | **5786** | **11,87** | **0,49** |
| 17 | 28 | 4 | | | | | 5813 | 5988 | 5775 | **5775** | **11,89** | **0,42** |
| 18 | 32 | 4 | | | | | 5590 | 5580 | 5638 | **5580** | **12,31** | **0,38** |
| 19 | 2 | 12 | 68702 | 68770 | 71195 | **68702** | 35512 | 37437 | 35676 | **35512** | **1,93** | **0,97** |
| 20 | 4 | 12 | | | | | 18532 | 19204 | 18569 | **18532** | **3,71** | **0,93** |
| 21 | 8 | 12 | | | | | 9797 | 9805 | 9989 | **9797** | **7,01** | **0,88** |
| 22 | 12 | 12 | | | | | 7275 | 7049 | 7343 | **7049** | **9,75** | **0,81** |
| 23 | 16 | 12 | | | | | 5938 | 6026 | 5929 | **5929** | **11,59** | **0,72** |
| 24 | 20 | 12 | | | | | 6041 | 6095 | 5913 | **5913** | **11,62** | **0,58** |
| 25 | 24 | 12 | | | | | 5758 | 5690 | 5628 | **5628** | **12,21** | **0,51** |
| 26 | 28 | 12 | | | | | 5482 | 5673 | 5618 | **5482** | **12,53** | **0,45** |
| 27 | 32 | 12 | | | | | 5522 | 5473 | 5548 | **5473** | **12,55** | **0,39** |

