Проект по Системи за Паралелна Обработка

Тема: Изобразяване на фрактал – зад. 19

Изготвил: Добромир Антонов Атанасов, Ф.Н. [redacted], Компютърни науки, 1-ви поток, 3-ти курс, 3-та група

Ръководители: проф. Васил Георгиев, ас. Христо Христов

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

1. Цел на проекта

Изобразяване на множеството на Маделброт, определено с формулата $z_n = exp(z_{n-1}^2 + c)$, като за целта се използват паралелни процеси (нишки), чрез които се разпределя работата по търсенето на точките от множеството на повече от един процесор.

Генерирането на множеството на Манделброт е една от тези изчислителни задачи, познати на английски като “embarrassingly parallel”. Това означава, че на практика не се нуждаем от никаква синхронизация между нишките, т.е. нишките работят асинхронно. Определянето дали дадена точка от комплексната равнина е част от множеството на Манделброт по никакъв начин не ни дава информация дали съседните на нея точки принадлежат на множеството. Точно затова при тази задача можем да получим много добро ускорение със средствата на паралелното програмиране.

2. Анализ

Първо ще разгледаме няколко паралелни реализации, генериращи множеството на Манделброт. Те генерират “класическото” множество на Манделброт, т.е. това, което се изчислява чрез формулата $z_n = z_{n-1}^2 + c$.

При [1] архитектурата на програмата е ***Master-Slaves***. Използваният паралелизъм е по данни (***SPMD***). Балансирането е ***статично***, като декомпозицията на проблема е извършена по интересен начин. Използвана е ***блокова декомпозиция***. Интересното в случая е, че въз основа на корен квадратен от броя нишки, които ще бъдат използвани, се изчислява височината и ширината на блоковете, на които ще бъде разделено изображението. По този начин изображението се разбива на редове и колони. Например, ако изображението е 1600x1600 и се използват 64 нишки, то тогава всеки ред и колона се обработва от sqrt(64) = 8 нишки. Понякога, при определен брой нишки, изображението не може а бъде добре разделено на блокове, т.е. остават пиксели, които не са част от образуваните блокове. Реализацията се грижи за това - пускат се отделни нишки, които да обработват тези остатъчни части.

- В горната реализация е използван езикът C++.

При [2] архитектура на програмата е ***Master-Slaves***, реализираният паралелизъм е по данни - ***SPMD*.** Използван е ***static load balancing*.** Декомпозицията на матрицата е ***блокова по редове*.** Използвана е ***едра грануларност*.** Всяка от нишките взима по (*height / number of threads)* на брой последователни редове.

Подобно разделяне на проблема често води до ***неравномерно разпределяне на работата между нишките, вследствие на което някои нишки работят много по-дълго от други, а програмата завършва тогава, когато завърши и последната нишка*.** Въпреки, че не е предоставен анализ на ускорението, смятам, че такава декомпозиция води до лоши резултати. В проекта ще бъде реализирана тази декомпозиция с цел сравняване на резултатите с други решения.

- В горната реализация е използван езикът C++, като за изобразяването на фрактала е използвана библиотеката OpenGL (Open Graphics Library).

В [3] са описани множество реализации. Избрал съм да анализирам две от тях. Всяко едно от решенията, описани в статията, е на псевдокод.

При първата реализация архитектура на програмата е ***Master-Slaves***, реализиран е ***SPMD*** паралелизъм, т.е. паралелизъм по данни. Използван е ***static load balancing***, т.е. предварително е ясно коя нишка кои части от матрицата ще обработва. Декомпозицията е ***блокова***. В конкретния случай се работи със 6 нишки и матрицата се разделя на 6 равни части. Получават се 2 реда, като всеки от тях се състои от по 3 матрици.

Подобно разделяне на товара между нишките често *не е оптимално*. ***Грануларността е едра*,** което е предпоставка за неравномерно разпределяне на работата между нишките. От направените тестове се вижда, че това разделяне на матрицата не води до добри резултати.

При втората реализация от [3] архитектурата на програмата отново е ***Master-Slaves*,** паралелизмът е по данни (***SPMD***), а балансирането на товара - ***статично***. Матрицата се декомпозира по редове и ***циклично всяка една от нишките взима по един ред*** за обработка, т.е. реализирано *е* ***статично циклично разпределяне на товара между нишките*** при ***фина грануларност*.** От получените резултати се вижда, че тази декомпозиция е по-добра от предишната - получава се по-добро ускорение. В този случай разпределението на товара е *равномерно между нишките* –

всяка нишка получава *както трудни, така и лесни* за изчисление редове. Получават се добри резултати.

Разгледаните отдолу решения на задачата са разпределени, като е използван MPI (Message Passing Interface). Въпреки концептуалните различия между *мултипроцесор и мултикомпютър***,** има идеи и части от решенията, които могат да бъдат реализирани и на мултипроцесор.

Отново ще разгледаме само част от решенията, описани в статия [4]. Всички решения от [4] са на C++.

При първото от тях архитектурата на програмата е ***Master-Slaves***, реализиран е ***SPMD*** паралелизъм и е използван ***dynamic load balancing*****.** Има един *главен процес*, който раздава задачи на работниците и след това получава резултатите от тяхната работа. В тази реализация главният процес *раздава по един пиксел* за обработка на всеки един от процесите *от опашката на чакащите.*

Получава се ***комуникационен свръхтовар***, защото всеки процес има ***много малко работа за вършене***, след което той се нарежда на опашката да предаде получените резултати и да получи нова задача. Времето, в което процесът-работник прекарва в чакане на опашка и синхронизация с главния процес доминира над времето, в което той върши съществена работа. Получените резултати не са добри.

При втората реализация от [4] отново архитектурата на програмата е ***Master-Slaves***, реализираният паралелизъм е по данни (***SPMD***) и се използва ***dynamic load balancing***. Този път обаче на всеки един процес от опашката *се дава цял ред за обработка***.**

В този случай *времето за комуникация и синхронизация с главният процес* не доминира времето на работа на *процеса-работник и не се получава комуникационен свръхтовар.* Получените резултати са по-добри.

