Тест на Манделброт: изследване на ускорението при изпълнение върху мултипроцесор

Борислав Карапанов, специалност: софтуерно инженерство, курс 3, ФН: 62280

Изходен код: https://github.com/Borislav-K/Parallel-Mandelbrot-Set

1. Анализ

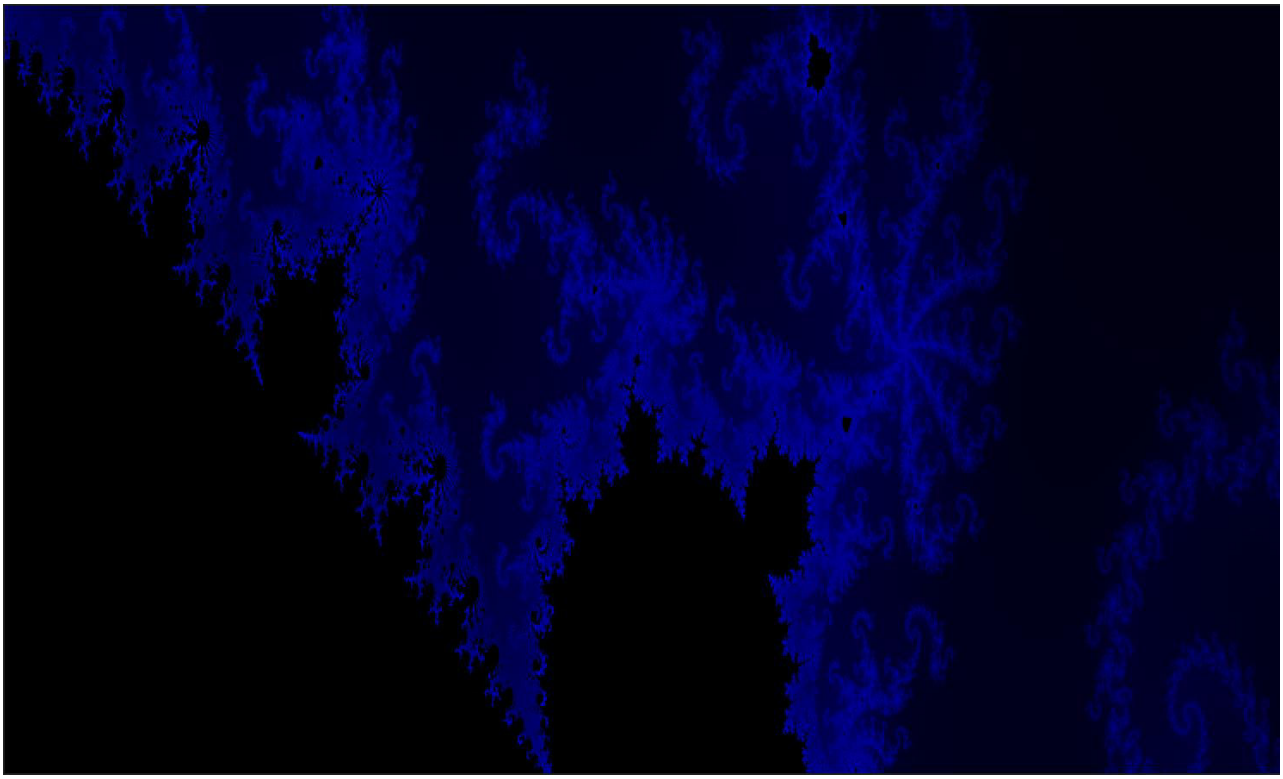
1.1. Разглеждане и анализ на образци с подобна имплементация

Източник	Характеристика	Тип балансиране	Тип декомпозиция	Коментар
[1]	Размер на изображението – 10000x10000. Броят итерации не е посочен. Максимален паралелизъм=4 Параметрите на хардуера не са посочени Реализиран на C++, с MPI	Статично	Рандомизирано разпределение на пикселите между нишките	Полученото ускорение е почти линейно. Целим се към същия ефект с циклично разпределение по редове (допускането е, че балансът ще е достатъчен и няма да има нужда от разпределение по произволен начин)
[2]	Размер на изображението - 160x120 Итерации: 5000,50000,500000 Максимален паралелизъм=6 Параметрите на хардуера не са посочени Реализиран на C++, с MPI	Статично	1. Делене на p залепени правоъгълника 2. Циклично по ред	При увеличаване броя на итерациите, ускорението се подобрява значително. Ниската изчислителна сложност би могла да се компенсира и с по-голям размер на изображението. Ще възпроизведем декомпозицията на правоъгълници, като ще въведем и параметър за грануларност, чиито стойности ще изследваме и за по-високо ускорение (фиг.2 и фиг. 3)

1.2. Основни параметри и декомпозиция на данните

- Размер на изображението – 15000x9000 пиксела.
- Максимален брой итерации на пиксел – 256.
- Цветови модел – RGBA, като всяка точка се оцветява с цвета (0,0,it,0), където it е броят на итерациите за пиксела (изключение е it=0 – черен цвят, ако итерациите са 256).

