ParMandelbrot:

Изследване на грануларността и адаптиране спрямо L1 d-cache при статично циклично разпределение и съпоставяне на получените резултати с динамично централизирано решение

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

Изготвил:
Калоян Николов
Софтуерно инженерство
Курс: III, ф. н. 62252

Съдържание:

0. Въведение

В рамките на настоящия проект – ParMandelbrot, се разглеждат редица въпроси, свързани с паралелизма, като за целта е използвано множеството на Манделброт. Ето защо в тази секция се разглежда какво представлява то и как може да се определи дали произволна точка му принадлежи.

Множеството на Манделброт – открито и описано за първи път от Беноа Манделброт, е най-известният пример за фрактал в днешно време. Въпреки, че няма строга дефиниция за понятието „фрактал“, то това е геометричен модел със сложна структура при каквото и да е приближение, точно или приблизително самоподобен и/или притежава дробна размерност.

Множеството на Манделброт съдържа комплексните числа **c**, за които функцията:

$f_C(z) = z^2 + c$ не е разходяща при N итерации за N, клонящо към безкрайност, и започвайки с z = 0, т.е. искаме редицата $f_C(0), f_C(f_C(0)), \ldots$ да остава ограничена по абсолютна стойност за всяко N. Казано по-формално множеството на Манделброт M е [8]:

$$M = \{c \in \mathbb{C}: \exists R \forall n: |z_n| < R\}$$

Множеството на Манделброт е компактно множество, тъй като е затворено и ограничено от окръжност с радиус 2 и център (0;0). Доказано е, че ако за дадено число **c**, някой член N на съответстващата редицата е по-голям от 2, то редицата не е ограничена и съответното число **c** не принадлежи на множеството на Манделброт [8].

Друга особеност на множеството на Манделброт е, че е свързано множество, което ще бъде реферирано в „Секция 1.1.2.“ [8].

Защо сме избрали точно множеството на Манделброт?

Това е пример за т. нар. „embarrassingly parallel“ проблеми, които се поддават изключително лесно на паралелна обработка. Причината за това е, че разделянето на голямата задача на няколко подзадачи е лесно за осъществяване, като не е необходима постоянна комуникация между нишките, изпълняващи съответните подзадачи.

1. Анализ

1.1. Функционален анализ

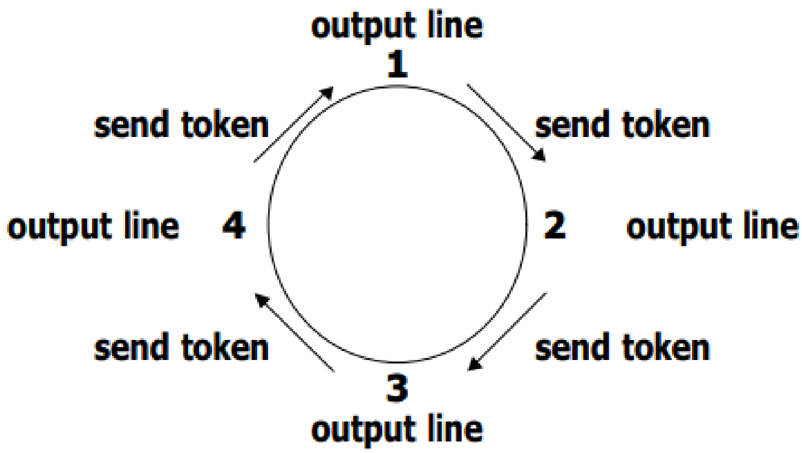
В тази секция са представени няколко образеца, които съдържат полезни анализи и изследвания върху разгледания в този проект проблем.

1.1.1. Parallel Fractal Image Generation – A Study of Generating Sequential Data With Parallel Algorithms, Matthias Book, The University of Montana, Missoula – Spring Semester 2001

Първоначално се описват стъпките, които трябва да следваме при изграждане на паралелен алгоритъм. Определяме коя е най-малката отделна задача или атом, която може да се направи – в случая това е изчисляването на 1 пиксел дали принадлежи на множеството на Манделброт. Така се избягва нуждата от постоянна комуникация между нишките. След това трябва да решим как да обединим тези минимални задачи или атоми в цялостни задачи, които ще дадем на отделните нишки. Разгледан е сценарий, при който се дават цели блокове от редове, но авторът се фокусира основно върху подаването на отделни редове като задачи на нишките с цел по-добро балансиране.

Това, което авторът отбелязва е, че, за да е възможно генерирането на много големи изображения е необходимо нито 1 нишка да не съхранява цялото изображение. Въпреки че за самите изчисления не е нужна комуникация, то такава е необходима за създаването на цялостното изображение. Ако всяка нишка принтира съответния ред от изображението веднага след като го изчисли, то е ясно, че изображението няма да бъде в правилна последователност. Ето защо авторът се опитва да намери механизъм за синхронизация.

Разгледан е алгоритъм, при който нишките си предават жетон или token. Нишката, която получи жетона, трябва да принтира съответния ред от изображението и след това да предаде жетона на следващата нишка. Алгоритъмът може да се проследи на фигура 1.



Фигура 1 – Опит за синхронизация чрез предаване на жетон между нишките [1].

Разгледан е вариант, при който нишките блокират до получаване на жетона и такъв, при който само проверяват дали са получили жетона след завършване на всяка задача и ако не са – продължават да изпълняват оставащите им задачи.

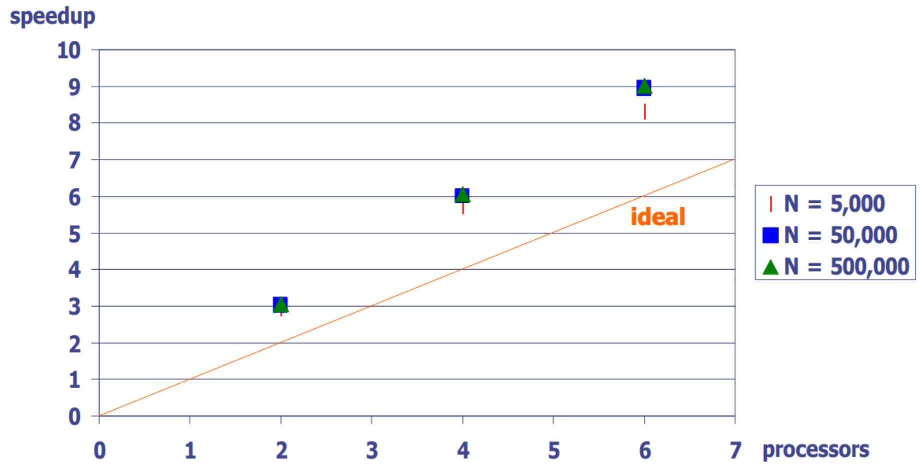
За съжаление, такъв алгоритъм не работи коректно и авторът описва внимателно каква е причината – нишките нямат контрол върху това как отделните редове от изображението ще бъдат подредени в изходния файл. Ето защо се стига до заключение, че **само 1 нишка трябва да принтира цялото изображение**. За да не нарушаваме изискването да няма нишка, която да съхранява цялото изображение, се взима следното решение – нишката, която ще принтира изображнието (както авторът я нарича master нишката), ще притежава контейнер, където да се съхранява следващият ред, който трябва да се получи от всяка работеща нишка. Когато контейнерът се напълни, нишката принтира своя ред и тези на останалите нишки в правилната последователност и изпразва контейнера. По този начин се освобождава място за следващото „парче“ от изображението. Ако master нишката изчисли своя ред от изображението, а не е получила съответните редове от останалите нишки, тя просто продължава със следващите редове, които трябва да изчисли. Тъй като master нишката изпраща заявка за редовете в правилната последователност, а другите нишки проверяват дали са получили такава заявка след изчисляването на всеки ред, то е гарантирано, че винаги ще могат да дадат желания ред.

Друг вариант е в master нишката да има буфер, в който да се съхраняват всички редове, които са изчислени от другите нишки. По този начин е възможно в даден момент, master нишката да съхранява почти цялото изображение в буфера, а ние се опитваме да избегнем точно това. Ето защо авторът не предпочита този вариант.

Изследва се постигнатото ускорение при гореописаното паралелно решение и се представят получите резултати чрез няколко таблици и диаграми. Тук ще представя само 1 от таблиците и построената на база на нея графика:

$S_{P,N}$	$P = 2$	$P = 4$	$P = 6$
$N = 5{,}000$	2,93061915	5,72014816	8,31788581
$N = 50{,}000$	3,03581113	6,00517219	8,95976592
$N = 500{,}000$	3,05658884	6,04407973	9,03326882

Фигура 2 – Постигнато ускорение от автора [1]



Фигура 3 – Графично представяне на постигнатото ускорение – N е максималният брой (дълбочина) на итерациите [1].

Авторът е постигнал ускорение, по-добро от идеалното линейно ускорение, към което трябва да се стремим при паралелните алгоритми. Тези „неочаквано добри“ резултати се обясняват по следния начин:

При последователната програма ние просто изчисляваме даден пиксел и го принтираме, след което изчисляваме следващия пиксел, принтираме и него и т.н. При паралелната програма, това не е така – имаме няколко нишки, които осъществяват изчисленията за съответните пиксели и поради използването на неблокиращи механизми за комуникация, се получава второ ниво на паралелизъм. Изпращането на вече изчисления ред от нишката j се осъществява на заден фон (тъй като това е входно-изходна операция) и нишката не спира да изчислява.

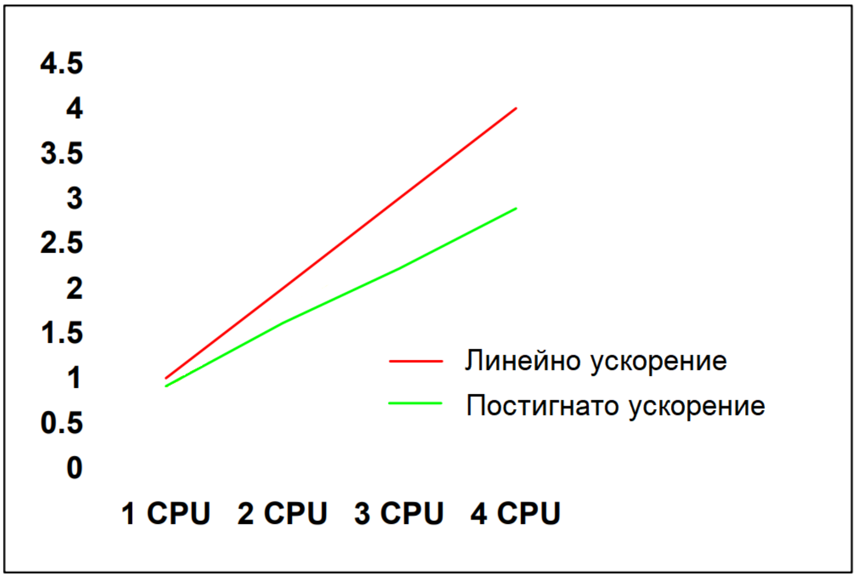
1.1.2. Isaac K. Gäng, David Dobson, Jean Gourd and Dia Ali, Parallel Implementation and Analysis of Mandelbrot Set Construction, University of Southern Mississippi, 2008

Авторът използва т. нар. **Escape time** алгоритъм при определянето дали дадена точка принадлежи на множеството на Манделброт. Този подход ще използваме и ние при нашите изследвания. Съгласно него, първо определяме стойността на комплекната константа C – това е точката, която искаме да проверим дали принадлежи на множеството на Манделброт. Последователно изпълняваме до N на брой присвоявания (итерации) от вида $Z = Z^2 + C$, където първоначално $Z = 0$. След всяка от тези итерации се проверява дали Z е изпълнила условието за „изход“ – „escape“ от множеството на Манделброт (дали абсолютната стойност на Z е надвишила 2). Ако е изпълнено условието, то прекратяваме изпълнението на итерациите – точката не е от множеството, и определяме цвета ѝ спрямо броя итерации, които са били необходими за изпълнението на условието за „изход“. Някои точки, отговарят на условието за изход след само няколко итерации, но за други – милиони итерации не са достатъчни. Ето защо получаваме все по-точни изображение на множеството на Манделброт при увеличаване на максималния брой итерации N. Разбира се, точките, принаджещи на множеството, никога няма да изпълнят условието за изход.

Важно е да отбележим, че съществуват и други алгоритми за определяне на точките от множеството на Манделброт. Пример за това е **Mariani-Silver алгоритъмът**, който се основава на едно от свойствата на множеството на Манделброт, а именно, че то е свързано [8]. Не съществува точка, която да принадлежи на множеството на Манделброт и да не бъде свързана с други точки, принадлежащи на множеството. Това означава, че ако имаме област с произволна форма, за която знаем, че границата ѝ напълно принадлежи на множеството на Манделброт, то и всички точки от вътрешността ѝ принадлежат на множеството [7].

По отношение на паралелната обработка, авторът коментира разделянето на изображението на блокове от няколко реда или колони като неефективно, поради неравномерното разпределение между нишките на изчисленията, които трябва да се направят. Той се спира на следното решение: на случаен принцип избира един по един пикселите, които да бъдат обработени от всяка от нишките, игнорирайки недостатъците на това свое решение – не добрата употреба на **Level 1 d-cache**, както и нуждата от доста **повече предварителни изчисления**, преди нишките да могат да започнат своята работа.

Също се коментира, че при по-големи изображение се получават по-оптимални стойности на ускорение. Ето защо в изложената по-долу графика се генерира изображение с размер 10000x10000 пиксела. Въпреки това, постигнатото ускорение, както се и очаква, поради вече споменатите недостатъци на алгоритъма, е доста по-лошо в сравнение с това, постигнато в разгледания в „секция 1.1.1.“ източник.



Фигура 4 – Постигнато ускорение [2]

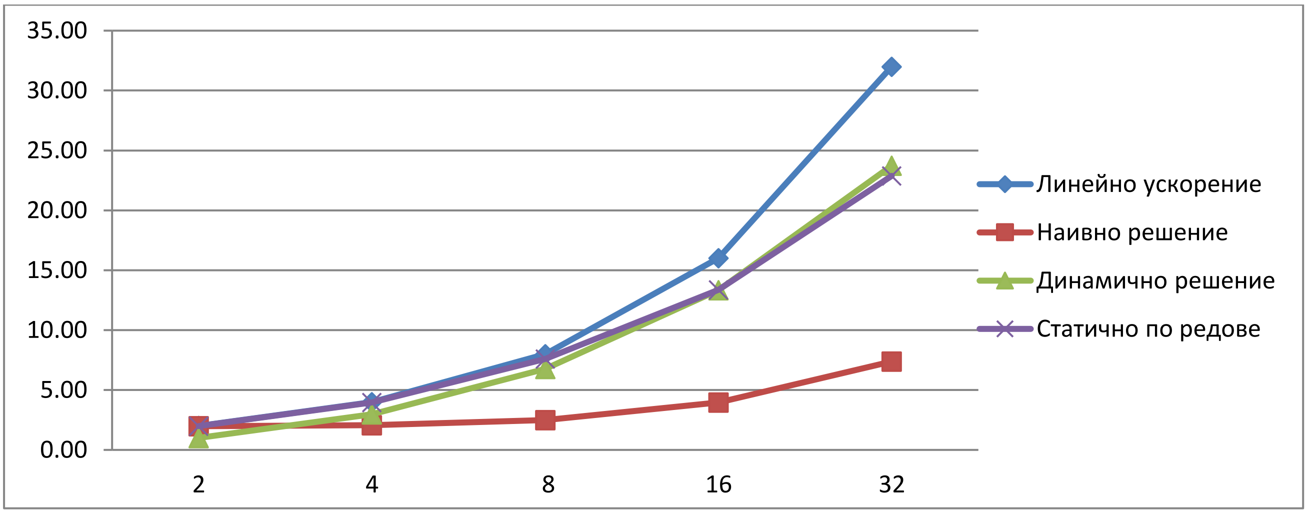
Този алгоритъм свежда комуникацията между нишките до минимум – те си комуникират единствено преди и след извършването на всички изчисления (за разлика от използвания алгоритъм в „секция 1.1.1.“). Това, разбира се, има своята цена – за генерирането на изображения с размер 10000x10000 пиксела е необходима памет от над 400MB.

