Parall*e*l: Паралелно изчисление на цифри на числото *e* чрез числов ред

Автор: Станимир Петров, ф.н. 62285

Съдържание

1. Описание на проблема

Настоящият документ разглежда реализирато и изследването на ускорението при различни параметри на паралелното изчисление на цифри на Неперовото число, използвайки следния числов ред:

$$e = \sum_{k=0}^{\infty} \frac{1}{k!}$$

2. Анализ на образци

2.1. Функционален анализ

[1] y-cruncher

y-cruncher е програма, която може да изчислява π, e и други константи с точност до трилиони цифри. Тя позволява многонишково изпълнение на изчисленията и е със скалируема ефективност за многоядрени системи. Най-често се използва за сравнителен анализ и стрес-тестване на хардуера.

Паралелизмът се постига чрез рекурсивния fork-join модел, приложен върху многобройните алгоритми „разделяй и владей“ използвани от програмата.

Програмата позволява както изпълнение на много нишки, така и изпълнение на само една нишка. При изпълнение на много нишки, потребителят може да задава брой нишки и брой подзадания, на които да бъде разделена работата. Тези стойности, обаче, са само препоръчителни и често програмата ги закръгля до най-близката степен на двойката. Това е така понеже се цели възможно най-добра производителност, а повечето от използваните „разделяй и владей“ алгоритмни са двоични и се държат най-добре при разделяне на брой подзадания, който е степен на двойката, а свръхтоварът от създаването и унищожаването на нишките е пренебрежимо малък в сравнение със сложността на изчисленията, които програмата прави.

[2] Notable Large Computations: 50 000 000 000 decimal digits of Pi, WaiKin Wong & Rickie Chang, 2009

Резултати от паралелното изчисление на 50 000 000 000 цифри след десетичната точка на константата π върху машина с 4 процесора AMD Opteron 8356 @ 2.3 GHz. Тъй като за такова голямо изчисление е необходима много памет е използвано SSD FusionIO 600 GB за swap-памет. Това води до bottleneck и оттам ефективността на паралелната обработка е по-ниска.

Изчисленията са направени чрез програмата y-cruncher, като използваният алгоритъм е алгоритъмът на Chudnovsky. Изчислението е отнело 23 часа и 44 минути.

[3] Y-CRUNCHER - PI-1B, benchmark on Intel® Core™ i9-10980XE

Резултати от паралелното изчисление на 1 000 000 000 цифри след десетичната точка на константата π върху машина с процесор Intel® Core™ i9-10980XE.

Изчисленията са направени чрез програмата y-cruncher 0.7.8.9507 (Windows/17-SKX ~ Kotori) като използваният алгоритъм е алгоритъмът на Chudnovsky. Изчислението е отнело 14.387 секунди.

2.2. Сравнителна таблица

Образец	Константа	Брой цифри след десетичната точка	Паралелизъм	Балансиране	Грануларност	Ефективност на паралелната обработка
[2]	π	50 000 000 000	16	Статично	? (определена динамично от програмата)	74.7%*
[3]	π	1 000 000 000	18	Статично	? (определена динамично от програмата)	95.9%*
[Parallel]	e	159 330 955	16	Статично, циклично	едра, средна, фина	75.4%

* Дадената ефективност е на база на статистически изчисления на програмата за използването на процесора по време на изпълнение, а не на емпирично тестване.