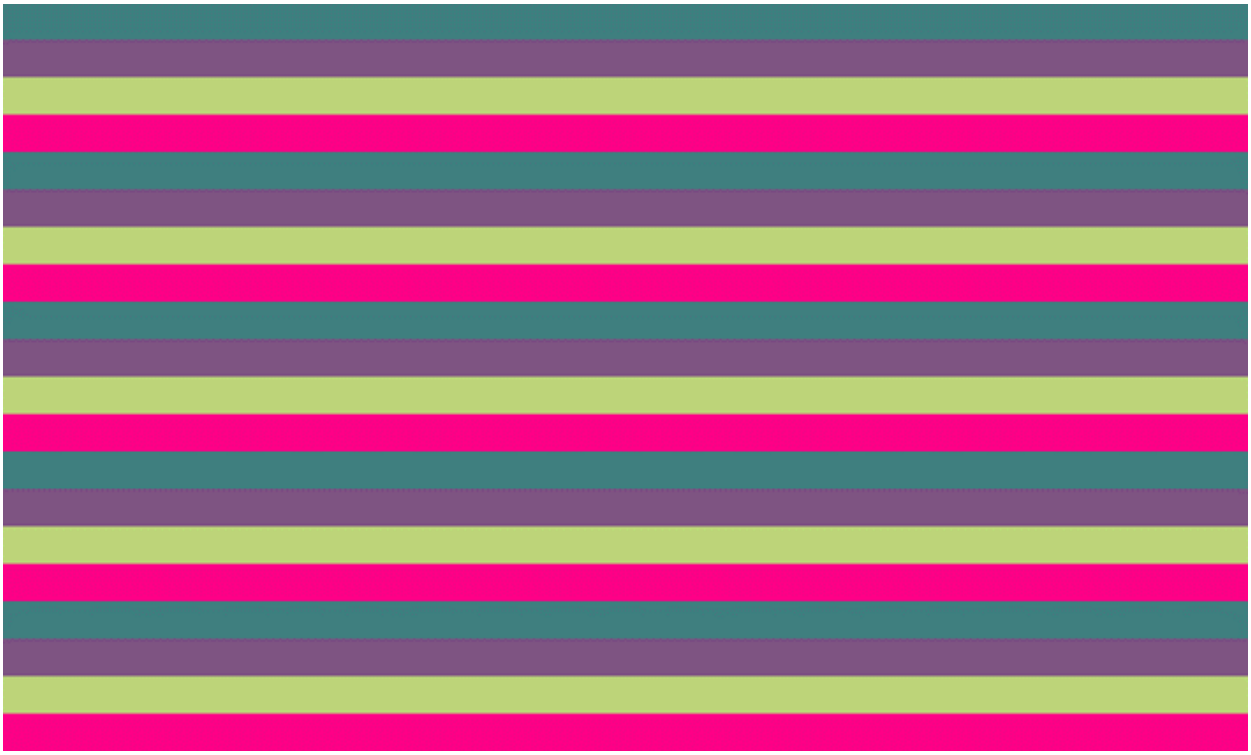
- Област от комплексното пространство - правоъгълникът, който е разположен между точките (-0.87,-0.215) и (-0.814,-0.1976) с цел в матрицата да има дисбаланс и да се демонстрира как по-фината грануларност ще подобри ускорението.



Фигура 1. Примерен резултат от изпълнението на програмата за избраната област

Балансирането на заданията между нишките е статично. Декомпозицията на данните е с 2 разновидности – при първата екранът се разделя на блокове от последователни редове и се изследва ускорението в зависимост от грануларността (фиг. 2-3). При втората, разпределението на редовете между нишките е циклично (т.е. нишка № i взема задания i,i+parallelism,i+2*parallelism,…)

Фиг.2 – декомпозиция на данните при софтуерен паралелизъм 4 и грануларност 1 (всеки цвят отговаря на една нишка)



Фиг.3 – декомпозиция на данните при софтуерен паралелизъм 4 и гранулярност 5 (всеки цвят отговаря на една и съща нишка)

**Ако броят редове не се дели целочислено на g*p, тогава последната нишка взима остатъка от редовете – важно за тестовия случай при g=15.*

2. Проектиране

2.1. Хардуерна инфраструктура

DEV инфраструктура

- Процесор: Intel Core i5-9300HF @ 2.40GHz, 4 ядра
- Размер на процесорната дума: 64 бита
- L1 data cache: 32kb (на ядро)
- Размер на кеш линиите: 64 байта
- L2 cache: 1mb
- L3 cache: 8mb
- RAM: 16GB

