MultiMandel

Изследване на гранулярността при статично циклично разпределение и съпоставка с динамично централизирано решение

Изготвил:
Лъчезар Младенов
Инфомационни системи
3-ти курс, ф.н 71915

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

София
2021

Съдържание

Увод

Настоящия проект - MultiMandel има за цел да разгледа редица въпроси свързани с паралелизма, като за целта е избрана задача, която изобразява множеството на Манделброт. Поради тази причина в текущата глава ще бъде описано какво представлява това множество и как може да се определи дали дадена точка попада в него.

Множеството на Манделброт е кръстено в чест на своя откривател и изследовател математика Беноа Манделброт, като има връзка с множеството на Жюлиа, тъй като и двете образуват сложни фрактални фигури.

Това множество съдържа комплексните числата c и се дефинира от функцията $f_c(z) = z^2 + c$, която не е разходяща при итерация с z = 0. Казано с други думи редицата $f_c(0), f_c(f_c(0)),...$ остава ограничена по абсолютна стойност. Можем да кажем, че дадено комплексно число c е елемент на това множеството, когато започвайки с $z_0 = 0$ и прилагайки повтаряща се итерация, абсолютната стойност на z_n остава ограничена за всички $n>0$. Самото множество е компактно, тъй като е затворено и ограничено от окръжност с радиус 2 около началото на координатната система (0;0).

Изображения на множеството на Манделброт могат да се създадат чрез тестване на комплексни числа дали гореописаната редица за всяка точка с е разходяща до безкрайност. Нанасянето на реалната и имагинерната част на комплексното число като координати върху комплексната равнина позволява да се оцветят пикселите обикновено в черен цвят, когато точката принадлежи. Оцветяването на останалите точки, непринадлежащи на множеството, се определя от степента, с която получената от тях редица достига определена граница, отвъд която няма елементи на множеството.

Важно е да се спомене и причината защо точно тази задача е избрана за текущите изследвания. Причината е проста и се дължи на факта, че създаването на желаното изображение лесно може да бъде декомпозирано на много на брой малки подзадачи, като не е необходима синхронизация между отделните нишки, които изпълняват съответните подзадачи.

1. Анализ

1.1 Функционален анализ

В текущата глава ще бъдат кратко описани няколко източника, които представят различни анализи и изследвания при паралелното решаване на същия проблем.

1.1.1 Parallel generation of a Mandelbrot set [1]

Авторът е имплементирал два подхода за генериране на множеството на Манделброт - статично циклично и динамично централизирано решение. Също така са описани и различни начини за разделянето на подзадачите.

От представените диаграми лесно може да се забележи, че при статичното разпределение, декомпозицията по редове е много по-ефективна, от тази по колони. Преглеждайки резултатното изображение веднага можем да разберем каква е причината за този резултат - а именно факта, че колоните са по-малко балансирани спрямо редовете. Преглеждайки обаче диаграмата, изобразяваща ускорението при динамичното балансиране, ясно се вижда, че това небалансиране е компенсирано, а това се дължи на разпределението на подзадачите по време на изпълнението.

Интересно сравнение е направено и при разделянето на изображението на най-малката градивна единица - пиксел. При такова разделяне статичното балансиране е доста неефективно за разлика от динамичното, което постига по-задоволителни резултати.

1.1.2 MPI vs OpenMP: A case study on parallel generation of Mandelbrot set [2]

В този източник авторът сравнява различни библиотеки като е имплементирал Master-Slave решение, при което главната нишка разделя матрицата от пиксели на p брой части, където p е броят на процесорите. Така всяка част се поема от различен процесор, като накрая резултатите се обединяват в едно цяло. Отново има сравнение на статично спрямо динамично балансиране, като този път не се сравнява начина на декомпозицията, а броя на итерациите.

Важно е да се отбележи и използването на алгоритъма наречен “Escape time”, чиято идея е да определи дали дадена точка принадлежи на множеството на Манделброт. Послеводавателно се изпълняват итерациите и се проверява, дали абсолютната стойност на $z = z^2 + c$ не надвишава 2. Част от точките могат да бъдат проверени дали са от множеството за няколко итерации, но за други може да са необходими повече от милион итерации. Както е показано в статията, при по-голям брой итерации се получават и по-точни изображения на множеството. Когато дадена точка не е от това множество, тя се оцветява зависимост от броя на извършените итерации. Този алгоритъм ще бъде използван и в моето решение.

1.1.3 Parallel Fractal Image Generation [3]

Доста подробно са представени и описани стъпките при изграждането на паралелен алгоритъм. В текущата задача отново става въпрос за “escape” алгоритъма и се подчертава важността при избора на броя итерации. Описани са начини за синхронизация при създаването на изходното изображение, тъй като е възможно при отпечатването на изхода да няма правилна последователност при приключването на работата.

Авторът е представил случай, при който се дават отделни блокове на всяка нишка, но в крайна сметка се е спрял върху декомпозиция на редове с цел по-равномерно натоварване на нишките. Разгледан е въпроса за грануларността - в случая едра, и в крайна сметка авторът е избрал първия процес да изчислява редовете: 1, 5, 9 и тн., втория - 2, 6, 10 и тн и аналогично за останалите процеси.

В заключителните резултати става ясно, че се получава ускорение, което е по-добро от линейното, а причината за този резултат се крие в това, че когато дадена нишка свърши работата си, отпечатването става във фонов режим. Това отпечатване само по себе си не блокира главната работа и нишката продължава с “важната” работа, която е новото изчисление.

1.1.4 Efficient Generation of Mandelbrot Set usingMessage Passing Interface [4]

В тази статия е описано ускорението при 3 различни подхода на разделяне на подзадачите като и при трите начина само една(гравната) нишка отпечатва резултатите.

Наивно разделяне - тук изображението се разделя на блокове с равен брой редове и всяка нишка получава по един блок, който да обработи.

Динамично разпределяне по редове на принципа “първи дошъл първи обслужен” - в този случай самото изображение се разделя на множество задачи, като всяка задача е един ред от изображението. Когато някоя “slave” нишка приключи работата си, главната “master” нишка дава следващия необработен ред и така докато изображението не е създадено.

Статично декомпозиране по редове - при това разделяне предварително работата е разпределена върху нишките и по този начин се гарантира, че всяка нишка ще получи равен брой задачи за изпълнение, а остатъка се дава на главната нишка за обработка.

От получените резултати и тяхното графично представяне, ясно може да се направи извода, че при повече от две нишки последните два начина за разделяне на задачи, дават най-добро ускорение, за разлика от наивния подход, който е най-добър при 2 нишки.

1.1.5 Сравнителна таблица

Образец	Декомпозиция	Тип балансиране	Грануларност	Брой итерации	Размер на изображение
[1]	редове, колони, пиксели и блокове	статично циклично и динамично централизирано	едра и средна	10000	100x100 до 12800x12800
[2]	блокове от редове	статично циклично и динамично	едра	100, 1000, 10000, 100000	1024x1024
[3]	редове, блокове от редове	статично циклично	едра и финна	5000, 50000, 500000	160x120
[4]	редове, блокове от редове	статично циклично и динамично	едра и финна	2000	8000x8000
MultiMandel	редове	статично циклично, динамично централизирано	едра и средна	10000, 20000	3840x2160

1.2 Технологичен анализ

След като вече бяха съпоставени различните източници на ниво функционалност и използван подход, следва да представим и тестовите среди, използвани за провеждане на опитите.

Parallel generation of a Mandelbrot set [1]

В този източник не е написан конкретния модел процесор, единствената информация е за двете използвани платформи - клъстер от 11 двуядрени 32 битови процесора Intel и клъстер от 13 четириядрени 64 битов процесор Intel.

