Софийски университет „Св. Климент Охридски”

Факултет по математика и информатика

Детерминанта на матрица

Изготвил: Иван Арабаджийски, КН III курс ФН: 81631

Проверил: …….
(ас. Христо Христов)

Съдържание

1.Условие на задачата

Да се напише програма която пресмята det A, където A е прочетена от файл или случайно генерираната матрица с размерност nxn. Работата на програмата по пресмятането на детерминантата да се раздели по подходящ начин на две или повече нишки (задачи).

2.Анализ на възможните решения

2.1. Формула на Лайбниц [1]

Един от наивните подходи за пресмятането на детерминанта на матрица е формулата на Лайбниц. В нея се генерират всички пермутации на индексите на редовете и стълбовете и се изчислява всеки член на сумата:

$$\det(A) = \sum_{\tau \in S_n} \operatorname{sgn}(\tau) \prod_{i=1}^{n} a_{i,\tau(i)} = \sum_{\sigma \in S_n} \operatorname{sgn}(\sigma) \prod_{i=1}^{n} a_{\sigma(i),i},$$

Най-интуитивния начин за паралелизирането на тази формула е като се пуснат няколко нишки, всяка от които генерира пермутация, пресмята произведението и го добавя към споделен между нишките масив от произведения. След приключване на работата на нишките резултатът се агрегира. Използва се метода Master-Slaves[2].

Предимства.

Първоначалната матрица не се изменя по време на изчислението. Изменянето на матрицата по време на изпълнението на програмата влияе отрицателно на ускорението. И обратно. Неизменението на матрицата по време на изчисленията позволява тя да се подава по референция на нишките и това влияе положително на ускорението.

Недостатъци.

Сложността на алгоритъма е O(N!). Това означава, че за доста малки матрици (от порядъка на 16x16) изчислението на детерминантата ще отнема много време, а дори при по-големи матрици това време ще бъде толкова голямо, че пресмятането няма да приключи докато сме живи. Друг недостатък на алгоритъма е това, че генерирането на пермутация е времеемко и ако използваме някакви синхронизационни примитиви това ще влияе отрицателно на ускорението.

2.2. Развитие на детерминанта по ред [3]

В условието на задачата е предложено да се използват адюлгирани количества и развитие на детерминанта по ред или стълб, метод изучаван в курса по линейна алгебра, затова се спираме и на този метод. Имаме квадратната матрица B от ред n, адюлгирано количество дефинираме като

$$C_{ij} = (-1)^{i+j} M_{ij} ,$$

където Mij е i,j-тия минор на матрицата B – детерминантата на (n – 1) x (n – 1) матрицата, която се получава като премахнем i-тия ред и j-тата колона на матрицата B.

Детерминантата на B предсмятаме по формуалта:

$$\begin{aligned} |B| &= b_{i1}C_{i1} + b_{i2}C_{i2} + \cdots + b_{in}C_{in} \\ &= b_{1j}C_{1j} + b_{2j}C_{2j} + \cdots + b_{nj}C_{nj} \\ &= \sum_{j'=1}^{n} b_{ij'}C_{ij'} = \sum_{i'=1}^{n} b_{i'j}C_{i'j} \end{aligned}$$

За да паралелизираме тази формула[4] пускаме нишки и назначаваме дадено развитие на B по ред, като след това всяка нишка рекурсивно смята детерминантата. След приключване на работата резултатът се агрегира. Използва се метода Master-Slaves[2].

Предимства.

Матрицата отново не се изменя по време на изчислението. Подматрицата, чиято детерминанта пресмятаме, може да се подаде по референция на нишките, което влияе положително на ускорението.

Недостатъци.

Сложността на алгоритъма е O(N!), което е прекалено бавно за да има истинско приложение в света. Изисква се и допълнителна обработка на матрицата, понякога и допълнително разбиване, за да се постигне по-добро балансиране на товара.

2.3. Гаусова елиминация [5]

Ще сведем дадената ни квадратна матрица B от ред n до триъгълна (тоест такава с елементи под главния диагонал нули) като използваме Гаусова елиминация[5]. След това ще използваме факта, че детерминантата на такава матрица се пресмята като произведение на елементите по главния диагонал[6].

$$\det(\mathbf{T}_n) = \prod_{k=1}^{n} a_{kk}$$

Паралелизирането на гаусовата елиминация е малко по-трудно, защото матрицата се изменя по време на изчисленията. Затова делигираме изменението на матрицата на множество нишки, докато частта от алгоритъма, която не може да се паралелизира се извършва от главната нишка. Тук също използваме Master-Slaves модела, като определянето на оста (pivot) се извършва от master нишката, а останалите изчисления от slaves нишките.

Предимства.

Сложност O(rN^3). Значително по-добра сложност от предишните разгледани варианти. Вече за времето, за което преди можехме да изчислим детерминанта на матрица 16x16 сега изчисляваме детерминанта на матрица 2048x2048 и нагоре.

Недостатъци.

Изчислението променя матрицата (mutability), което влияе на ускорението негативно, защото тъй като всяка итерация променя матрицата тези промени трябва да бъдат изчаквани от нишките. Имаме нетривиална синхронизация. Всеки ред/стълб се обработва след като е бил обработен предишния. Губи се точност.