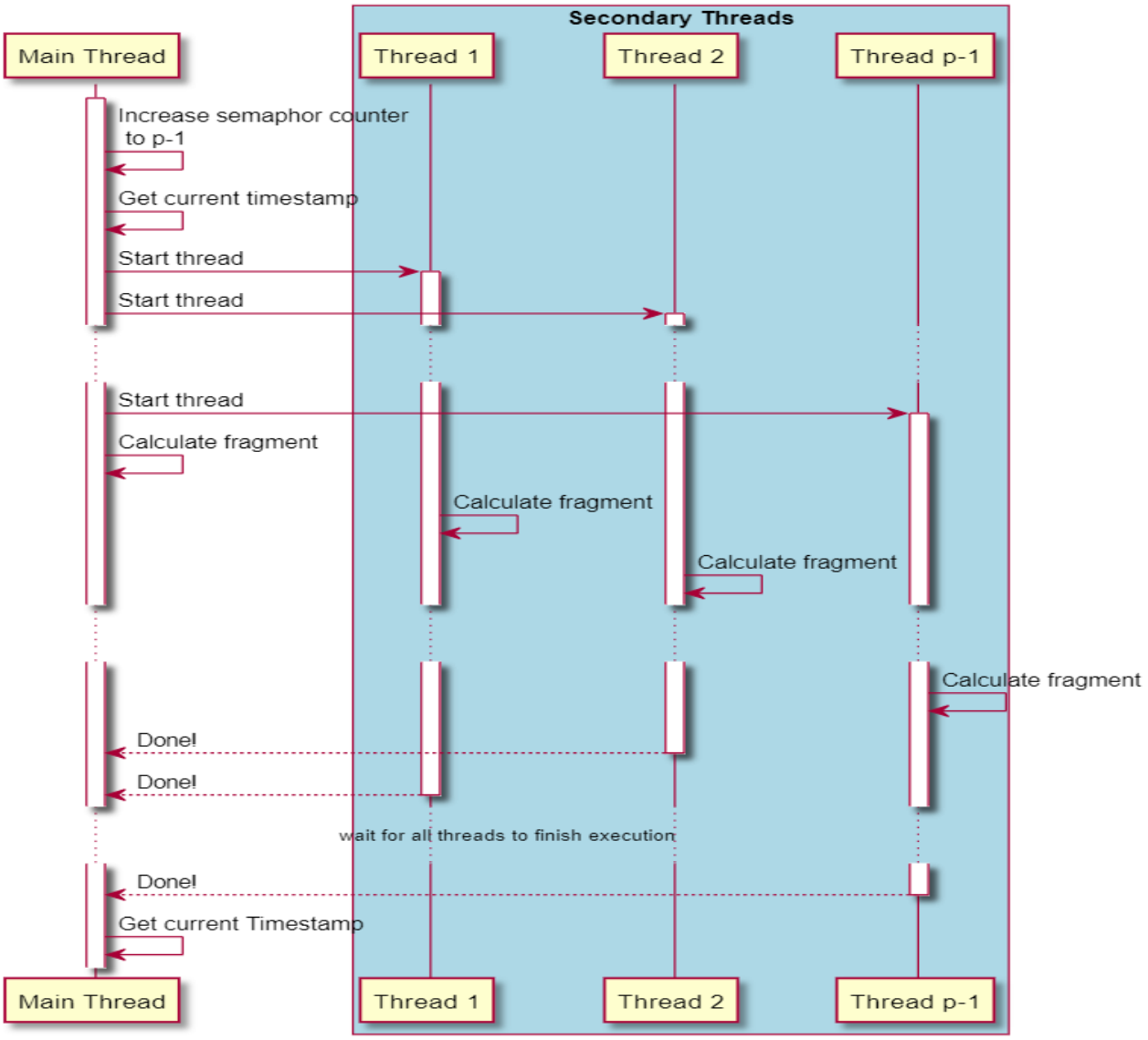
Инфраструктура, върху която са изпълнени тестовете

- Процесор: 2x Intel Xeon E5-2660 @ 2.2GHz, по 8 ядра (16 общо)
- Размер на процесорната дума: 64 бита
- L1 data cache: 32kb (на ядро)
- Размер на кеш линиите: 64 байта
- L2 cache: 256kb

- L3 cache: 20480kb
- RAM: 62GB

2.2. Модел на паралелизма

Използваният модел за паралелизация на изчислението е SPMD. При софтуерен паралелизъм p, главната нишка заделя необходимата памет за пикселите на резултатното изображение, засича времето и стартира p-1 нишки, които обработват заданията си. След като стартира нишките, главната нишка също изчислява нейните задания. След обработката на заданията си, вторичните нишки приключват изпълнението си, а главната изчаква всички вторични да приключат, след което засича времето и създава изображението (фиг. 4). Тъй като отделните пиксели са независими помежду си, няма нужда от допълнителна синхронизация между нишките.



Фигура 4. Стартиране на и синхронизация между нишките

2.3. Софтуерни примитиви за стартиране на нишки и синхронизация между тях

Програмата е реализирана на езика **Go**. И на двете машини версията е **1.15**. Параметрите за грануларност и софтуерен паралелизъм се подават като командни параметри с имена **g** и **p**. Има и други командни параметри, но те не се изменят в тестовете. При подаден параметър **g=0**, разпределянето е циклично по ред.