MPI vs OpenMP: A case study on parallel generation of Mandelbrot set [2]

AMD Ryzen™ 5 2500U Quad-Core

Level 1 icache	256 KiB
Level 1 dcache	128 KiB
Level 2 cache	2 MiB
Level 3 cache	4 MiB
Cores	4

Parallel Fractal Image Generation [3]

Няма информация

Efficient Generation of Mandelbrot Set usingMessage Passing Interface [4]

Intel Xeon Gold 6150

Level 1 icache	576 KiB
Level 1 dcache	576 KiB
Level 2 cache	16 MiB
Level 3 cache	24.75 MiB
Cores	18

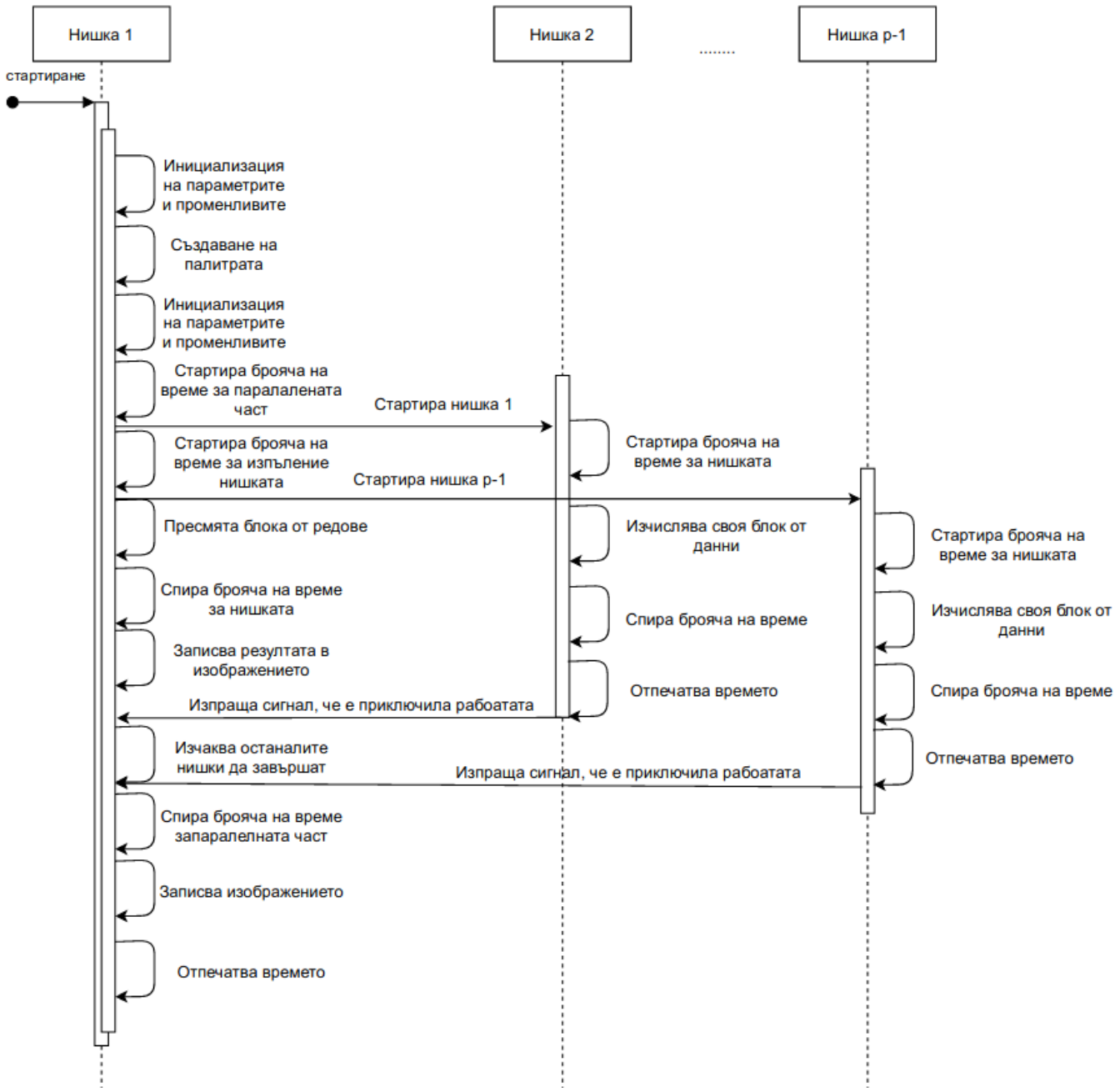
2. Проектиране

2.1 Функционално проектиране

2.1.1 Модел на статично паралелно решение

Повлиян от представените източници, в текущия проект съм реализирал декомпозиция по редове. Реализиран е моделът “***Single Program Multiple Data***”, при който има една главна нишка, която стартира останалите *p-1* нишки, извършва своята работа по създаване на част от изображението, изчаква останалите нишки да приключат и създава крайния резултат. Задачата на останалите *p-1* нишки е да пресметнат своите задачи, които се състоят от блокове от редове и да запишат получените резултати. Синхронизация не се налага да бъде имплементирана, тъй като трябва само да декомпозираме данните и различните нишки да обработят различните части от крайното изображение. Всяка нишка с изключение на първата ще извършва една и съща работа, като е важно да се уточни, че нишките работят независимо една от друга и не можем да гарантираме еднаква последователност на изпълнение. Това може да бъде видяно нагледно в следващата глава - “*Тестови резултати*”.

За да добием по-ясна представа от операциите извършващи се по време на изпълнението на приложението, можем да разгледаме следната диаграма на последователността.



Фиг, 1 - Диаграма на последователността

2.1.2 Модел на динамично паралелно решение

В текущия проект е реализирано и динамично централизирано разпределение на задачите, като отново е използвана декомпозиция по редове. За този модел е важно да се спомене наличието на една главна (master) нишка, която създава другите(slave) нишки и създава отделните задачи, като след това ги добавя в опашката от задачи, изчаква опашката да се изпразни и създава крайното изображение. От тук идва и името на модела “**Master-Slave**”. Чрез този подход се улеснява значително балансирането на задачите между нишките и се постига максимално еднакво време за приключването на всяка нишка. Въпреки предоставената функционалност, важно е да кажем и на каква цена ще я получим. Един от проблемите при тази архитектура е постоянното запитване на главната нишка дали работата е свършена. Това може да доведе до намаляване на постигнатото ускорение. Друг проблем е, че имаме нишка която не върши почти никаква натоварваща задача и при нисък паралелизъм е възможно ускорението да бъде много по-ниско сравнено със статичното разпределение на задачите(това може да бъде видяно и в диаграмите на описаните източници в глава “***Анализ***”).

Задачата свързана с изобразяване на множеството на Манделброт не е подходяща за решава с такъв тип модел, тъй като се обезсмисля динамичното разпределение на задачите. Казано с други думи по-добра ефективност бихме имали, ако дадена задача трябва да се добави към опашката по време на изпълняващи се нишки, а не както в случая - предварително добавяне преди стартиране на работните нишки. Въпреки това, е реализиран такъв тип модел с идеята, той да бъде сравнен със статичното решение и да проверим дали ще се наблюдава намалено ускорение.

2.1.3 Ръководство на потребителя

Програмата може да бъде изпълнена посредством bash скрипт:

```
./multimandel <OPTIONS>
```

За по-лесна работа с приложението са добавени незадължителни командни параметри описани в следващата таблица.

Кратък параметър	Дълъг параметър	Стойност по подразбиране	Описание
-s	--size	3840x2160	Размер на изходното изображение
-d	--domain	-2.5:1.5:-1:1	Областта от комплексната равнина, в която множеството ще бъде изобразено. Въвежда се областта по X и Y
-t	--threads	1	Броят нишки, които ще бъдат стартирани
-o	--output	multimandel.png	Името на изходното изображение
-v	--verbose	Да	Отпечатва информация за работата на нишките и тяхното време за изпълнение
-g	--granularity	-	Задава грануларността, с която програмата да работи
-t	--type	static	Оказва начина на разпределение на задачите (static/dynamic)

2.2 Технологично проектиране

За тестването на програмата бяха използвани следните две тестови среди, като в следващата глава - "**3 Тестови резултати**" са представени резултатите само от тестова машина "***t5600.rmi.yaht.net***", поради възможността за по-висок паралелизъм.