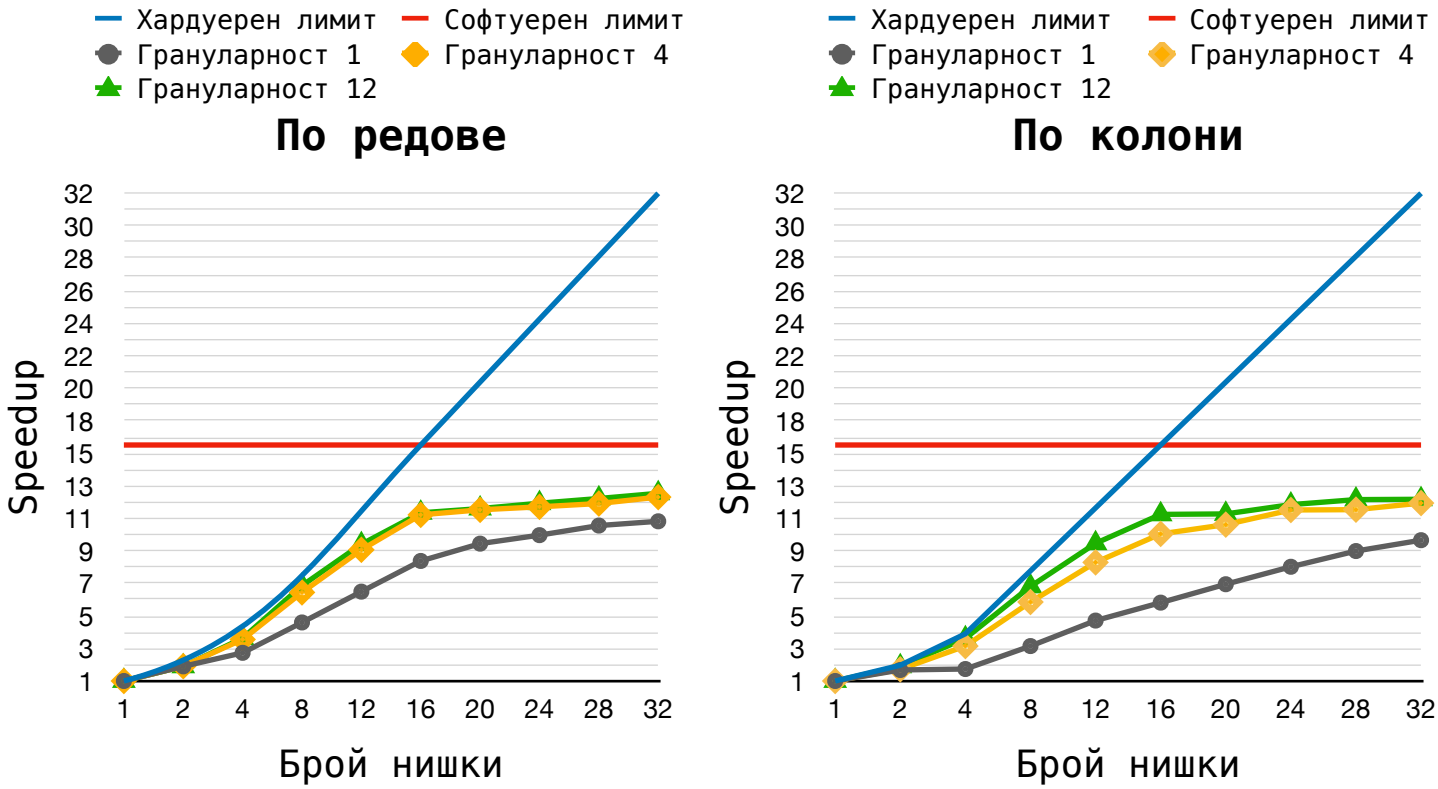



**Фигура 9.** Ускорение при декомпозиция по колони




**Фигура 10.** Ефективност при декомпозиция по колони

### 5.3. Сравнение между декомпозицията по редове и декомпозицията по колони



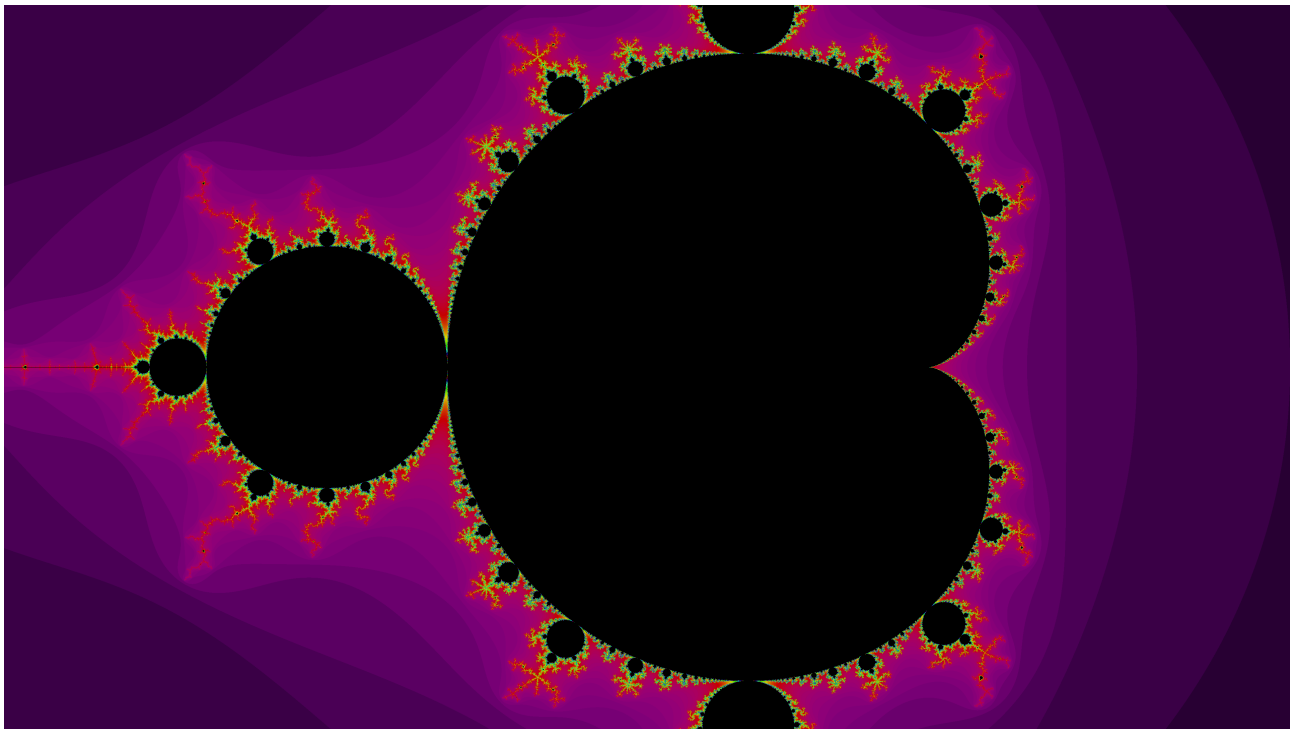

От горните две графики на ускорението и от таблиците с резултатите от тестовете забелязваме, че има разлика между декомпозицията по редове и по колони, макар и малка. Това означава, че задачите, които се образуват при декомпозиция по колони не могат да се балансират толкова добре, колкото при декомпозицията по редове. Най-ясно това може да се види при грануларност 1. Там разликата е най-голяма между двете декомпозиции и графиката на декомпозицията по колони показва, че наистина задачите при тази декомпозиция са не толкова балансирани помежду си. Този дисбаланс се пренася и при другите грануларности, като той започва да се премахва с растенето на грануларността, поради това, че задачите стават все по-малки и по-малки. Този дисбаланс, който имахме първоначално при грануларност 1, изчезва. Това е причината резултатите при грануларност 12 да са сходни.

В приложението е използван двумерен масив от байтове от чисти съображения за оптимизиране на операциите по прехвърляне от области от паметта в L1-d кеша било то от по-долните кешове или от главната памет (колкото по-надолу в йерархията се взима дадената област, толкова по-бавно става тази операция). Това означава, че използваме един байт за един пиксел, което значително намалява размера на масива в сравнение с двумерен int масив. Самото използване на масив от байтове за всеки

пиксел увеличава броят на пикселите, с които всяка нишка ще разполага в своя L1-d кеш, което намалява необходимостта от непрекъснатото му презареждане. Както виждаме, когато използваме декомпозиция по редове, няма голяма разлика между грануларност 4 и 12, тъй като тогава и в двата случая размера на областите, върху които дадена нишка ще работи са най-приспособени към размера на L1-d кеша.

Друг важен коментар върху получените резултати е хипертрединга. Тъй като тестовата машина разполага с 16 физически ядра, резултатите, които са получени след 16 нишки се дължат именно на хипертрединга.

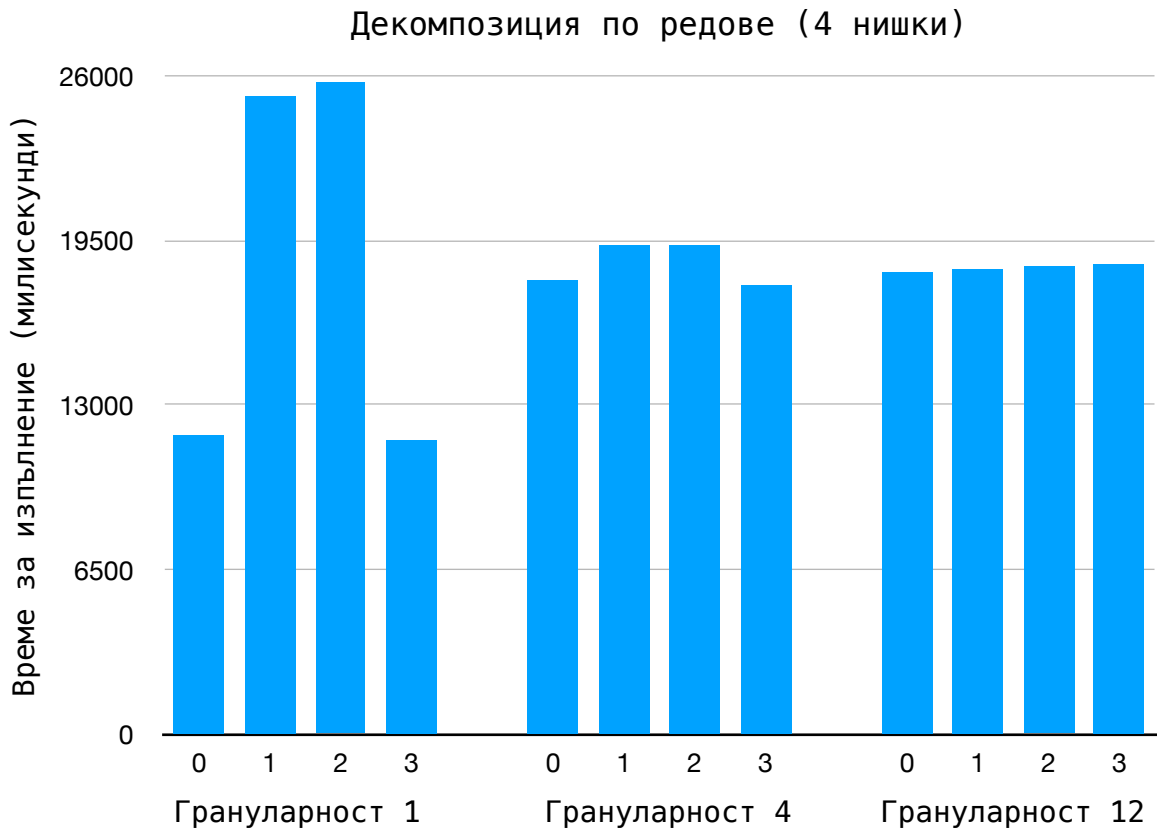
Окончателно, можем да заключим, че при изображение с размер 3840x2160 и размер на изследвана област $a \in [-1.67; 1]$, $b \in [-0{,}75; 0{,}75]$, където $a$ и $b$ са съответно реалната и имагинерната част на комплексното число $z = a + i * b$, най-добро ускорение и адаптивност към L1-d кеш получаваме при декомпозиция по редове.