2.3. Технологичен анализ

[2] Notable Large Computations: 50 000 000 000 decimal digits of Pi, WaiKin Wong & Rickie Chang, 2009

4 x **AMD Opteron 8356 @ 2.3 GHz**
Брой ядра: 4x4 (общо 16)
L1d cache: 64 KB
RAM: 128 GB

[3] Y-CRUNCHER - PI-1B, benchmark on Intel® Core™ i9-10980XE

Intel® Core™ i9-10980XE Extreme Edition Processor
Брой ядра: 18
L1d cache: 32 KB
RAM: 32 GB

3. Реализация

3.1. Използвани алгоритми и библиотеки

Програмата е реализирана на C++. За работа с големи числа е използвана библиотеката **GNU Multiple Precision Arithmetic Library (GMP)**. Използваният модел в програмата е **SPMD (single program, multiple data)**. Подзадачите се разпределят и биват изпълнявани едновременно на няколко нишки. Архитектурата на програмата следва модела Master/slave. Главната нишка (main thread) стартира нишките за изчисление на междинните резултати и изчаква те да завършат. След това обединява резултатите (като отново стартира нишки, ако все още има достатъчно работа за няколко нишки). Накрая, главната нишка преобразува получената обикновена дроб в десетична (с необходимата точност) и записва резултата във файл или го извежда на стандартния изход. В този случай Master/slave архитектурата не е абсолютно задължителна, но е избрана, тъй като прави кода много по-лесен за четене и дебъгване. Освен това моделът, при който главната нишка стартира P-1 на брой други нишки и извършва изчисления докато те работят, няма да донесе

почти никакво подобрение на производителността. Това е така, защото докато главната нишка чака останалите да завършат, операционната система я поставя в състояние на изчакване и тя не отнема процесорно време.

За стартирането на нишките е използвана функцията `std::async` (от C++11). Пример от кода за стартиране на i-тата нишка.

```
futures[i] = std::async(std::launch::async, partiallyCompute, start,
std::min(start+step, lastTermIndex), id--);
```

`std::async` връща обект от тип `std::future`, чрез който може да се достъпи върнатият резултат от изпълнението. Това става чрез метода `std::future::get()`, който чака нишката да завърши и извлича върнатия резултат. Пример от кода за изчакване и взимане на резултата от i-тата нишка:

```
BigFraction tmp = futures[i].get();
```

Забелязваме необходимостта от умножение на големи числа за изчисление на $k!$. Стандартният алгоритъм за умножение има сложност по време $O(n^2)$ за умножение на две n-цифрени числа. По-подходящо в случая би било да използваме алгоритъм за умножение на големи числа, базиран на Fast Fourier Transform (FFT). Такъв например е алгоримът на Schönhage–Strassen. Изборът на GMP библиотеката за работа с големи числа се дължи точно на това, тъй като тя имплементира FFT-базирани алгоритми за умножение на големи числа.[4] Сложността по време на тези алгоритми е от порядъка на $O(n * \log n * \log\log n)$.[5]

Нека разгледаме подходите за изчисляване на $n!$.

Наивният подход би бил да изчисляваме стойностите последователно: $x_1 = 1, x_2 = 2 * x_1, x_3 = 3 * x_2, \dots, x_n = n * x_{n-1}$. На итерация номер k, числото x_k съдържа $\mathrm{O}(k * \log k)$ цифри, следователно изчислението на x_{k+1} има цена $\mathrm{O}(k * \log^2 k)$. Така общата цена за изчислението на x_n е $\mathrm{O}(2 * \log^2 2) + \cdots + \mathrm{O}(n * \log^2 n) = \mathrm{O}(n^2 * \log^2 n)$.

Добрият подход е *binary splitting*[6]: той рекурсивно разделя произведението на последователни цели числа на две части. По-точно, нека $p(a,b) = (a+1) * (a+2) * \dots * (b-1) * b = \frac{b!}{a!}$. Изчислението на $p(a,b)$ се извършва по следния начин:

$$p(a,b) = p\left(a, \left\lfloor \frac{a+b}{2} \right\rfloor\right) * p\left(\left\lfloor \frac{a+b}{2} \right\rfloor, b\right),$$

където двата множителя се изчисляват рекурсивно по същия начин. Това позволява да изчислим $n! = p(0,n)$.