2.3.1 LU декомпозиция [7]

LU декомпозицията е един от начините за намиране на детерминанта чрез декомпозиция и е модифицирана версия на Гаусовата елиминация. Работи като “голяма” матрица B се разбива на произведение на две матрици, долната матрица L(lower) и горната матрица U(upper), които са по-лесни за пресмятане. Декомпозицията се прави като матрицата A се разбива диагонално, така че всички елементи над диагонала на L са нули, а всички елементи под диагонала на U са нули. По този начин техните детерминанти се намират лесно като произведение на елементите по диагонала. В допълнение, елементите на диагонала на матрицата L са единици за улеснение. Използваме формулата за пресмятане:

$$u_{ij} = a_{ij} - \sum_{k=1}^{i-1} u_{kj} \cdot l_{ik}$$

$$l_{ij} = \frac{1}{u_{jj}} \left(a_{ij} - \sum_{k=1}^{j-1} u_{kj} \cdot l_{ik} \right)$$

Декомпозираме:

$$|A| = \begin{vmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{vmatrix} = \begin{vmatrix} 1 & 0 & 0 \\ l_{21} & 1 & 0 \\ l_{31} & l_{32} & 1 \end{vmatrix} \cdot \begin{vmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{vmatrix}$$

Така ще получим 9 неизвестни и 9 уравнения, които трябва да решим.

Основно подобрение пред гаусовата елиминация е сложността O(N^3+r*N^2).

3.Проектиране

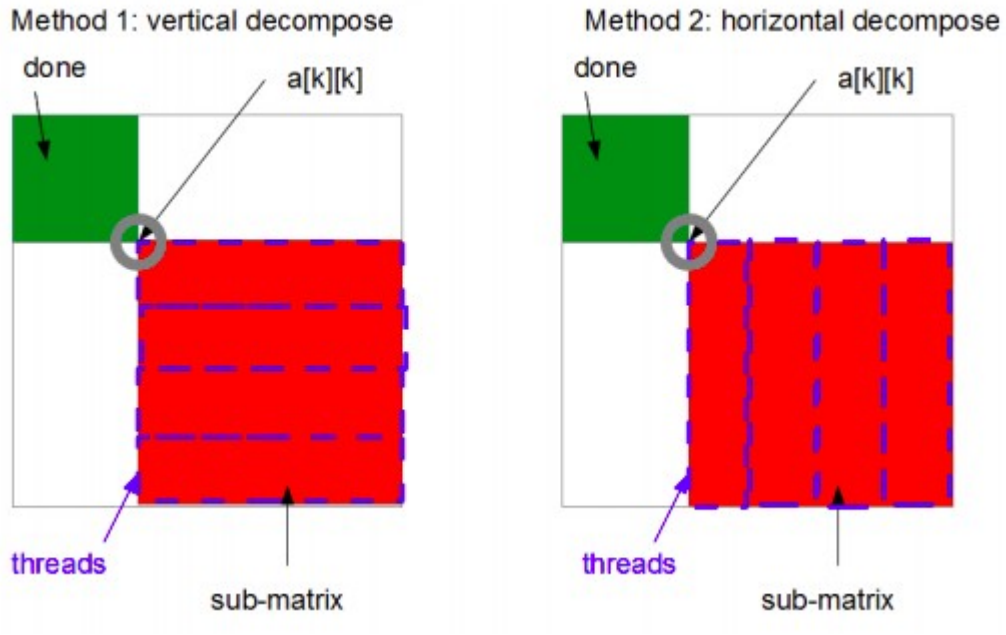
Първоначално подходих към проблема наивно, имплементирайки паралелен алгоритъм за развитие на матрица по ред и стълб. Макар и доволен от резултатите според мен сложността на самия алгоритъм го прави практически неизползваем, тъй като не можем да изчисляваме детерминанта на матрица от висок ред. След това имплементирах второ решение, използващо гаусова елиминация. По-надолу ще разгледаме всяко едно от решенията като анализираме проведени тестове и сравним резултатите. Проектът е писан на Java 8, която носи удобство при програми използващи паралелна обработка – класовете Thread и ThreadPool. Реализиран е паралелизъм по данни (Single Program-Multiple Data) или SPMD. Master-Slaves моделът е реализиран и в двата алгоритъма с оглед на анализа по горе.

3.1 Развитие по ред

Използваме design pattern-a Divide and Conquer – разделяме голямата задача на малки подзадачи и ги “раздаваме” на нишките, които извършват изчисленията директно променяйки общата променлива отговаряща за крайния резултат. Това е възможно поради имплементацията на клас AtomicBigInteger и в този случай няма нужда от синхронизиране. След получаване на задача всяка нишка пресмята детерминантата рекурсивно. Предвид факта, че можем да работим само с много малки матрици може да се случи така, че да имаме работещи повече нишки отколкото задачи сме приготвили. Стандартно при брой нишки <= n, където n е редът на матрицата, разбиваме матрицата на n задачи – по 1 за всеки елемент на първия ред. При брой нишки > n използваме по-фина грануларност – разбиваме задачите още веднъж като използваме рекурсията още веднъж в master нишката преди да сме пуснали slave нишките. Така получаваме n(n – 1) на брой задачи, които са винаги достатъчно при n <= 16 (тоест случаите, които можем да сметнем в човешко време) .

3.2 Гаусова елиминация

Тук използваме опашка от хомогенни задачи като всяка нишка взима следващата поред задача от опашката и я изпълява. Секцията с елиминирането може да бъде паралелизирана по два начина[8]:



Вертикално и хоризонтално. При хоризонталната декомпозиция всяка нишка работи по ред с индекс I и понеже няма връзка между редовете това позволява на нишките да работят паралелно. След като всички са готови резултатът се синхронизира и изчисленията преминават към следващия диагонален елемент. Аналогично за хоризонталното.

Други стандартни шаблони като pipeline и производител-потребител модела в нашия случай са неприложими с оглед на анализа по-горе. Имаме също реализирано динамично балансиране – на всяка итерация динамично намираме най-подходящия диагонален елемнт, според който раздаваме задачите след това.