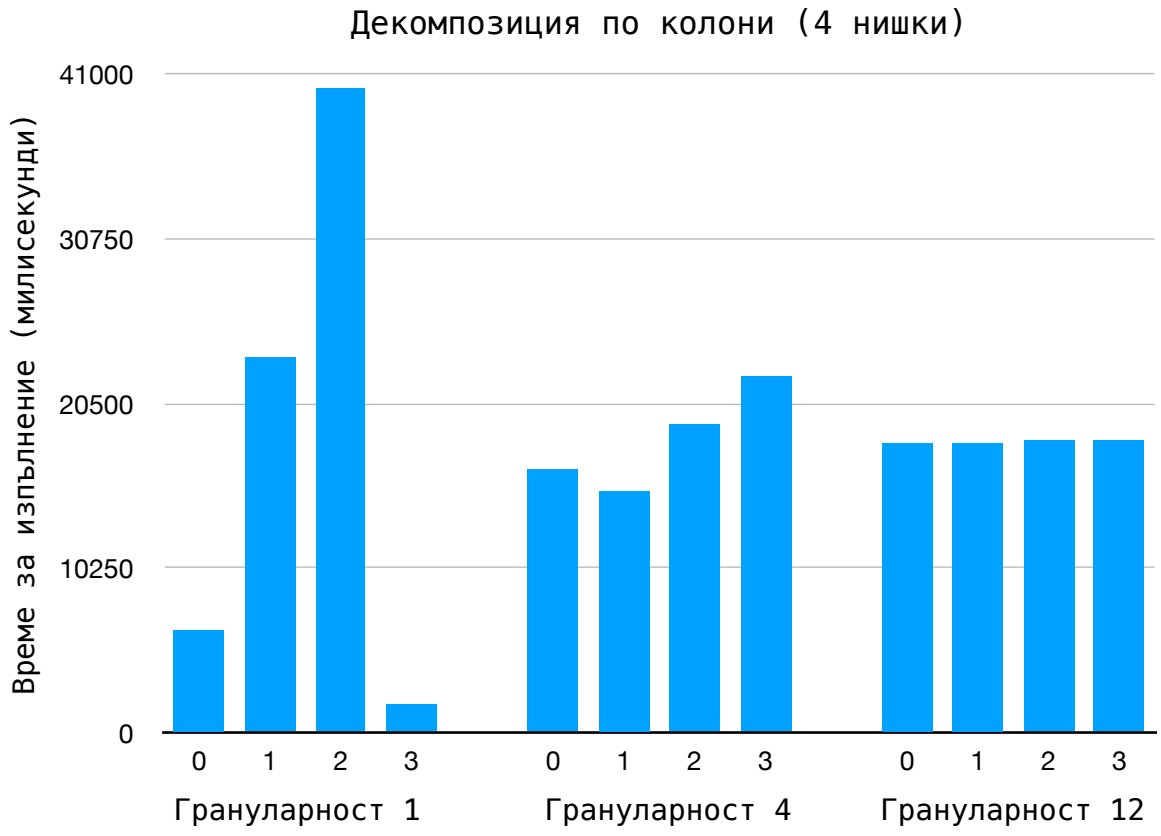


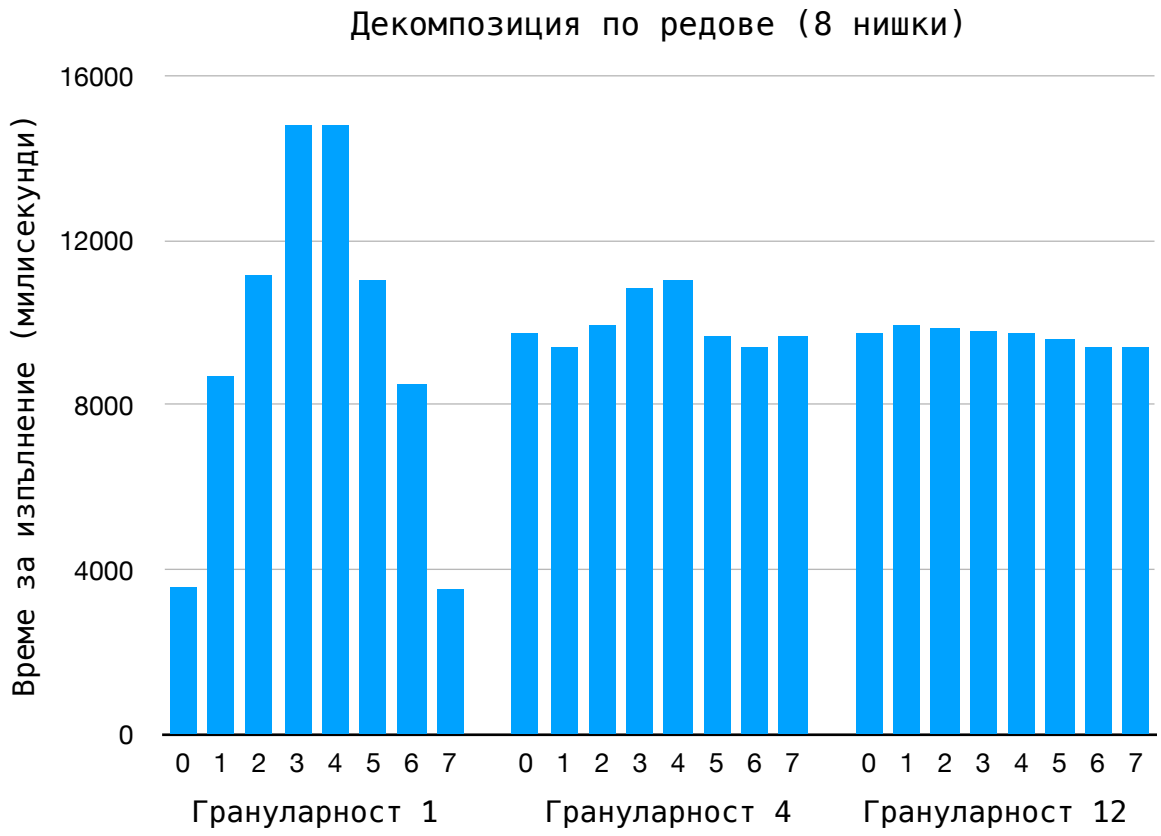
**Фигура 11.** Полученото от горните тестове изображение на множеството на Манделброт в съответната област от комплексната равнина