Сравнителна таблица на разгледаните решения

	Архитектура на програмата	**Вид паралелизъм**	**Load balancing**	**Декомпозиция по данни**	**Коментар:**
[1]	**Master-Slaves**	**SPMD (по данни)**	**Static**	**Блокова декомпозиция**	Интересен начин на определяне на блоковете и разделяне на работата между нишките;
[2]	**Master-Slaves**	**SPMD (по данни)**	**Static**	**Блокова декомпозиция**	Едра грануларност; неравномерно разпределение на товара между нишките; лоши резултати;
[3], първа реализация	**Master-Slaves**	**SPMD (по данни)**	**Static**	**Блокова декомпозиция**	Едра грануларност; неравномерно разпределение на товара; лоши резултати;

[3], втора реализация	**Master-Slaves**	**SPMD (по данни)**	**Static**	**Статична циклична по редове**	Фина грануларност; по-равномерно разпределяне на работата между нишките; добри резултати;
[4], първа реализация	**Master-Slaves**	**SPMD (по данни)**	**Dynamic**	**Декомпозиция по клетките на матрицата**	Много фина грануларност; комуникационен свръхтовар; лоши резултати;
[4], втора реализация	**Master-Slaves**	**SPMD (по данни)**	**Dynamic**	**Декомпозиция на задачата по редове**	Грануларността е по-едра в сравнение с предната реализация; вече няма комуникационен свръхтовар; по-добри резултати в сравнение с предното решение;

3. Проектиране

След направения анализ на някои възможни решения, моят проект ще бъде реализиран по следния начин:

- софтуерна архитектура - ***Master-Slaves***;
- load balancing - ***static***;
- вид паралелизъм - по данни, т.е. ***SPMD***, т.е. ***множество нишки ще изпълняват една и съща подпрограма върху различни данни*** – тези различни данни ще се дефинират от начина, по който ***ще разпределя работата по генерирането на множеството между отделните нишки-работници***. Избрах да използвам паралелизъм по данни (***SPMD***), тъй като в конкретната задача определянето на принадлежност към множеството на дадена точка от комплексната *равнина по никакъв начин не ни дава информация за останалите точки.* ***Нуждаем се от колкото се може повече работници – нишки, които да извършват едно и също действие за всяка една точка от равнината – чрез дадената формула да определят дали точката е от множеството;***

Ще бъде използван ***static load balancing***, защото в конкретния проект ***няма динамично постъпване на данни*** *и* ***формиране на динамични задания.*** От сполучливите образци е видно, че могат да се получат *много добри* резултати със ***static load balancing*.**

Алгоритъм за определяне на принадлежност към множеството:

Програмата реализира подобен на класическия алгоритъм за генериране на множеството на Манделброт. Разликата е във формулата, чрез която се определя дали дадена точка от комплексната равнина е в множеството. Оригиналната формула е $z_n = z_{n-1}^2 + c$, а в конкретната задача проверката става с формулата $z_n = exp(z_{n-1}^2 + c)$.

Взимаме точка от комплексната равнина и прилагаме върху нея формулата $z_n = exp(z_{n-1}^2 + c)$:

1. Ако модулът на комплексното число, описващо дадената точка, стане по-голям от фиксирана стойност (в програмата тази стойност се задава чрез командния параметър **"-bound"**), то алгоритъмът определя точката като неучастваща в множеството на Манделброт и я оцветява в бяло. Стойността по подразбиране на командния параметър е 8.

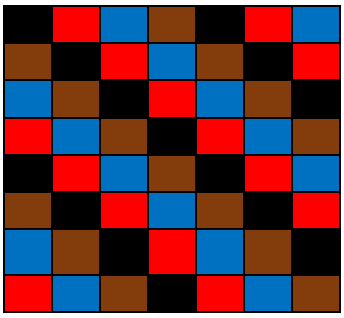
2. Ако пък броят пъти, които сме приложили рекурентната формула, стане равен на предварително зададено число (програмата приема това число чрез командния параметър **"-max-iter"**), което има стойност по подразбиране 256, и модулът на комплексното число все още не е станал по-голям от зададената граница в параметъра "**-bound"**, то алгоритъмът определя точката като част от множеството на Манделброт и я оцветява в черно. Това представлява ***escape-time*** алгоритъмът.

Разбира се, има точки, които много бързо клонят към безкрайност, при които са необходими много малко итерации, за да се излезе от множеството на Манделброт, но има и такива, които много по-бавно клонят към безкрайност. За получаване на по-добър визуален ефект може да се броят итерациите, необходими на точка да излезе от множеството на Манделброт и в зависимост от този брой съответната точка може да се оцвети с подходящ цвят, оказващ, че тя не е толкова далеч от множеството или пък, че е много далеч от него. В проекта е използван по-различен подход – или точка е част от множеството на Манделброт според зададените критерии и се оцветява в черно, или не е част от него и се оцветява в бяло.