4.Проведени тестове и измервания

Тестовете са проведени на машини с характеристиките:

Персонален компютър:
Architecture: x86_64
CPU op-mode(s): 32-bit, 64-bit
Byte Order: Little Endian
CPU(s): 4
On-line CPU(s) list: 0-3
Thread(s) per core: 1
Core(s) per socket: 4
Socket(s): 1
NUMA node(s): 1
Vendor ID: GenuineIntel
CPU family: 6
Model: 158
Model name: Intel(R) Core(TM) i5-7300HQ CPU @ 2.50GHz
Stepping: 9
CPU MHz: 900.059
CPU max MHz: 3500.0000
CPU min MHz: 800.0000
BogoMIPS: 4999.90
Virtualization: VT-x
L1d cache: 32K
L1i cache: 32K
L2 cache: 256K
L3 cache: 6144K
NUMA node0 CPU(s): 0-3

ФМИ сървър:
Architecture: x86_64
CPU op-mode(s): 32-bit, 64-bit
Byte Order: Little Endian
CPU(s): 32
On-line CPU(s) list: 0-31
Thread(s) per core: 2
Core(s) per socket: 8
Socket(s): 2
NUMA node(s): 2
Vendor ID: GenuineIntel
CPU family: 6
Model: 45
Model name: Intel(R) Xeon(R) CPU E5-2660 0 @ 2.20GHz

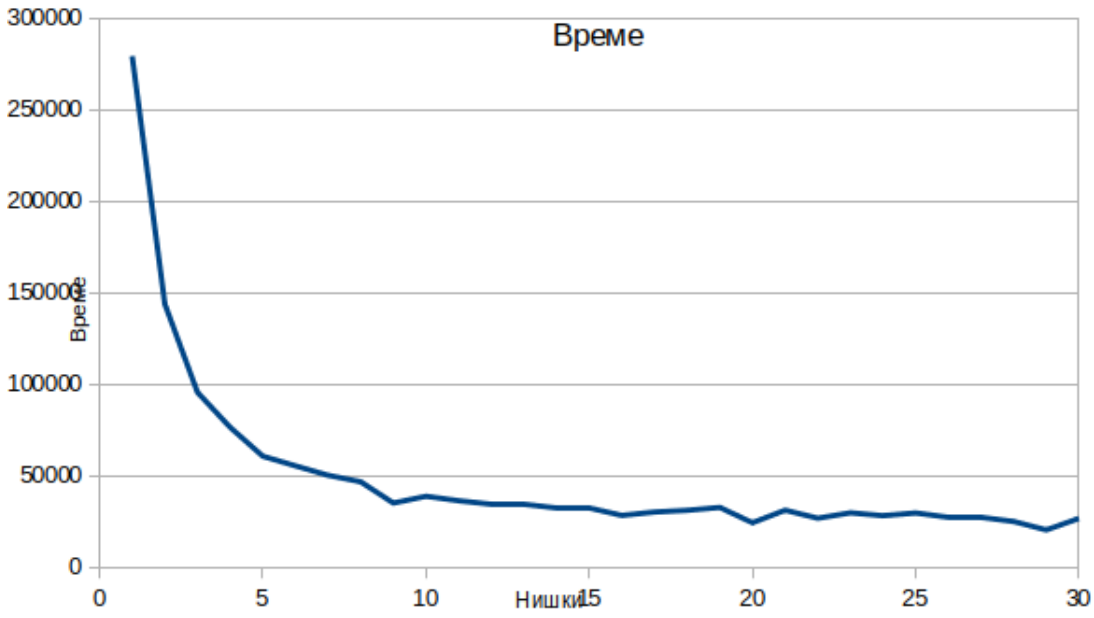
Stepping: 7
CPU MHz: 2087.072
CPU max MHz: 3000.0000
CPU min MHz: 1200.0000
BogoMIPS: 4389.58
Virtualization: VT-x
L1d cache: 32K
L1i cache: 32K
L2 cache: 256K
L3 cache: 20480K
NUMA node0 CPU(s): 0-7,16-23
NUMA node1 CPU(s): 8-15,24-31

За алгоритъмът, който използва развитие по ред, използваме матрица от ред 12, а за този, който използва гаусова елиминация използваме матрица от ред 1024. Елементите на матриците са в интервала [0, n], където n е редът на матрицата. За аномалия се приема, когато даден елемент на матрица е 0 – тогава изчисленията на нишката стават 0 и нейаната работа приключва значително по-бързо от тази на останалите нишки. Това влия на ускорението положително, тъй като имаме опашка от задачи и нишката, чиято работа приключи по-бързо ще си вземе нова задача от опашката веднага. Всеки тест е проведен 3 пъти и е взет най-добрият резултат.