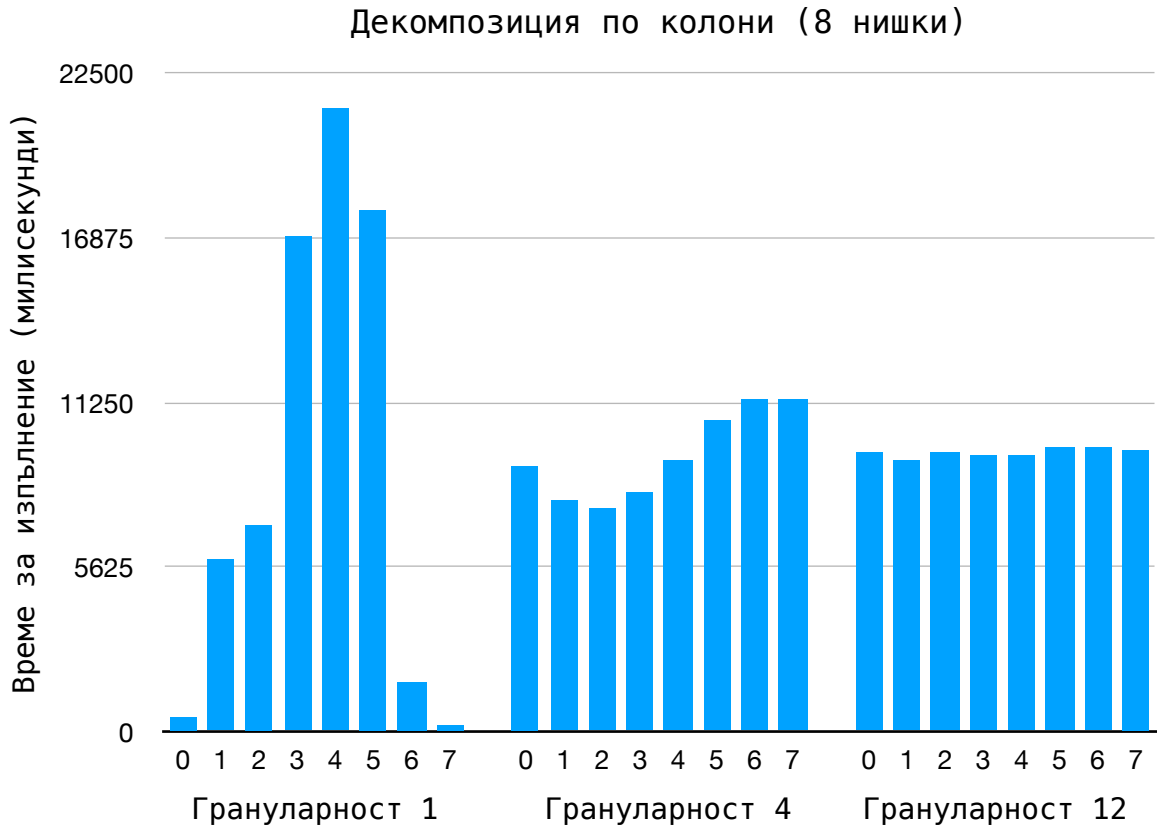
## 5.4. Сравнение на времената за работа на нишките при декомпозиция по редове и при декомпозиция по колони

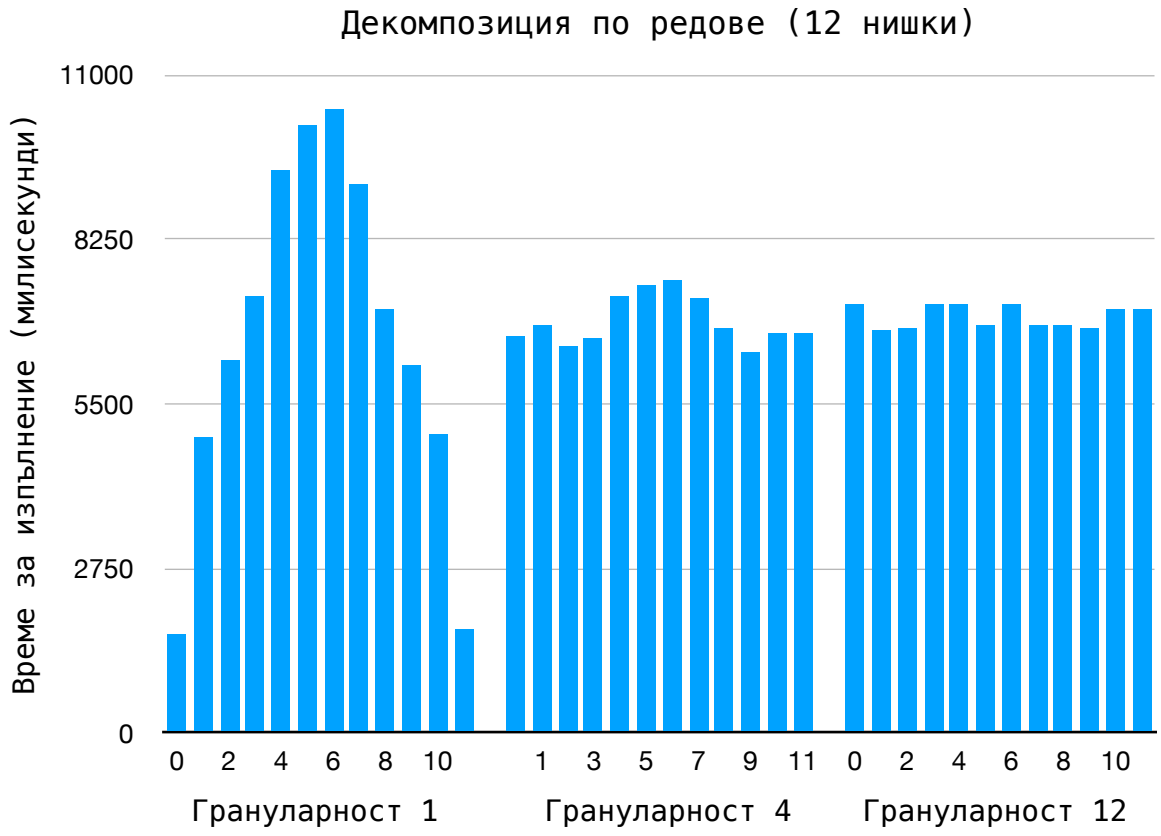
В тази точка ще разгледаме дисперсията на времената за работа на отделните нишки при двете декомпозиции. Това от една страна ще ни даде оценка на това по какъв начин грануларността балансира времената за работа на нишките, а от друга ще покаже визуално, че при декомпозиция по колони колкото по-едра е грануларността, толкова повече отделните задачи не са балансирани.