1.1.3. Bhanuka Manesha Samarasekara Vitharana Gamage and Vishnu Monn Baskaran, Efficient Generation of Mandelbrot Set using Message Passing Interface, Monash University Malaysia, July 2020

Сравнява се ускорението при 3 различни подхода, но общото е, че **винаги има само 1 нишка, която принтира цялото изображение**. Трите представени начина за разделяне на множеството на Манделброт на подзадачи са:

1) **Наивно решение**: Изображението се разделя на блокове с равен брой редове (ако не е възможно, последният блок съдържа и оставащите редове от делението). Всяка нишка получава само 1 блок, който трябва да обработи.
2) **Динамично централизирано решение, следващо принципа First Come First Served**: Изображението, което трябда да се генерира се разделя на множество задачи – всяка задача е 1 ред от изображението. Последователно се дава по 1 ред за всяка slave нишка. Когато master нишката разбере, че дадена slave нишка е изпълнила задачата си, тя ѝ дава следващия неизчислен ред от множеството на Манделброт.
3) **Разпределение, базирано на редове**: Предварително се определя за всяка нишка, кои редове ще трябва да обработи. Така се гарантира, че всички нишки ще обработят по равен брой редове. Ако има оставащи редове, след като всички нишки са получили максимален равен брой задачи за обработка, оставащите редове се дават на master нишката.

Получените резултати от проведените тестове могат да бъдат проследени на следната фигура:



Фигура 5 – Сравнение на полученото ускорение при различните подходи за определяне на подзадачи [3]

Коментар към получените резултати:

Наивното решение постига високо ускорение при 2 нишки – 1.98, тъй като тогава изображението се разделя на 2 еднакви части – частта от множеството на Манделброт „над“ и „под“ реалната ос са огледални една на друга и за изчисляването им се изискват еднакво количество изичисления. Когато обаче започнем да използваме повече нишки, постигнатото ускорение се отдалечава от линейното, тъй като вече работата, която трябва да се извърши от отделните нишки става различна. Това решение постига най-ниски стойности на ускорение.

Динамично централизираното решение не постига ускорение при 2 нишки. Това е напълно очаквано, тъй като master нишката единствено „разпределя“ задачите, а всички те се изпълняват от единствената slave нишка. По този начин, всички необходими изчисления се извършват от само една нишка. Разбира се, при увеличаване броя на нишките, се наблюдава доста по-добро ускорение. При 4 нишки – има 3 slave нишки, които ще осъществяват изчисленията. Ето защо и ускорението е 2.98, което е близко до максималното теоретично ускорение, което можем да очакваме. Може да се обобщи, че това решение за малко на брой нишки постигна слаби резултати, но с увеличавеното на паралелизма, постигнатото ускорение се подобрява чувствително.

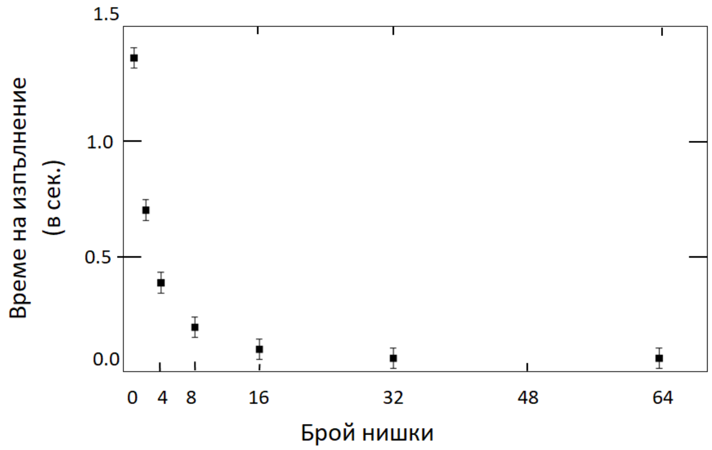
При **разпределението, базирано на редове**, при 2 нишки се наблюдава ускорение, близко до линейното: 1.99. От фигура 5, можем да видим, че това решение, при увеличаване на броя на нишките, запазва подлинейно ускорение, но сравнително близко до линейното. Вижда се тенденция, постепенно, при увеличаване броя на използваните нишки, полученото ускорение да се отдалечава от линейното. Въпреки това, получените резултати са по-добри от тези при наивното решение, тъй като постигнатото разпределение на задачите е много по-балансирано.

1.1.4. Други образци, използвани при изготвянето на проекта

1.1.4.1. Vito Simonka, Estimating potential parallelism and parallelizing of Mandelbrot set with Tareador and OmpSs, Faculty of Natural Science and Mathematics, University of Maribor, Slovenia, September 2013

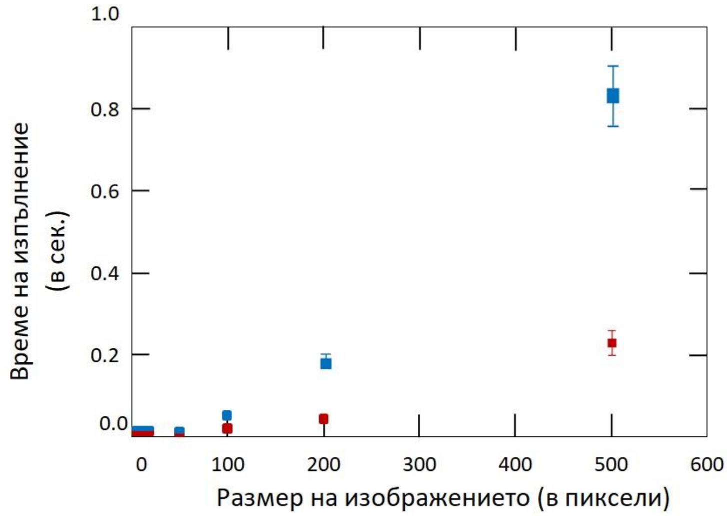
Разбирането на зависимостите на данните е от ключово значение при паралелните алгоритми. Авторът използва Tareador, за да създаде граф, показващ зависимостите на данни при конкретен избор на подзадачи. В статията е представен граф, показващ зависимостите между инициализацията, изчисленията и записването на резултата за даден пиксел.

Също така са представени резултати от тестове, показващи, че времето за изпълнение намаля експоненциално при увеличаване на броя паралелно работещи нишки.



Фигура 6 – Зависимост между времето за изпълнение и броя нишки [4]

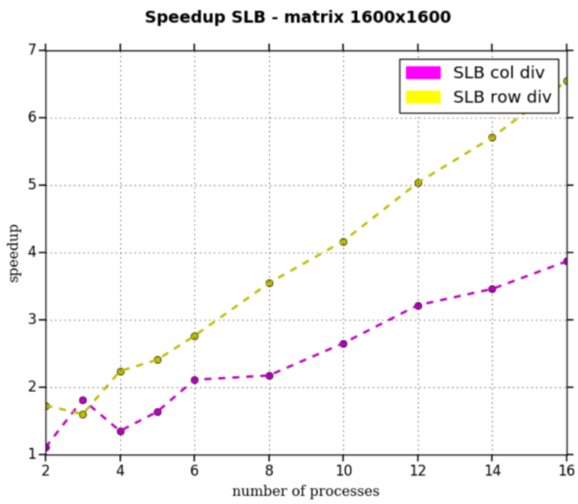
Друго, което се коментира в статията е, че **при увеличаване размера на изображението** на множеството на Манделброт, **времето за изпълнение се увеличава експоненциално** както при последователната, така и при паралелната програма.



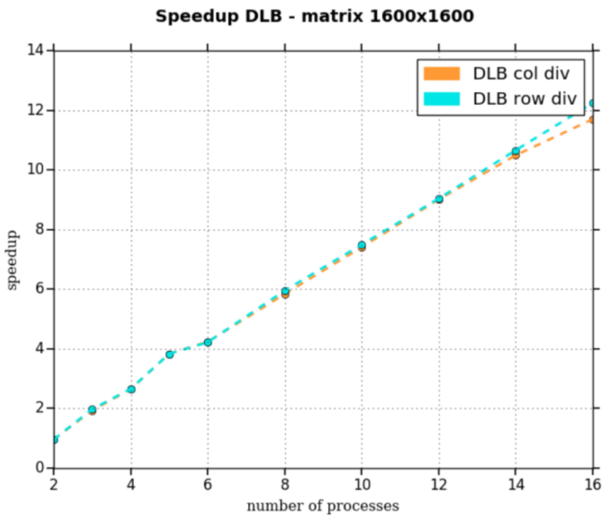
Фигура 7 – Зависимост между времето за изпълнение и размера на изображението [4]

1.1.4.2. Mirco Tracolli, Parallel generation of a Mandelbrot set, Department of Mathematics and Computer Sciences, University of Perugia, April 2016

Представени са 2 подхода за генериране на множеството на Манделброт – статично циклично и динамично централизирано решение. Сравнява се постигнатото ускорение при различни методи за определяне на подзадачите.



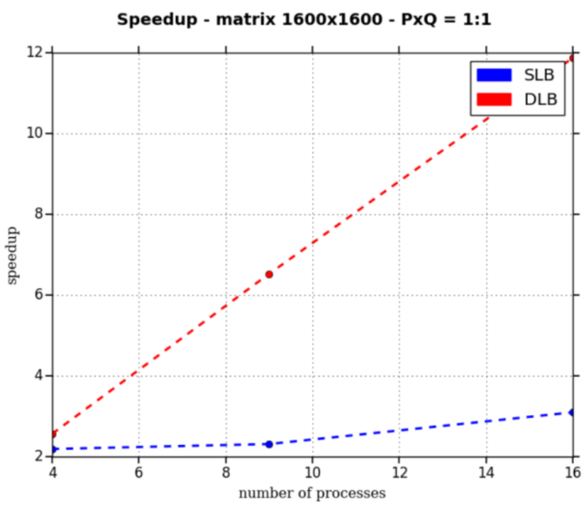
Фигура 8 – Сравнение на ускорението при статично разпределение по редове и по колони [5]



Фигура 9 – Сравнение на ускорението при динамично разпределение по редове и по колони [5]

При статичното разпределение се забелязва, че декомпозицията по редове е много по-ефективна от тази по колони – почти двойно при 16 работещи нишки. Главна причина за това е, че колоните са много по небалансирани в сравнение с редовете при множеството на Манделброт. Тази липса на балансираност при колоните се превъзмогва от динамичното балансиране, тъй като там разпределението на задачите се осъществява по време на изпълнението на програмата.

Друго интересно сравнение в рамките на източника е показано на фигура 10, където са представени резултатите от разделянето на изображнението на подзадачи от отделни пиксели. Виждаме, че статичното балансиране не може да се справи с толкова малки атомарни задачи, за разлика от динамичното, което отново постига приемливи резултати.



Фигура 10 – Сравнение на ускорението при статично и динамично разпределение, ако всяка задача се състои от само 1 пиксел [5]

1.1.4.3. Craig S. Bosma, Parallel Mandelbrot in Julia, C++, and OpenCL, January 2015

Авторът реализира динамично разпределение на задачите за изчисляването на Мандеброт на различни езици и представя получените резултати. Те са най-добри при използването на C++ и затова тук ще включим само тях:

C++: Parallel Speedup

	Speedup	Cores
MBA	3.79	2
MBP	6.98	4
MP	10.16	6

Фигура 11 – Сравнение на полученото ускорение при динамично разпределение на различни тестови среди:

MBA - MacBook Air,

MBP - MacBook Pro,

MP - Mac Pro

Постигнатото ускорение, по-високо от линейното, се аргументира с наличието на хипертрединг, чрез който всяко ядро може да изпълнява до 2 нишки едновременно (което, за съжаление, не е точно това, което се случва на практика, и не можем да очакваме двойно увеличаване на ускорението).

1.1.5. Сравнителна таблица

Образец	максимален паралелизъм	декомпозиция	тип балансиране	грануларност	дълбочина на итерациите	размер на изображението	побитово кодиране
[1]	6	по редове и блокове от редове	статично циклично	максимално едра (1) и фина	5000, 50000 и 500000	5100x6600px	не
[2]	4	по пиксели	статично стохастично	фина	1000	10000x10000px	не
[3]	32	по редове и по блокове от редове	статично циклично и динамично централизирано	максимално едра (1) и фина	2000	8000x8000px	не
[4]	64	по редове	статично циклично	фина	не е посочена	от 100x100 до 600x600px	не
[5]	16	по редове, по колони и пиксели	статично циклично и динамично централизирано	от максимално едра (1) до фина	10000	от 100x100 до 12800x12800px	не
[6]	6	по редове	динамично централизирано	фина	200	3500x2500px	не
ParMandelbrot	16	по редове и по колони	статично циклично и динамично централизирано	от максимално едра (1) до фина	1024	3840x2160px	да

1.2. Технологичен анализ

В тази секция ще представим тестовите среди, които са използвани от описаните в секция „1.1. Функционален анализ“ образци.

Parallel Fractal Image Generation – A Study of Generating Sequential Data With Parallel Algorithms, Matthias Book, The University of Montana, Missoula – Spring Semester 2001

2x 22-core IBM Power9 processors

Level 1 dcache	704 KiB
Level 1 icache	704 KiB
Level 2 Cache	5.5 MiB
Level 3 Cache	110 MiB

Isaac K. Gäng, David Dobson, Jean Gourd and Dia Ali, Parallel Implementation and Analysis of Mandelbrot Set Construction, University of Southern Mississippi, 2008

IMB Power9 Processor 24 cores

Level 1 dcache	768 KiB
Level 1 icache	768 KiB
Level 2 Cache	6 MiB
Level 3 Cache	120 MiB

Bhanuka Manesha Samarasekara Vitharana Gamage and Vishnu Monn Baskaran, Efficient Generation of Mandelbrot Set using Message Passing Interface, Monash University Malaysia, July 2020

Intel Xeon Gold 6150 Processor @ 2.7GHz

Level 1 dcache	576 KiB
Level 1 icache	576 KiB
Level 2 Cache	18 MiB
Level 3 Cache	24.75 MB
Number of Cores / Threads	18 / 36

MacBook Air – 2-core Intel Core i7- -620LM @ 2.0GHz

Level 1 dcache	64 KiB
Level 1 icache	64 KiB
Level 2 Cache	512 MiB
Level 3 Cache	4 MiB
Number of Cores / Threads	2 / 4

MacBook Pro – 4-core Intel Core i7-4960HQ @ 2.6GHz:

Level 1 dcache	128 KiB
Level 1 icache	128 KiB
Level 2 Cache	1 MiB
Level 3 Cache	6 MiB
Number of Cores / Threads	4 / 8

Mac Pro – 6-core Intel Xeon D-1528 @ 2.5GHz

Level 1 dcache	192 KiB
Level 1 icache	192 KiB
Level 2 Cache	1.5 MiB
Level 3 Cache	9 MiB
Number of Cores / Threads	6 / 12

2. Проектиране

2.1. Функционално проектиране

2.1.1. Модел на статичното паралелно приложение

В рамките на проекта са реализирани 3 приложения със статично циклично разпределение на задачите с:

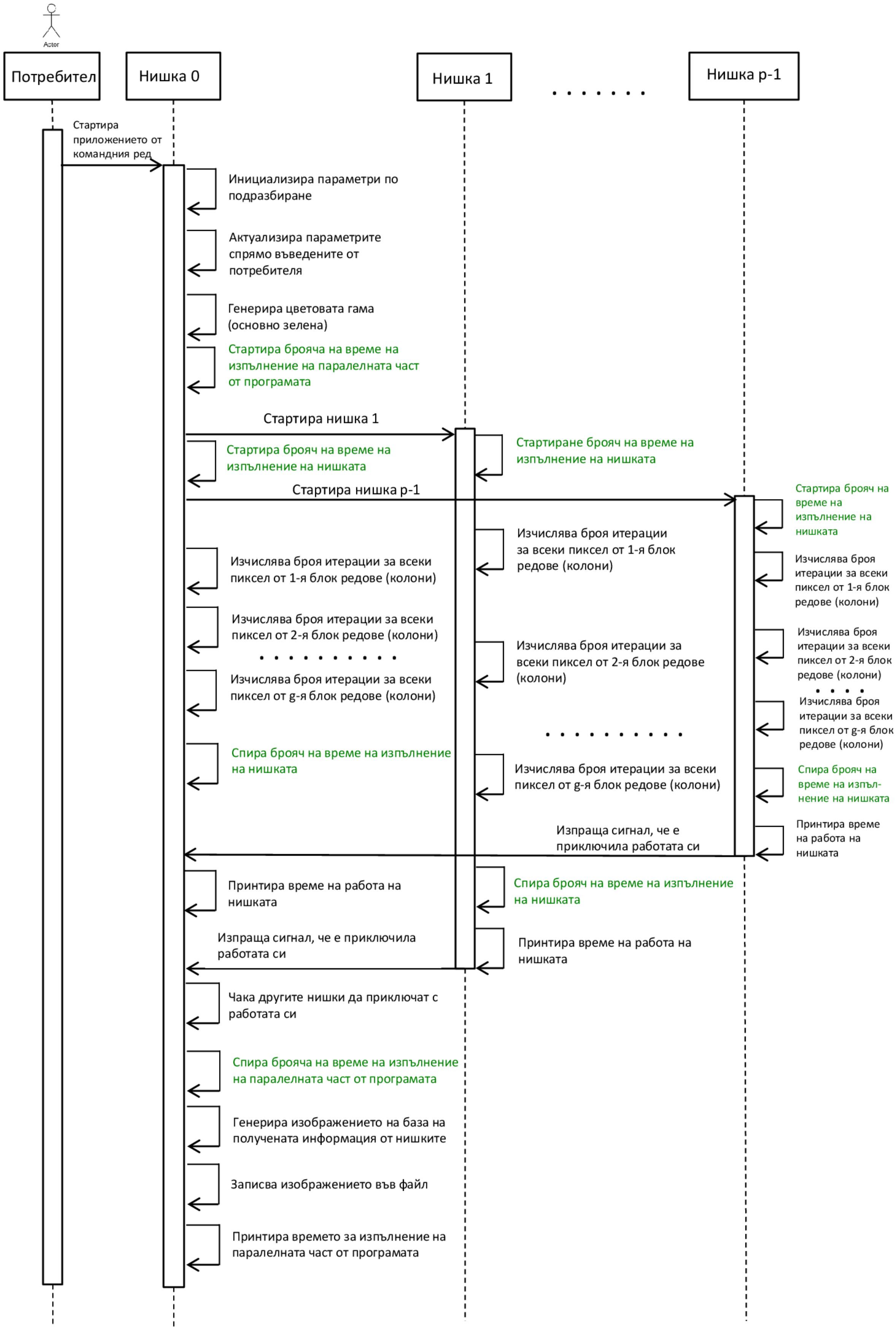
A. декомпозиция по редове;

B. декомпозиция по колони;

C. декомпозиция по колони с побитово кодиране;

Общото между тези приложения е, че главната нишка генерира цветовата палитра, стартира останалите p – 1 нишки, генерира част от множеството на Манделброт, изчаква другите нишки да завършат задачите си и генерира финалното изображение. Другите нишки единствено изчисляват съответните блокове от редове или колони и записват получените резултати – тази дейност се осъществява и от главната нишка 0. Това показва, че използваният модел е **Single Instruction Multiple Data** или **SPMD**. Причината за това е, че както беше посочено още в секция „0. Въведение", множеството на Манделброт е „embarrassingly parallel" проблем, който се поддава изключително лесно на паралелна обработка. Можем да го разделим на подобласти по произволен начин и изчисляването на това кои точки от дадената област принадлежат на множеството по никакъв начин не зависи от другите подобласти. Това означава, че можем да осъществим декомпозиция по данни и различните нишки да обработват различни части от областта по напълно аналогичен начин без нужда от интензивна комуникация и опасност да настъпи съревнование за общи ресурси и мъртва хватка.

За да онагледим по-добре точната последователност от операции, които се извършват при изпълнението на приложението ще представим и диаграма на последователностите:



Фигура 12 – Диаграма на последователностите

Както можем да видим от последователностната диаграма, всяка нишка с изключение на нишка 0, извършва една и съща последователност от действия като тези действия се изпълняват и от нишка 0 наред с другите ѝ задължения. В диаграмата е показано, че нишка p – 1 завършва първа и след това приключва и нишка 1. Важно е да се уточни, че нишките работят независимо една от друга и в зависимост от операционната система последователността, в която завършват, може да варира при различните изпълнения на програмата. Разбира се, от голямо значение за времето на завършване на съответната нишка е и какви задачи или задача е получила да изчислява както ще видим в секция „3. Тестови резултати“.

Различните варианти на приложението, които са реализирани, целят да дадат възможност да изследваме каква грануларност ще бъде най-подходяща, при какъв фиксиран брой задачи приложението постига най-добро ускорение, дали по редове или по колони балансирането е по-успешно и дали използването на побитово кодиране ще намали достатъчно зареждането на данни в Level 1 d-cache-a, за да се подобри постигнатото ускорение.

2.1.2. Модел на динамичното паралелно приложение

Като част от настоящия проект е реализирано приложение за изчисляване на множеството на Манделброт с динамично централизирано разпределение на задачите и декомпозиция по редове. Обособена е главна master нишка, която създава другите нишки (slave нишките), създава опашката от задачи, от където те ще взимат задачи, добавя задачите в опашката, изчаква докато всички задачи от опашката са завършени и генерира крайното изображение на множеството на Манделброт. Slave нишките единствено взимат задачи от опашката със задачи и ги изпълняват т.е. изчисляват броя итерации за всеки пиксел за дадената последователност от редове. Всичко това определя модела на приложението като **Master-Slave**. По този начин се улеснява значително балансираното разпределение на задачите между нишките и приключването на slave нишките в максимално еднакъв момент. За съжаление, този модел носи със себе си и минуси, най-сериозният от които е, че постоянното допитване до master нишката може да доведе до bottlenect или понижаване на постигнатото ускорение от алгоритъма. Друг недостатък е това, че има master нишка, която единствено стартира другите нишки и създава и разпределя задачите. Това означава, че при нисък паралелизъм (2-8 стартирани нишки) ще се наблюдава чувствително по-ниско ускорение в сравнение със статичното разпределение на задачите, тъй като тук имаме нишка, която не изчислява никаква част от финалното изображение.

Master-Slave моделът е подходящ в случаите, когато нови задачи трябва да се добавят в опашката от задачи динамично по време на изпълнението на приложението, а не както тук – всички задачи могат да бъдат създадени и добавени в опашката преди стартирането на другите нишки и реално преди започването на паралелната част от алгоритъма. Изчисляването на множеството на Манделброт НЕ е пример за проблем, който е подходящо да се реши посредством Master-Slave модела. В рамките на проекта е реализирано такова приложение единствено с цел да го сравним с вече представеното в „секция 2.1.1.“ статично приложение и да изследваме дали ще се наблюдава bottleneck (ограничение на ускорението). При реализацията е следван принципът First Come First Served, описан в източника от „секция 1.1.2.“.

2.1.3. Ръководство за потребителя

Всеки от вариантите на приложението се компилира посредством:

```
./makeMe.sh
```

След това, приложението може да се изпълни чрез следното извикване:

```
./runMe.sh <OPTIONS>
```

За улесняване процеса на тестване както и цялостното използване на приложението е добавена възможност за добавяне на командни параметри. На мястото на <OPTIONS> може да стоят 0, 1 или повече от тези параметри. Наличните параметри са:

Кратка опция	Дълга опция	Приема аргумент	Стойност по подразбиране	Описание
-s	--size	Да	3840x2160	Размерът на изображнението, което ще бъде генерирано.
-d	--domain	Да	-2.2:1.2:-1:1	Областта от комплексната равнина, където множеството на Манделброт ще бъде изобразено. Първо въвеждаме началната и крайната точка на областта по X (реалната ос) и след това началната и крайната точка по Y (имагинерната ос), разделени с двуеточие „:“.
-t	--threads	Да	При статично: 1 При динамично: 2	Броят на нишките, които ще бъдат стартирани. При динамично разпределение, броят на slave нишките е с 1 по-малък от въведената стойност.
-o	--output	Да	MandelbrotSetImage.png	Името на файла, които ще съдържа изходното изображение.
-q	--quiet	Не	n/a	Приложението няма да принтира времето за работа на всяка нишка (при статично) и няма да принтира броя задачи, изпълнени от всяка нишка (при динамично)
-r	--rowBlock	Да	1	Броят последователни редове, които ще представляват една отделна задача за дадена нишка. (Параметърът е наличен само при декомпозиция по редове)

-c	--columnBlock	Да	1	Броят последователни колони, които ще представляват една отделна задача за дадена нишка. (Параметърът е наличен само при декомпозиция по колони)
-h	--help	Не	n/a	Принтира информация за наличните параметри.

2.2. Технологично проектиране

2.2.1. Тестови среди

При тестването на програмата бяха използвани следните тестови среди:

Lenovo Z50-70

Architecture	x86_64
CPU op-mode(s)	32-bit, 64-bit
Byte Order	Little Endian
CPU(s)	4 *
Thread(s) per core	2
Model name	Intel(R) Core(TM) i5-4210U CPU @ 1.70GHz
CPU MHz	798.199
CPU max MHz	2700,0000
CPU min MHz	800,0000
L1d cache	32K
L1i cache	32K
L2 cache	256K
L3 cache	3072K

t5600.rmi.yaht.net

Architecture	x86_64
CPU op-mode(s)	32-bit, 64-bit
Byte Order	Little Endian
CPU(s)	32 *
Thread(s) per core	2
Model name	Intel(R) Xeon(R) CPU E5-2660 0 @ 2.20GHz
CPU MHz	1622.741
CPU max MHz	3000,0000
CPU min MHz	1200,0000
L1d cache	32K
L1i cache	32K
L2 cache	256K
L3 cache	20480K

* за да получим броя на реалните физически ядра трябва да разделим числото на 2 (съответно получаваме 2 и 16 ядра за гореописаните тестови среди), тъй като използвана команда (**lscpu**) извежда броят на физическите ядра, умножен по коефициента 2, поради използването на хипертрединг.

Представените резултати в секция 3 са единствено от тестовете, които са проведени на t5600.rmi.yaht.net, тъй като тя дава възможност да експлоатираме в пъти по-висок паралелизъм.

2.2.2. Използван програмен език

За разработването на всички варианти на приложението е използван програмният език **Java**: статично циклично разпределение по редове и по колони, статично циклично разпределение по колони с побитово кодиране и динамично централизирано разпределение с декомпозиция по редове.

Използвани са 2 външни библиотеки:

A. commons-math3-3.6.1.jar

Тази библиотека позволява да използваме готови функции за умножение, степенуване и намиране на абсолютна стойност на комплексни числа.

B. commons-cli-1.4.jar

Тази библиотека дава възможност за по-лесна работа с и обработка на входните параметри на приложението.

2.2.3. Стартиране и управление на използваните нишки

A. Статично циклично разпределение

Представеният по-долу код показва как първата нишка (на практика нишката с номер 0) създава останалите numThreads-1 нишки.

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

След като другите нишки са стартирани, нишка 0 също започва да смята тази част от изображението, която се пада на нея:

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

След като завърши работата си, изпълнявайки последователно

```
threads[i].join();
```

за всяка от стартираните нишки, тя ги изчаква да приключат своите задачи и генерира крайното изображение.

B. Динамично централизирано разпределение

Показаният по-долу код представя как се създава нов басейн (пул) от нишки:

```
ThreadPool threadPool = new ThreadPool(allSlavesCount, allTasksCount);
```

Този ред довежда до изпълнението на следните 2 for цикъла:

```
for (int i = 0; i < allSlavesCount; i++) {
        this.runnables.add(new ThreadPoolRunnable(this.taskQueue, i));
}
for (ThreadPoolRunnable runnable : this.runnables) {
        new Thread(runnable).start();
}
```

Първият цикъл отговаря за добавянето на allSlavesCount на брой нишки към списъка с нишки, които ще взимат задачи от байсена, а вторият цикъл – стартира нишките, които вече са добавени в този списък.

Добавянето на задачи към басейна (пул-а) със задачи се онагледява посредством:

```
for (int currTask = 0; currTask < allTasksCount; currTask++) {
        threadPool.execute(() -> {
                // определяме коя част от изображението трябва да бъде
                // обработена в рамките на задачата

                incrementTaskCount(); // като последна стъпка от изпълнението
                // на задачата, съответната нишка ще актуализира броя на
                // завършените задачи.
        });
}
```

Master нишката изчаква всички задачи да бъдат изпълнени:

```
while (finishedTasks < allTasksCount) {
        Thread.sleep(20);
}
```

В края на програмата, Master нишката инициира унищожаването на създадените slave нишки посредством:

```
threadPool.destroyThreads();
```

3. Тестови резултати

При всички представени по-долу тестове, използваната област е от -2.2 до 1.2 по X и от -1 до 1 по У.

В таблиците, показващи тестовия план, са използвани следните означения:

#	Уникален идентификационен номер на тестовия случай
p	Брой стартирани нишки
g	Грануларност
N	Общ брой задачи, които ще се изпълнят от всички нишки
$T_p^{(i)}$	Времето за изпълнение на приложението в **милисекунди** при **i**-тото стартиране с **p** на брой нишки.
T_p=min()	Минималното време за изпълнение на програмата при **p** стартирани нишки
$S_p = T_1/T_p$	Speedup или постигнатото ускорение, което получаваме като разделим времето за изпълнение на програмата с 1 нишка на времето за изпълнение при **p** стартирани нишки

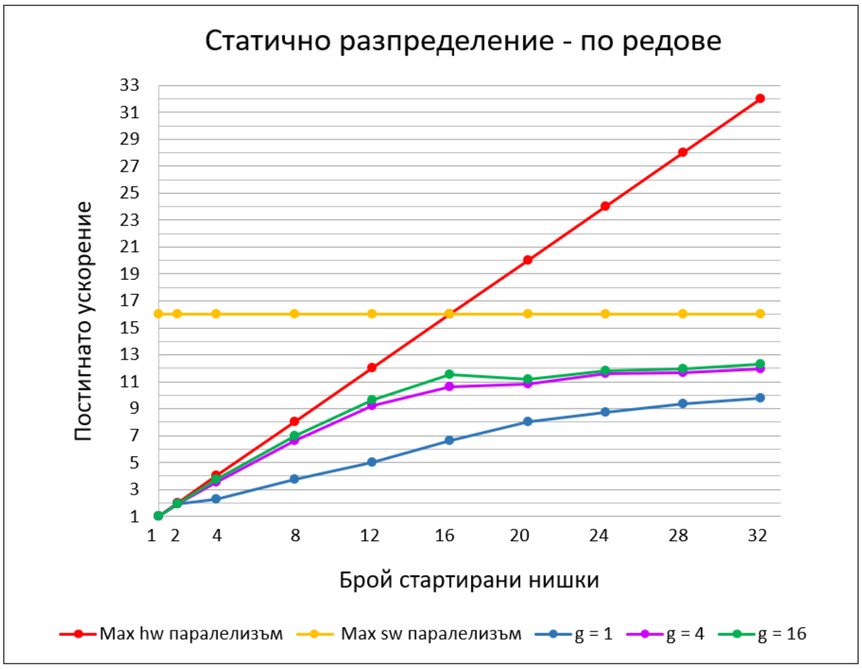
3.1. Сравняване на ускорението при статично циклично разпределение при грануларност g = 1, 4, 16

В тази секция са представени резултатите от тестване на приложението с декомпозиция по редове и колони с цел да намерим оптимална грануларност т.е. брой задачи, които да се дават на всяка от стартираните нишки.