В проекта се реализират няколко схеми на *разпределяне на товара между нишките:*

1-ва схема на декомпозиция:

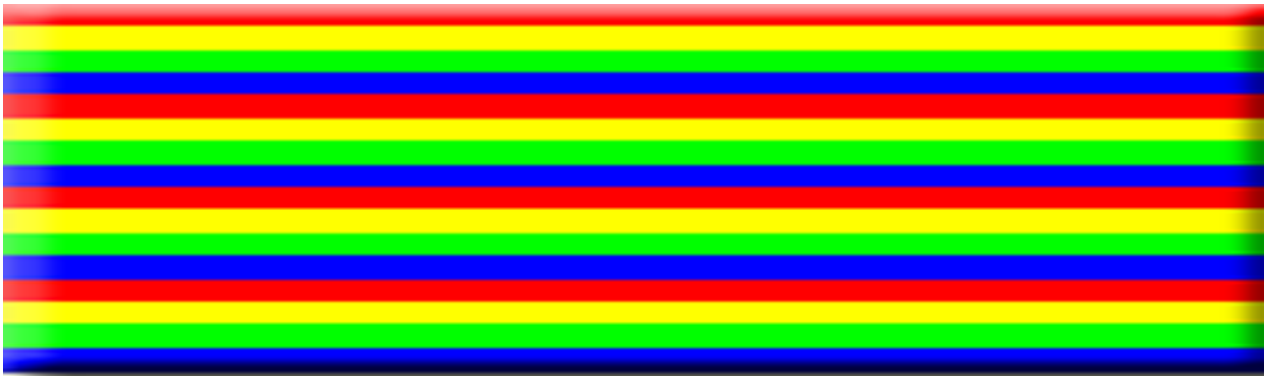
Схемата на декомпозиция при наличие на 4 нишки изглежда така:



- На картинката полетата, оцветени в един цвят, се обработват от една и съща нишка;

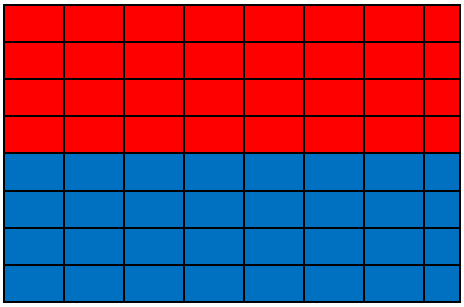
2-ра схема на декомпозиция:

Статична циклична декомпозиция по редове, като е използвана фина грануларност – нишка №1 обработва ред №1, ред № 1 + броя налични нишки, ред № 1 + (броя налични нишки * 2) и т.н. Нишка № 2 обработва редове № 2, № 2 + броя на наличните нишки, № 2 + (броя на наличните нишки * 2) и т.н. Декомпозицията изглежда по следния начин:



3-та схема на декомпозиция:

По-наивна схема на декомпозиция. Грануларността е едра, което често води до небалансирано разпределяне на работата между нишките. При наличие на 2 нишки декомпозиция 3 изглежда така:



Избрах да реализирам гореописаните схеми, защото:

1) открих много примерни реализации, използващи схема 2, но почти не намерих решения, базирани на схема 1. По същество схема 1, подобно на схема 2, реализира статично циклично разпределяне на работата между нишките, но разпределянето става по различен начин. Получените резултати ще бъдат сравнени;
2) схема 3 е реализирана с цел да се сравнят резултатите при една по-наивна декомпозиция с резултатите, получени при по-хитри декомпозиции;

Използвани технологии

За реализацията на проекта е използван езикът C++. Използваните нишки в реализацията са ***std::thread*** нишки от тип ***joinable***. Използвани са стандартни структури от данни като ***std::vector*****,** библиотека за работа със системен часовник – ***std::chrono*****,** библиотека за работа с комплексни числа – ***std::complex****.*

Описание на основните процедури и функции от кода

Процедурата initialize_variables прочита подадените командни параметри и чрез тях инициализира променливи, които ще бъдат използвани по-нататък в програмата. Не е необходимо да се подават всички командни параметри. Тези, които не са зададени, получават стойност по подразбиране.

Структурата thread_struct съдържа в себе си нишка std::thread, както и две променливи от тип time_point – thread_start_time и thread_end_time. По-нататък в програмата двете променливи от тип time_point ще бъдат *инициализирани съответно с времената на стартиране и завършване на нишката.* Структурата съдържа и променлива initial от целочислен тип, на която по-късно при изпълнението на програмата ще ѝ бъде присвоена стойност – идентификатор на нишката.

Процедурата executor е тази, която извършва основната работа и записва генерираното изображение накрая. Тя приема множество от параметри, един от които е vector<thread_struct>& vector_of_structs. В този вектор се пазят толкова обекти от тип thread_struct, колкото нишки са пуснати при изпълнението на програмата. Процедурата *стартира хронометър* – програмата е *започнала изпълнението си.* След това се пускат *една по една нишите,* пазени в обектите, елементи на подадения вектор. На всяка една от тези нишки ѝ се подава процедурата thread_runnable[1,2,3]. Процедурата thread_runnable1 реализира разбиването на работата между нишките по първата описана схема, thread_runnable2 – по втората, а thread_runnable3 – по третата.

При изпълнението на thread_runnable процедура от дадена нишка - *веднага при започването на процедурата се пуска хронометъра на нишката*, изпълняваща процедурата. След това по идентификатора на нишката, т.е. *индекса във вектора*, на който се пази обекта, от който нишката е част, *всяка една нишка определя клетките от матрицата* (при thread_runnable1) или *целите редове* (при thread_runnable2), или пък *последователността от редове* (при thread_runnable3), които ѝ се падат за обработка. След това точките от *комплексната равнина се скалират* въз основа на размерите на правоъгълника, подадени чрез команден параметър. Започва проверката дали точка е част от множеството. След като нишката *завърши* цялата работа, която ѝ се полага, *се спира хронометърът на нишката* и времето на завършване се записва в обекта, от който нишката е част.

Мастър-нишката (първоначалната нишка в програмата), пуснала всички нишки-работници, *изчаква завършването на всяка една от тях и веднага след това спира хронометъра, засичащ времето на програмата*, тъй като всички паралелни изчисления са завършени и на мастър-нишката остава само да запише изображението.

4. Тестов план

Програмата е изпълнявана върху тестовия сървър (тестова среда) и върху лаптоп (развойна среда).