Acer Aspire E1-570g

Архитектура	x86_64
CPU	Intel Core i4-3337U
Cores	2
L1 dcache	64 KiB
L1 icache	64 KiB
L2 cache	512 KiB
L3 cache	3 MiB

t5600.rmi.yaht.net

Архитектура	x86_64
CPU	Intel Xeon E5-2660
Cores	16
L1 dcache	32 KiB
L1 icache	32 KiB
L2 cache	256 KiB
L3 cache	20 MiB

3. Тестови резултати

При всички следващи тестове са използвани следните аргументи на програмата - размер на изходното изображение 3840x2160 и област на комплексната равнина от -2.5 до 1.5 по абцисната ос и от -1 до 1 по ординатната ос. За статичното централизирано решение са направени по два теста - един при 10000 и един при 20000 итерации.

За следващите тестови планове ще въведем следните означения.

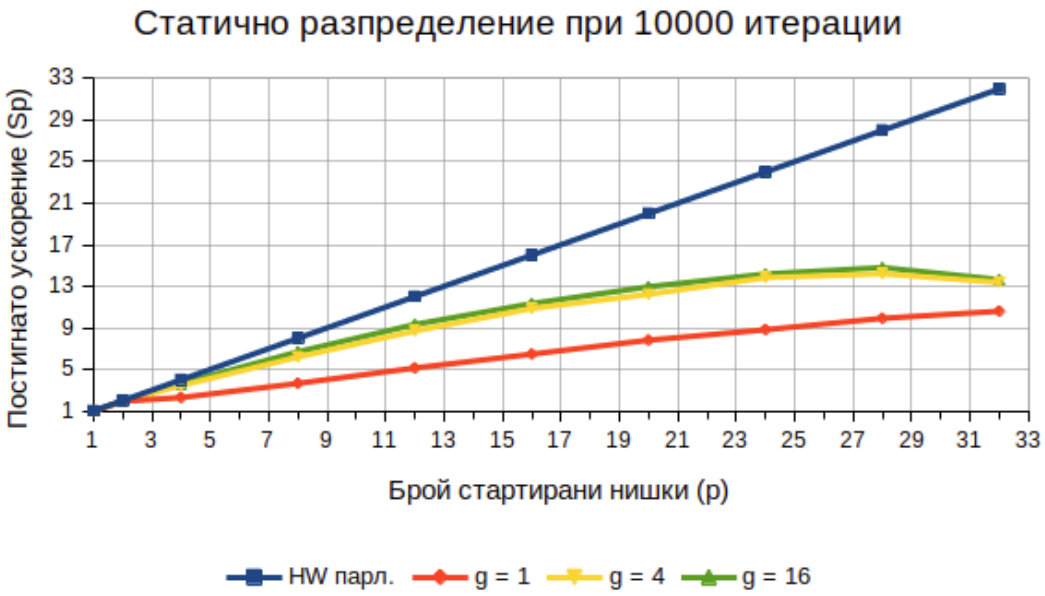
#	Уникален идентификатор на тестовия случай
p	Брой стартирани нишки
g	Грануларност
$T_p^{(i)}$	Времето за изпълнение на програмата в милисекунди при *i*-тото стартиране с *p* брой нишки
$T_p = min()$	Минималното изпълнение на програмата при *p* стартирани нишки
$S_p = T_1/T_p$	Постигната ускорение
$E_p = S_p/p$	Ефективност

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

3.1.1 Брой на итерациите - 10000

#	p	g	Tp (1)	Tp (2)	Tp (3)	Tp=min(Tp(i))	Sp = T1/Tp	Ep = Sp/p
1	1	1	***88292***	***82753***	***81643***	**81643**	1	1
2	2	1	*42447*	*42193*	*42476*	**42193**	1.935	0.968
3	4	1	*36053*	*35703*	*35661*	**35661**	2.289	0.572
4	8	1	*22451*	*22359*	*22493*	**22359**	3.651	0.456
5	12	1	*16007*	*16090*	*15908*	**15908**	5.132	0.428
6	16	1	*12675*	*12917*	*12632*	**12632**	6.463	0.404
7	20	1	*10455*	*10463*	*10954*	**10455**	7.809	0.39
8	24	1	*9407*	*9336*	*9258*	**9258**	8.819	0.367
9	28	1	*8382*	*8335*	*8234*	**8234**	9.915	0.354
10	32	1	*7706*	*8022*	*7966*	**7706**	10.595	0.331
11	1	4	***82307***	***82614***	***81256***	**81256**	1	1
12	2	4	*42047*	*44881*	*42099*	**42047**	1.933	0.967
13	4	4	*23775*	*23532*	*23718*	**23532**	3.453	0.863
14	8	4	*13308*	*13106*	*13177*	**13106**	6.2	0.775
15	12	4	*9420*	*9339*	*9456*	**9339**	8.701	0.725
16	16	4	*7528*	*7479*	*7675*	**7479**	10.865	0.679
17	20	4	*6864*	*6758*	*6647*	**6647**	12.224	0.611
18	24	4	*5873*	*5980*	*6160*	**5873**	13.836	0.577
19	28	4	*5720*	*5740*	*5720*	**5720**	14.206	0.507
20	32	4	*6072*	*6252*	*6248*	**6072**	13.382	0.418
21	1	16	***81326***	***82733***	***81280***	**81280**	1	1

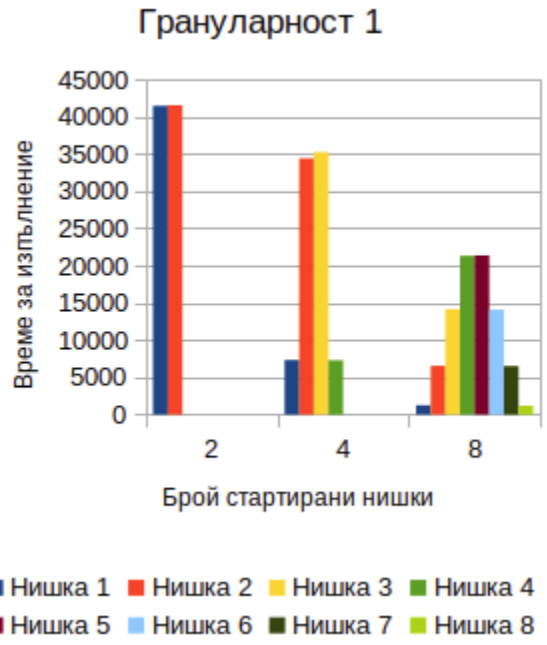
22	2	16	*42097*	*42500*	*42443*	**42097**	1.931	0.966
23	4	16	*22268*	*22591*	*22473*	**22268**	3.65	0.913
24	8	16	*12138*	*12226*	*12207*	**12138**	6.696	0.837
25	12	16	*8839*	*8719*	*8777*	**8719**	9.322	0.777
26	16	16	*7174*	*7276*	*7268*	**7174**	11.33	0.708
27	20	16	*6336*	*6276*	*6285*	**6276**	12.951	0.648
28	24	16	*5822*	*5775*	*5730*	**5730**	14.185	0.591
29	28	16	*5538*	*5491*	*5668*	**5491**	14.802	0.529
30	32	16	*5959*	*5995*	*6105*	**5959**	13.64	0.426



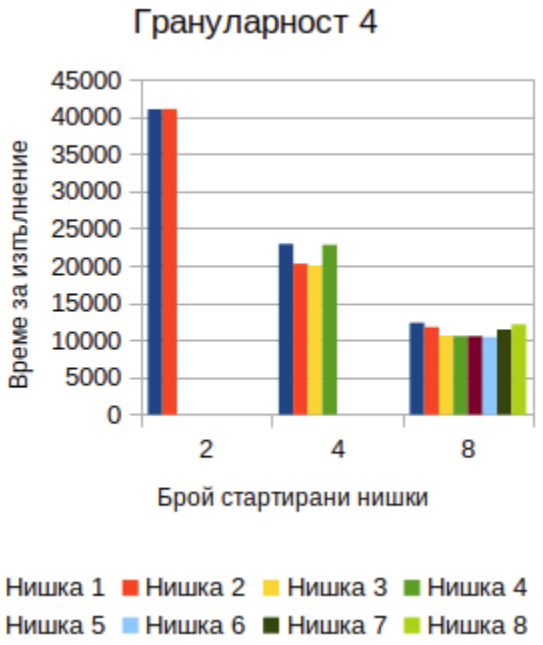
Фиг. 2 - Сравнение на ускорението при статично циклично разпределение с 10000 итерации, g = 1, 4, 16

Гледайки графиката, веднага се забелязва, че при грануларност 1 постигнатото ускорение е по-малко сравние с това получено при 4 и 16. Причината е по-доброто разпределение на работата между нишките. Подзадачите се разделят на по-голям брой, но за сметка на това те са по-малки като последователни редове от изображението и по този начин дадена нишка успява да получи както сложни, така и лесни области за обработка. При средната грануларност ще имаме голямо количество задачи и бихме могли да получим оптималното балансиране.