Използваните примитиви за стартиране на нишки са **goroutines**. Те представляват функции, които се стартират асинхронно (в отделна нишка) и като wrapper на нишките на ОС предоставят оптимизации за системни извиквания([3]). Тъй като нишките само и единствено взимат системното време, тези оптимизации няма да представляват значителни подобрения в тестовите резултати, но и в никакъв случай няма да ги влошат.

Единствената комуникация между нишките е стартирането на 1,2,…,p-1-ва нишка от главната(0) и изчакването на 1,2,…,p-1-ва да завършат. Това се реализира с **waitGroup**([4]), който е примитив на Golang, предоставящ функционалността на семафор. Програмата съдържа глобален waitGroup, до който всички нишки имат достъп.

За да работи този семафор, преди да стартираме нишките, в главната(0) извикваме `wg.Add(parallelism)`, което увеличава брояча на семафора с parallelism. Всяка нишка, след приключването на изчисленията си, извиква `wg.Done()`, което намалява брояча с 1. След като главната нишка изпълни своето задание, тя извиква `wg.Wait()`, което блокира изпълнението и, докато броячът на wg не стане 0 (т.е. докато всички останали нишки приключат). След това се засича за втори път системното време, с цел рисуването на фрактала да не участва в тестовите резултати.

Функцията, която нишките ще изпълняват асинхронно се нарича **calculateMandelbrotSetFragment**. Приема като параметър номера на нишката:

```
func calculateMandelbrotSetFragment(i int) {
```

Стартирането на нишките от главната става в следния цикъл:

```
for i := 1; i <= parallelism-1; i++ {
    go calculateMandelbrotSetFragment(i)
}
```

С go calculateMandelbrotSetFragment(i) стартираме нишка с номер i.

Времето засичаме със стандартната библиотека **time** на Golang.

2.4. Оптимизации за по-ефективно използване на кеш паметта

2.4.1. Матриците в Go са разположени в паметта по редове. Поради тази причина, декомпозицията е по редове, а не по колони. По този начин ще се използва оптимално размерът на кеш линиите.

2.4.2. За да се минимизират времевите разходи, свързани с достъп до по-отдалечените памети от процесора, целим да компресираме колкото се може повече необходимата памет за цветовете. За тази цел всеки елемент на матрицата е от тип uint8, което намалява размера 4 пъти, спрямо този който би бил необходим при използване на color.RGBA, което е структура с 4 uint-а за всяка компонента.

3. Тестови план и резултати

Тестовете за ускорение представляват изпълнение на програмата с различна конфигурация за софтуерен паралелизъм и грануларност. Изследва се ускорението при едра (g=1), средна (g=5) и фина (g=15) грануларности, както и при циклично разпределение на редовете между нишките (g=0).

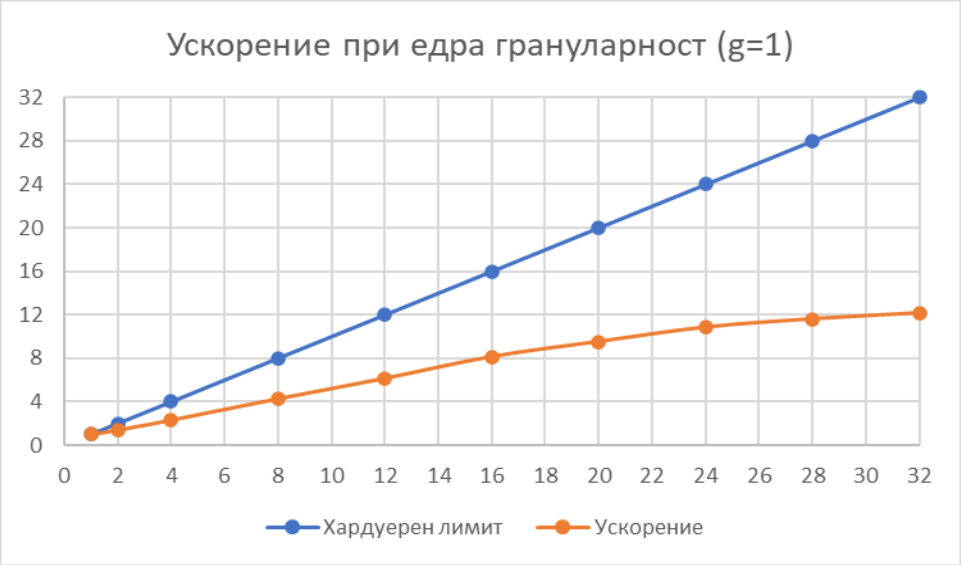
Всеки тестови случай се изпълнява по 3 пъти с цел да се елиминират подвеждащи резултати, възникнали заради фоново натоварване на машината.