Параметри на **тестовата среда**:

- два процесора Intel XEON E5-2660 (2.20 GHz, L1d cache: 32KB, L1i cache: 32KB, L2 cache: 256KB, L3 cache: 20480KB) всеки с по 8 ядра и hyperthreading;
- RAM: 64GB;
- g++ компилатор за C++ с флагове -std=c++11 и -O3(ниво на оптимизация);

Параметри на **развойната среда**:

- 4-ядрен процесор Intel Core I5-6300HQ (2.30 GHz, с Intel Turbo Boost до 3.20 GHz, L1 cache: 256KB, L2 cache: 1.0MB, L3 cache: 6.0 MB) без hyperthreading;
- RAM: 16GB;

В тестовата среда са проведени по 5 теста за всяка една от описаните декомпозиции. Всеки един от тестовете се състои в пускане на програмата при брой нишки p = 1, 2, 4, 6, 8, … , 32. Взети са най-добрите получени резултати. Генерираното изображение е с размери *4096x4096.*

В развойната среда са проведени по 5 теста за всяка една от описаните декомпозиции. При всеки един от тях програмата е пускана с брой нишки p = 1, 2, 4. Пускат се най-много 4 нишки, тъй като хардуерния паралелизъм е 4 и машината е без hyperthreading. Генерираното изображение е с размери *1024x1024.*

Времето се измерва в ***милисекунди.***

В следващите таблици и графики ***d1*** означава, че е използвана ***1-ва схема на декомпозиция, d2 – 2-ра схема на декомпозиция, d3 – 3-та схема на декомпозиция*****,*** където съответните схеми са описани в точката “Проектиране”.

Резултати от тестовете, изпълнени на сървъра:

1) При **d1:**

THREADS	T1	T2	T3	T4	T5	BEST	SP	EP
1	436228	385242	366516	354292	356110	354292	1	1
2	227012	187533	183689	179985	179686	179686	1.97172846	0.98586423
4	92768	99031	93939	92441	91321	91321	3.879633381	0.96990835
6	61493	69493	65302	61485	61752	61485	5.762250956	0.96037516
8	46749	47121	50935	48359	46788	46749	7.578600612	0.94732508
10	41511	38173	38386	38138	38307	38138	9.28973727	0.92897373
12	42659	33519	34512	37150	32160	32160	11.01654229	0.91804519
14	36633	30862	28212	34496	28372	28212	12.55820218	0.89701444
16	30687	33893	26939	28143	25399	25399	13.94905311	0.87181582

18	52790	28972	25793	27232	24810	24810	14.28020959	0.79334498
20	41793	28380	26623	28687	24074	24074	14.7167899	0.73583949
22	35796	23216	24025	28314	23113	23113	15.32868948	0.69675861
24	31697	30518	22956	22453	22574	22453	15.77927226	0.65746968
26	32034	36284	22213	24365	21880	21880	16.19250457	0.62278864
28	21939	22150	21554	25831	21132	21132	16.76566345	0.59877369
30	20543	20880	20870	27329	20524	20524	17.26232703	0.5754109
32	20054	21374	21733	21278	20422	20054	17.66689937	0.55209061

2) При **d2:**

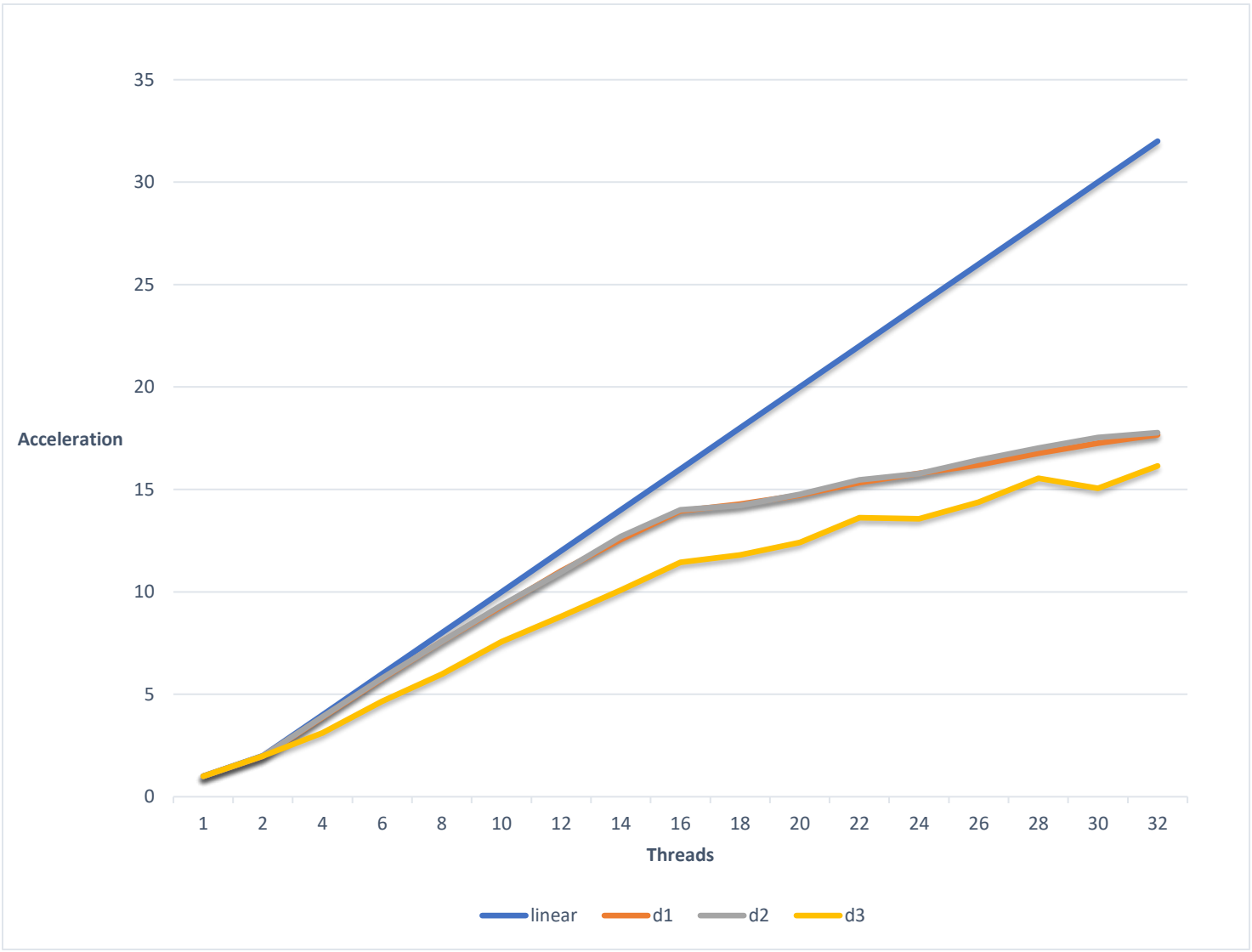
THREADS	T1	T2	T3	T4	T5	BEST	SP	EP
1	362935	404743	369072	371284	361556	361556	1	1
2	183214	185465	192491	189409	182305	182305	1.983247854	0.99162393
4	102691	97078	101490	106625	92814	92814	3.895489905	0.97387248
6	77435	71536	65558	63452	62432	62432	5.791196822	0.96519947
8	53759	54718	48035	48825	47701	47701	7.579631454	0.94745393
10	38947	39226	38986	39774	38673	38673	9.349054896	0.93490549
12	37633	41228	33023	37126	32989	32989	10.95989572	0.91332464
14	28594	35834	28627	34417	28471	28471	12.69909733	0.90707838
16	26194	33143	27702	32447	25828	25828	13.99860616	0.87491289
18	25438	29367	26406	32073	25433	25433	14.21601856	0.78977881
20	24485	25795	25229	30558	24536	24485	14.76642843	0.73832142
22	26642	24083	24137	30217	23386	23386	15.4603609	0.70274368
24	26344	30363	24943	23163	22925	22925	15.77125409	0.65713559
26	21997	36216	22553	22097	22170	21997	16.43660499	0.63217712
28	21421	29289	21728	21245	21319	21245	17.01840433	0.60780015
30	20720	24024	21064	20670	20620	20620	17.5342386	0.58447462
32	20804	23221	22045	20346	20352	20346	17.77037255	0.55532414