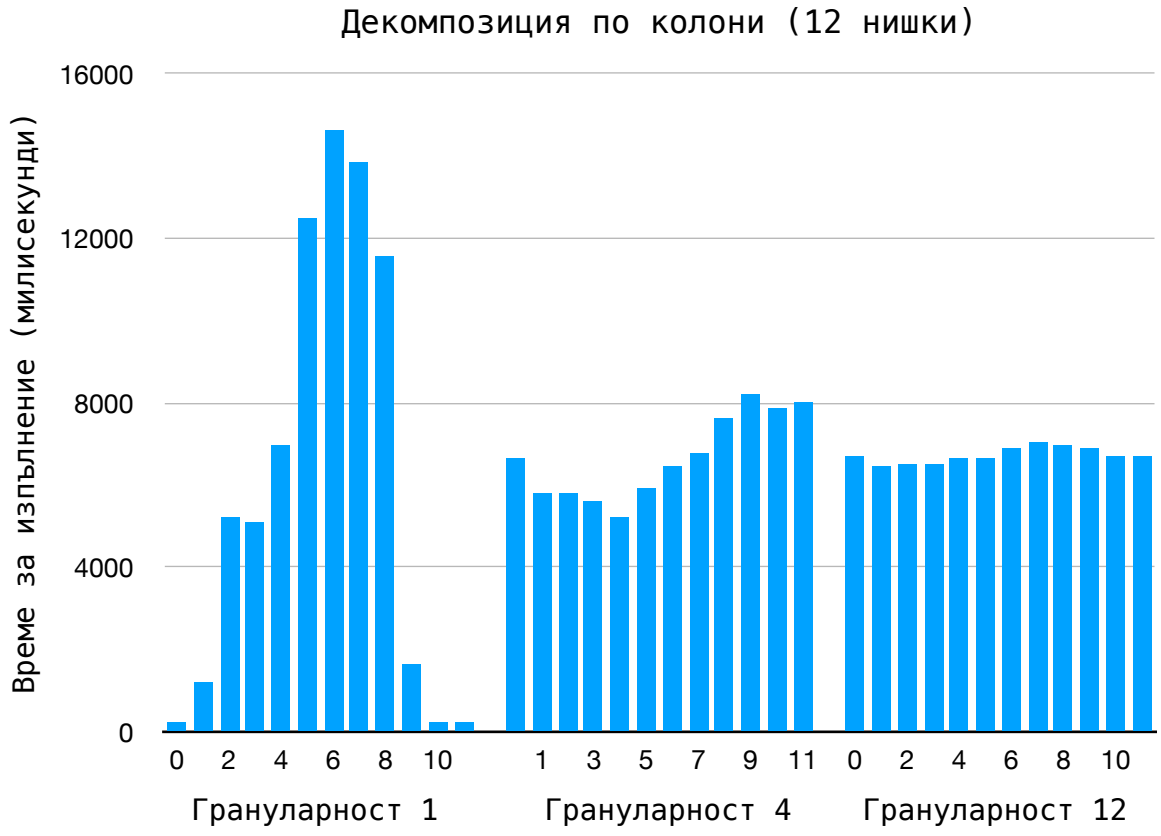

Декомпозиция по редове (4 нишки)
Време за изпълнение (милисекунди)
26000
19500
13000
6500
0
0 1 2 3
0 1 2 3
0 1 2 3
Грануларност 1
Грануларност 4
Грануларност 12


Декомпозиция по колони (4 нишки)
Време за изпълнение (милисекунди)
41000
30750
20500
10250
0
0 1 2 3
0 1 2 3
0 1 2 3
Грануларност 1
Грануларност 4
Грануларност 12


Декомпозиция по редове (8 нишки)
Време за изпълнение (милисекунди)
16000
12000
8000
4000
0
0 1 2 3 4 5 6 7
0 1 2 3 4 5 6 7
0 1 2 3 4 5 6 7
Грануларност 1
Грануларност 4
Грануларност 12


Декомпозиция по колони (8 нишки)
Време за изпълнение (милисекунди)
22500
16875
11250
5625
0
0 1 2 3 4 5 6 7
0 1 2 3 4 5 6 7
0 1 2 3 4 5 6 7
Грануларност 1
Грануларност 4
Грануларност 12


Декомпозиция по редове (12 нишки)
Време за изпълнение (милисекунди)
11000
8250
5500
2750
0
0 2 4 6 8 10 1 3 5 7 9 11 0 2 4 6 8 10
Грануларност 1
Грануларност 4
Грануларност 12


Декомпозиция по колони (12 нишки)
Време за изпълнение (милисекунди)
16000
12000
8000
4000
0
0 2 4 6 8 10 1 3 5 7 9 11 0 2 4 6 8 10
Грануларност 1
Грануларност 4
Грануларност 12

На хистограмите можем да забележим, че има съществена разлика при гранулярности 1 и 4 при двете декомпозиции. При декомпозицията по колони виждаме, че различните задачи са значително небалансирани и се случва така, че цялата програма чака една нишка да завърши своята работа, докато другите са свършили. Но този проблем постепенно се оправя с увеличаването на грануларността и двете декомпозиции практически стават почти балансирани при грануларност 12. Това означава, че при средна грануларност двете декомпозиции са почти еднакви по отношение на балансираност на задачите, а и по ускорение.

# 6. Динамично централизирано балансиране

## 6.1. Функционално проектиране

### 6.1.1. Модел на приложението

Моделът на приложението е Master–Slave. Това означава, че когато стартираме например 4 нишки, една нишка (master) ще разпределя задачите, а другите 3 ще вършат работата (затова биват наричани slaves). Първоначално, master нишката раздава последователно задачи и след това следи за завършването на нишките. Когато една slave нишка завърши работа по задачата си, тя взима следващата задача от master нишката.

В рамките на тази реализация е използвана декомпозиция по редове с цел да се сравнят резултатите със статичното циклично балансиране с декомпозиция по редове, което даде най-добрите резултати.

### 6.1.2. Описание на командни параметри

Командите, реализирани тук не се различават от използваните в миналото приложение. Те са подмножество на тях:

| Опция | Описание |
|---|---|
| **r, rect** | Чрез нея се задават границите на областта от комплексната равнина, в която ще търсим да визуализираме множеството на Манделброт. Например: -r (-rect) -1.0:1.0:0.5:2.0, което се интерпретира като: $a \in [-1.0; 1.0], b \in [0.5; 2.0]$, където $a, b \in R$ и $z = a + i * b$. По подразбиране областта е $a \in [-2.0; 2.0], b \in [-2.0; 2.0]$. |
| **t, tasks** | Броят на нишките, които ще трябва да работят паралелно. По подразбиране, програмата стартира с 1 нишка. |
| **o, output** | Задава се името на изображението, което ще се генерира. По подразбиране, това е “Mandelbrot.png”. |
| **s, size** | Размерът на изображението, което ще се генерира. По подразбиране, това е 3840x2160 (4K). |

| Опция | Описание |
|---|---|
| **h, help** | Когато този команден параметър бъде добавен, програмата извежда списък с наличните команди и тяхното значение на командния ред. След това програмата прекратява своето изпълнение. |
| **g, granularity** | С този параметър може да се посочи грануларността (или броят задания, които всяка нишка трябва да изпълни). По подразбиране, това е най-едрата грануларност $g = 1$. |

Програмата извежда на командния ред единствено съобщение, когато приключи работа:

**Total execution time: <exec time> ms.**

## 6.2. Технологично проектиране

### 6.2.1. Тестова среда

Тестовата машина е абсолютно същата, която бе използвана и за тестване на програмата със статично циклично балансиране.

### 6.2.2. Използван език и външни библиотеки

Използваният език и външни библиотеки отново са същите, както при програмата, реализираща статично балансиране.

### 6.2.3. Пример за стартиране на програма

Пример за това как се стартира програмата:

```
./runDynamicMandelbrot.sh -t 12 -g 16 -r -1.67:1:-0.75:0.75
```

С този ред се извиква shell скрипта runDynamicMandelbrot.sh, който извиква програмата с параметри командните параметри t, g и r. По този начин програмата ще се изпълни посредством употребата на 12 нишки, грануларността ще е 16 (тоест всички задания ще са 16*12 = 192) и ще се извърши динамично централизирано балансиране по редове в областта [-1.67;1] за реалната ос и [-0.75;0.75] за имагинерната ос.