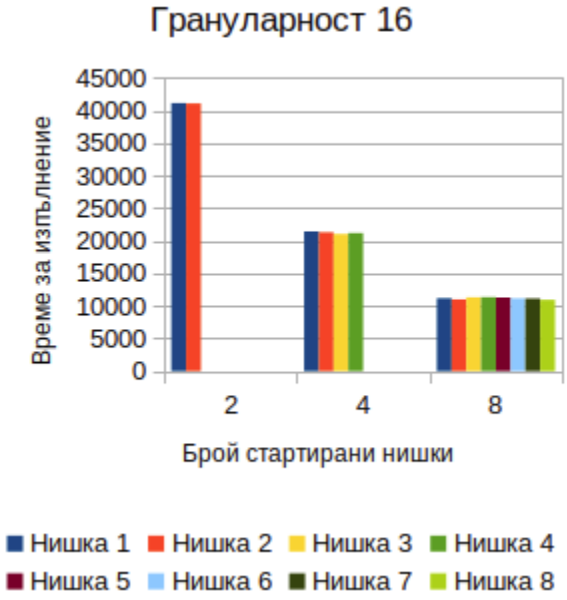
На следващите фигури можем да видим, че с увеличаването на броя на задачите, които всяка нишка трябва да обработи, се подобрява и балансирането на необходимите изчисления, като по този начин нишките ще приключат своята работа почти по едно и също време. В случая са представени данните при 2, 4 и 8 нишки., тъй като при по-голям брой закономерността се запазва и няма смисъл да претрупваме диаграмите.



Фиг. 3 - Време за изпълнение на нишките при g = 1

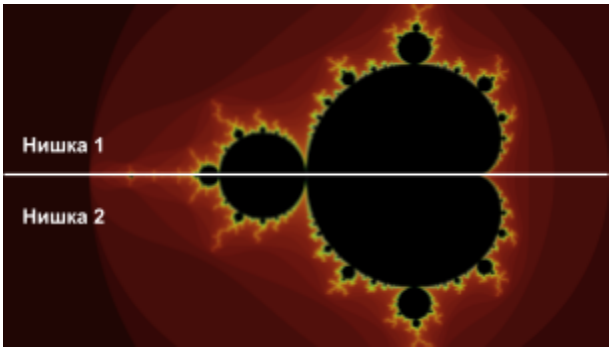


Фиг. 4 - Време за изпълнение на нишките при g = 4

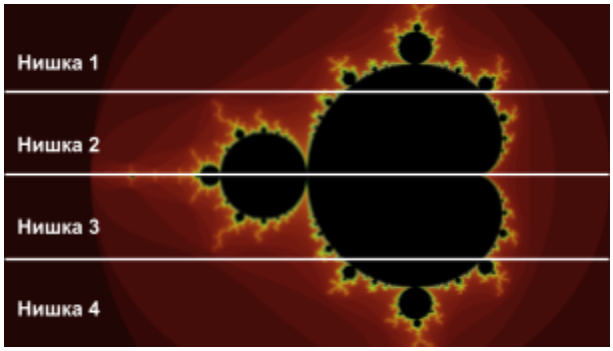


Фиг. 5 - Време за изпълнение на нишките при g = 16

Във всеки един от случаите се забелязва, че при 2 нишки се постига балансирано разпределение на работата, тъй като изображението е разделено на две еднакви части. В първия случай при стартирането на 4 нишки и *g = 1* се забелязва, че няма осезаемо ускорение спрямо стартирането на 2 нишки, а причината за това се крие в начина, по който е разделена работата между нишките. Следващите две фигури ще покажат условното разделяне на изображението между 2 и 4 нишки при грануларност 1.



Фиг. 6 - Разделяне между две нишки при g = 1



Фиг. 7 - Разделяне между четири нишки при g = 1

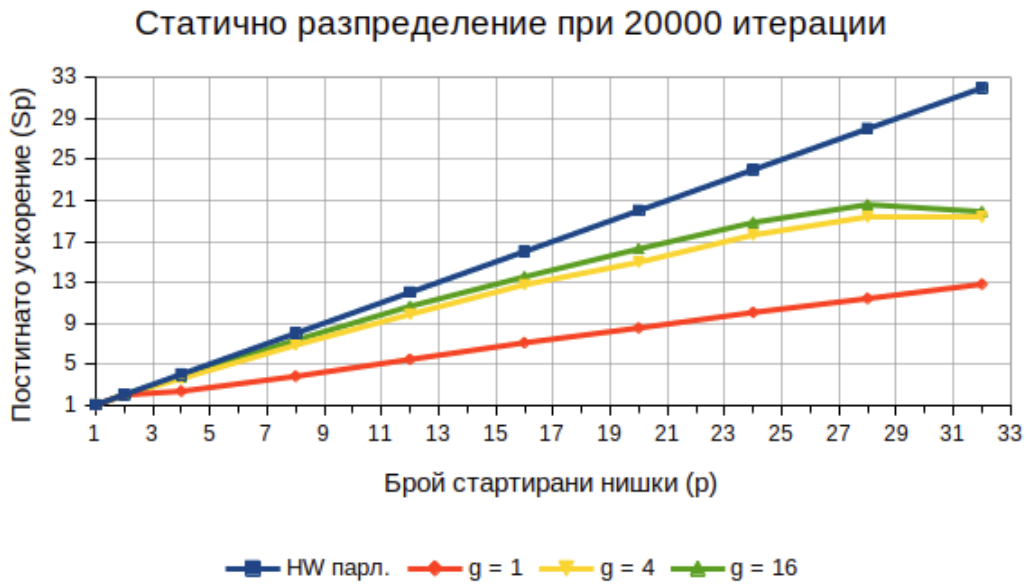
Както се вижда от по-горните илюстрации когато работата по обработката на изображението е разделена между две нишки, всяка от тях получава една и съща област, но огледална спрямо реалната ос. Това е причината за еднаквите времена на работа на нишките представени на *фиг. 3*. На съседната *фиг. 7* обаче, при наличието на 4 обработващи нишки, работата на първата ще е същата с работата на четвъртата, а втората ще обработва същата облааст като третата. Знаем, че черната област е скъпа за обработка и на практика нишки “2” и “3” получават почти същата област за работа както на фиг. 6, което се оказва причината за отсъствието на по-сериозно ускорение, предвид факта, че имаме двойно повече работещи нишки.

3.1.2 Брой на итерациите - 20000

Същия тест беше проведен и при двойно по-голям брой итерации с цел по-голямо натоварване. По-долу ще бъде представена само таблицата с тестовете и диаграмата на ускорението, тъй като останалите анализи са аналогични.