3) При **d3:**

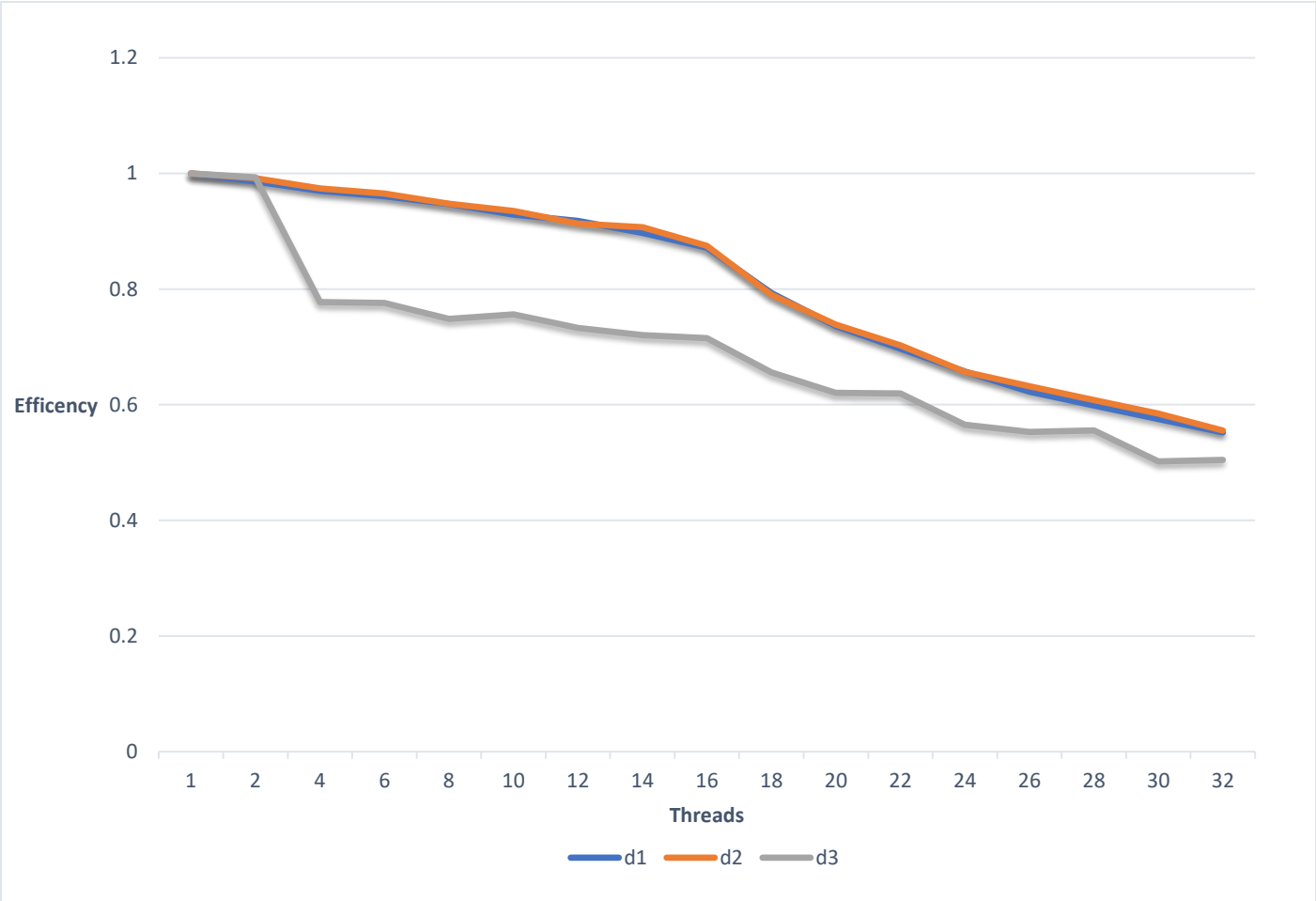
THREADS	T1	T2	T3	T4	T5	BEST	SP	EP
1	432286	411649	363982	360977	364237	360977	1	1
2	215162	236720	185807	181699	184108	181699	1.986675766	0.99333788
4	152309	119940	116716	118467	116033	116033	3.110985668	0.77774642
6	103681	77947	77947	77564	77525	77525	4.656265721	0.77604429
8	71385	60557	61357	60291	60490	60291	5.987245194	0.74840565
10	54008	48519	49021	47993	47733	47733	7.562420129	0.75624201