### 6.2.4. Стартиране и използване на нишки

```
ExecutorService pool = Executors.newFixedThreadPool(numThreads - 1);

Runnable[] tasks = new Runnable[rows];
for (int i = 0; i < rows; i++) {
      tasks[i] = new Runnable(i);
}

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

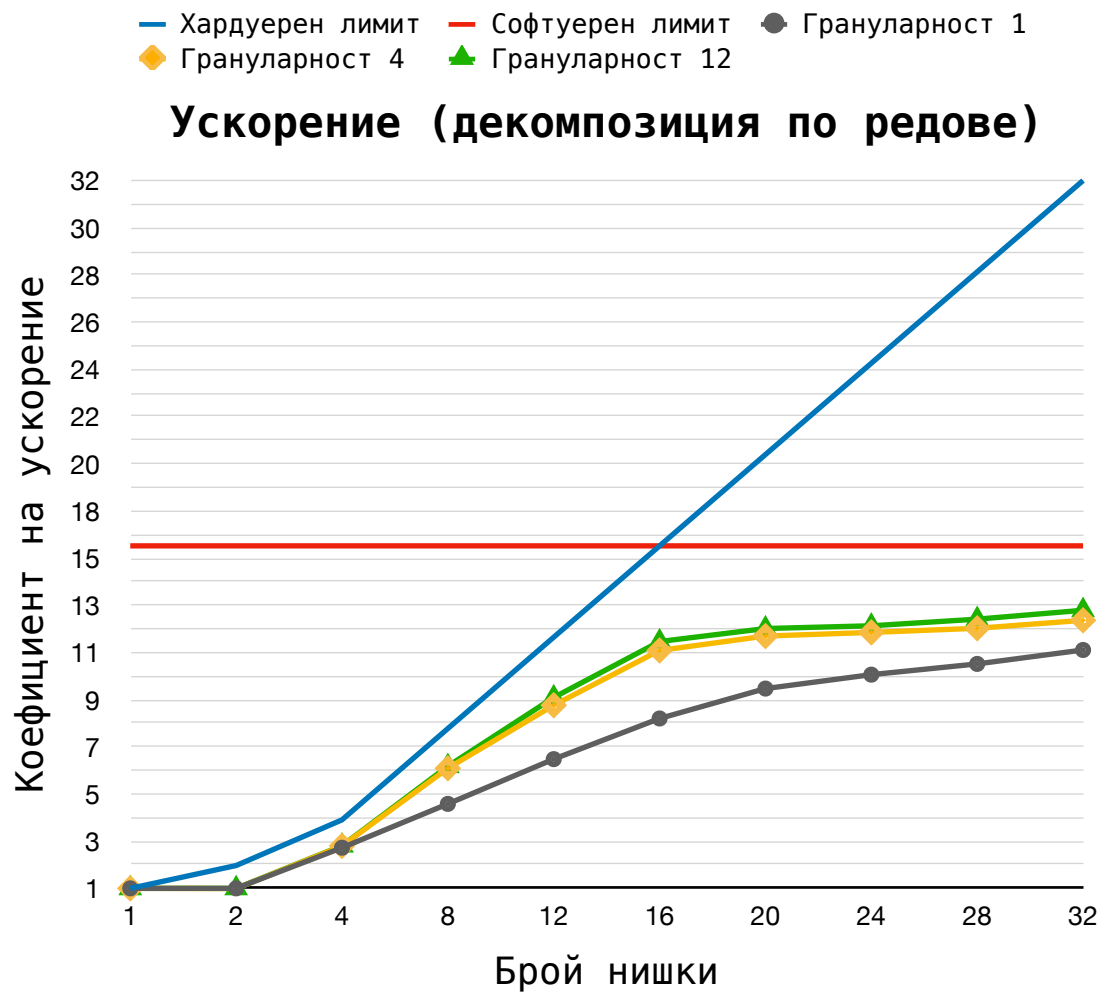
```
}
pool.shutdown();

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

Нишките се създават чрез вградената функция Executors.newFixedThreadPool() в библиотеката java.util.concurrent.Executors (трябва да се отбележи че това е вградена библиотека, а не външна). Чрез тази функция се създава басейн от нишки с фиксиран размер, тоест в този басейн винаги ще се намират точно (numThreads – 1) нишки (колкото сме задали в кода горе). Тук numThreads представлява аргумента, подаден от командния ред, задаващ колко нишки трябва да работят едновременно. Създаваме (numThreads – 1) допълнителни нишки, поради причината, че master нишката ще е част от тази обща бройка (част от паралелизма), въпреки че тя не изпълнява никакви задачи. Ако в този басейн в дадено време постъпят задачи, но всички нишки са заети, те ще отидат в опашка, от която тези нишки ще взимат следващото си задание. След като създадем басейна от нишки, следва напълването му със задачи, които нишките да изпълняват. Тези задачи се създават в първия for цикъл, като на всяка задача й се подава аргумент i, съответстващ на това коя задача по поредност е тя, и по която да се ориентира коя част от изображението ще обхваща тя. След тази процедура следва изпълването на басейна с нишки със задачи посредством pool.execute(). След като задачите бъдат добавени в опашката на басейна, се извиква функцията pool.shutdown(), която спира постъпването на нови задачи. Накрая, посредством pool.awaitTermination() master нишката изчаква slave нишките да се терминират. Подадените на функцията awaitTermination() параметри съответстват на това колко най-много ще чака master нишката – а именно Long.MAX_VALUE наносекунди.