#	p	g	Tp (1)	Tp (2)	Tp (3)	Tp=min(Tp(i))	Sp = T1/Tp	Ep = Sp/p
1	1	1	***160866***	***162671***	***161057***	**160866**	1	1
2	2	1	*82188*	*81511*	*81933*	**81511**	1.974	0.987
3	4	1	*69168*	*69205*	*68786*	**68786**	2.339	0.585
4	8	1	*42550*	*42629*	*42420*	**42420**	3.792	0.474
5	12	1	*29890*	*29773*	*29520*	**29520**	5.449	0.454
6	16	1	*22733*	*23041*	*22989*	**22733**	7.076	0.442
7	20	1	*18965*	*18933*	*18879*	**18879**	8.521	0.426
8	24	1	*16024*	*16298*	*16259*	**16024**	10.039	0.418
9	28	1	*14108*	*14133*	*14313*	**14108**	11.402	0.407
10	32	1	*12595*	*12666*	*12567*	**12567**	12.801	0.4
11	1	4	***161513***	***162452***	***162396***	**161513**	1	1
12	2	4	*82874*	*82442*	*82306*	**82306**	1.962	0.981
13	4	4	*45500*	*45104*	*45508*	**45104**	3.581	0.895
14	8	4	*23552*	*23560*	*23828*	**23552**	6.858	0.857
15	12	4	*16368*	*16505*	*16478*	**16368**	9.868	0.822
16	16	4	*12741*	*12827*	*12668*	**12668**	12.75	0.797
17	20	4	*10889*	*10885*	*10799*	**10799**	14.956	0.748
18	24	4	*9163*	*9253*	*9411*	**9163**	17.627	0.734
19	28	4	*8336*	*8334*	*8455*	**8334**	19.38	0.692
20	32	4	*8454*	*8347*	*8595*	**8347**	19.35	0.605
21	1	16	***162580***	***162840***	***161302***	**161302**	1	1
22	2	16	*82381*	*81972*	*82612*	**81972**	1.968	0.984
23	4	16	*42317*	*42403*	*42781*	**42317**	3.812	0.953

24	8	16	*21958*	*22168*	*22206*	**21958**	7.346	0.918
25	12	16	*15305*	*15222*	*15155*	**15155**	10.643	0.887
26	16	16	*11924*	*12011*	*12029*	**11924**	13.528	0.846
27	20	16	*10064*	*10046*	*9918*	**9918**	16.264	0.813
28	24	16	*8562*	*8628*	*8640*	**8562**	18.839	0.785
29	28	16	*7900*	*7883*	*7833*	**7833**	20.593	0.735
30	32	16	*8442*	*8304*	*8104*	**8104**	19.904	0.622

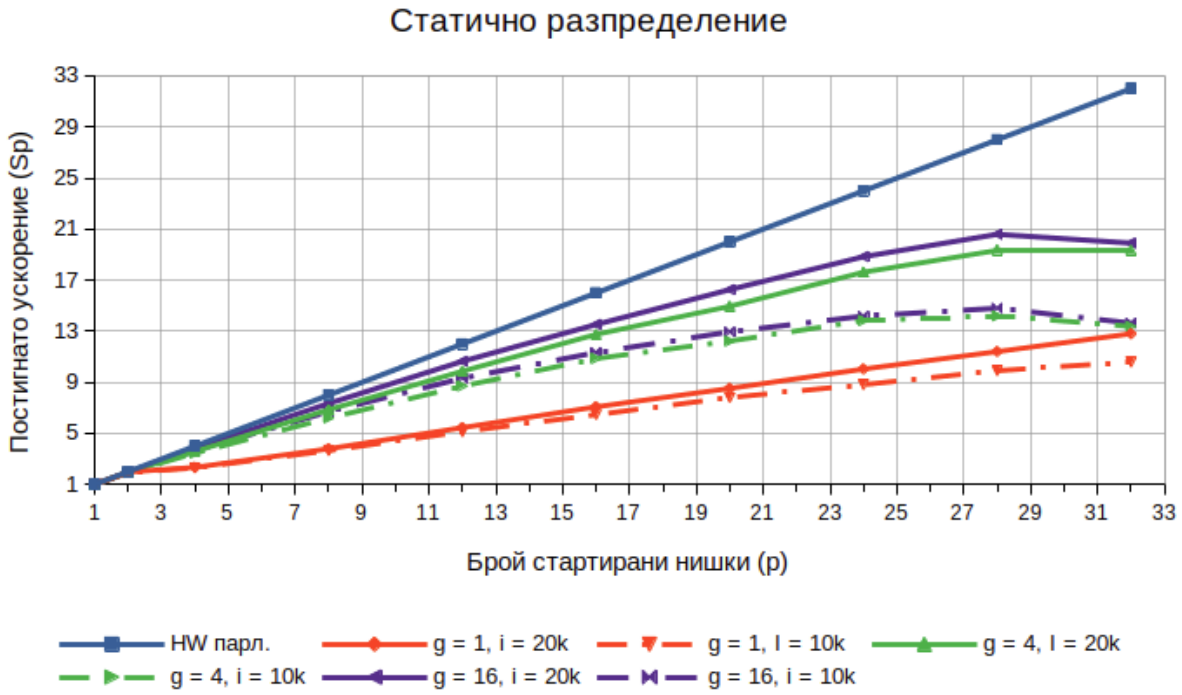


Фиг. 8 - Сравнение на ускорението при статично циклично разпределение с 20000 итерации, g = 1, 4, 16

Създавайки по-голям товар, чрез увеличаването на броя итерации, от представените резултати ясно се вижда и по-голямото ускорение, което е постигнато, като още повече се доближаваме до идеалното. Възможна причина може да се крие в по-рядкото презаписване на стойностите в Level 1d кеша или пък в хипертрединга. Ако данните са прекалено малко или са прекалено лесни за обработка, това също би могло да се разглежда като причина за това ускорение, тъй като по-често ще се правят излишни инструкции.

3.1.3 Съпоставка на получените резултати

Постигнатите резултати показват по-добро ускорение при грануларност 4 и 16, като разликата между двете е почти минимална. Също така се наблюдава, че върховата точка на това ускорение е било винаги при използването на 28 нишки, с изключение при опитите с грануларност 1, където е било при 32 нишки. Като крайно заключение бихме могли да обобщим следното, независимо от броя итерации, и в двата случая получаваме най-добро ускорение при грануларност 16 с използването на 28 нишки. Това от своя страна означава, че най-оптималното разделяне е на блокове от 7-8 реда.



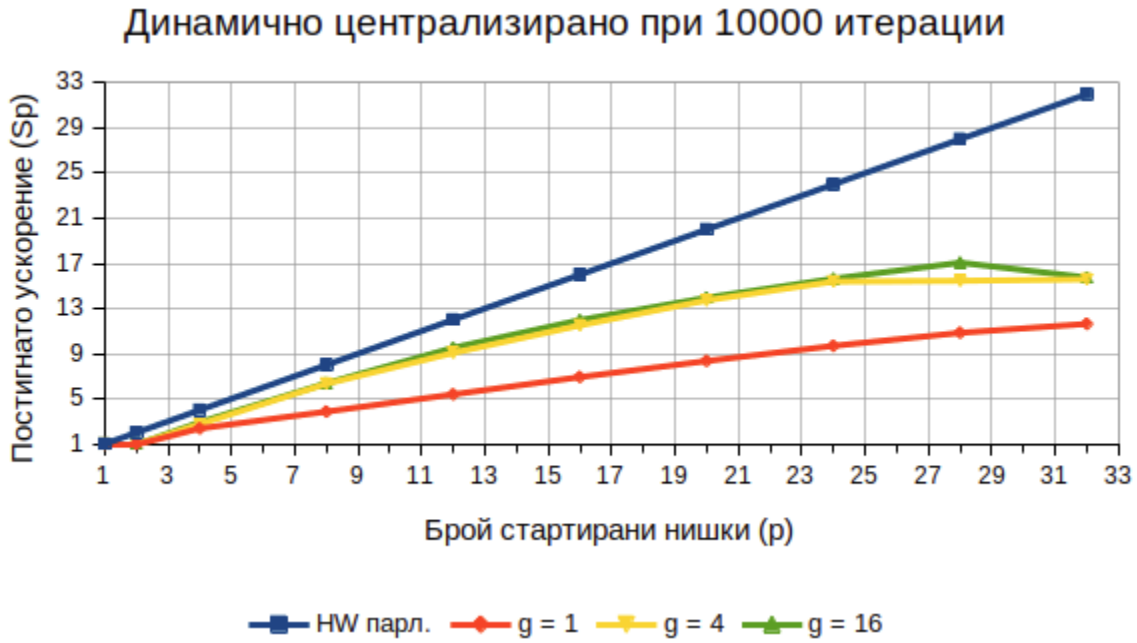
Фиг. 9 - Сравнение на ускорението при 10000 и 20000 итерации

3.2 Сравняване на ускорението при динамично централизирано разпределение при грануларност g = 1, 4, 16

В текущата глава са представени резултати от тестването на програмата, използваща динамично централизирано разпределение на задачите. Този път ще се проведат тестове само при 10000 итерации и накрая резултата ще бъде сравнен с този от статично цикличното решение.