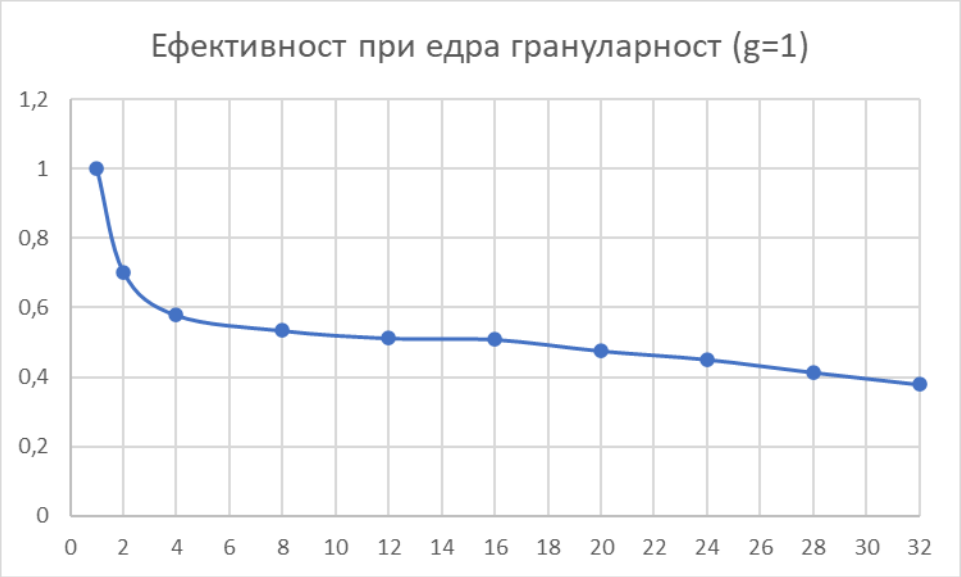
- Данни от тестовете(всичките времена са в милисекунди)

#	p	g	$T_1^{(1)}$	$T_1^{(2)}$	$T_1^{(3)}$	T_1=min()	$T_p^{(1)}$	$T_p^{(2)}$	$T_p^{(3)}$	T_p = min()	$S_p = T_1/T_p$
1	2	1	***91026***	***92248***	***91667***	**91026**	*65097*	*64773*	*64774*	64773	1,41
2	4						*39755*	*39880*	*39319*	39319	2,32
3	8						*21292*	*21513*	*21827*	21292	4,28
4	12						*14801*	*14881*	*14873*	14801	6,15
5	16						*11349*	*11538*	*11172*	11172	8,15
6	20						*9557*	*9592*	*9563*	9557	9,52
7	24						*8468*	*8453*	*8396*	8396	10,84
8	28						*7875*	*7852*	*7844*	7844	11,60
9	32						*7538*	*7501*	*7490*	7490	12,15
10	2	5	***91154***	***91300***	***92057***	**91154**	*51472*	*51107*	*51504*	51107	1,78
11	4						*26365*	*26204*	*26433*	26204	3,48
12	8						*13715*	*13830*	*13690*	13690	6,66
13	12						*9420*	*9407*	*9353*	9353	9,75
14	16						*7175*	*7125*	*7222*	7125	12,79
15	20						*6730*	*6755*	*6748*	6730	13,54
16	24						*6586*	*6568*	*6547*	6547	13,92

#	*p*	*g*	$T_1^{(1)}$	$T_1^{(2)}$	$T_1^{(3)}$	T_1=min()	$T_p^{(1)}$	$T_p^{(2)}$	$T_p^{(3)}$	T_p = min()	$S_p = T_1/T_p$
17	28						*6466*	*6462*	*6450*	6450	14,13
18	32						*6443*	*6415*	*6505*	6415	14,21
19	2	15	***92230***	***90753***	***91038***	**90753**	*47483*	*48153*	*47860*	47483	1,91
20	4						*24993*	*24575*	*24849*	24575	3,69
21	8						*12800*	*12867*	*12829*	12800	7,09
22	12						*8763*	*8744*	*8766*	8744	10,38
23	16						*6663*	*6697*	*6693*	6663	13,62
24	20						*6543*	*6539*	*6536*	6536	13,89
25	24						*7341*	*7461*	*7384*	7341	12,36
26	28						*7724*	*7718*	*7713*	7713	11,77
27	32						*6773*	*6715*	*6733*	6715	13,51
28	2	0	***91031***	***91375***	***91146***	**91031**	*46270*	*46129*	*46008*	46008	1,98
29	4						*23511*	*23503*	*23612*	23503	3,87
30	8						*12155*	*12211*	*12146*	12146	7,49
31	12						*8310*	*8341*	*8373*	8310	10,95
32	16						*6411*	*6375*	*6361*	6361	14,31
33	20						*6315*	*6330*	*6331*	6315	14,42
34	24						*6318*	*6316*	*6351*	6316	14,41
35	28						*6334*	*6327*	*6339*	6327	14,39
36	32						*6449*	*6338*	*6354*	6338	14,36