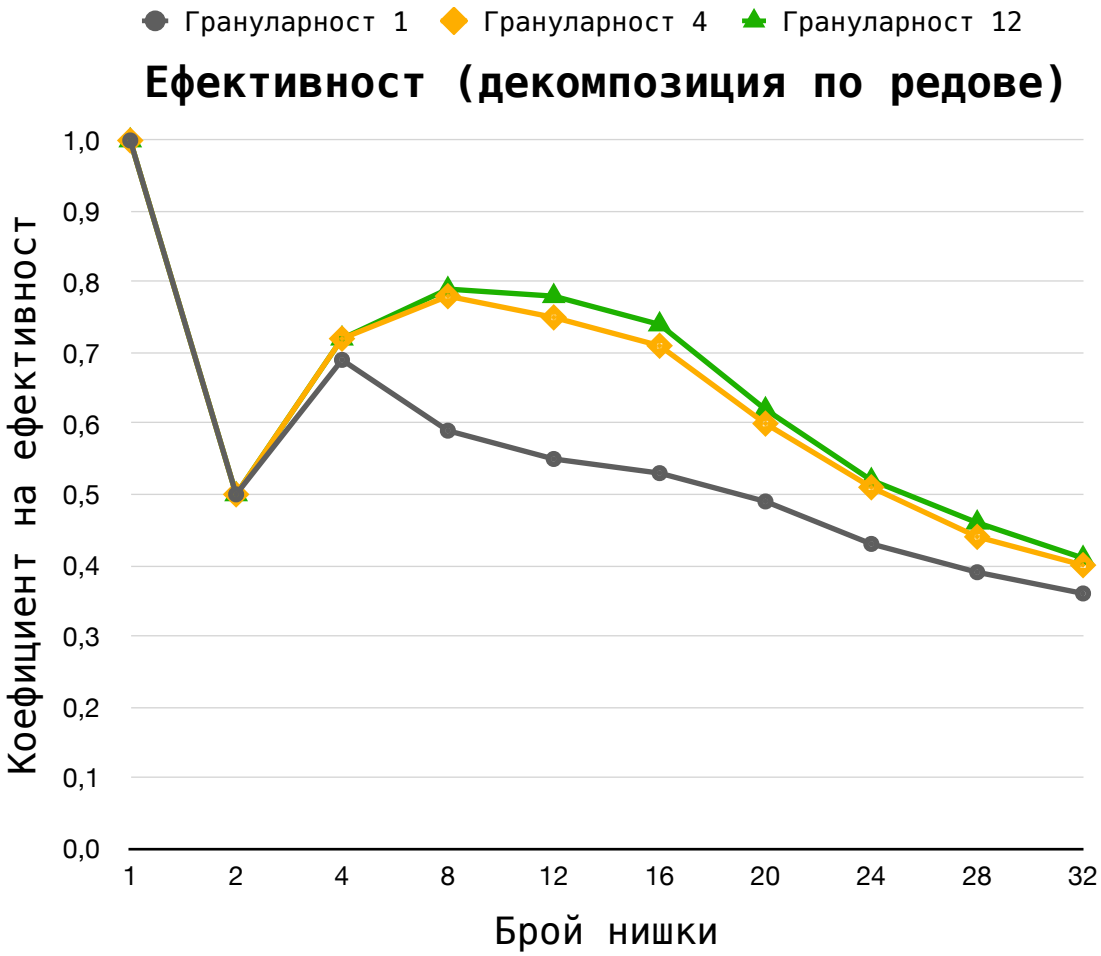
## 7. Резултати от тестване на динамично централизирано балансиране

Използваните параметри в тази таблица са аналогични на използваните в предишните две таблици по-горе. Изследваната област също е [–1.67;1] и [–0.75;0.75] съответно за реалната и имагинерната ос. Тук, обаче, сме използвали измерванията при 1 нишка от статичното циклично балансиране с декомпозиция по редове, тъй като от една страна не можем да направим динамично балансиране само с 1 нишка и от друга понеже да имаме обща база, на която да сравним двете балансирания. Трябва да се отбележи, че тук g (грануларност) има по-различен смисъл. Това всъщност е средния брой задачи, която една задача ще изпълни. Може да не са точно толкова (или по-малко, или повече, но средно ще се изпълняват точно толкова задачи). То присъства тук с цел да се извършват същия брой задачи като при статичното циклично балансиране с декомпозиция по редове. По този

| # | p | g | $T_1$ (1) | $T_1$ (2) | $T_1$ (3) | $T_1$=min() | $T_p$(1) | $T_p$(2) | $T_p$(3) | $T_p$ = min() | $S_p$ = $T_1/T_p$ | $E_p$ = $S_p/p$ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 1 | 70895 | 69751 | 69411 | **69411** | 69481 | 70037 | 70003 | **69481** | **1,00** | **0,50** |
| 2 | 4 | 1 | | | | | 25099 | 25331 | 24994 | **24994** | **2,78** | **0,69** |
| 3 | 8 | 1 | | | | | 14764 | 15050 | 15446 | **14764** | **4,70** | **0,59** |
| 4 | 12 | 1 | | | | | 10428 | 10488 | 10507 | **10428** | **6,66** | **0,55** |
| 5 | 16 | 1 | | | | | 8406 | 8395 | 8225 | **8225** | **8,44** | **0,53** |
| 6 | 20 | 1 | | | | | 7117 | 7638 | 7230 | **7117** | **9,75** | **0,49** |
| 7 | 24 | 1 | | | | | 7049 | 6697 | 6832 | **6697** | **10,36** | **0,43** |
| 8 | 28 | 1 | | | | | 6504 | 6632 | 6411 | **6411** | **10,83** | **0,39** |
| 9 | 32 | 1 | | | | | 6200 | 6070 | 6185 | **6070** | **11,44** | **0,36** |
| 10 | 2 | 4 | 69542 | 70564 | 69701 | **69542** | 69953 | 73482 | 71699 | **69953** | **0,99** | **0,50** |
| 11 | 4 | 4 | | | | | 24307 | 26298 | 24478 | **24307** | **2,86** | **0,72** |
| 12 | 8 | 4 | | | | | 11318 | 11115 | 11497 | **11115** | **6,26** | **0,78** |
| 13 | 12 | 4 | | | | | 7713 | 7712 | 7808 | **7712** | **9,02** | **0,75** |
| 14 | 16 | 4 | | | | | 6090 | 6115 | 6149 | **6090** | **11,42** | **0,71** |
| 15 | 20 | 4 | | | | | 6056 | 5770 | 5884 | **5770** | **12,05** | **0,60** |
| 16 | 24 | 4 | | | | | 5753 | 5795 | 5695 | **5695** | **12,21** | **0,51** |
| 17 | 28 | 4 | | | | | 5611 | 5611 | 5618 | **5611** | **12,39** | **0,44** |
| 18 | 32 | 4 | | | | | 5467 | 5685 | 5459 | **5459** | **12,74** | **0,40** |
| 19 | 2 | 12 | 69765 | 69472 | 69915 | **69472** | 69459 | 69529 | 69341 | **69341** | **1,00** | **0,50** |
| 20 | 4 | 12 | | | | | 24248 | 24312 | 24365 | **24248** | **2,87** | **0,72** |
| 21 | 8 | 12 | | | | | 10935 | 11019 | 11018 | **10935** | **6,35** | **0,79** |
| 22 | 12 | 12 | | | | | 7431 | 7425 | 7424 | **7424** | **9,36** | **0,78** |
| 23 | 16 | 12 | | | | | 5916 | 5882 | 5918 | **5882** | **11,81** | **0,74** |
| 24 | 20 | 12 | | | | | 5610 | 5673 | 5702 | **5610** | **12,38** | **0,62** |
| 25 | 24 | 12 | | | | | 5556 | 5558 | 5558 | **5556** | **12,50** | **0,52** |
| 26 | 28 | 12 | | | | | 5438 | 5582 | 5430 | **5430** | **12,79** | **0,46** |
| 27 | 32 | 12 | | | | | 5400 | 5349 | 5271 | **5271** | **13,18** | **0,41** |