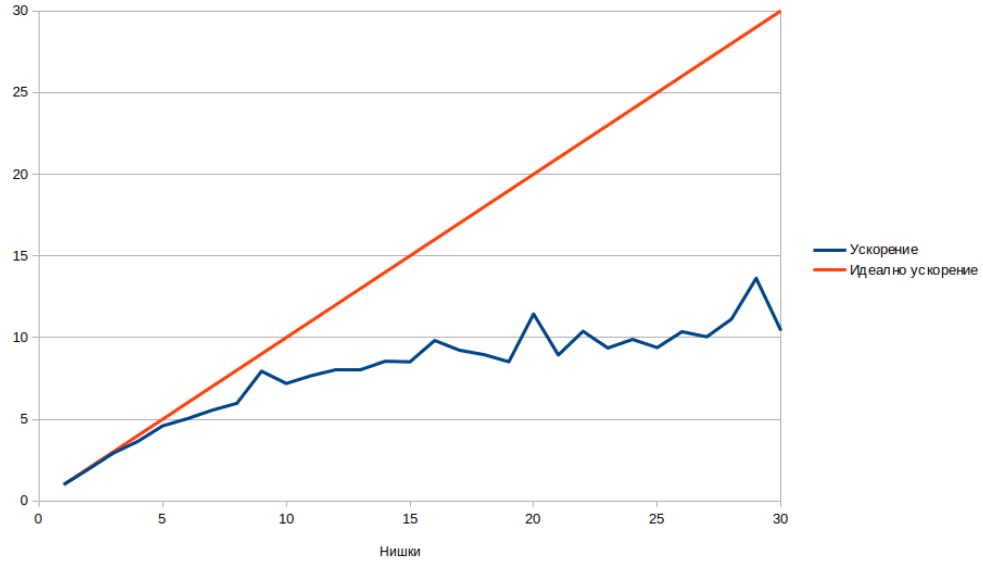
4.1 Тест 1

Ред: 1024
Алгоритъм: Gaussian Elimination
Машина: ФМИ сървър

Брой нишки	Време	Ускорение	Ефективност
1	279268	1	1
2	143848	1.94141037762082	0.970705188810411
3	95668	2.91913701551198	0.973045671837326
4	76578	3.64684374102223	0.911710935255556
5	60789	4.59405484544901	0.918810969089802
6	55474	5.03421422648448	0.83903570441408
7	50318	5.55006160817203	0.792865944024575
8	46780	5.96981616075246	0.746227020094057
9	35179	7.93848602859661	0.882054003177401
10	38859	7.18670063563139	0.718670063563138
11	36450	7.66167352537723	0.696515775034294
12	34845	8.01457884918927	0.667881570765772
13	34815	8.02148499210111	0.6170373070847
14	32661	8.55050365879795	0.610750261342711
15	32810	8.51167327034441	0.567444884689627
16	28437	9.82058585645462	0.613786616028414
17	30280	9.22285336856011	0.542520786385889
18	31198	8.95147124815693	0.497303958230941
19	32788	8.5173844089301	0.448283389943689
20	24406	11.4425960829304	0.572129804146521
21	31270	8.93086024944036	0.42527905949716
22	26896	10.383254015467	0.471966091612136
23	29832	9.36135693215339	0.407015518789278
24	28256	9.88349377123443	0.411812240468101
25	29763	9.38305950341027	0.375322380136411
26	26965	10.3566845911367	0.398334022736025
27	27800	10.0456115107914	0.372059685584865
28	25100	11.1262151394422	0.397364826408651
29	20495	13.6261527201757	0.469867335178471
30	26788	10.4251157234583	0.347503857448609
31	24615	11.3454397724964	0.365981928145047
32	29165	9.57545002571576	0.299232813303617


Време
300000
250000
200000
150000
100000
50000
0
Време
0
5
10
15
20
25
30
Нишки

Ускорение

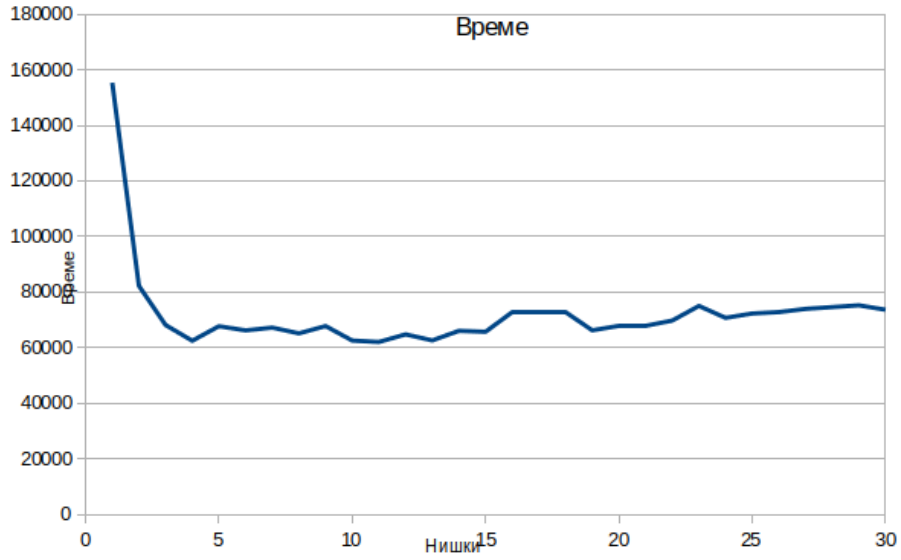

30
25
20
15
10
5
0
0
5
10
15
20
25
30
Нишки
Ускорение
Идеално ускорение

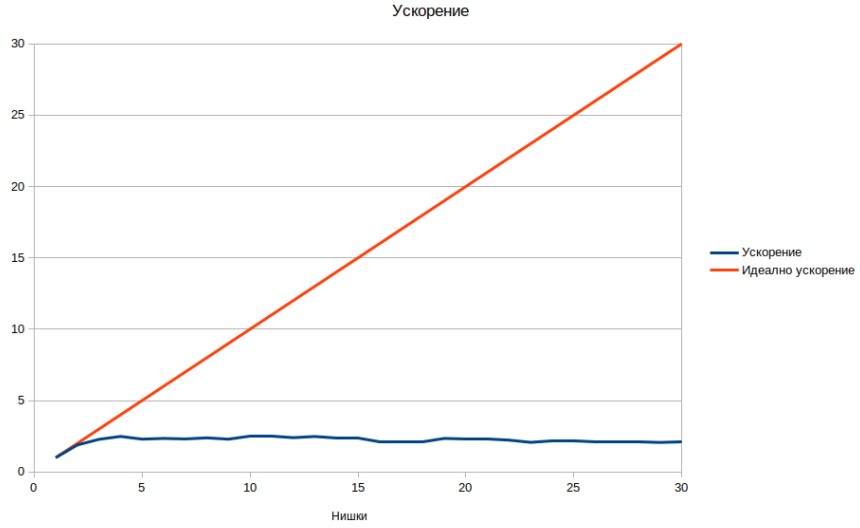

Ефективност
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
Ефективност
0
5
10
15
20
25
30
35
Нишки