12	44639	41387	41817	41530	41043	41043	8.795092951	0.73292441
14	57436	36562	37822	35789	35811	35789	10.08625555	0.72044683
16	47498	32746	33020	31541	31609	31541	11.44469104	0.71529319
18	34844	31582	30838	30583	30892	30583	11.80319132	0.65573285
20	35791	30516	29992	29076	29368	29076	12.41494704	0.62074735
22	30005	27245	27395	26496	26775	26496	13.62383001	0.619265
24	27030	27183	28075	26880	26604	26604	13.56852353	0.56535515
26	25139	25677	26858	25113	25351	25113	14.37410903	0.55285035
28	23995	24308	25651	23213	23772	23213	15.55063973	0.55537999
30	23973	24978	25622	24463	24908	23973	15.05764819	0.50192161
32	22354	23780	24653	22514	22565	22354	16.14820614	0.50463144

Графика на ускорението:



Графика на ефективността:



Резултати от тестовете, изпълнени върху лаптоп с гореописаните параметри:

При **d1:**

THREADS	T1	T2	T3	T4	T5	BEST	SP	EP
1	163306	165755	163048	164977	165237	163048	1	1
2	89978	83256	84161	84413	84508	83256	1.958393389	0.97919669
4	44994	44216	43766	44435	45276	43766	3.725448979	0.93136224

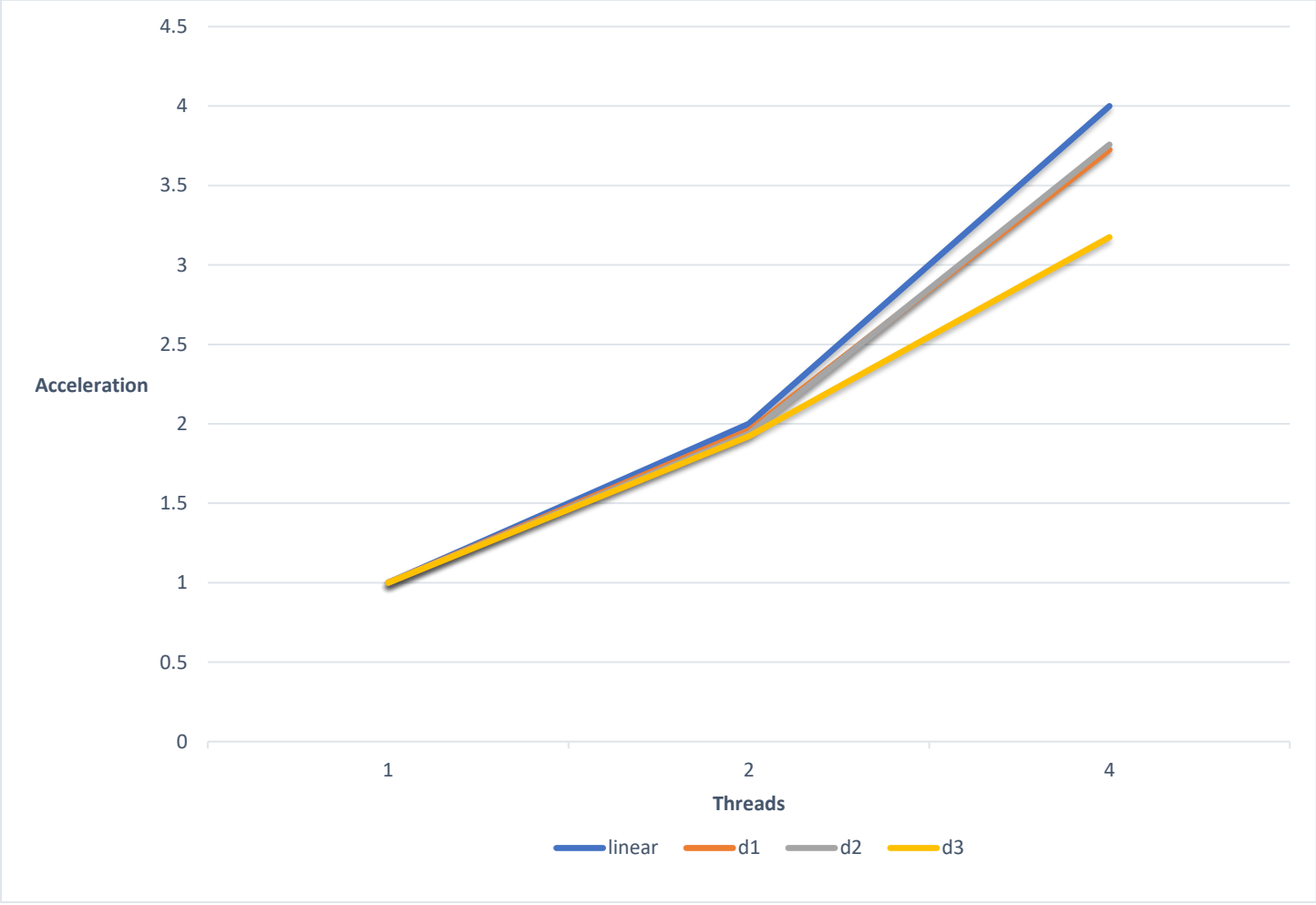
При **d2:**

THREADS	T1	T2	T3	T4	T5	BEST	SP	EP
1	163148	165118	164548	164589	165004	163148	1	1
2	85385	84193	87568	84669	85108	84193	1.937785802	0.9688929
4	46679	43588	43418	44611	45103	43418	3.75761205	0.93940301