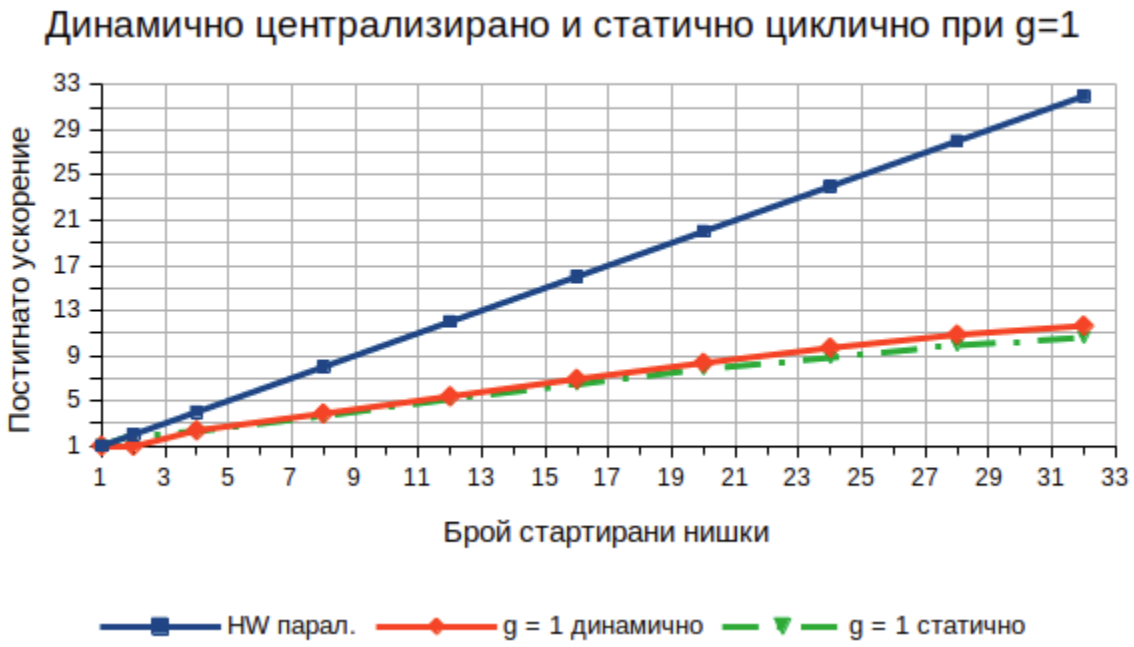
#	p	g	Tp (1)	Tp (2)	Tp (3)	Tp=min(Tp(i))	Sp = T1/Tp	Ep = Sp/p
1	1	1	***88292***	***82753***	***81643***	**81643**	1	1
2	2	1	*81295*	*81030*	*80695*	**80695**	1.012	0.506
3	4	1	*34554*	*34423*	*35596*	**34423**	2.372	0.593
4	8	1	*21282*	*21146*	*21180*	**21146**	3.861	0.483
5	12	1	*15359*	*15166*	*15152*	**15152**	5.388	0.449
6	16	1	*11826*	*11794*	*11866*	**11794**	6.922	0.433
7	20	1	*9945*	*10119*	*9783*	**9783**	8.345	0.417
8	24	1	*8427*	*8696*	*8547*	**8427**	9.688	0.404
9	28	1	*7791*	*7845*	*7527*	**7527**	10.847	0.387
10	32	1	*7018*	*7017*	*7263*	**7017**	11.635	0.364
11	1	4	***82307***	***82614***	***81256***	**81256**	1	1
12	2	4	*81293*	*80295*	*79757*	**79757**	1.019	0.51
13	4	4	*29954*	*29472*	*29695*	**29472**	2.757	0.689
14	8	4	*13242*	*13242*	*12838*	**12838**	6.329	0.791
15	12	4	*9031*	*9070*	*8964*	**8964**	9.065	0.755
16	16	4	*7152*	*7071*	*7060*	**7060**	11.509	0.719
17	20	4	*6229*	*6055*	*5918*	**5918**	13.73	0.687

18	24	4	*6165*	*5280*	*5389*	**5280**	15.389	0.641
19	28	4	*5258*	*5272*	*5293*	**5258**	15.454	0.552
20	32	4	*5325*	*5212*	*5212*	**5212**	15.59	0.487
21	1	16	***81326***	***82733***	***81280***	**81280**	1	1
22	2	16	*81173*	*80681*	*80112*	**80112**	1.015	0.508
23	4	16	*28143*	*27816*	*28145*	**27816**	2.922	0.731
24	8	16	*12748*	*12854*	*12788*	**12748**	6.376	0.797
25	12	16	*8541*	*8664*	*8742*	**8541**	9.516	0.793
26	16	16	*6794*	*6949*	*6786*	**6786**	11.978	0.749
27	20	16	*5890*	*5822*	*5836*	**5822**	13.961	0.698
28	24	16	*5196*	*5220*	*5224*	**5196**	15.643	0.652
29	28	16	*4764*	*4893*	*4925*	**4764**	17.061	0.609
30	32	16	*5161*	*5273*	*5242*	**5161**	15.749	0.492



Фиг. 10 - Ускорение при динамично централизирано решение с 10000 итерации

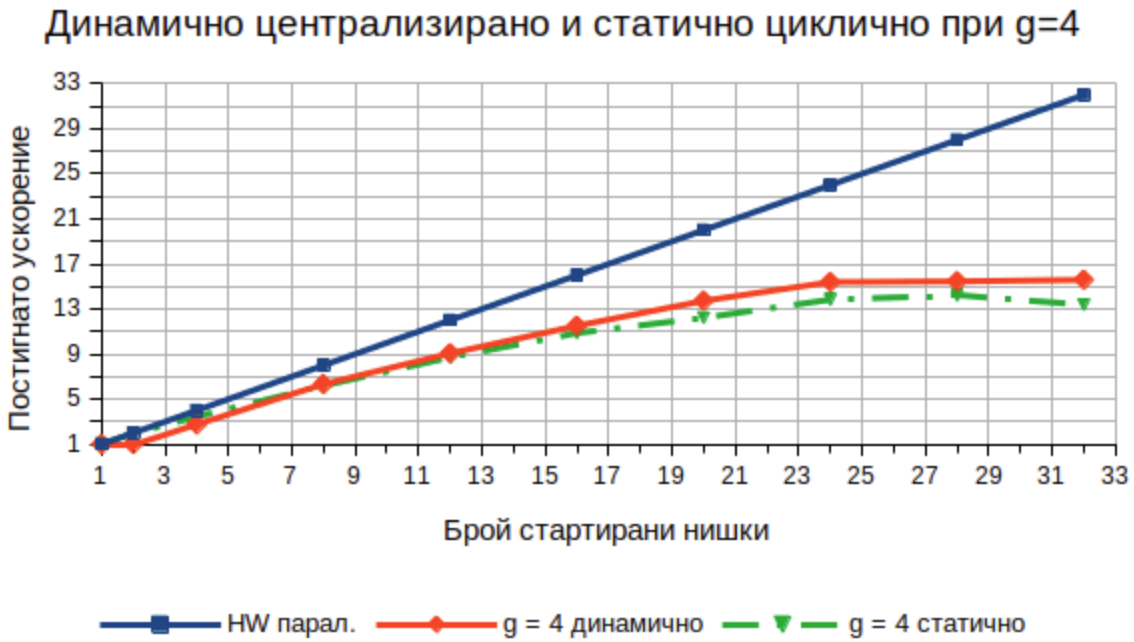
За изчисление на ускорението съм използвал времето при изпълнението на статичното решение с 1 нишка. Ясно се вижда, че при малък брой нишки ускорение не се наблюдава, тъй като имаме една главна нишка, която само разпределя задачите и чака свършването на “slave” нишките. От предишната фигура се забелязва отново, че при грануларност 1 ускорението е най малко, докато при 4 и 16 е почти еднакво. Все пак за по-точни стойности можем да погледнем в таблицата.