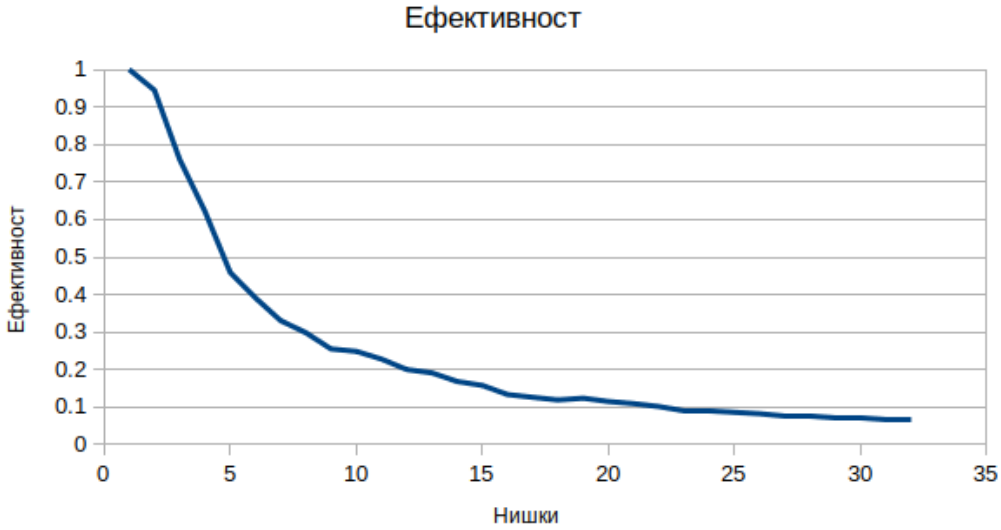
4.2 Тест 2

Ред: 1024
Алгоритъм: Gaussian Elimination
Машина: Персонален компютър

Брой нишки	Време	Ускорение	Ефективност
1	155297	1	1
2	82195	1.88937283289738	0.944686416448689
3	68114	2.27995713069266	0.759985710230887
4	62440	2.48713965406791	0.621784913516976
5	67673	2.29481477103128	0.458962954206257
6	66173	2.34683330059088	0.391138883431813
7	67163	2.31224037044206	0.330320052920294
8	65097	2.38562452954821	0.298203066193527
9	67738	2.29261271369099	0.254734745965665
10	62504	2.48459298604889	0.248459298604889
11	62007	2.50450755559856	0.227682505054415
12	64747	2.39852039476732	0.199876699563944
13	62572	2.48189285942594	0.190914835340457
14	66011	2.35259274969323	0.168042339263802
15	65689	2.36412489153435	0.15760832610229
16	72900	2.1302743484225	0.133142146776406
17	72595	2.1392244644948	0.125836733205576
18	72646	2.13772265506704	0.118762369725947
19	66190	2.34623054842121	0.123485818337959
20	67727	2.29298507242311	0.114649253621156
21	67875	2.28798526703499	0.108951679382619
22	69705	2.22791765296607	0.101268984225731
23	74970	2.07145524876617	0.090063271685486
24	70673	2.19740211962135	0.09155842165089
25	72250	2.14943944636678	0.085977577854671
26	72753	2.13457864280511	0.082099178569427
27	73921	2.10085090840221	0.077809292903786
28	74558	2.08290190187505	0.074389353638395
29	75191	2.06536686571531	0.071219547093631
30	73610	2.10972693927456	0.070324231309152
31	76029	2.04260216496337	0.065890392418173
32	72377	2.14566782265084	0.067052119457839


Време
180000
160000
140000
120000
100000
80000
60000
40000
20000
0
Време
0
5
10
15
20
25
30
Нишки


Ускорение
30
25
20
15
10
5
0
0
5
10
15
20
25
30
Нишки
Ускорение
Идеално ускорение


Ефективност
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
Ефективност
0
5
10
15
20
25
30
35
Нишки