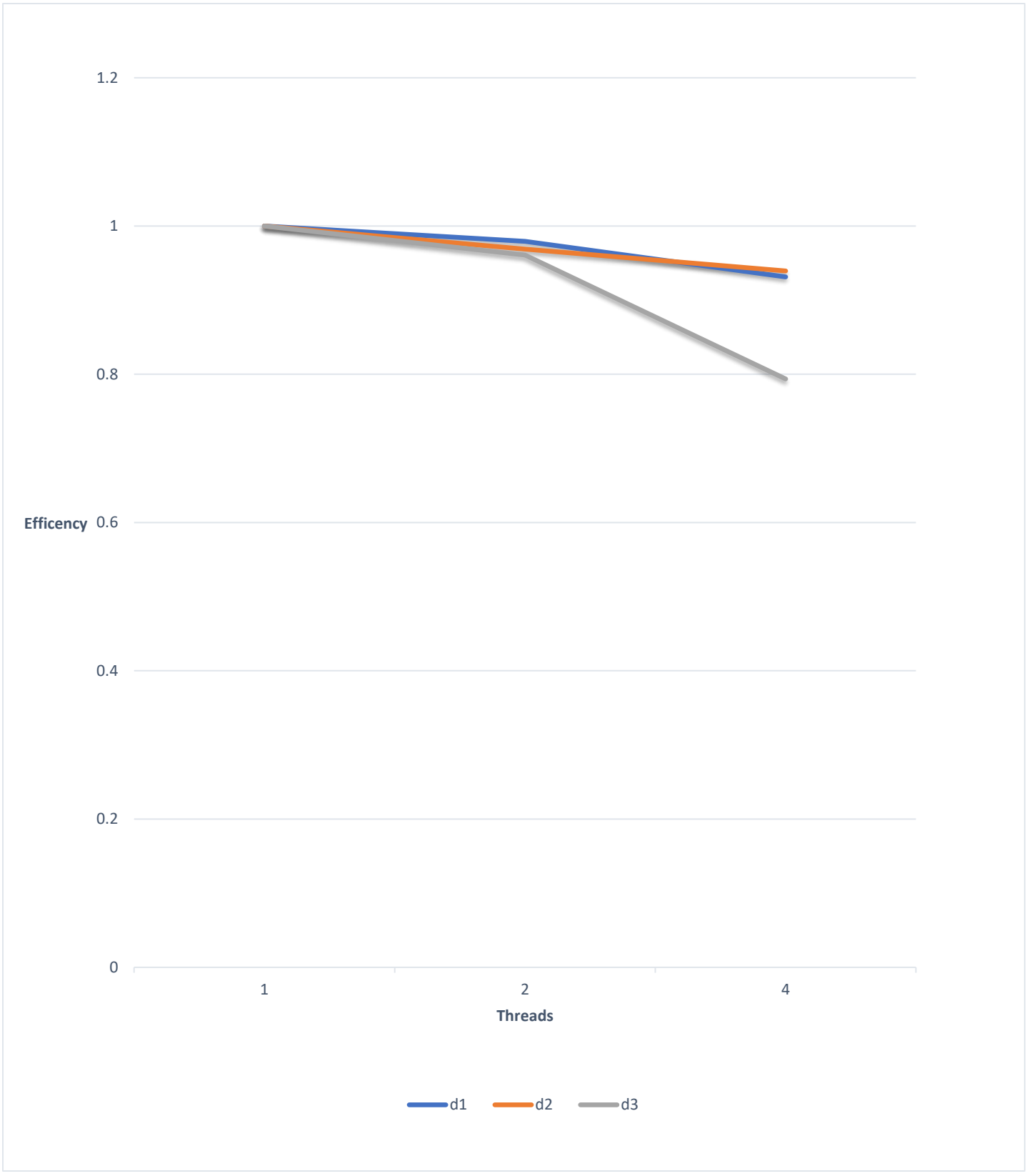
При **d3:**

THREADS	T1	T2	T3	T4	T5	BEST	SP	EP
1	168749	167150	164372	165347	166123	164372	1	1
2	85608	87184	86149	87310	85528	85528	1.921850154	0.96092508
4	53808	52150	53894	51773	52961	51773	3.174859483	0.79371487