3.1.1. Декомпозиция по редове

#	*p*	*g*	*N*	$T_1^{(1)}$	$T_1^{(2)}$	$T_1^{(3)}$	T_1=min()	$T_p^{(1)}$	$T_p^{(2)}$	$T_p^{(3)}$	T_p = min()	$S_p = T_1/T_p$
1	2	1	2	38793	37861	38617	37861	19973	20120	19805	19805	1.91
2	4		4					16448	16562	16763	16448	2.30
3	8		8					10123	10027	10529	10027	3.78
4	12		12					7620	7539	7742	7539	5.02
5	16		16					5779	6186	5714	5714	6.63
6	20		20					5683	4714	4897	4714	8.03
7	24		24					4613	4328	4502	4328	8.75
8	28		28					4121	4247	4037	4037	9.38
9	32		32					3892	3866	3880	3866	9.79
10	2	4	8	39180	38431	38999	38431	20042	19903	19877	19877	1.93
11	4		16					10831	10961	10874	10831	3.55
12	8		32					6124	5821	5897	5821	6.60
13	12		48					4180	4465	4193	4180	9.19
14	16		64					3624	3662	3654	3624	10.60

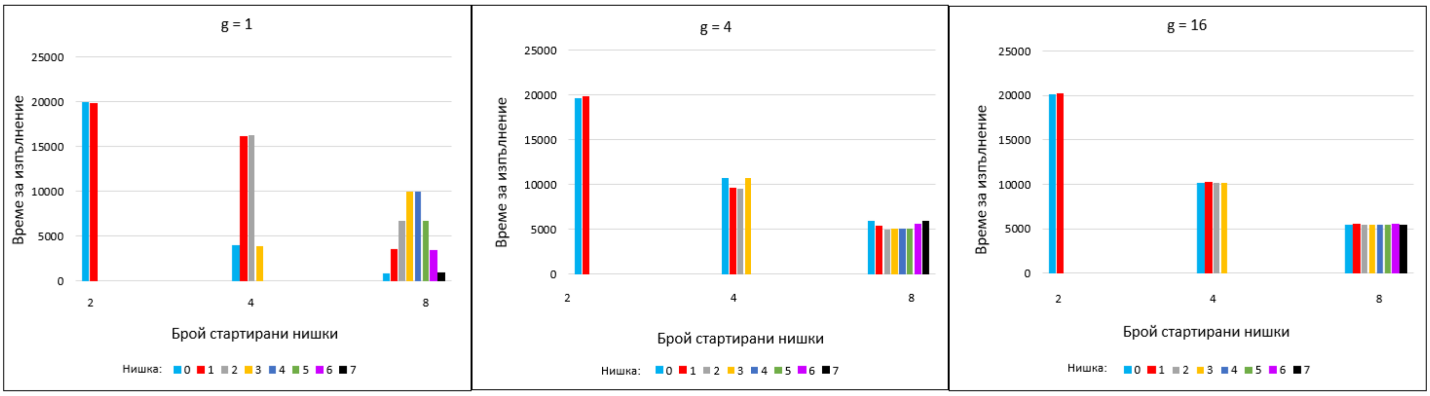
15	20		80					3717	3552	3582	3552	10.82
16	24		96					3304	3391	3370	3304	11.63
17	28		112					3290	3304	3288	3288	11.69
18	32		128					3273	3212	3248	3212	11.96
19	2	16	32	38219	38660	38237	38219	20255	20051	20004	20004	1.91
20	4		64					10213	10508	10484	10213	3.74
21	8		128					5471	5506	5496	5471	6.99
22	12		192					3955	3992	3989	3955	9.66
23	16		256					3640	3320	3403	3320	11.51
24	20		320					3425	3459	3608	3425	11.16
25	24		384					3238	3440	3320	3238	11.80
26	28		448					3250	3244	3194	3194	11.97
27	32		512					3106	3227	3194	3106	12.30



Фигура 13 – Сравнение на ускорението при статично циклично разпределение, декомпозиция по редове и g = 1, 4, 16

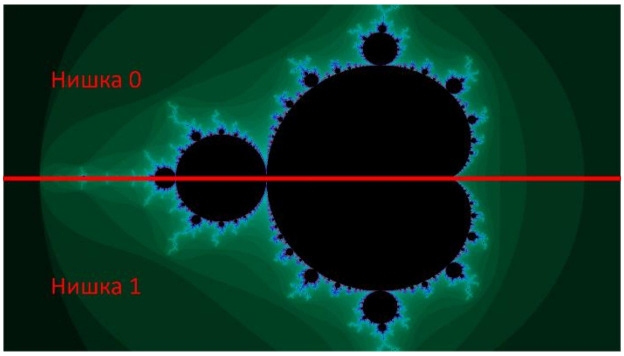
Това, което се забелязва веднага е, че при грануларност 1 постигнатото ускорение е доста по-ниско. Сравнявайки ускоренията, получени при g = 4 и g = 16, виждаме че макар и да са близки, ускорението при g = 16 е винаги малко по-високо. Причината за това е, че като увеличаваме броя задачи, които получава всяка нишка, се постига по-добро балансиране на изчисленията, които трябва да направи всяка нишка. При g = 16 задачите са едновременно достатъчно много, за да получим близко до оптималното балансиране и същевременно всяка задача се състои от достатъчно реда, така че да не е необходимо постоянно презаписване на стойности в Level 1 d-cache-a.

Логично е при увеличаване на броя на задачите, поемани от всяка нишка, да се подобрява балансирането на необходимите изчисления и по този начин нишките да свършват в почти еднакво време. Такава тенденция ясно се забелязва и на фигура 14:

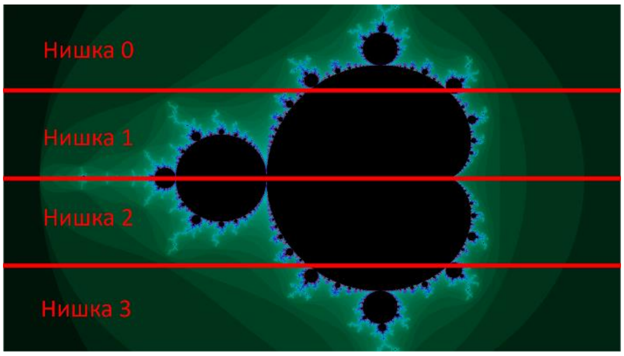


Фиг 14 – Време за изпълнение (в милисекунди) на нишките при различна грануларност при декомпозиция по редове.

Това, което прави впечатление е, че при 2 нишки дори при грануларност 1 се постига балансирано разпределение на работата между 2-те нишки, а когато стартираме още 2 нишки – т.е. изпълним приложението с 4 нишки и грануларност 1, то получаваме ускорение, което НЕ е много по-добро спрямо това при 2 нишки, грануларност 1. Причината за това се крие в начина, по който разделяме работата между нишките:



Фиг 15 – Разделяне на множеството на Манделброт при декомпозиция по редове, 2 нишки и грануларност 1

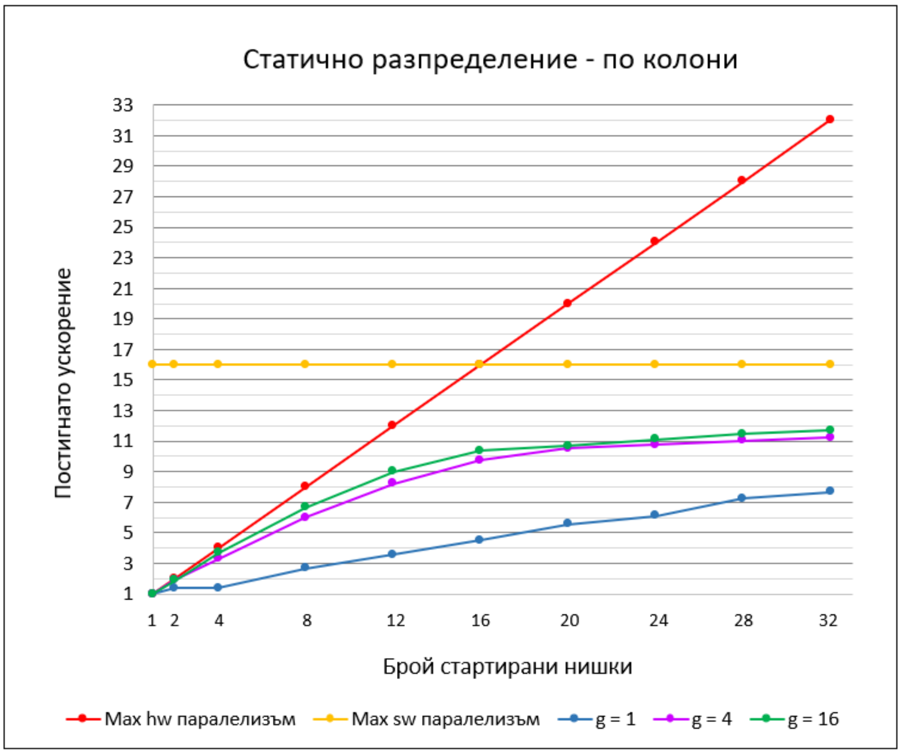


Фиг 16 – Разделяне на множеството на Манделброт при декомпозиция по редове, 4 нишки и грануларност 1

Както виждаме от фигура 15 – множеството на Манделброт е огледално спрямо реалната ос и когато го разделим на 2 равни части – горната и долната половина са напълно единтични. Ето защо двете нишки завършват по едно и също време. Когато обаче добавим още 2 нишки, то полученият резултат е далеч от желания – както се вижда от фигура 16, нишка 0 и нишка 3 получават много малка част от точките, които са част от множеството и следователно завършват работата си много по-бързо от нишки 1 и 2. Така въпреки, че буквално са се умножили стартираните нишки по 2, то полученото ускорение се увеличава едва от 1.91 на 2.30.

3.1.2. Декомпозиция по колони

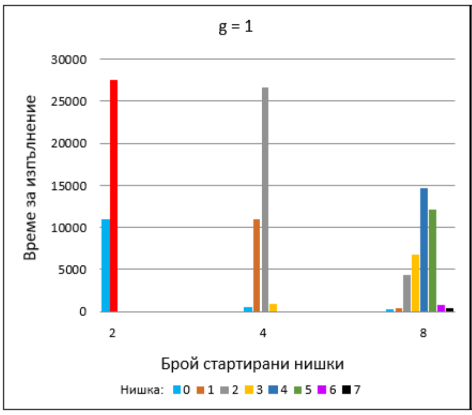
#	*p*	*g*	*N*	$T_1^{(1)}$	$T_1^{(2)}$	$T_1^{(3)}$	T_1=min()	$T_p^{(1)}$	$T_p^{(2)}$	$T_p^{(3)}$	T_p = min()	$S_p = T_1/T_p$
28	2	1	2	37784	38943	37531	37531	27270	28115	27025	27025	1.39
29	4		4					27218	27124	26945	26945	1.39
30	8		8					14072	14211	14516	14072	2.67
31	12		12					10744	10723	10490	10490	3.58
32	16		16					8297	8361	8407	8297	4.52
33	20		20					6769	6910	6862	6769	5.54
34	24		24					6320	6139	6496	6139	6.11
35	28		28					5193	5255	5282	5193	7.23
36	32		32					4890	5140	5050	4890	7.68
37	2	4	8	37463	37344	37435	37344	19696	19378	19546	19378	1.93
38	4		16					11281	11287	11309	11281	3.31
39	8		32					6456	6242	6534	6242	5.98
40	12		48					4636	4554	4589	4554	8.20
41	16		64					3950	4106	3836	3836	9.74
42	20		80					3656	3743	3547	3547	10.53
43	24		96					3623	3544	3465	3465	10.78
44	28		112					3457	3386	3509	3386	11.03
45	32		128					3372	3374	3325	3325	11.23
46	2	16	32	37450	37531	37596	37450	20505	20055	19716	19716	1.90
47	4		64					10484	10122	10505	10122	3.70
48	8		128					6394	5690	5637	5637	6.64
49	12		192					4152	4298	4271	4152	9.02
50	16		256					3613	3708	3788	3613	10.37
51	20		320					3636	3502	3646	3502	10.69
52	24		384					3387	3368	3422	3368	11.12
53	28		448					3274	3282	3270	3270	11.45
54	32		512					3224	3253	3205	3205	11.68

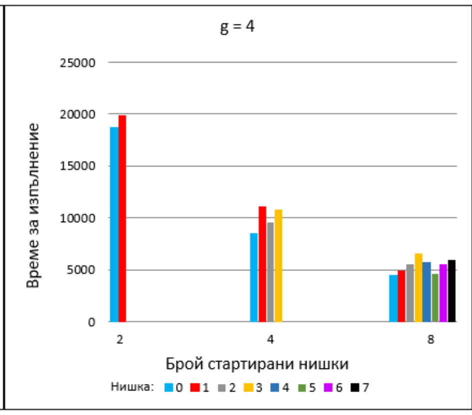


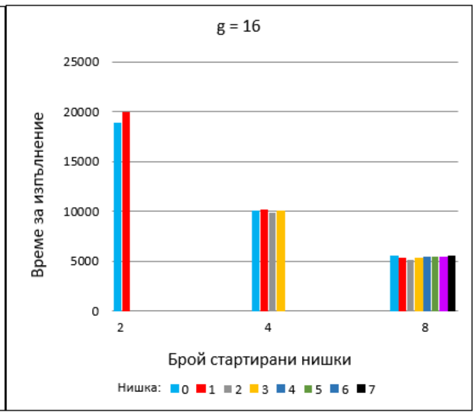
Фигура 17 – Сравнение на ускорението при статично циклично разпределение, декомпозиция по колони и g = 1, 4, 16

Забелязва се, че при грануларност 1 отново получаваме най-ниско ускорение, а ускоренията при g = 4 и g = 16 са близки, но това при g = 16, т.е. когато всяка от стартираните нишки получава по 16 задачи, които трябва да пресметне, води до резултати, които са малко по-добри. Главната причина за това е, че се намира баланс между броя задачи, които получава всяка нишка и техния размер – така едновременно задачите са достатъчно много, за да са разпределени изчисленията приблизително по равно и същевременно, задачите не са прекалено малки, за да се отрази това на Level 1 d-cache-a.

Точно както и при декомпозицията по редове, тук виждаме, че при увеличаване на грануларността – времето, нужно на всяка нишка, за да извърши съответните изчисления, се изравнява:

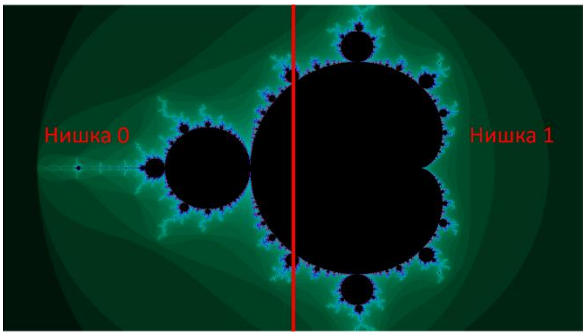




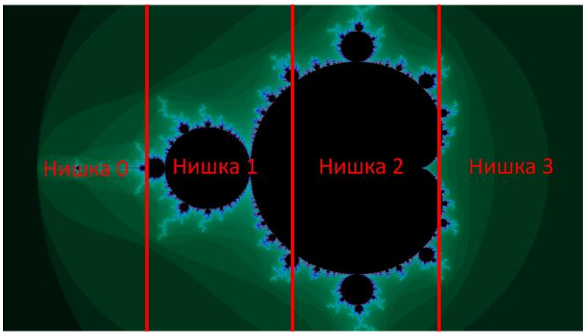


Фиг 18 – Време за изпълнение (в милисекунди) на нишките при различна грануларност при декомпозиция по колони.