Нека разгледаме цената по време $C(a,b)$ на този процес. Нека $M(d)$ е цената за умножение на две цели числа с по d цифри. Множителите $p\left(a, \left\lfloor \frac{a+b}{2} \right\rfloor\right)$ и $p\left(\left\lfloor \frac{a+b}{2} \right\rfloor, b\right)$ съдържат $O\big((b-a)\log b\big)$ цифри, следователно

$$\mathrm{C}(a,b) = O\left(M\big((b-a)\log b\big)\right) + C\left(a, \left\lfloor \frac{a+b}{2} \right\rfloor\right) + C\left(\left\lfloor \frac{a+b}{2} \right\rfloor, b\right) \leq O\left(M\big((b-a)\log b\big)\right) + 2 * C\left(\left\lfloor \frac{a+b}{2} \right\rfloor, b\right)$$

откъдето следва че $\mathrm{C}(a,b) = O\left(\log(b-a) * M\big((b-a)\log b\big)\right)$.

Показахме, че $\mathrm{C}(0,n) = O\big(\log(n) * M(n\log n)\big)$. Използвайки, че сложността на умножението в нашия случай е от порядъка на $O(n * \log n * \log\log n)$, получаваме

$$\mathrm{C}(0,n) = O(n * \log^3(n) * \log\log n).$$

Следователно, за големи n е по-добре да използваме подхода *binary splitting*. Модификация на този подход е използван в програмата.

Нека

$$S(a,b) = (a-1)! * \sum_{k=a}^{b-1} \frac{1}{k!} = \frac{p_{a,b}}{q_{a,b}}, \qquad 0 < a < b$$

Нека дефинираме операцията $\circledast$ по следния начин: $\frac{a}{b} \circledast \frac{c}{d} = \frac{a*d+c}{b*d}$.

Забелязваме, че

(1) $$S(a,b) = \frac{p_{a,m}}{q_{a,m}} \circledast \frac{p_{m,b}}{q_{m,b}} = S(a,m) \circledast S(m,b)$$ за някое $a < m < b$.

Не е трудно да видим, че търсеният краен резултат е $1 + S(1,N)$, където N е броят членове, които потребителят е задал като точност на изчислението. Използвайки уравнение (1), разбиваме задачата на M (описано в предната точка) подзадания, които накрая трябва да обединим, използвайки операцията $\circledast$. Нещо повече – уравнение (1) ни позволява също да правим *binary splitting* при изчислението на всяко едно подзадание $S(i,j)$, като на всяка стъпка избираме $m = \left\lfloor \frac{i+j}{2} \right\rfloor$ и използваме $S(a,a+1) = \frac{1}{a}$ за дъно на рекурсията.

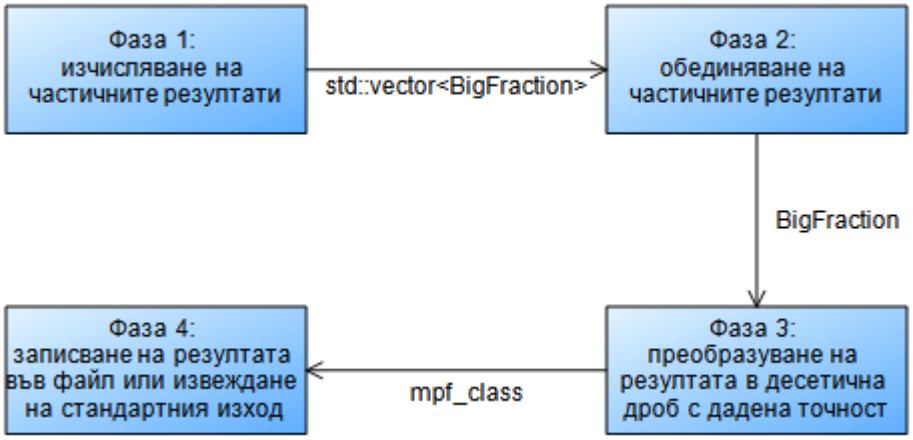
След като предсметнем крайния резултат $1 + S(1,N) = \frac{p_{final}}{q_{final}}$, остава да преобразуваме получената обикновена дроб в десетична. Тук идва моментът да кажем как преценяме с каква точност да изчислим десетичната дроб (колко цифри на числото ***e*** след десетичната точка). Един възможен подход е да изчислим N на брой цифри след десетичната точка. Това обаче не е добра идея – за големи N това не е никак оптимално, тъй като можем да получим доста повече цифри на база изчисленията, извършени в предните стъпки от хода на програмата. За да получим N десетични цифри е достатъчно да изчислим доста по-малко на брой членове на реда. За да достигнем до по-добър подход правим следното наблюдение:

Нека с d означим броя десетични цифри на числото $(N-1)! = q_{final}$. Ако $N \geq 10$, то $\frac{1}{N!} < 10^{-d}$. Разбира се оттук не следва, че със сигурност можем да твърдим, че сме пресметнали вярно числото ***e*** с точност d цифри след десетичната точка, тъй като грешката, която допускаме е $\sum_{k=N}^{\infty} \frac{1}{k!}$, а не просто $\frac{1}{N!}$. Въпреки този факт, това е доста добро приближение, тъй като редът е бързо сходящ. Все пак, за да направим вероятността за грешка близка до нула, в програмата „жертваме“ няколко цифри и изчисляваме ***e*** с точност $d - 10$ цифри след десетичната точка.

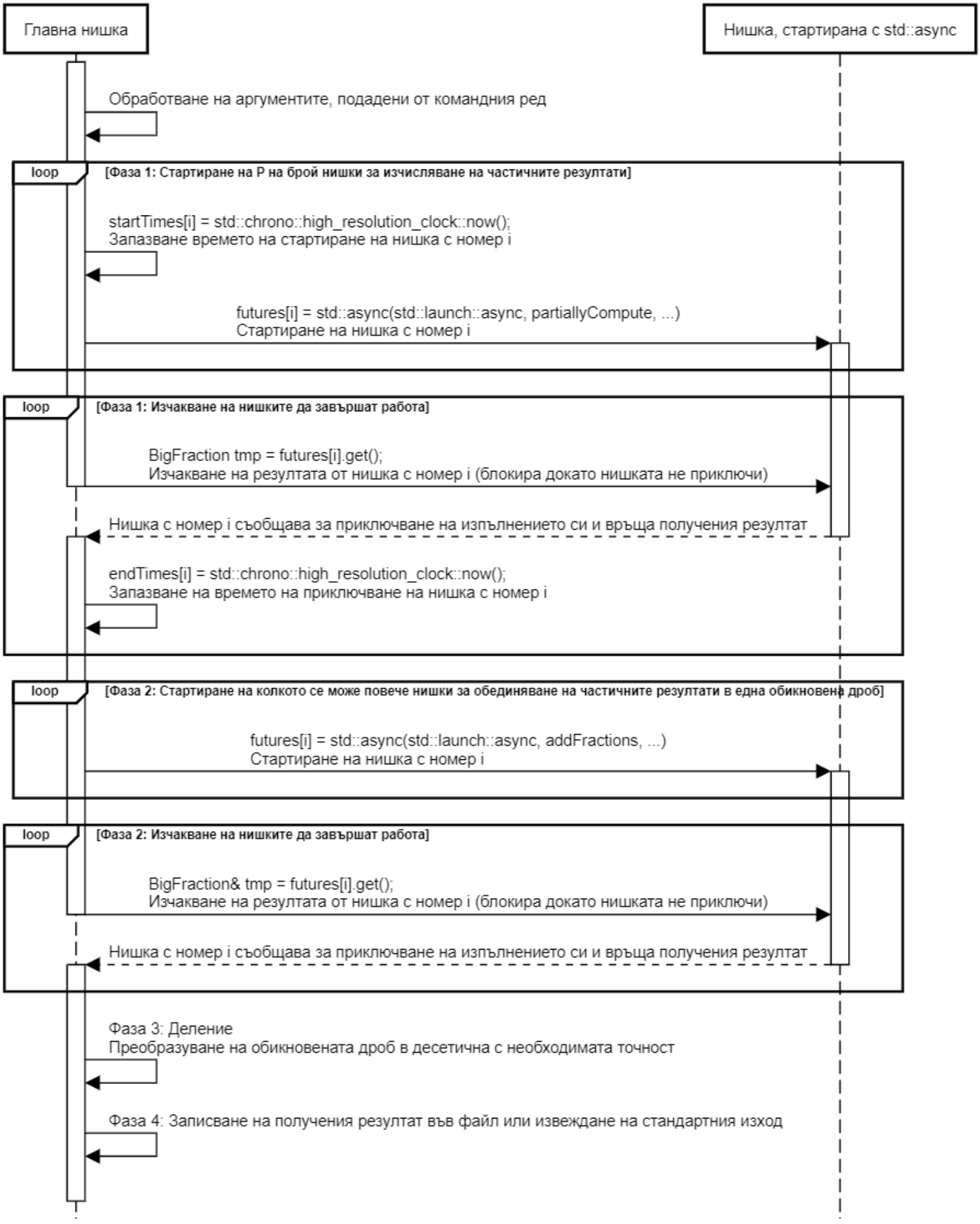
Коректността на програмата е проверена чрез сравнение на получените резултати с файл, съдържащ първите 160 000 000 цифри на числото ***e***.[7]