Фиг. 11 Съпоставка между статично и динамично при g = 1

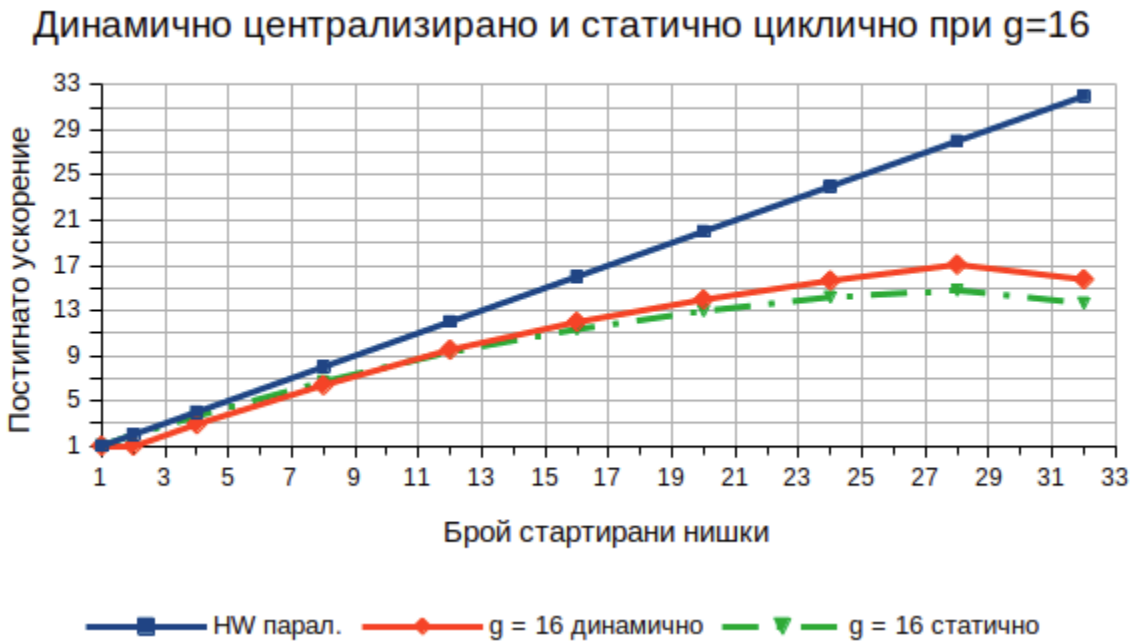
На фиг. 11 можем да видим разликата между динамично централизирано и статично циклично разпределение при грануларност 1. При по-слаб паралелизъм динамичното изостава спрямо другото разпределение, но с увеличаването на броя нишки дори го задминава по ускорение. Все пак сега разглеждаме случая с грануларност 1, което означава, че всяка нишка ще получи точно по една задача, което обезсмисля работата на главната разпределяща нишка.

На диаграмата по-долу може да се разгледа същата съпоставка, но при грануларност 4. Общо взето тенденцията е същата, както в предишния случай с тази разлика, че вече имаме по-голям брой задачи, които да се обработят от нишките.



Фиг. 12 Съпоставка между статично и динамично при g = 4

По-долу при средната грануларност броя задачи се увеличава значително, което оправдава наличието на “master” нишка, грижеща се за разпределението на задачите. Отново най-голям паралелизъм се получава при 28 нишки, както беше при статичното решение.

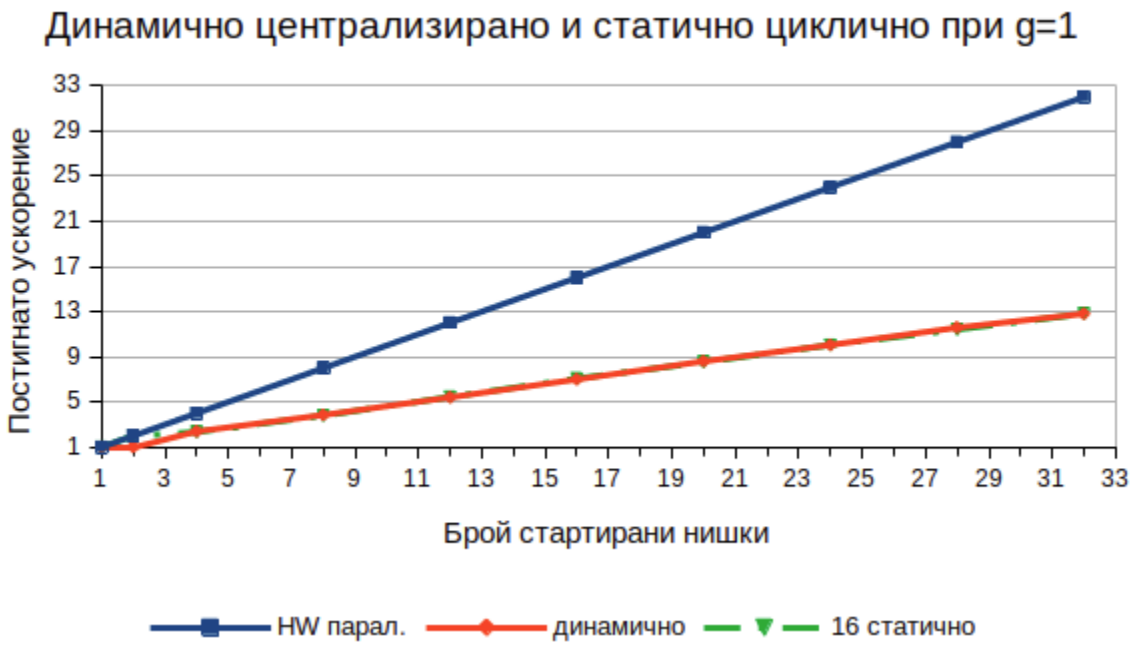


Фиг. 13 Съпоставка между статично и динамично при g = 16

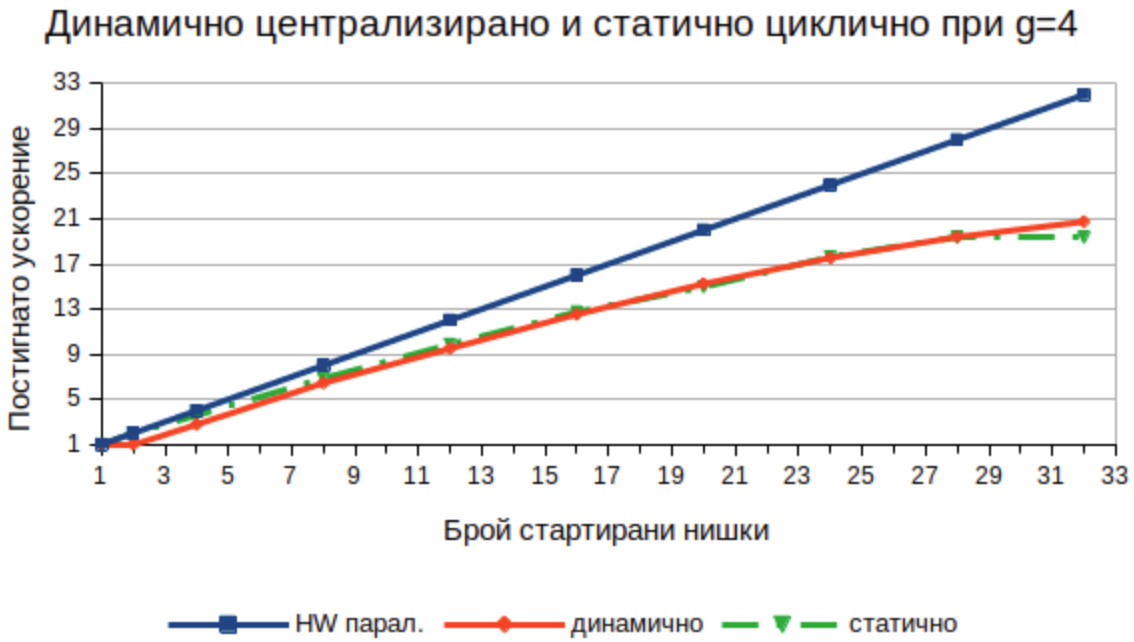
При използването на динамичния подход при **20000 итерации** картината не е по-различна.