Това, което прави впечатление е, че при g = 1, времето за изпълнение на нишките е изключително небалансирано – дори може да се каже, че при 4 стартирани нишки, 2 почти не са работили, а при 8 стартирани – 4 почти не са осъществявали пресмятания. Това се дължи на начина, по който разделяме множеството на Манделброт на подобласти, които даваме на нишките да изчисляват:



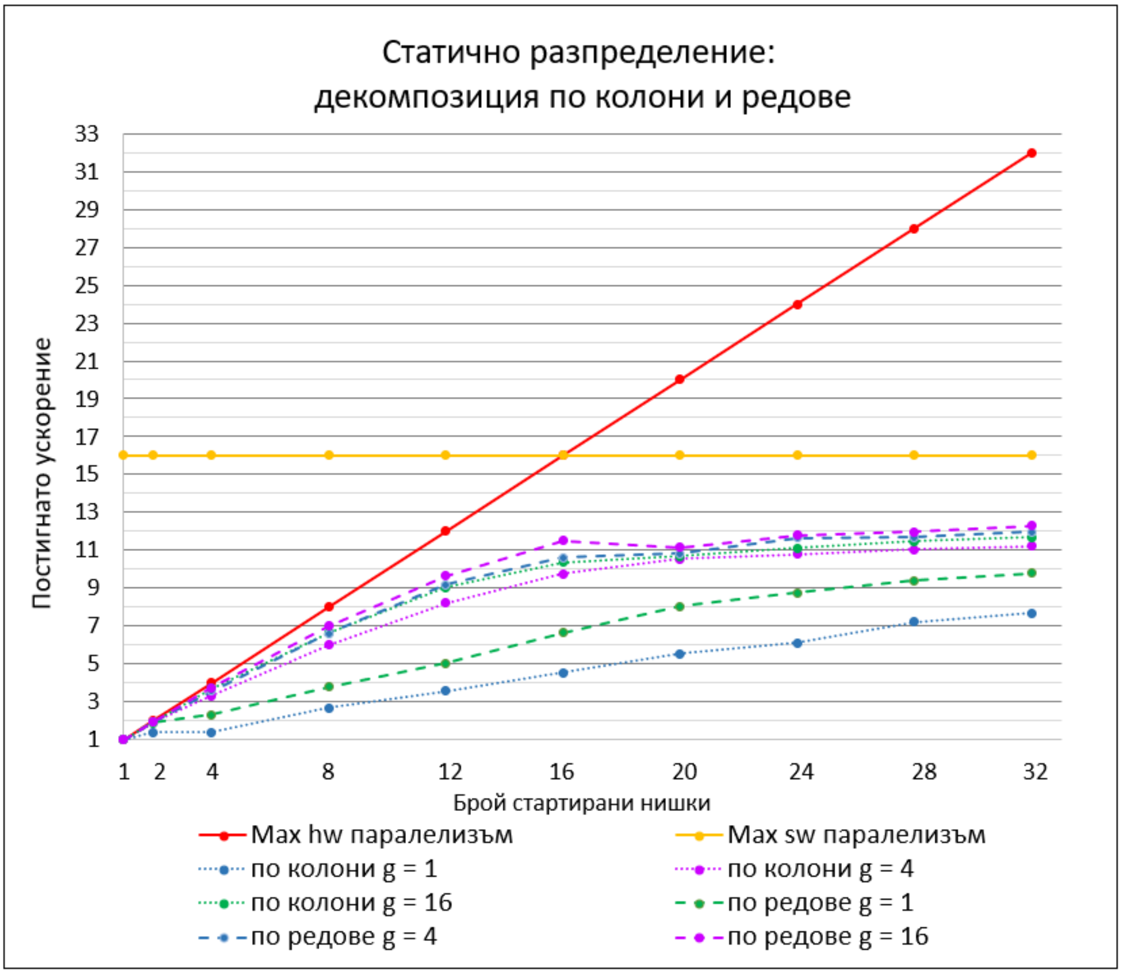
Фиг 19 – Разделяне на множеството на Манделброт при декомпозиция по колони, 2 нишки и грануларност 1



Фиг 20 – Разделяне на множеството на Манделброт при декомпозиция по колони, 4 нишки и грануларност 1

На фигура 19 ясно се вижда, че нишка 1 получава повече от 2/3 от точките в множеството на Мандеброт, което и се отразява върху времето за изпълнение на нишката. Това е причината тук да наблюдаваме доста лошо ускорение – 1.39. Ситуацията става дори по-лоша, когато стартираме 4 вместо 2 нишки – нишка 0 и нишка 3 почти не получават точки от множеството (фигура 20) и завършват изключително бързо своята работа. Същевременно ако сравним изчисленията, които трябва да се направят от нишка 1 и нишка 2 със съответните изчисления на нишка 0 и нишка 1 от фигура 19 (когато са стартирани само 2 нишки), то виждаме, че работата им е почти една и съща. Ето защо, въпреки че сме стартирали 4 нишки вместо 2, то подобрение на ускорението на практира няма.

3.1.3. Съпоставяне на получените резултати



Фигура 21 – Съпоставяне на полученото ускорение при статично циклично разпределение, декомпозиция по редове и колони и гранулярност g = 1, 4, 16

Постигнатите резултати са по-добри при декомпозиция по редове. Това най-вече си пролѝчава при гранулярност 1. Причината за това е начинът, по който разделяме множеството на Манделброт за различните нишки, както беше показано в „секция 3.1.1.“ и „секция 3.1.2.“. Подобна тенденция, макар и не толкова ярко изразена, се забелязва и при гранулярности 4 и 16. Прави впечатление, че ускорението при разделяне по редове при гранулярност 4 (което е по-бавно от по редове и гранулярност 16) постига по-добро ускорение от декомпозиция по колони както при гранулярност 4, така и гранулярност 16. Причината и тук може би е това, което се твърди в източника от „секция 1.1.4.2“, а именно, че при множеството на Манделброт редовете са по-добре балансирани в сравнение с колоните. Въпреки това, е важно да се отбележи, че съществува хипотеза за гранулярност 4 и 16, че по-доброто ускорение по редове може да се дължи на организацията на Level 1 d-cache-а, както и на хипертрединга, който е възможно тук да се възползва от по-малко зависимости на ниво инструкция.

Като обобщение и извод на тестовете, проведени в „секция 3.1.“ можем да кажем, че най-оптималната грануларност при статично циклично разпределение е 16, не зависимо дали декомпозицията е по редове или по колони. Също така другият извод е, че декомпозицията по редове постига по-добри резултати от тази по колони.

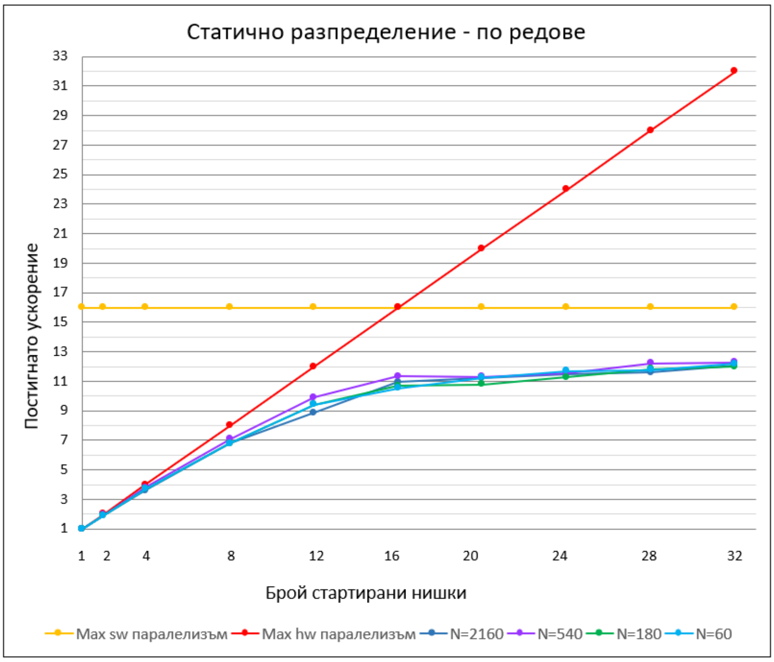
3.2. Сравняване на ускорението при статично циклично разпределение при фиксиран брой задачи N

Целта на тази част от тестовия план е да се намери оптимален размер на задачите, при който приложението да използва Level 1 d-cache-а по възможно най-оптимален начин.

3.2.1. Декомпозиция по редове – N = 2160, 540, 180, 60

#	*p*	*g*	*N*	$T_1^{(1)}$	$T_1^{(2)}$	$T_1^{(3)}$	T_1=min()	$T_p^{(1)}$	$T_p^{(2)}$	$T_p^{(3)}$	T_p = min()	$S_p = T_1/T_p$
55	2	1080	2160	37809	38287	38861	37809	20142	19920	20492	19920	1.90
56	4	540						10526	10911	10701	10526	3.59
57	8	270						5571	5926	5727	5571	6.79
58	12	180						4860	5423	4272	4272	8.85
59	16	135						3456	3496	3710	3456	10.94
60	20	108						3446	3426	3370	3370	11.22
61	24	90						3351	3318	3295	3295	11.47
62	28	18						3333	3282	3262	3262	11.59
63	32	68						3131	3164	3210	3131	12.08
64	2	270	540	38711	38556	38672	38556	20225	20644	19997	19997	1.93
65	4	135						10288	10150	10497	10150	3.80
66	8	68						5555	5521	5454	5454	7.07
67	12	45						4199	4211	3895	3895	9.90
68	16	34						3512	3726	3401	3401	11.34
69	20	27						3480	3415	3488	3415	11.29
70	24	23						3440	3526	3342	3342	11.54
71	28	20						3322	3156	3301	3156	12.22
72	32	17						3181	3265	3136	3136	12.29
73	2	90	180	37965	38252	37842	37842	20103	20024	20183	20024	1.89
74	4	45						10447	10269	10322	10269	3.69
75	8	23						5591	5824	5941	5591	6.77
76	12	15						4288	4022	4024	4022	9.41
77	16	12						3582	3538	3594	3538	10.70
78	20	9						3645	3507	3560	3507	10.79
79	24	8						3355	3359	3371	3355	11.28
80	28	7						3263	3201	3220	3201	11.82
81	32	6						3246	3160	3216	3160	11.98
82	2	30	60	38027	38198	37925	37925	19909	21490	20604	19909	1.90

83	4	15						10563	10244	11126	10244	3.70
84	8	8						5592	5657	5623	5592	6.78
85	12	5						4027	4089	4067	4027	9.42
86	16	4						3654	3605	3778	3605	10.52
87	20	3						3601	3538	3377	3377	11.23
88	24	3						3367	3365	3483	3365	11.27
89	28	3						3227	3221	3261	3221	11.77
90	32	2						3205	3345	3116	3116	12.17

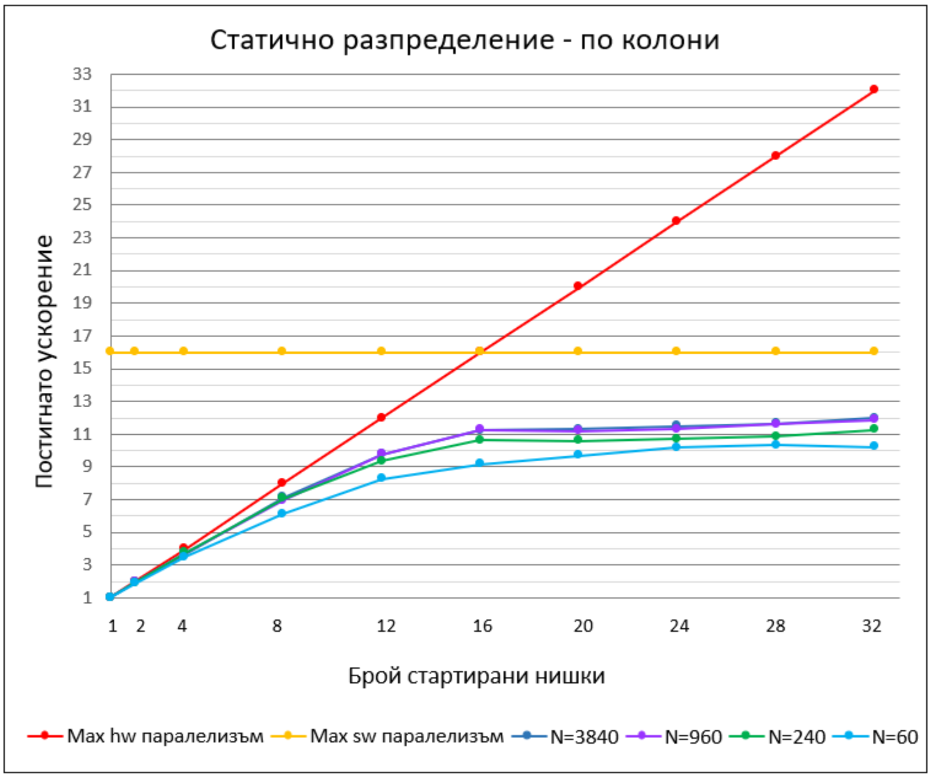


Фигура 22 – Сравнение на постигнатото ускорение при декомпозиция по редове и фиксиран брой задачи

Фигура 22 не разкрива ярка тенденция и възможност за конкретни сигурни изводи, тъй като постигнатите резултати са следствие от емпирични тестове. Въпреки това, внимателното изследване на данните показва, че при фиксиран брой на задачите N = 540 (т.е. всяка задача е 4 последователни реда), постигнатите резултати са малко по-добри, независимо от броя стартирани нишки. Също така при N = 60 (когато всяка задача е 36 последователни реда), т.е. когато задачите са доста по-малко на брой, забелязваме, че резултатите често са по-ниски. Причината за сравнително близките резултати, независимо от броя задачи, на които се разделя изследваната област, най-вероятно се крие в това, че се компенсира по-доброто балансиране на задачите при по-големи стойности на N от това, че алгоритъмът става по-неблагоприятен за Level 1 d-cache-а, тъй като задачите, които се дават на нишките се състоят от по-малко последователни редове.