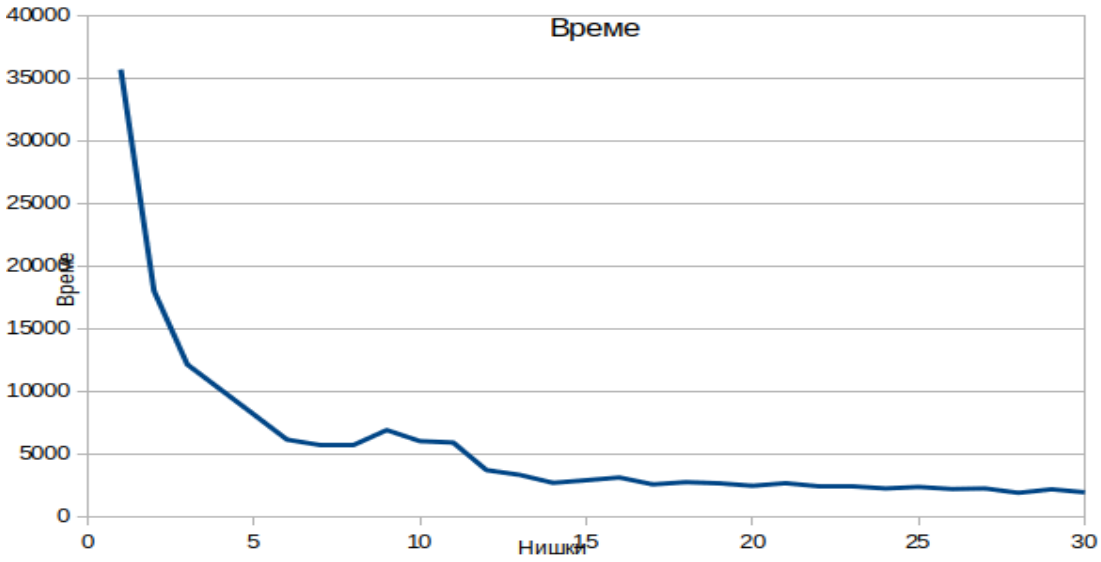
4.3 Анализ на резултатите

Първо ще разгледаме резултатите на ФМИ сървъра. Там успяваме да постигнем приблизително 10 пъти ускорение за 16 нишки. Обръщаме специално внимание и анализ на резултата за 16 пуснати нишки, защото знаем, че процесорът има 2x8 ядра, тоест можем да очакваме най-добри резултати в конкретно този случай. И действително най-добро ускорение получаваме точно там. Забелязваме, че сме ограничени хардуерно – можем максимално да полуичим ускорение 16 пъти. Нашите резултати са близо, но защо не можем да получим оптимакното ускорение? Това се дължи на естеството на алгоритъма. Както видяхме в анализа на Гаусовата елиминация можем да паралелизираме само частта от алгоритъма, която променя матрицата и изчислява детерминантата, но остава частта, която намира оста. Тя не може да се паралелизира и това допълнително ограничава ускорението. След това тестваме до 32 нишки заради hyperthreading-а на ядрата. Успяваме да получим минимално, но неконсистентно подобрение, което се дължи именно на него. Аналогични наблюдения можем да направим и при резулататите на персоналния компютър. Там сме ограничени хардуерно от четири пъти ускорение, заради четирите ядра, hyperthreading няма. И отново поради естеството на алгоритъма не можем да получим оптималното 4 пъти ускорение и виждаме, че най-доброто ускорение е 2.5 пъти при 4 нишки – отново достигнато в момента, в който очакваме. Също можем да наблюдаваме и забавяне при пускането на прекалено много нишки – след пускането на 23-4 нишки наблюдаваме забавяне, вероятно причинено от context switch-а на многото нишки.