#	p	g	Tp (1)	Tp (2)	Tp (3)	Tp=min(Tp(i))	Sp = T1/Tp	Ep = Sp/p
1	1	1	***160866***	***162671***	***161057***	**160866**	1	1
2	2	1	*160811*	*158750*	*160481*	**158750**	1.013	0.507
3	4	1	*68193*	*73922*	*67460*	**67460**	2.385	0.596
4	8	1	*42225*	*41884*	*42292*	**41884**	3.841	0.48
5	12	1	*29843*	*29748*	*29777*	**29748**	5.408	0.451
6	16	1	*23117*	*22999*	*23354*	**22999**	6.994	0.437
7	20	1	*18882*	*18714*	*18794*	**18714**	8.596	0.43
8	24	1	*16048*	*16074*	*16017*	**16017**	10.043	0.418
9	28	1	*14232*	*13897*	*14052*	**13897**	11.576	0.413
10	32	1	*12567*	*12616*	*12762*	**12567**	12.801	0.4
11	1	4	***161513***	***162452***	***162396***	**161513**	1	1
12	2	4	*159783*	*160274*	*160111*	**159783**	1.011	0.506
13	4	4	*58424*	*58447*	*58688*	**58424**	2.764	0.691
14	8	4	*25084*	*25559*	*25630*	**25084**	6.439	0.805
15	12	4	*17077*	*16990*	*17002*	**16990**	9.506	0.792
16	16	4	*12960*	*12968*	*12887*	**12887**	12.533	0.783
17	20	4	*10969*	*10596*	*11066*	**10596**	15.243	0.762
18	24	4	*9547*	*9211*	*9361*	**9211**	17.535	0.731
19	28	4	*8330*	*8339*	*8343*	**8330**	19.389	0.692

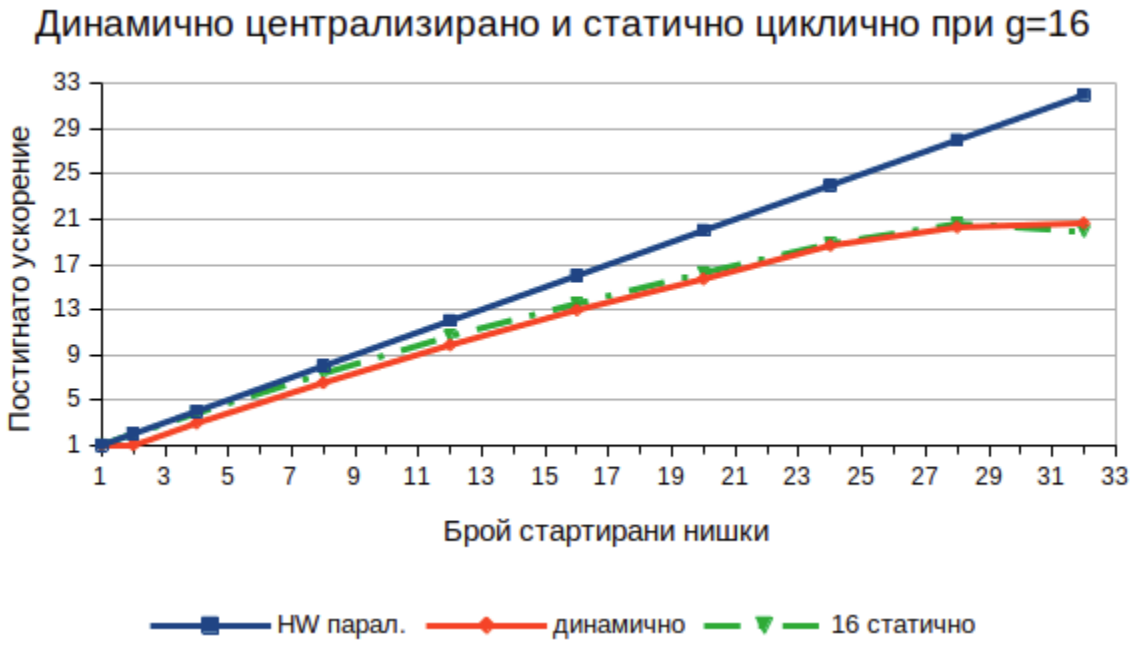
20	32	4	*7788*	*8027*	*8248*	**7788**	20.739	0.648
21	1	16	***162580***	***162840***	***161302***	**161302**	1	1
22	2	16	*159591*	*160035*	*159449*	**159449**	1.012	0.506
23	4	16	*55260*	*55209*	*54832*	**54832**	2.942	0.736
24	8	16	*24742*	*25027*	*24774*	**24742**	6.519	0.815
25	12	16	*16391*	*16424*	*16367*	**16367**	9.855	0.821
26	16	16	*12542*	*12531*	*12448*	**12448**	12.958	0.81
27	20	16	*10335*	*10318*	*10270*	**10270**	15.706	0.785
28	24	16	*8989*	*8841*	*8640*	**8640**	18.669	0.778
29	28	16	*7977*	*7951*	*8057*	**7951**	20.287	0.725
30	32	16	*8138*	*8125*	*7813*	**7813**	20.645	0.645



Фиг. 14 Съпоставка между статично и динамично при g = 1



Фиг. 15 Съпоставка между статично и динамично при g = 4



Фиг. 16 Съпоставка между статично и динамично при g = 16

Можем да направим заключението, че макар и неподходящо за конкретната задача, динамичното разпределение не ограничава по никакъв начин ускорението, а дори напротив - има леко подобрение. В “глава 3.1”стигнахме до извода, че най-високо ускорение се получава при 28 нишки с грануларност 16. При статичното разпределение то беше около 14.802(при 10000 итерации), докато при динамичното разпределение получихме 17.061, отново при 28 нишки за 10000 интерации. Забелязва се обаче, че благодарение на динамичното разпределение по редове успяхме да увеличим ускорението до 20.739 използвайки 32 нишки и грануларност 4. В заключение можем да кажем, че при грануларност 4 и 16 при всички тестове получаваме сходни резултати, но обработвайки блокове от изображението по 7-8 реда, получаваме най-задоволителни резултати.

Източници

[1] 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

[2] Ernesto Soto Gómez, MPI vs OpenMP: A case study on parallel generation of Mandelbrot set, Universidad de las Ciencias Informáticas, September 2020, https://www.researchgate.net/publication/344453347_MPI_vs_OpenMP_A_case_study_on_parallel_generation_of_Mandelbrot_set

[3] Matthias Book, Parallel Fractal Image Generation - A Study of Generating Sequential DataWith Parallel Algorithms, The University of Montana, Missoula, http://matthiasbook.de/papers/parallelfractals/

[4] Bhanuka Manesha Samarasekara Vitharana Gamage, Vishnu Monn Baskaran, Efficient Generation of Mandelbrot Set usingMessage Passing Interface, Monash University Malaysia https://arxiv.org/pdf/2007.00745.pdf