3.2.2. Декомпозиция по колони – N = 3940, 960, 240, 60

#	*p*	*g*	*N*	$T_1^{(1)}$	$T_1^{(2)}$	$T_1^{(3)}$	T_1=min()	$T_p^{(1)}$	$T_p^{(2)}$	$T_p^{(3)}$	T_p = min()	$S_p = T_1/T_p$
91	2	1920	3840	37786	40005	37884	37786	19790	20111	19355	19355	1.95
92	4	960						10275	10234	10364	10234	3.69
93	8	480						5283	5657	5589	5283	7.15
94	12	320						3872	3874	3947	3872	9.76
95	16	240						3495	3359	3418	3359	11.25
96	20	192						3363	3344	3373	3344	11.30
97	24	160						3378	3284	3338	3284	11.51
98	28	138						3370	3283	3246	3246	11.64
99	32	120						3151	3175	3198	3151	11.99
100	2	480	960	38024	37406	37378	37378	19865	19261	20067	19261	1.94
101	4	240						10189	10316	10014	10014	3.73
102	8	120						5393	5470	5387	5387	6.94
103	12	80						4002	4091	3826	3826	9.77
104	16	60						3787	3376	3319	3319	11.26
105	20	48						3339	3386	3490	3339	11.19
106	24	40						3373	3365	3302	3302	11.32
107	28	35						3222	3251	3240	3222	11.60
108	32	30						3216	3232	3149	3149	11.87
109	2	120	240	37380	37580	37543	37380	19387	20554	19964	19387	1.93
110	4	60						10346	10337	9938	9938	3.76
111	8	30						5538	5292	5577	5292	7.06
112	12	20						4147	4152	3990	3990	9.37
113	16	15						3518	3639	3535	3518	10.63
114	20	12						3528	3621	3524	3524	10.61
115	24	10						3525	3492	3571	3492	10.70
116	28	9						3562	3446	3493	3446	10.85
117	32	8						3347	3312	3324	3312	11.29
118	2	30	60	37481	37178	37311	37178	19950	19746	20253	19746	1.88
119	4	15						10607	10606	10828	10606	3.51
120	8	8						6064	6233	6194	6064	6.13
121	12	5						4488	4506	4666	4488	8.28
122	16	4						4070	4141	4058	4058	9.16
123	20	3						3833	3931	3967	3833	9.70
124	24	3						3646	3682	3652	3646	10.20
125	28	3						3593	3858	3718	3593	10.35
126	32	2						3638	3730	3799	3638	10.22



Фигура 23 – Сравнение на постигнатото ускорение при декомпозиция по колони и фиксиран брой задачи

Както можем да видим от фигура 23, най-добро ускорение – почти едно и също, получаваме при фиксиран брой задачи N = 3840 и N = 960, т.е. когато всяка задача се състои съответно от само 1 колона и 4 последователни колони. Забелязва се, че резултатите при N = 240 (всяка задача представлява 16 последователни колони) са по-лоши и най-ниско ускорение се получава при N = 60, когато всяка задача е 64 последователни колони. Причината за наблюдаваните резултати най-вероятно се крие в по-лошото балансиране на времето за изпълнение на различните нишки при малки стойности на N (60 и 240).

Друго, което можем да видим от фигура 23, е, че след 16 нишки, ускорението почти не се подобрява – например при N = 3840 при 16 нишки ускорението е 11.25, а при 32 нишки – 11.99. Причината за това е, че на машината, където са проведени тестовете, са налични само 16 физически ядра и стартирайки повече от 16 нишки, ние се надяваме да получим ускорение, поради хипертрединга – т.е. това, че всяко ядро може да изпълнява до 2 нишки едновременно, но както може да се убедим тук – хипертрединга рядко води до високо ускорение. Характерното за хипертрединга ускорение от 5-10% се наблюдава и тук – разликата между 11.25 и 11.99 е 6,67%.

3.2.3. Съпоставяне на получените резултати

Сравнявайки фигура 22 и фигура 23 виждаме, че резултатите, получени при декомпозиция по редове отново са по-добри, както беше посочено в „секция 3.1.3.“.

Прави впечатление, че резултатите, получени при различни стойности на N при декомпозиция по редове са доста по близи от тези, получени при различните стойности на N при декомпозиция по колони. Причината за това най-вероятно е казаното в източника от „секция 1.1.4.2“ – по редове множеството на Манделброт е много по-добре балансирано от по колони. Разбира се, възможно е причината да се крие и в Level 1 d-cache-а или хипертрединга, тъй като по редове може да има по-малко зависимости на ниво инструкция.

В заключение на тази секция можем да кажем, че най-добро ускорение – 12.3, при статично циклично разпределение се получава при декомпозиция по редове и по-конкретно успяваме да го получим в 2 случая - при фиксиран брой задачи N = 540, 32 стартирани нишки и при грануларност g = 16 и отново 32 стартирани нишки. Възможно е тези 2 случая на пръв поглед за изглеждат напълно отделни, но всъщност при първия случай: N = 540, то всяка задача се състои от 4 последователни реда, а при g = 16 и 32 стартирани нишки, всяка задача ще представлява 5 последователни реда. Така получаваме, че приложението работи най-оптимално, когато множеството на Манделброт се разделя на подобласти от 4 или 5 последователни реда. Вероятната причина за тези резултати е именно организацията на Level 1 d-cache-а.

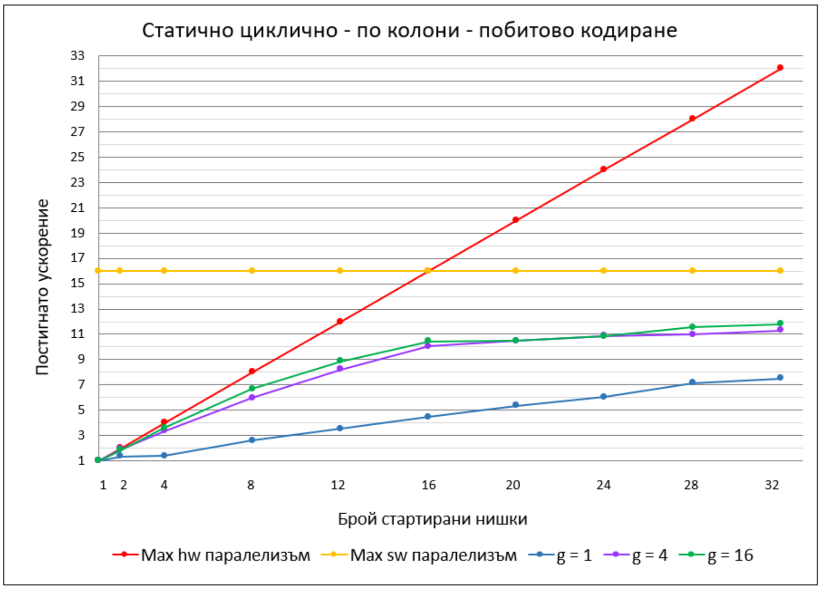
3.3. Сравняване на ускорението при статично циклично разпределение с декомпозицт по колони и побитово кодиране

В тази секция ще представим получените резултати при използването на побитово кодиране. Избрана е декомпозиция по колони, тъй като видяхме, че тя дава по-лоши резултати и по този начин ще проверим дали можем да подобрим тези резултати т.е. наблюдаваното ускорение като използваме побитово кодиране.

3.3.1. При грануларност g = 1, 4, 16

#	p	g	N	$T_1^{(1)}$	$T_1^{(2)}$	$T_1^{(3)}$	T_1=min()	$T_p^{(1)}$	$T_p^{(2)}$	$T_p^{(3)}$	T_p = min()	$S_p = T_1/T_p$
127	2	1	2	37185	36955	37019	36955	27034	27220	27398	27034	1.37
128	4		4					27443	26867	26664	26664	1.39
129	8		8					14233	14546	14495	14233	2.60
130	12		12					10565	10700	10435	10435	3.54
131	16		16					8422	8245	8313	8245	4.48
132	20		20					6949	6894	6901	6894	5.36
133	24		24					6119	6374	6172	6119	6.04
134	28		28					5171	5387	5164	5164	7.16
135	32		32					4929	5088	4961	4929	7.50
136	2	4	8	37198	37207	37279	37198	19634	19404	19624	19404	1.92
137	4		16					11118	11326	11422	11118	3.35
138	8		32					6491	6461	6221	6221	5.98

139	12		48					4563	4530	4569	4530	8.21
140	16		64					3824	3697	3822	3697	10.06
141	20		80					3550	3618	3647	3550	10.48
142	24		96					3569	3497	3425	3425	10.86
143	28		112					3383	3443	3495	3383	11.00
144	32		128					3293	3461	3302	3293	11.30
145	2	16	32	37544	39427	37407	37407	20389	20115	20293	20115	1.86
146	4		64					10337	10488	10520	10337	3.62
147	8		128					5609	5997	5622	5609	6.67
148	12		192					4430	4376	4215	4215	8.87
149	16		256					3586	3615	3888	3586	10.43
150	20		320					3615	3571	3682	3571	10.48
151	24		384					3501	3523	3452	3452	10.84
152	28		448					3237	3305	3282	3237	11.56
153	32		512					3293	3284	3168	3168	11.81

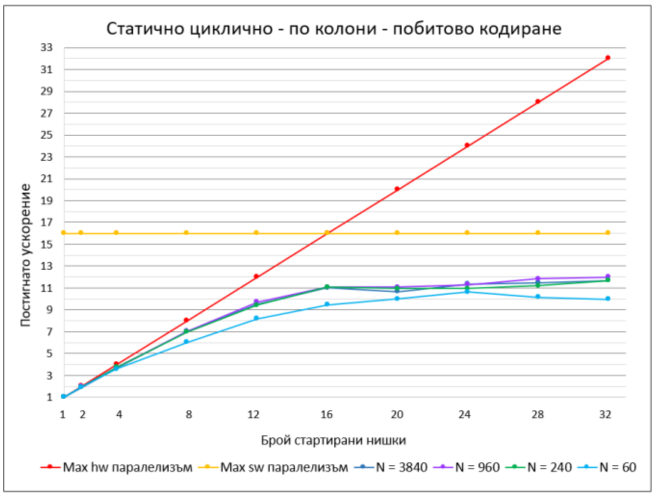


Фигура 24 – Сравняване на резултатите от статично циклично разпределение по колони с побитово кодиране при грануларност g = 1, 4, 16

Както очакваме, получаваме отчетливо най-лошо ускорение при грануларност 1 и почти равни ускорения при g = 4 и g = 16 като това при грануларност g = 16 е с малко по-висока стойност за всякакъв брой стартирани нишки – изключение са 20 и 24 нишки, тъй като ускорението в този случай при g = 4 и g = 16 на практика съвпада. Напълно логично е причината за наблюдаваните резултати да бъде същата както и при статично циклично разпределение с декомпозиция по колони БЕЗ побитово кодиране – по-лошо балансиране на множеството на Манделброт по колони и организацията на Level 1 d-cache.

3.3.2. При фиксиран брой задачи N = 3940, 960, 240, 60

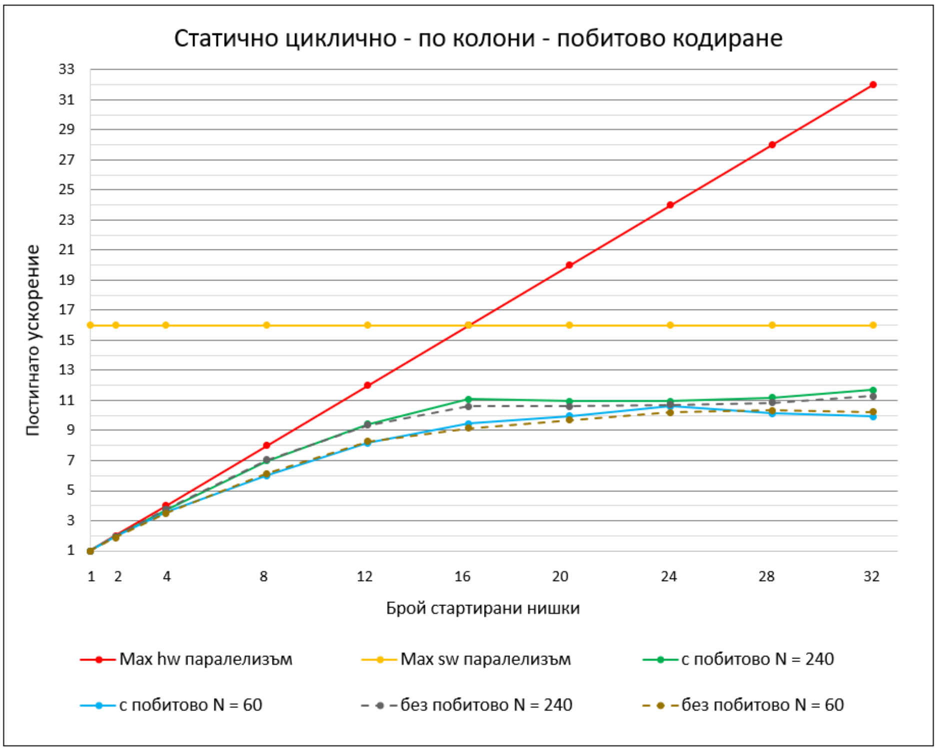
#	*p*	*g*	*N*	$T_1^{(1)}$	$T_1^{(2)}$	$T_1^{(3)}$	T_1=min()	$T_p^{(1)}$	$T_p^{(2)}$	$T_p^{(3)}$	T_p = min()	$S_p = T_1/T_p$
154	2	1920	3840	37551	37436	37184	37184	19948	19765	20045	19765	1.88
155	4	960						10562	10401	10167	10167	3.66
156	8	480						5308	5367	5461	5308	7.01
157	12	320						3925	4104	4041	3925	9.47
158	16	240						3505	3542	3379	3379	11.00
159	20	192						3491	3517	3510	3491	10.65
160	24	160						3389	3272	3410	3272	11.36
161	28	138						3234	3300	3259	3234	11.50
162	32	120						3245	3187	3244	3187	11.67
163	2	480	960	38350	37863	37879	37863	20077	19726	19396	19396	1.95
164	4	240						10233	10367	10324	10233	3.70
165	8	120						5378	5454	5423	5378	7.04
166	12	80						4081	4141	3903	3903	9.70
167	16	60						3496	3682	3418	3418	11.08
168	20	48						3620	3535	3419	3419	11.07
169	24	40						3352	3429	3412	3352	11.30
170	28	35						3203	3296	3278	3203	11.82
171	32	30						3271	3157	3248	3157	11.99
172	2	120	240	37578	38361	37462	37462	19396	19976	21488	19396	1.93
173	4	60						10037	10074	10249	10037	3.73
174	8	30						5446	5366	5540	5366	6.98
175	12	20						4204	4380	3974	3974	9.43
176	16	15						3445	3383	3547	3383	11.07
177	20	12						3424	3497	3497	3424	10.94
178	24	10						3582	3517	3424	3424	10.94
179	28	9						3345	3431	3460	3345	11.20
180	32	8						3313	3208	3295	3208	11.68
181	2	30	60	37062	37440	37072	37062	20068	20140	19328	19328	1.92
182	4	15						10506	10360	10435	10360	3.58
183	8	8						6249	6160	6272	6160	6.02
184	12	5						4933	4677	4528	4528	8.19
185	16	4						3920	4155	3945	3920	9.45
186	20	3						3934	3714	3858	3714	9.98
187	24	3						3491	3849	3719	3491	10.62
188	28	3						3750	3657	3870	3657	10.13
189	32	2						3726	3814	3781	3726	9.95



Фигура 25 – Сравняване на резултатите от статично циклично разпределение по колони с побитово кодиране при фиксиран брой задачи N = 3840, 960, 240, 60

Отново наблюдаваме резултати, подобни на тези без побитово кодиране. Най-ниско ускорение отново се постига при брой задачи N = 60, но вече постигнатото ускорение при N = 240 е по-високо и на места (16 стартирани нишки) дори настига ускорението, получено при N = 960 и N = 3840.

3.3.3. Съпоставяне на получените резултати с тези, получени БЕЗ побитово кодиране



Фигура 26 – Сравняване на резултатите от статично циклично разпределение по колони с и без побитово кодиране при фиксиран брой задачи N =240, 60

На фигура 26 е сравнено ускорението, постигнато с и без побитово кодиране при N = 60 и N = 240. С цел да не претрупваме графиката, е пропуснато сравнение на ускорението при N = 960 и N = 3840. Това, което забелязваме е слабо подобрение на ускорението както при N = 60, така и при N = 240. При 240 на брой задачи т.е. всяка задача представлява 16 последователни колони, подобрението е най-осезаемо, тъй като до 8 нишки ускоренията на практика съвпадат и след това винаги тестовете с побитово кодиране бележат по-добро ускорение. При 60 задачи, т.е. когато всяка задача е 48 последователни колони, ускоренията отново съвпадат до 8 нишки, след това приложението с побитово кодиране бележи по-добри резултати, но при 32 нишки, приложението без побитово кодиране е постигнало малко по-добро ускорение.