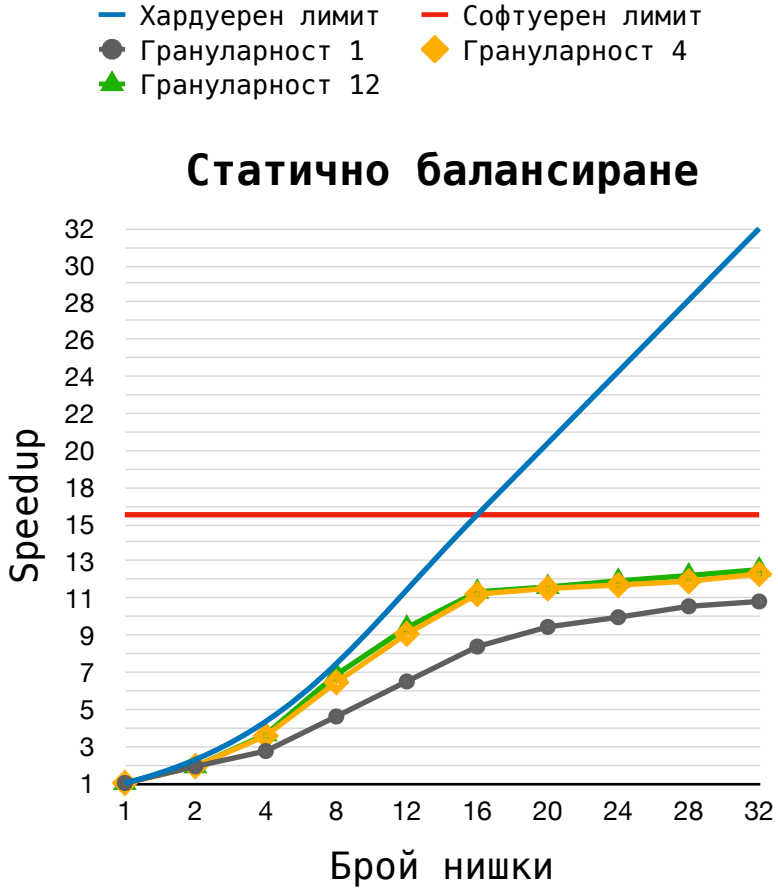


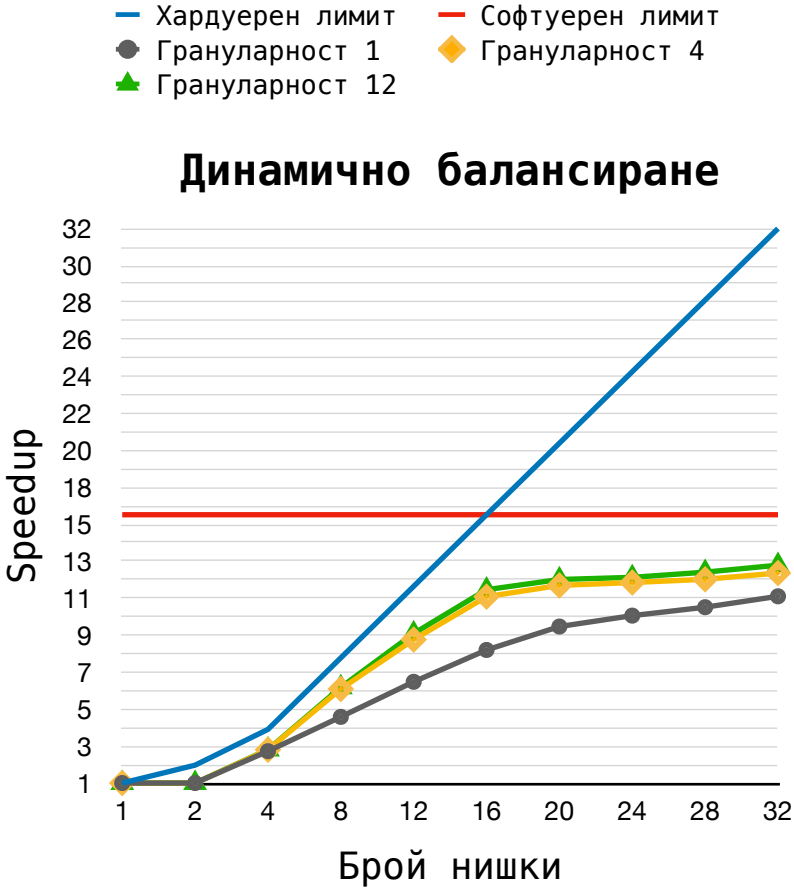

**Фигура 12.** Ускорение при динамично балансиране с декомпозиция по редове




**Фигура 13.** Ефективност при динамично балансиране с декомпозиция по редове

## 8. Сравнение между статично циклично и динамично централизирано балансиране




Нека отбележим първо, че в програмата с динамично балансиране е реализирана същата стратегия относно оптимизация на ниво L1-d кеш. Тоест, двумерния масив, който съдържа броя итерации, генериран за всеки пиксел от изображението, всъщност е масив от байтове. По този начин, както коментирахме по-горе, в L1-d кеша на всяка нишка ще се запази по-голяма част от самия масив, което спестява значителен брой операции по презареждане на кеша, които все пак отнемат някакво време (това време става по-голямо в зависимост от това от къде достъпваме данните, които искаме да презаредим в кеша – тоест достъпа до главната памет ще е най-тежка операция от към време).

Можем да забележим от двете графики, а и от таблиците, че няма съществена разлика в двете ускорения – те са сходни, като по-конкретно до 16 нишки статичното балансиране работи по-добре (по-бързо) от динамичното, но след това се случва обратното. Въпреки това, разликата между ускорението след 16 нишки между двете балансирания не е особено голяма.

Но все пак, самите данни, които сме получили от тестовете показват това, че динамичното балансиране се представя по-добре при голям брой нишки (над 16) от статичното. Това, противно на очакванията ни, от една страна може да се дължи на хипертрединга, но от друга, може да се дължи на това, че броят задачи все още не е достатъчно голям, за да се получи bottleneck-а, който е неминуем при употребата на централизирани решения. За да предизвикаме такъв ни трябват голям брой задания. От тук се появява проблема на поредовата декомпозиция, която е ограничена от към брой задачи. Максималният брой задачи е 3840, който не е достатъчно голям, за да предизвика bottleneck при 32 нишки. За да се получи този bottleneck, може би трябва да се увеличи размерът на изображението.

Като заключение, въпреки че не получихме bottleneck с динамичното централизирано балансиране, видяхме че ускоренията между двата вида балансирания се представят сходно, като динамичното представя по-добри резултати при голям брой нишки (над 16). Отново при грануларности 4 и 12 при динамичното балансиране, размера на изследваната област на всяка нишка е най-добре приспособена към размера на L1-d кеша.

## 9. Източници


[1] Mirco Tracolli, **Parallel generation of a Mandelbrot set**, Department of Mathematics and Computer Sciences, University of Perugia, April 21, 2016 (http://services.chm.unipg.it/ojs/index.php/virtlcomm/article/view/112)
[2] Bhanuka Manesha Samarasekara Vitharana Gamage, Vishnu Monn Baskaran, **Efficient Generation of Mandelbrot Set using Message Passing Interface**, School of Information Technology, Monash University Malaysia, July 1 2020 (https://arxiv.org/pdf/2007.00745.pdf)
[3] Alan Kaminsky, **Building Parallel Programs**, Course Technology, 2010 (https://ftp.utcluj.ro/pub/users/civan/CPD/1_RESURSE_CURS/Books/Book_2010_Kaminsky_buildingParalelPrograms.pdf)
[4] Renato Fonseca, **The Mandelbrot Set**, 9 May 2021 (https://renatofonseca.net/mandelbrotset)