Извършването на финалното деление за преобразуване се извършва винаги от главната нишка. Поради това, тази фаза от изпълнението не е взета предвид при тестовете в следващата точка. Не би било никак лесно да се напише деление на големи числа, което да работи на колкото нишки му зададем и в същото време има оптимизации, подобни на тези на делението от GMP библиотеката. Може делението да се извършва на всяка стъпка вместо накрая, но, въпреки че би донесло малко по-добро ускорение на няколко нишки, това прави програмата в пъти по-бавна, тъй като в този случай трябва да заделя много повече памет, трябва да работи през цялото време с floating-point числа и да се извършват много деления, вместо едно. Тестване на такъв вариант на програмата не е показано в следваща точка, тъй като документът би станал твърде дълъг.

Записването на резултата във файл не е обект на паралелизиране (или поне не при наличието на единствен диск). Поради тази причина то също не е взето предвид при тестването в следващата точка.



Фигура 1. Опростена диаграма, показваща последователността на различните фази от изпълнението на програмата.


Главна нишка
Нишка, стартирана с std::async
Обработване на аргументите, подадени от командния ред
loop
[Фаза 1: Стартиране на P на брой нишки за изчисляване на частичните резултати]
startTimes[i] = std::chrono::high_resolution_clock::now();
Запазване времето на стартиране на нишка с номер i
futures[i] = std::async(std::launch::async, partiallyCompute, ...)
Стартиране на нишка с номер i
loop
[Фаза 1: Изчакване на нишките да завършат работа]
BigFraction tmp = futures[i].get();
Изчакване на резултата от нишка с номер i (блокира докато нишката не приключи)
Нишка с номер i съобщава за приключване на изпълнението си и връща получения резултат
endTimes[i] = std::chrono::high_resolution_clock::now();
Запазване на времето на приключване на нишка с номер i
loop
[Фаза 2: Стартиране на колкото се може повече нишки за обединяване на частичните резултати в една обикновена дроб]
futures[i] = std::async(std::launch::async, addFractions, ...)
Стартиране на нишка с номер i
loop
[Фаза 2: Изчакване на нишките да завършат работа]
BigFraction& tmp = futures[i].get();
Изчакване на резултата от нишка с номер i (блокира докато нишката не приключи)
Нишка с номер i съобщава за приключване на изпълнението си и връща получения резултат
Фаза 3: Деление
Преобразуване на обикновената дроб в десетична с необходимата точност
Фаза 4: Записване на получения резултат във файл или извеждане на стандартния изход

Фигура 2. Диаграма на последователностите (sequence), показваща последователността на различните фази от изпълнението на програмата.