Като обобщение на тази секция от проекта, можем да кажем, че представеното побитово кодиране може да постигне малко по-високо ускорение в сравнение с аналогично приложение, не използващо побитово кодиране, но на практика получените разлики са почти пренебрежими. Причината за това най-вероятно е размерът на генерираното изображение – използван е размер 3840x2160px, което не е достатъчно, за да покаже значителна полза от използването на побитово кодиране.

3.4. Сравняване на ускорението при динамично централизирино разпределение с декомпозиция по редове

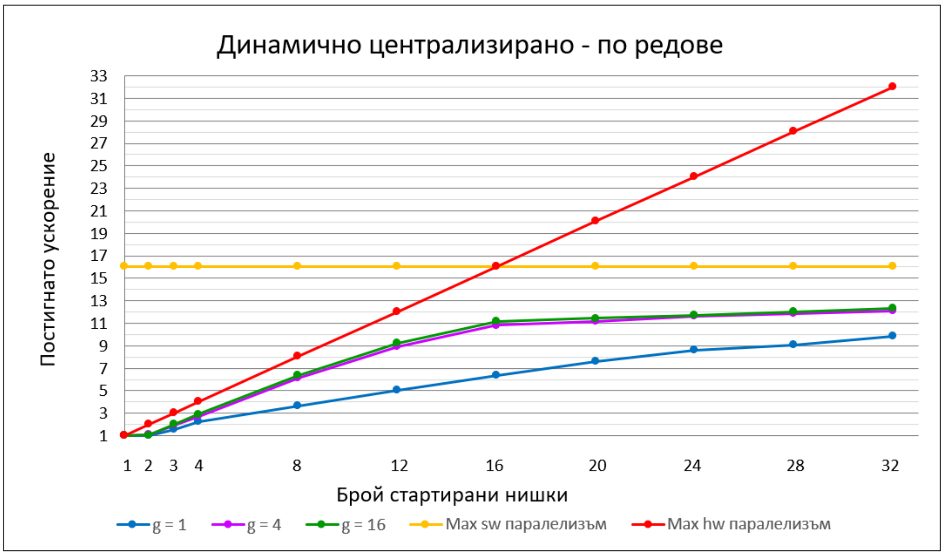
В тази секция са представени резултатите от тестването на приложението, използващо динамично централизирано разпределение на задачите. Избрана е декомпозиця по редове, тъй като в „секция 3.1.“ и „секция 3.2.“ видяхме, че тя води до по-добри резултати от по колони. Избирайки нея, ще проверим дали е възможно динамичното разпределение да подобри получените резултати от статичното, въпреки че то носи доста недостатъци, които се очаква да доведат до влошаване на ускорението при изчисляването на множеството на Манделброт.

Времето, на база на което изчисляваме постигнатото ускорение в долупредставените тестове, е времето на изпълнение на статичната програма с 1 нишка.

3.4.1. При грануларност g = 1, 4, 16

#	p	g	N	$T_1^{(1)}$	$T_1^{(2)}$	$T_1^{(3)}$	T_1=min()	$T_p^{(1)}$	$T_p^{(2)}$	$T_p^{(3)}$	T_p = min()	$S_p = T_1/T_p$
190	2	1	2	38793	37861	38617	37861	40565	39800	39371	39371	0.96
191	3		3					25288	25387	25624	25288	1.50
192	4		4					17014	17192	17184	17014	2.23
193	8		8					10569	10473	10563	10473	3.62
194	12		12					7619	7769	7521	7521	5.03
195	16		16					6031	5982	6128	5982	6.33
196	20		20					5339	5010	5379	5010	7.56
197	24		24					4420	4550	4467	4420	8.57

198	28		28					4350	4179	4444	4179	9.06
199	32		32					3867	4090	4123	3867	9.79
200	2	4	8	39180	38431	38999	38431	37556	37770	38350	37556	1.02
201	3		12					20738	19857	20073	19857	1.94
202	4		16					14681	14335	14428	14335	2.68
203	8		32					6344	6296	6473	6296	6.10
204	12		48					4344	4420	4377	4344	8.85
205	16		64					3725	3568	3704	3568	10.77
206	20		80					3482	3457	3528	3457	11.12
207	24		96					3472	3534	3316	3316	11.59
208	28		112					3253	3382	3301	3253	11.81
209	32		128					3243	3263	3198	3198	12.02
210	2	16	32	38219	38660	38237	38219	38151	37767	39815	37767	1.01
211	3		48					19827	19725	19641	19641	1.95
212	4		64					13777	13317	13312	13312	2.87
213	8		128					6228	6142	6068	6068	6.30
214	12		192					4178	4204	4183	4178	9.15
215	16		256					3590	3433	3529	3433	11.13
216	20		320					3404	3355	3452	3355	11.39
217	24		384					3282	3377	3420	3282	11.65
218	28		448					3269	3185	3207	3185	12.00
219	32		512					3169	3113	3151	3113	12.28



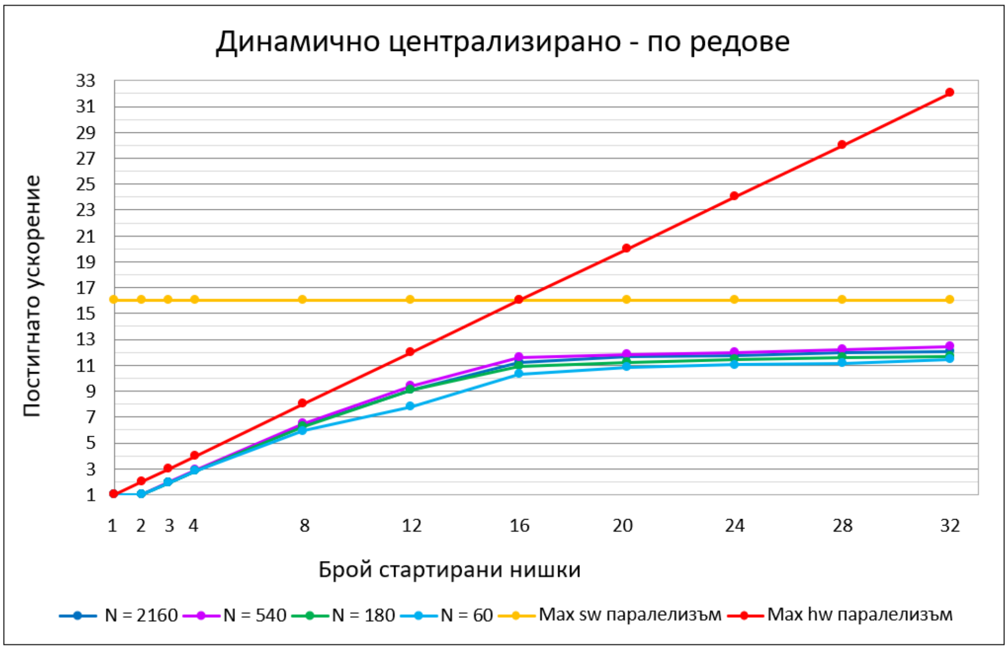
Фигура 27 – Сравнение на ускорението при динамично централизирано разпределение по редове при грануларност g = 1, 4, 16

Както се и очаква, когато стартираме приложението с 2 нишки, т.е. 1 slave нишка, която трябва да изпълни всички задачи и master нишка, която само разпределя задачите т.е. ги предава на slave нишката, ускорение не се наблюдава. След това, при увеличаване на броя slave нишки, ускорението започва да расте. Това, което можем да видим на фигура 27 е, че при грануларност 1, ускорението е най-ниско. Причината за това е, че всяка нишка трябва да вземе средно по само една задача и по този начин дейността на master нишката, която разпределя задачите, е нищожна и единствено пилее процесорната мощност. Увеличавайки грануларността, забелязаваме, че резултатите при g = 4 и g = 16 са почти идентични, но внимателно изследване на данните показва, че при грануларност 16, постигнатото ускорение винаги е малко по-високо – например при 32 стартирани нишки (31 slave нишки, които ще обработват задачите) ускорението при g = 4 е 12.02, а при g = 16 е 12.28.

3.4.2. При фиксиран брой задачи N = 2160, 540, 180, 60

#	*p*	*g*	*N*	$T_1^{(1)}$	$T_1^{(2)}$	$T_1^{(3)}$	T_1=min()	$T_p^{(1)}$	$T_p^{(2)}$	$T_p^{(3)}$	T_p = min()	$S_p = T_1/T_p$
220	2	1080	2160	37809	38287	38861	37809	37601	37692	37777	37601	1.01
221	3	720						19578	19682	19677	19578	1.93
222	4	540						13469	13354	13330	13330	2.84
223	8	270						5990	6109	6048	5990	6.31
224	12	180						4151	4203	4233	4151	9.11
225	16	135						3504	3379	3375	3375	11.20
226	20	108						3315	3243	3301	3243	11.66
227	24	90						3332	3413	3210	3210	11.78
228	28	18						3187	3219	3155	3155	11.98
229	32	68						3129	3136	3140	3129	12.08
230	2	270	540	38711	38556	38672	38556	37947	37473	37912	37473	1.03
231	3	180						19770	19760	19692	19692	1.96
232	4	135						13301	13402	13438	13301	2.90
233	8	68						6055	5996	5955	5955	6.47
234	12	45						4117	4147	4192	4117	9.37
235	16	34						3404	3379	3334	3334	11.56
236	20	27						3256	3286	3366	3256	11.84
237	24	23						3278	3300	3212	3212	12.00
238	28	20						3164	3254	3220	3164	12.19
239	32	17						3173	3181	3104	3104	12.42
240	2	90	180	37965	38252	37842	37842	37754	37532	37300	37300	1.01
241	3	60						19694	19761	19912	19694	1.92
242	4	45						13371	13376	13379	13371	2.83
243	8	23						6047	6087	6026	6026	6.28
244	12	15						4161	4195	4169	4161	9.09
245	16	12						3462	3475	3461	3461	10.93
246	20	9						3405	3393	3377	3377	11.21

247	24	8						3312	3353	3316	3312	11.43
248	28	7						3313	3294	3260	3260	11.61
249	32	6						3250	3231	3242	3231	11.71
250	2	30	60	38027	38198	37925	37925	38835	36823	37219	36823	1.03
251	3	20						19618	19635	19762	19618	1.93
252	4	15						13760	13421	13437	13421	2.83
253	8	8						6918	6492	6408	6408	5.92
254	12	5						4921	4890	5121	4890	7.76
255	16	4						3702	3711	3685	3685	10.29
256	20	3						3502	3495	3552	3495	10.85
257	24	3						3457	3439	3475	3439	11.03
258	28	3						3432	3395	3407	3395	11.17
259	32	2						3353	3315	3327	3315	11.44

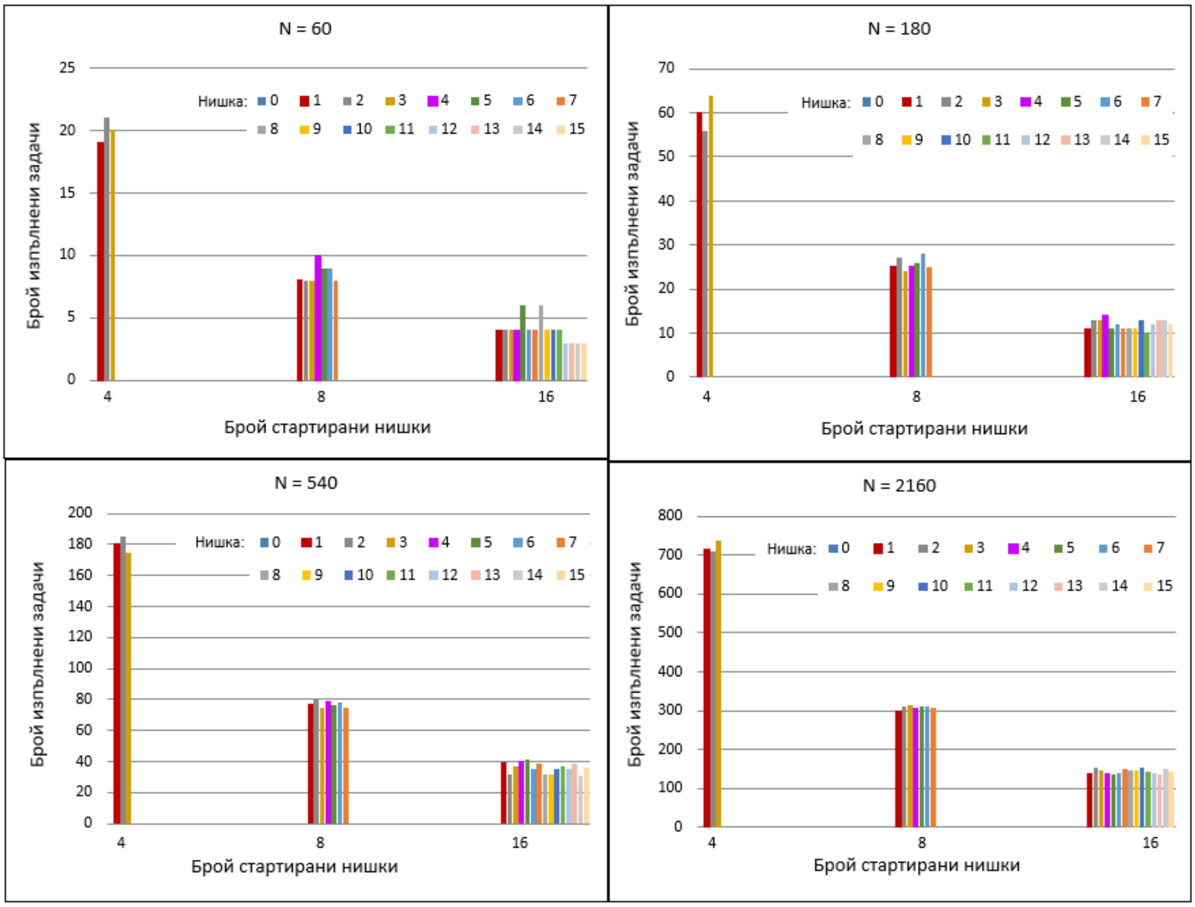


Фигура 28 – Сравнение на ускорението при динамично централизирано разпределение по редове при брой задачи N = 2160, 540, 180, 60

Представените в тази секция тестови резултати целят да изследваме дали при увеличаване на броя задачи, ще се наблюдава bottleneck или ограничение на ускорението, породено от по-високата комуникация при разпределяне на задачите. Както виждаме от фигура 28, най-добрите резултати се наблюдават при брой задачи N = 540. Тук, наблюдаваното ускорение, макар и с малко, е винаги по-добро както от постигнатото при N = 2160, така и при N = 180. За съжаление, толкова малки разлики в ускорението, не могат да бъдат сигурна индикация за bottleneck и свръхтовар, породен от големия брой задачи, които трябва да бъдат разпределени между изпълняващите нишки. Причината е възможно да се крие и в Level 1 d-cache, тъй като при N = 2160 всяка задача представлява само 1 ред от множеството на Манделброт, което вероятно прави алгоритъмът не толкова благоприятен за Level 1 d-cache.

Същевременно, при по-малък брой задачи (180 и особено 60) полученото по-ниско ускорение може да се обясни с това, че поради намаления брой задачи, е по-трудно нишките да си разпределят необходимите изчисления равномерно във времето, тъй като парчетата, които трябва да вземат са сравнително големи (12 последователни реда при 180 задачи и 36 последователни реда при 60 задачи).