- Анализ на получените резултати

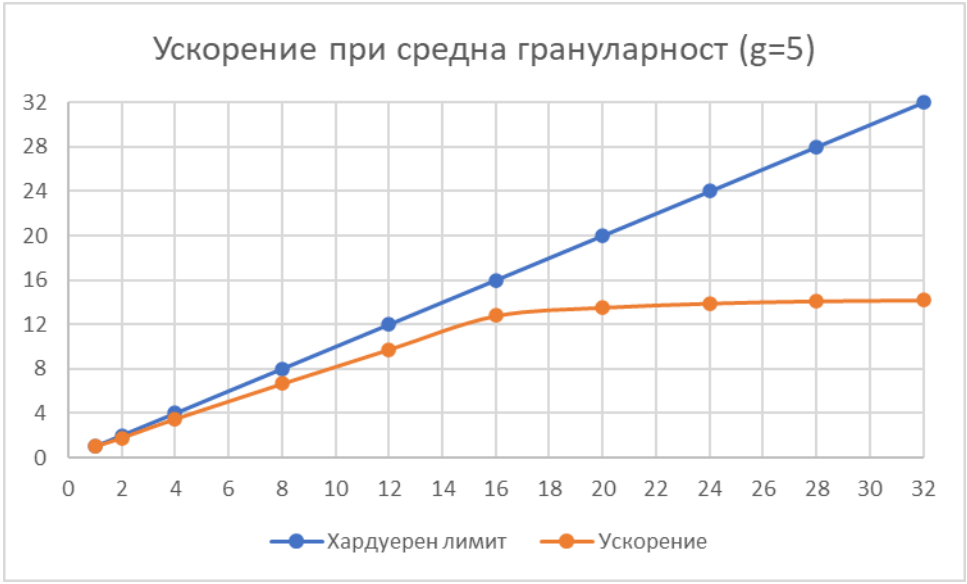


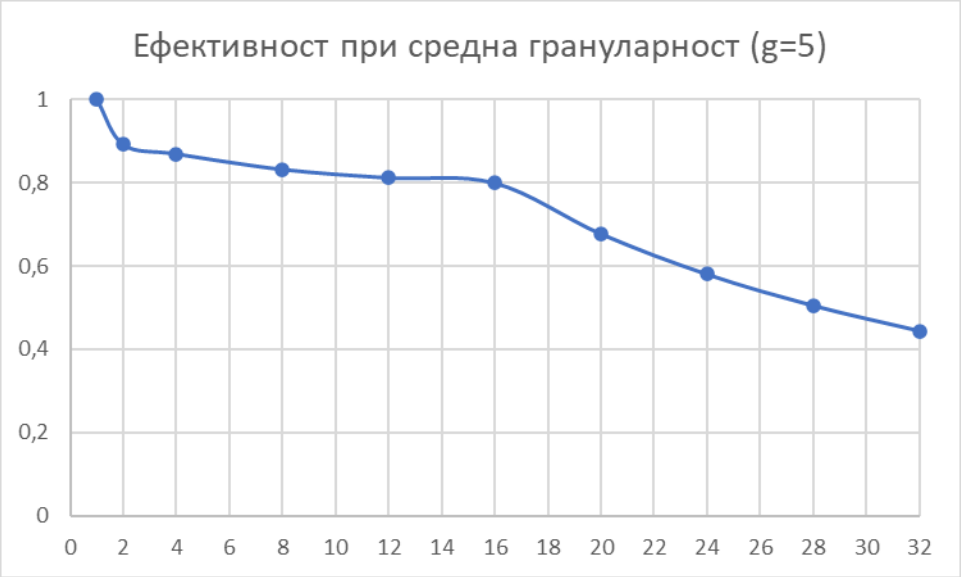


Полученото ускорение при едра грануларност не е много впечатляващо – едва 8.15 при p=16. След p=16, заради факта, че броят на заданията е g*p грануларността изкуствено се увеличава, което е и причината да постигаме по-високо ускорение с повече от 16 нишки. Изглежда, че при g=1 заданията са толкова небалансирани измежду отделните процеси, че загубеното системно време, необходимо да се стартират още нишки, не е дори близко до това, което печелим от по-високата грануларност. Все пак е добре да потвърдим, че това е причината за ускорението като съпоставим времената за изпълнение на отделните нишки при p=16 и p=32:

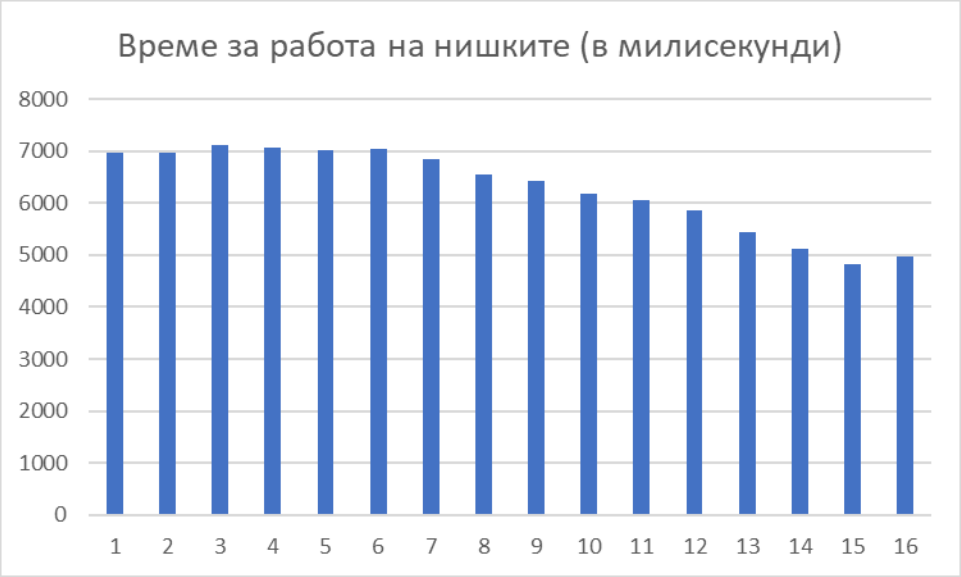
Стандартното отклонение във времето за изпълнение на нишките е 3378 милисекунди при g=1, p=16, а при g=1,p=32 е 2072 милисекунди. Следователно, можем да потвърдим хипотезата, че по-равномерното разпределение на заданията е главната причината за ускорението след p=16.