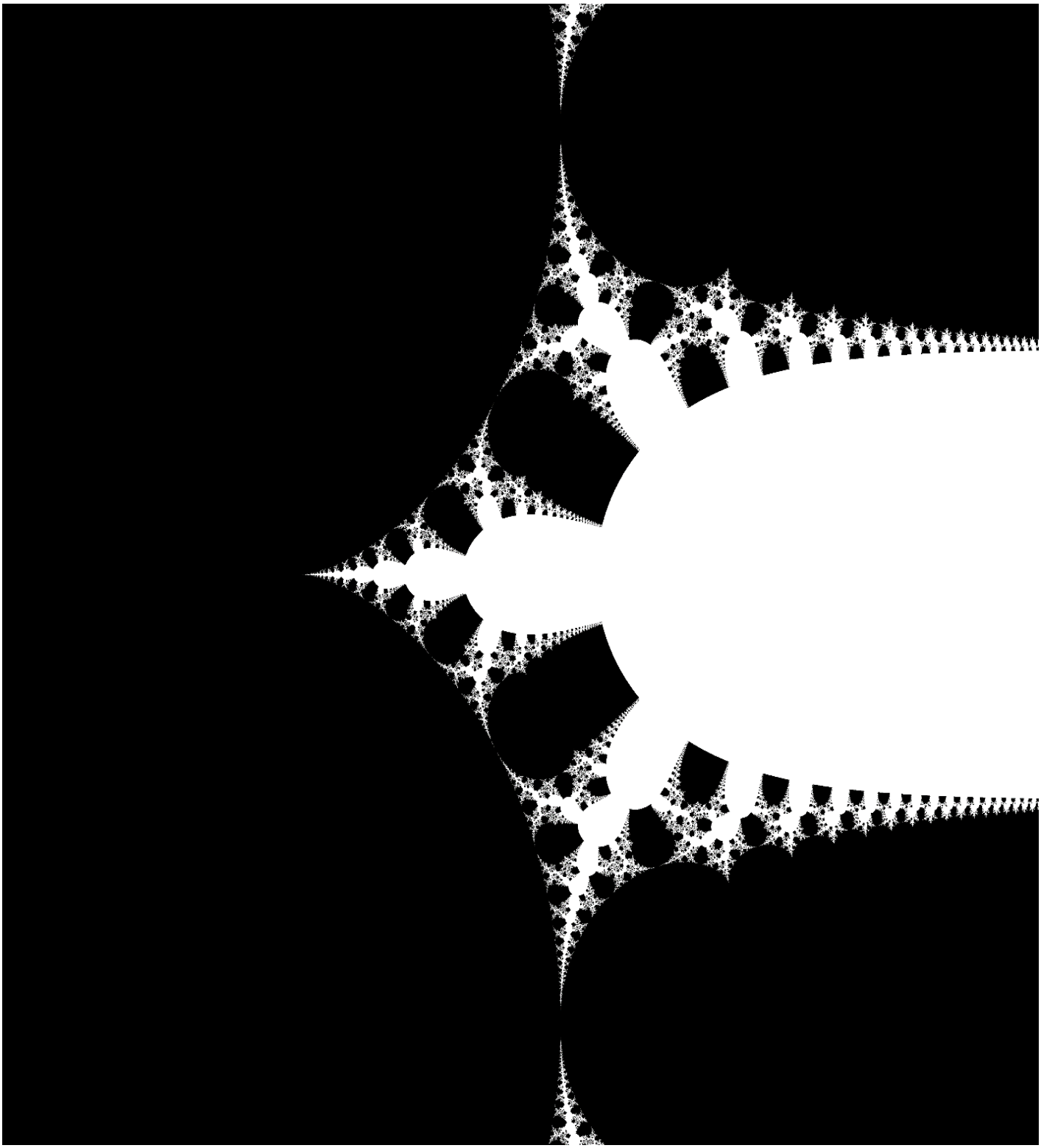
Графика на ускорението:



Графика на ефективността:



Краен резултат:



4096x4096

5. Изводи

От проведените тестове на сървъра и получените резултати е видно, че най-добро ускорение се получава при **d1** и **d2.** При тези две декомпозиции ускорението стига до около 18 при 32 нишки. Тестовият сървър има 16 ядра, така че ускорението, което получавам след 16-тата нишка, е в резултат на hyperthreading. От получените криви се вижда, че hyperthreading помага, тъй като след 16-тата нишка ускорението продължава да нараства с увеличаване на броя на нишките, въпреки че нарастването е незначително. Наивното разпределяне на товара между нишките - **d3** дава по-лоши резултати, което е напълно очаквано, тъй като грануларността при него е едра, а при едра грануларност е трудно товарът да се разпредели равномерно между нишките. В резултат на това някои нишки завършват много рано и бездействат, докато други завършват късно, тъй като им се е паднала много работа за вършене. При по-добрите декомпозиции грануларността е фина и това води до по-добри резултати. Причината е, че при фина грануларност е по-лесно балансирането на товара между нишките.

От тестовете, проведени в развойната среда, отново се вижда, че при **d1** и **d2** се получават най-добри резултати, а **d3** не води до толкова добро ускорение.

6. Източници

[1] Jofairden, C++ Mandelbrot with gradiented colors Multithreaded generation, 14 February 2018 (https://gist.github.com/Jofairden/56507b3184c7e16635ba66cd08c3d215)

[2] Andrey Gasparyan, mandelbrot_cpp, 8 January 2019 (https://github.com/gasparian/mandelbrot_cpp/blob/master/mandelbrot.cpp)

[3] Mirco Tracolli, Antonio Lagana, Leonardo Pacifici, Parallel generation of a Mandelbrot set, Department of Mathematics and Computer Sciences, Department of Chemistry, Biology and Biotechnology, University of Perugia, 21 April 2016 (http://services.chm.unipg.it/ojs/index.php/virtlcomm/article/view/112/108)

[4] Géza Várady, Bogdán Zaválnij, Mandelbrot, 2014 (https://regi.tankonyvtar.hu/hu/tartalom/tamop412A/2011-0063_23_introduction_mpi/ar01s12.html)

[5] Blaise Barney, Introduction to Parallel Computing, Lawrence Livermore National Laboratory (https://computing.llnl.gov/tutorials/parallel_comp/)

[6] R. Makinen, An Introduction to Parallel Computing (http://users.jyu.fi/~tro/TIEA342/material_for_tuesdays/intro_2.pdf)

[7] Joni, Let's draw the Mandelbrot set!, The Mindful Programmer, 17 November 2013 (https://jonisalonen.com/2013/lets-draw-the-mandelbrot-set/)

[8] Mandelbrot set, Wikipedia (https://en.wikipedia.org/wiki/Mandelbrot_set)

[9] Dr. Matthias Book, Parallel Fractal Image Generation, The Mandelbrot Set (http://matthiasbook.de/papers/parallelfractals/mandelbrot.html)

[10] Andrew Williams, Computing the Mandelbrot set, 1 September 1999
(https://plus.maths.org/content/computing-mandelbrot-set)