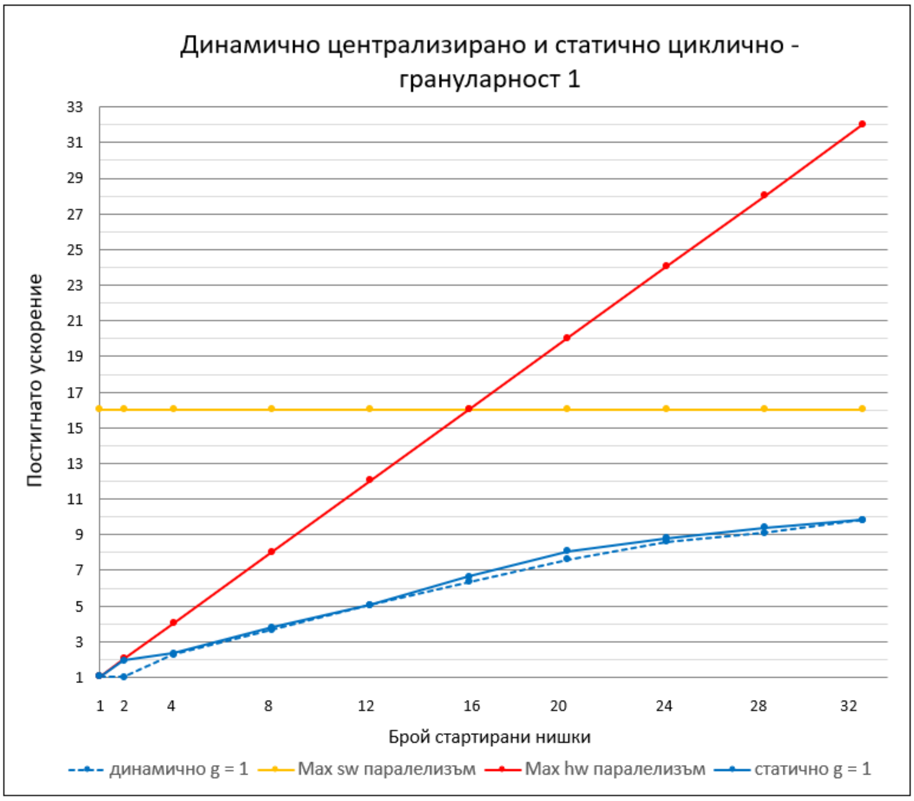
3.4.3. Сравняване на броя задачи, изпълнени от всяка нишка



Фигура 29 – Сравнение на разпределението на задачите при динамично централизирано разпределение по редове при брой задачи N =60, 180, 540, 2160

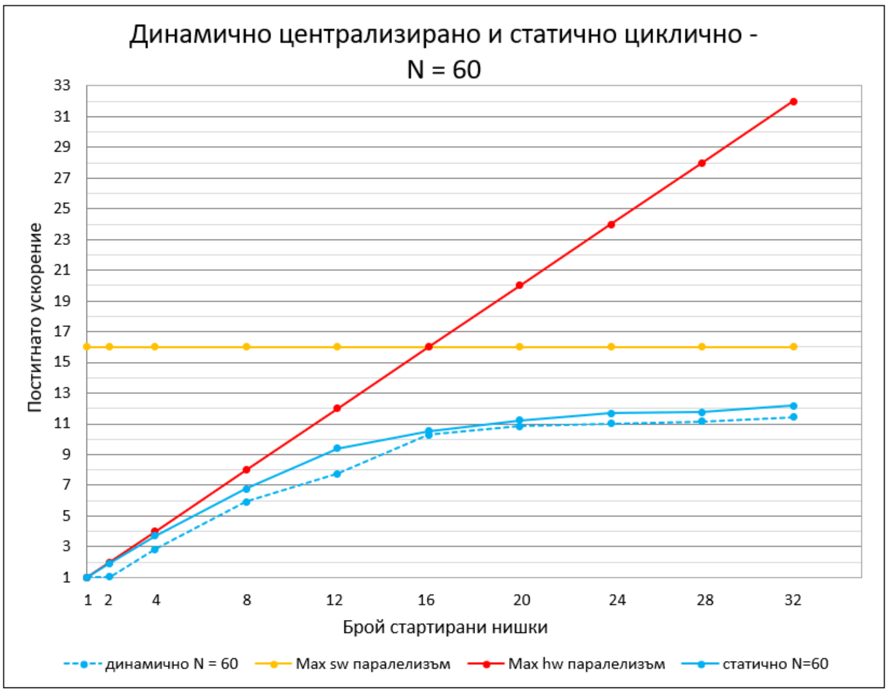
Различните диаграми от фигура 29 показват, че броя задачи, които се изпълняват от всяка нишки не се изравнява нито при увеличаване на общия брой задачи, които трябва да се изпълнят, нито при увеличаване на броя нишки при фиксиран брой задачи. Това е напълно очакван резултат, тъй като тук при динамичното разпределение, за разлика от при статичното, нишките нямат предварително зададено множество от задачи, а получават нова задача едва когато изпълнят текущата. По този начин се цели всички нишки да приключат работата си в максимално едно и също време и така цялото приложение да завърши работата си възможно най-бързо.

3.4.4. Съпоставяне на получитените резултати при динамично разпределение по редове с тези при статично разпределение по редове



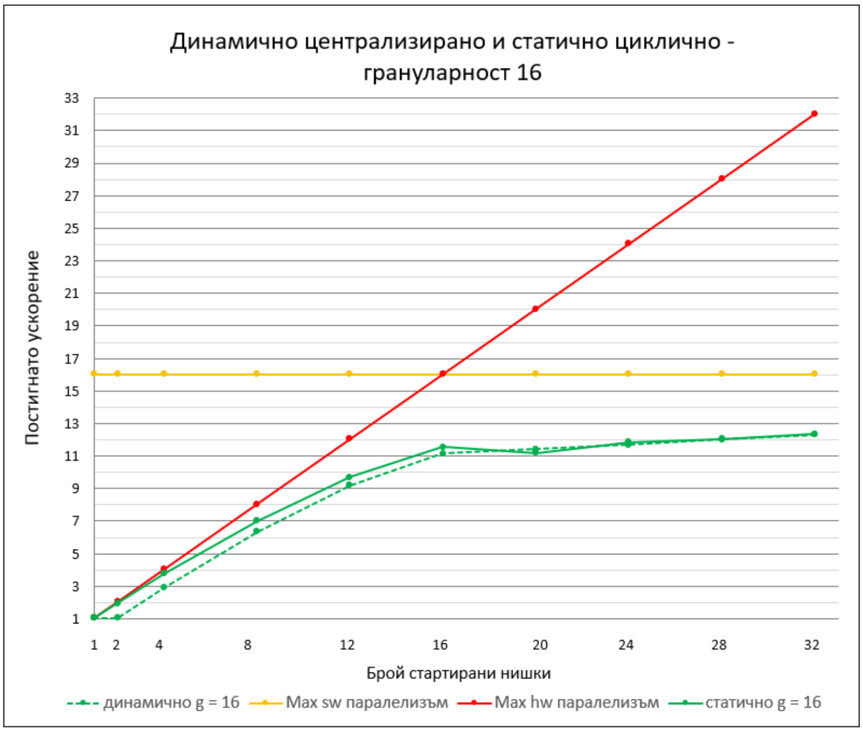
Фигура 30 – Сравнение на ускорението при динамично и статично разпределение по редове при грануларност 1

Както можем да видим на фигура 30, най-голяма разлика между статично и динамично разпределение при грануларност 1 виждаме при 2 стартирани нишки. При статично разпределение постигаме почти линейно ускорение – 1.91 (в „секция 3.1.1.“ е обяснено защо се наблюдава толкова високо ускорение, въпреки че всяка нишка получава само по 1 задача). Същевременно, при динамичното няма ускорение, тъй като цялата изследвана област трябва да бъде пресметната от единствената slave нишка. При увеличаване на броя стартирани нишки, динамичното разпределение започва да постига резултати, близки до тези на статичното. Въпреки това, при статичното разпределение при грануларност 1 постигаме по-добри резултати при всякакъв брой стартирани нишки. Логична причина за това е, че при динамичното имаме 1 нишка, която не осъществява никакви изчисления, а единствено разпределя задачи. Разбира се, тук говорим за грануларност 1 – това означава, че всяка нишка средно ще получи по само 1 задача т.е. няма осезаема ползата от разпределящата нишка.



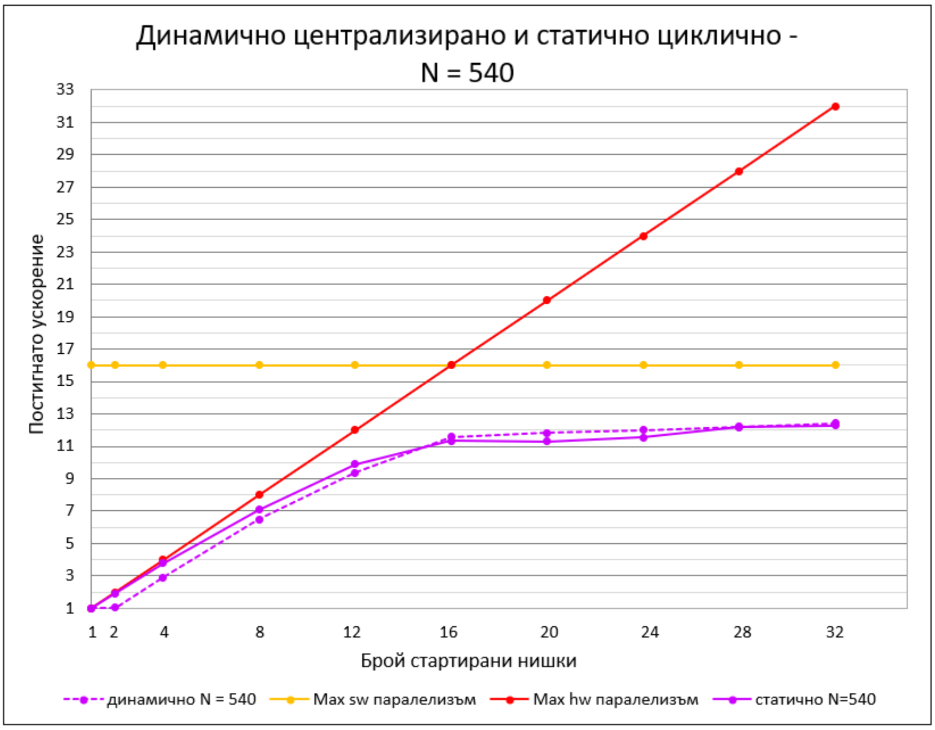
Фигура 31 – Сравнение на ускорението при динамично и статично разпределение по редове при фиксиран брой задачи N = 60

Фигура 31 разкрива подобни тенденции и при фиксиран малък брой задачи, които трябва да се разпределят между нишките. Отново при 2 нишки, статичното бележи ускорение, близко до линейното, а при динамичното няма ускорение изобщо. При увеличаване на броя стартирани нишки, динамичното балансиране започва да настига резултатите на статичното, но те никога не се изравняват. Статичното води до отчетливо по-добри резултати. Причините за това вероятно са вече посочените при анализа на фигура 30 – master нишката почти не е натоварена, тъй като задачите, които трябва да разпредели са малко на брой и същевременно този малък брой задачи е недостатъчен за изравняване на необходимите изчисления между отделните обработващи (slave) нишки.



Фигура 32 – Сравнение на ускорението при динамично и статично разпределение по редове при грануларност g = 16

Тук вече броят задачи се е увеличил чувствително, но основните наблюдавани тенденции се запазват – динамичното разпределение трябва да „настига“ статичното по резултати. Разликата при фигура 32 е, че динамичното не просто настигна, но в някои точки (20 и 32 стартирани нишки) показва макар и минимално по-добри резултати от статичното разпределение. Главната причина за това се очаква да бъде именно по-доброто разпределение на изчисленията във времето между нишките. Както виждаме, тук дори може да кажем, че донякъде се оправдава определянето на master нишка, която да не извършва изчисления, а единствено да разпределя задачи.



Фигура 33 – Сравнение на ускорението при динамично и статично разпределение по редове при фиксиран брой задачи N = 540

Фигура 33 съпоставя най-добрите резултати на статичното и динамичното разпределение, постигнати при емпиричните тестове на съответните приложения. Отново, в началото динамичното разпределение се характеризира с по-ниско ускорение, тъй като имаме 1 нишка, която не осъществява изчисления, но тук при 16 стартирани нишки динамичното изпреварва (макар минимално) статичното и запазва предмината си. Така при 32 стартирани нишки получаваме най-доброто ускорение на приложението, което успяваме да получим в рамките на този проект – 12.44.

Като обобщение на представената информация в тази секция, отново се наблюдават резултати, които подкрепят изказаната в „секция 3.2.“ хипотеза – приложението показва най-оптимални резултати при 540 на брой задачи т.е. когато всяка задача се състои от 4 последователни реда. При зададения размер на изображението – 3840x2160px, отчетливи признаци за bottleneck или ограничение на ускорението, поради голяма комуникация, не се забелязват. Най-доброто ускорение, получено при динамично централизирано разпределение е 12.44 – при N = 540 и 32 стартирани нишки, а при статично циклично разпределение – 12.3 – отново при N = 540 и 32 стартирани нишки. Тъй като тези резултати са в следствие на емпирични тестове, можем да заключим, че при така избраните параметри на изследвания проблем, статичното и динамичното балансиране довят до сходни резултати.

4. Използвани източници

[1] Parallel Fractal Image Generation - A Study of Generating Sequential Data With Parallel Algorithms, Matthias Book, The University of Montana, Missoula – Spring Semester 2001, http://matthiasbook.de/papers/parallelfractals/introduction.html

[2] Isaac K. Gäng, David Dobson, Jean Gourd and Dia Ali, Parallel Implementation and Analysis of Mandelbrot Set Construction, University of Southern Mississippi, 2008, https://www.academia.edu/1399383/Parallel_Implementation_and_Analysis_of_Mandelbrot_Set_Construction

[3] Bhanuka Manesha Samarasekara Vitharana Gamage and Vishnu Monn Baskaran, Efficient Generation of Mandelbrot Set using Message Passing Interface, Monash University Malaysia, July 2020, https://www.researchgate.net/publication/342655570_Efficient_Generation_of_Mandelbrot_Set_using_Message_Passing_Interface

[4] Vito Simonka, Estimating potential parallelism and parallelizing of Mandelbrot set with Tareador and OmpSs, Faculty of Natural Science and Mathematics, University of Maribor, Slovenia, September 2013, https://summerofhpc.prace-ri.eu/wp-content/uploads/2013/09/vito.simonka_reportsmallpdf.com_.pdf

[5] Mirco Tracolli, Parallel generation of a Mandelbrot set, Department of Mathematics and Computer Sciences, University of Perugia, April 2016, http://services.chm.unipg.it/ojs/index.php/virtlcomm/article/view/112/108

[6] Craig S. Bosma, Parallel Mandelbrot in Julia, C++, and OpenCL, January 2015 http://distrustsimplicity.net/articles/mandelbrot-speed-comparison/

[7] Andy Adinets, Adaptive Parallel Computation with CUDA Dynamic Parallelism, May 2014, https://developer.nvidia.com/blog/introduction-cuda-dynamic-parallelism/

[8] Kenneth Falconer, Fractal Geometry: Mathematical Foundations and Applications, 3rd Edition, February 2014, https://www.wiley.com/en-us/Fractal+Geometry%3A+Mathematical+Foundations+and+Applications%2C+3rd+Edition-p-9781119942399