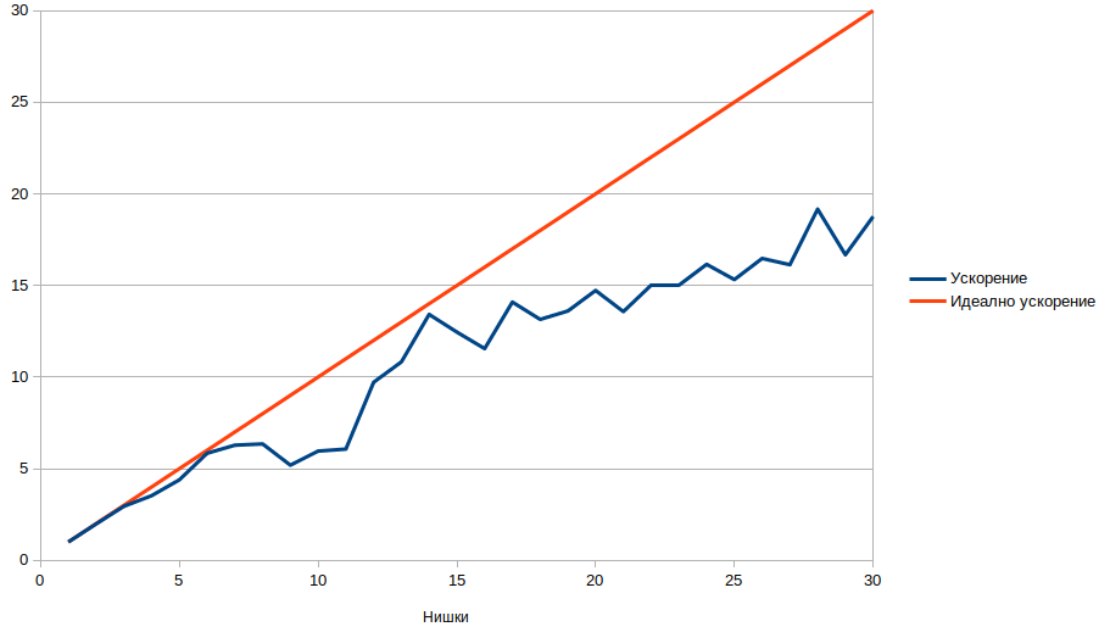
4.4 Тест 3

Ред: 12
Алгоритъм: Laplace expansion
Машина: ФМИ сървър

Брой нишки	Време	Ускорение	Ефективност
1	35666	1	1
2	16360	2.18007334963325	1.09003667481663
3	10812	3.29874213836478	1.09958071278826
4	7911	4.50840601693844	1.12710150423461
5	9122	3.90988818241614	0.781977636483227
6	5506	6.47766073374501	1.07961012229083
7	5678	6.2814371257485	0.897348160821215
8	5617	6.34965283959409	0.793706604949261
9	6869	5.19231329159994	0.57692369906666
10	5983	5.96122346648838	0.596122346648838
11	5876	6.06977535738598	0.551797759762362
12	3671	9.71560882593299	0.809634068827749
13	3294	10.8275652701882	0.832889636168325
14	2658	13.418359668924	0.958454262066
15	2864	12.4532122905028	0.830214152700186
16	3088	11.5498704663212	0.721866904145078
17	2531	14.0916633741604	0.828921374950613
18	2713	13.1463324732768	0.730351804070934
19	2622	13.602593440122	0.715925970532739
20	2422	14.7258464079273	0.736292320396367
21	2628	13.5715372907154	0.646263680510256
22	2373	15.0299199325748	0.6831781787534
23	2371	15.0425980598903	0.654026002603928
24	2208	16.1530797101449	0.673044987922705
25	2328	15.3204467353952	0.612817869415808
26	2165	16.4739030023095	0.63361165393498
27	2211	16.1311623699683	0.597450458146976
28	1861	19.1649650725416	0.684463038305059
29	2139	16.674146797569	0.574970579226516
30	1901	18.7617043661231	0.625390145537436
31	2163	16.4891354600092	0.531907595484169
32	1926	18.5181723779855	0.578692886812046


Време
40000
35000
30000
25000
20000
15000
10000
5000
0
Време
0
5
10
15
20
25
30
Нишки

Ускорение

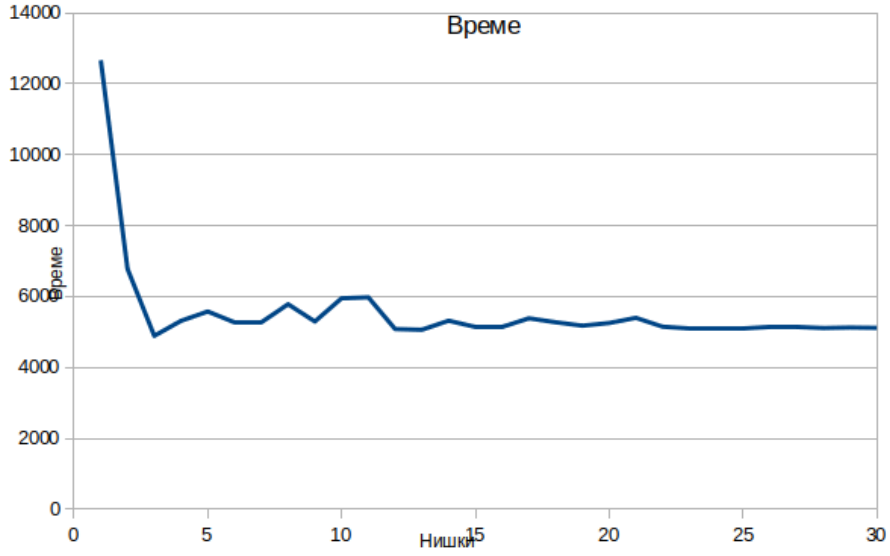

30
25
20
15
10
5
0
0
5
10
15
20
25
30
Нишки
Ускорение
Идеално ускорение

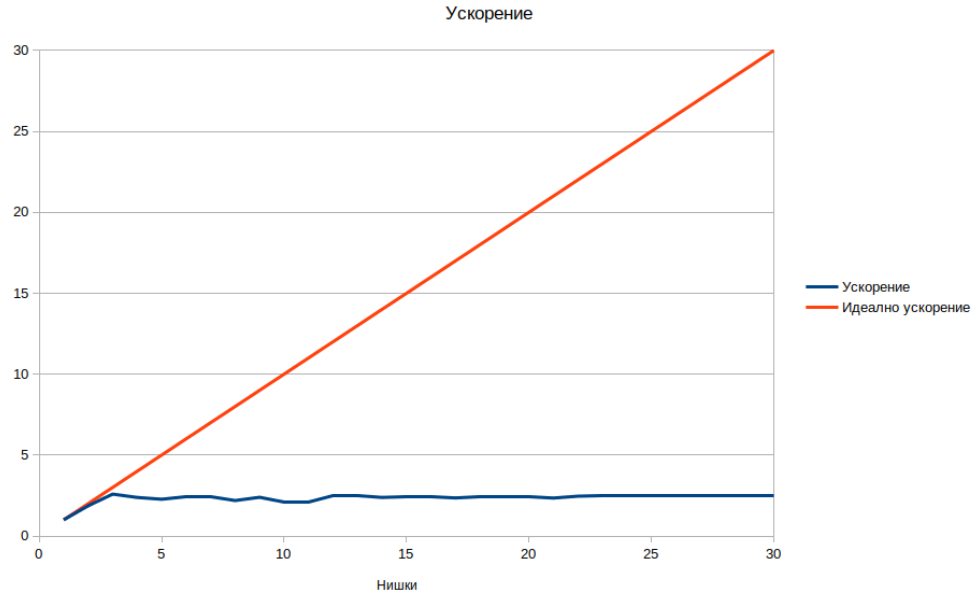

Ефективност
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
Ефективност
0
5
10
15
20
25
30
35
Нишки