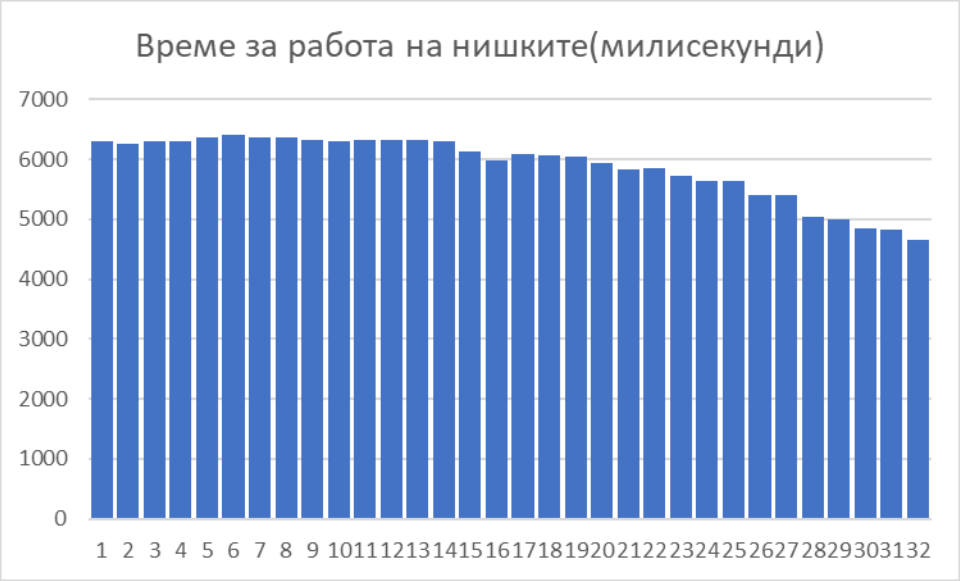
Все пак разликата между изпълнението на най-бързите и най-бавните нишки е все още твърде голяма и стартирането на повече нишки от броя ядра, само за да получим по-добро балансиране е излишен системен товар. Можем да получим по-добро ускорение при средна грануларност:





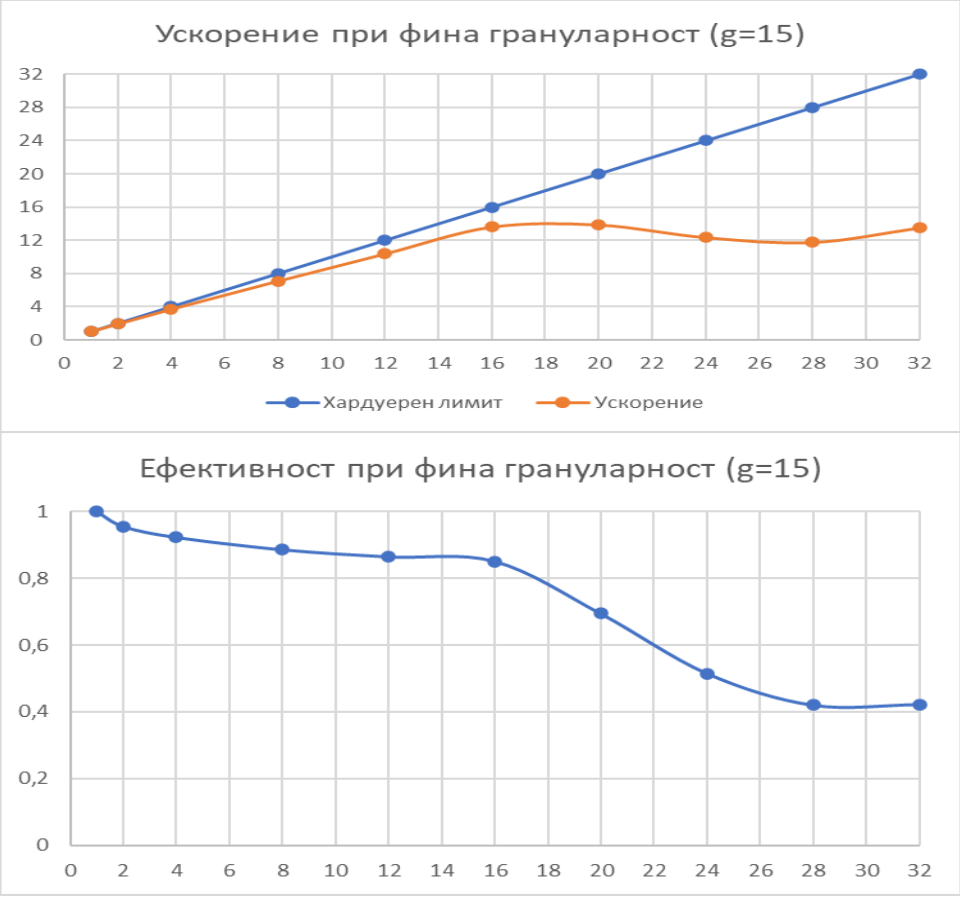
Вече полученото ускорение е доста по-впечатляващо – 12.79 при p=16 и 14.21 при p=32. Тук отново важи същата хипотеза – че допълнителната грануларност при p=32 е причина за повишаването на ускорението с 1.5 при 32 нишки, въпреки допълнителният системен товар. Хипотезата отново потвърждаваме с диаграми на продължителността на изпълнение на отделните нишки съответно при p=16 и p=32:





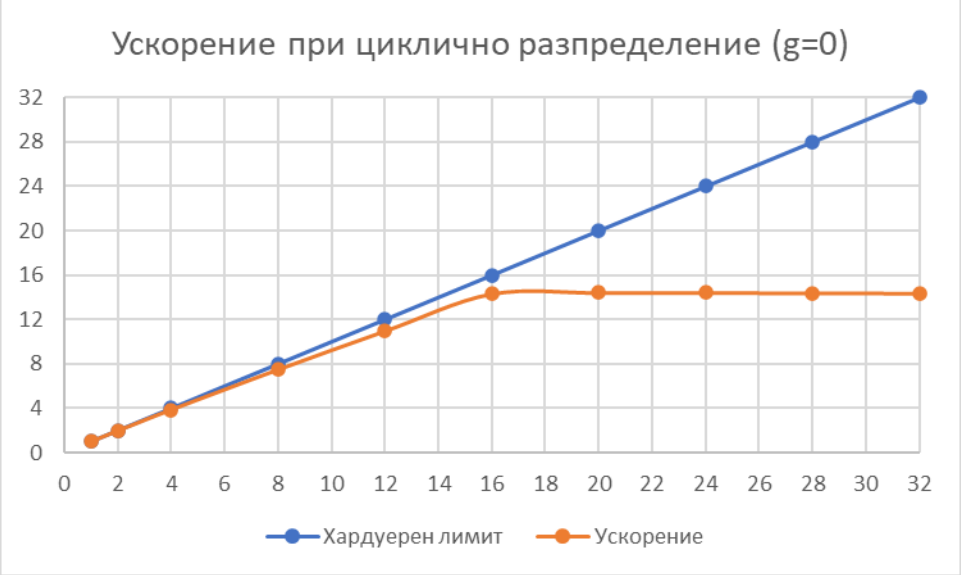
Тук стандартните отклонение са съответно 812 и 529 милисекунди при p=16 и p=32. Разликата в ускоренията е 1.5. Склонни сме отново да приемем хипотезата, че допълнителното балансиране го повишава. Възможно е и малка порция от ускорението след p=16 да е благодарение на hyperthreading.

Следващият тест е при коефициент на грануларност g=15:



Тук полученото ускорение при p=16 е 13.62, което е малко повече от това при средна грануларност. Пикът е при p=20 – 13.89. Колебанията в графиката след p=16 изглеждат неестествени. Причината за това е, че ако броят задания не се дели на g*p, тогава последната нишка заема остатъка от всяко g-то задание. В точките p=24 и p=28 се случва така, че последната нишка се товари с твърде много задания (заради неделимост на броя редове на 24*15 и 28*15). Можем да забележим това на следната хистограма за p=28:

Заедно с постигане на най-добро досега балансиране на заданията между нишките, последните тестове с циклично разпределение ще превъзмогнат и този недостатък:



Тук ускорението до p=16 е почти линейно и при p=16 достига 14.31. Върхът е при p=20 – 14.42, което е достатъчно малко увеличение, за да го обясним с hyperthreading. След това получаваме и очакваната немонотонна аномалия, тъй като хардуерният паралелизъм е 16, а баланса между заданията не се увеличава при по-висок от 16 паралелизъм.

4. Източници

[1] Isaac K. Gang , Parallel Implementation and Analysis of Mandelbrot Set Construction, School of Computing, University of Southern Mississippi, 2008

[2] Matthias Book, Parallel Fractal Image Generation, 2001

[3] Vincent Blanchon, Go: Goroutine, OS Thread and CPU Management, 20 ноември 2019

[4] Official Golang documentation - WaitGroup