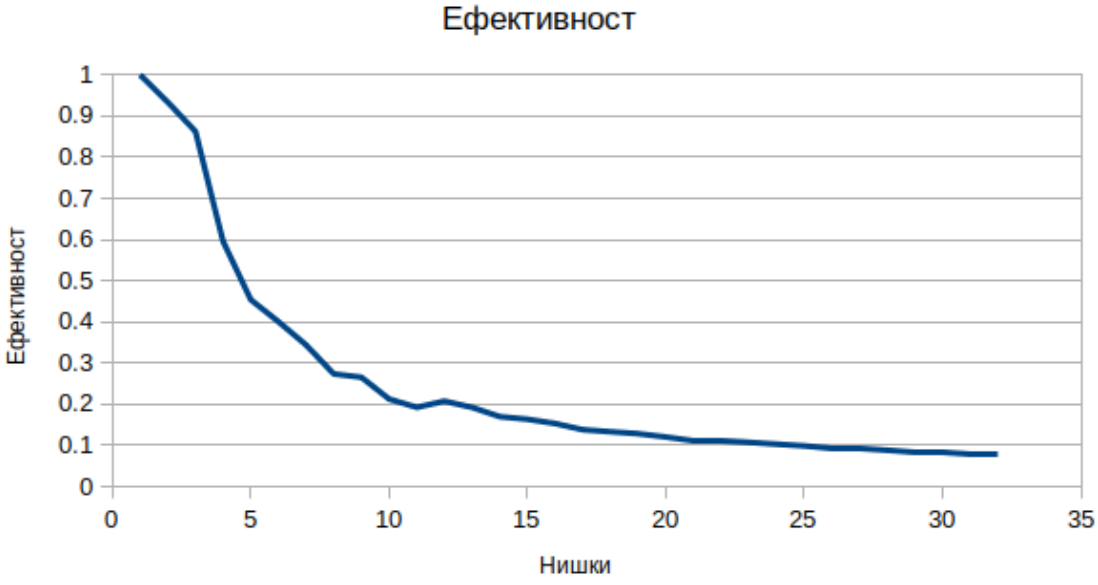
4.5 Тест 4

Ред: 12
Алгоритъм: Laplace expansion
Машина: Персонален компютър

Брой нишки	Време	Ускорение	Ефективност
1	12660	1	1
2	6780	1.8672566372	0.93362831858407
3	4891	2.5884277244	0.86280924146391
4	5315	2.3819379116	0.59548447789276
5	5578	2.269630692	0.45392613840086
6	5268	2.4031890661	0.400531511009871
7	5257	2.4082176146	0.3440310878013
8	5782	2.1895537876	0.27369422345209
9	5295	2.3909348442	0.26565942713252
10	5951	2.1273735507	0.21273735506638
11	5976	2.1184738956	0.19258853596203
12	5082	2.4911452184	0.20759543486816
13	5064	2.5	0.19230769230769
14	5318	2.3805942083	0.1700424434535
15	5159	2.4539639465	0.16359759643342
16	5158	2.4544397053	0.15340248158201
17	5386	2.350538433	0.13826696664555
18	5275	2.4	0.13333333333333
19	5178	2.4449594438	0.12868207598951
20	5253	2.4100513992	0.12050256996002
21	5399	2.3448786812	0.111660889582727
22	5148	2.4592074592	0.111782157236703
23	5118	2.4736225088	0.10754880473011
24	5112	2.4765258216	0.10318857589984
25	5111	2.4770103698	0.09908041479163
26	5127	2.4692802809	0.09497231849485
27	5125	2.4702439024	0.09149051490515
28	5112	2.4765258216	0.08844735077129
29	5126	2.4697619977	0.08516420681583
30	5117	2.4741059214	0.08247019738128
31	5097	2.4838140082	0.08012303252388
32	5099	2.4828397725	0.07758874289076


Време
14000
12000
10000
8000
6000
4000
2000
0
Време
0
5
10
15
20
25
30
Нишки


Ускорение
30
25
20
15
10
5
0
0
5
10
15
20
25
30
Нишки
Ускорение
Идеално ускорение


Ефективност
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
Ефективност
0
5
10
15
20
25
30
35
Нишки

4.6 Анализ на резултатите

Първо разглеждаме резултатите от ФМИ сървъра. Там забелязваме неконсистентно забавяне в промеждутъка 5-10 нишки, което може би се дължи на специфика на алгоритъма. Тъй като матрицата е от ред 12 тя бива разделена на 12 радачи, които се раздават на нишките работници (едра грануларност). Затова при 1-4 нишки ускорението се държи добре. След това при 5-10 нишки се получава така, че винаги има нишка, която да остане да извършва повече работа от другите – тя забавя цялостното изпълнение. Тук е моментът да уточним, че е тествана и по-фина грануларност – рекурсивно спускане още едно ниво надолу в алгоритъма. Това прави броят на задачите по-голям, а тяхната тежест – по-малка. Така заради сложността на алгоритъма (O(N!)) вече получаваме 12*11 = 132 задачи. След тестване се оказва, че по-фината грануларност не променя резултата – времето, отделено за създаването на новите задачи е приблизително толкова, колкото е спестеното време след това. След 12 нишки вече няма задача за всяка мишка и се налага задължителното прилагане на по-фина грануларност, за да можем да продължим да получаваме добро ускорение. Проблемът при резултатите на персоналния компютър е ясен – не можем да получим по-добро ускорение, защото сме ограничени хардуерно.

4.6 Източници

[1] Формула на Лайбниц за намиране на детерминанта - уикипедия
https://en.wikipedia.org/wiki/Leibniz_formula_for_determinants

[2] Master-Slaves
https://www.tutorialspoint.com/software_architecture_design/hierarchical_archit ecture.htm

[3] Laplace expansion – развитие на детерминанта по ред
https://en.wikipedia.org/wiki/Laplace_expansion#:~:text=In%20linear%20algebra%2C%20the%20Laplace,%C3%97%20(n%20%E2%88%92%201).

[4] New parallel algorithms for finding determinants of NxN matrices
https://www.researchgate.net/publication/261263046_New_parallel_algorithms_for_finding_determinants_of_NN_matrices

[5] Gaussian elimination
https://en.wikipedia.org/wiki/Gaussian_elimination

[6] Determinant of Triangular Matrix
https://proofwiki.org/wiki/Determinant_of_Triangular_Matrix

[7] LU decomposition
https://en.wikipedia.org/wiki/LU_decomposition

[8] Parallel Gaussian Elimination
Shule Li