3.2. Ръководство за потребителя

Програмата изчислява числото ***e*** със зададена от потребителя точност. Програмата получава следните параметри от командния ред.

- **-p** *N* – Задава точността на изчислението. $N \geq 20$ е броят членове на числовия ред, които да бъдат пресметнати. Параметърът е незадължителен. Стойност по подразбиране: 1000000.
- **-t** *P* – Задава максималния брой нишки, на които да бъде разпределена работата. *P* е число по-голямо или равно на 1. Параметърът е незадължителен. Стойност по подразбиране: 1.
- **-q** – „Тих“ режим на работа. Ако бъде зададен, програмата не извежда на стандартния изход подробна информация за етапите от работата си.
- **-o** *FILE* – Програмата записва резултата от работата си във файла *FILE*. Параметърът е незадължителен – ако не бъде зададен, програмата извежда резултата на стандартния изход.
- **-g** G – Задава грануларност. *G* е число по-голямо или равно на 1. Параметърът е незадължителен. Стойност по подразбиране: 1. В зависимост от версията на програмата има различно поведение:
    - **Версия с брой подзадания, зависещ от броя на нишките –** Разделя работата на $M = P * G$ подзадания.
    - **Версия с фиксиран брой подзадания –** Разделя работата на $M = G$ подзадания. В този случай е препоръчително $G \geq P$ или в противен случай ще има нишки, за които няма да има никаква работа.

4. Тестване и резултати

Резултатите от тестването са получени при изчисление на първите 23 000 000 члена на числовия ред (това са повече от 159 330 000 десетични цифри на числото ***e***).

Тестването е извършено на машина със следните параметри:

- Процесор: 2x Intel® Xeon® Processor E5-2660
- Брой ядра: 2x8 (общо 16)
- L1d cache: 32 KB
- RAM: 64 GB

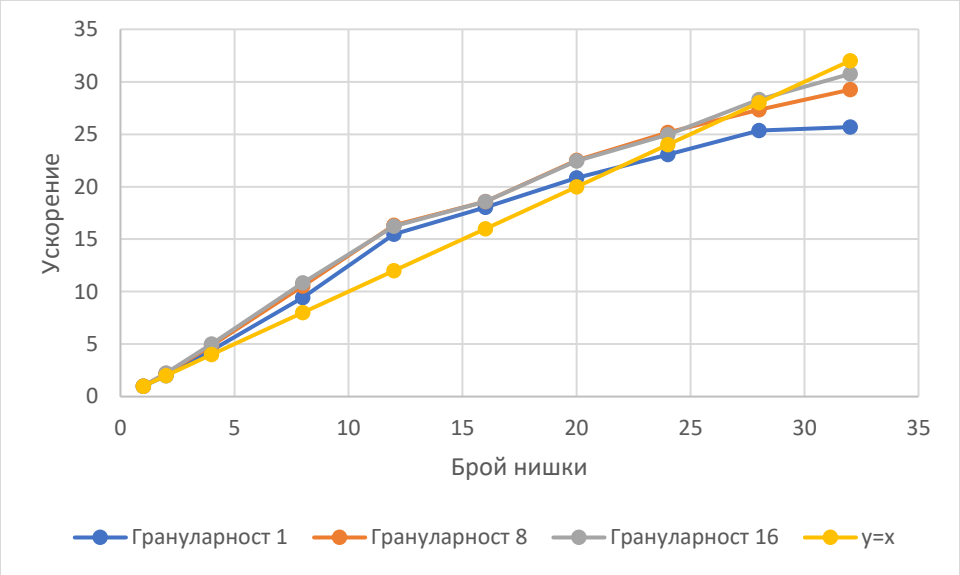
4.1. Тестване на версията с брой подзадания, зависещ от броя нишки

В тази версия на програмата броят подзадания е $M = g * p$.

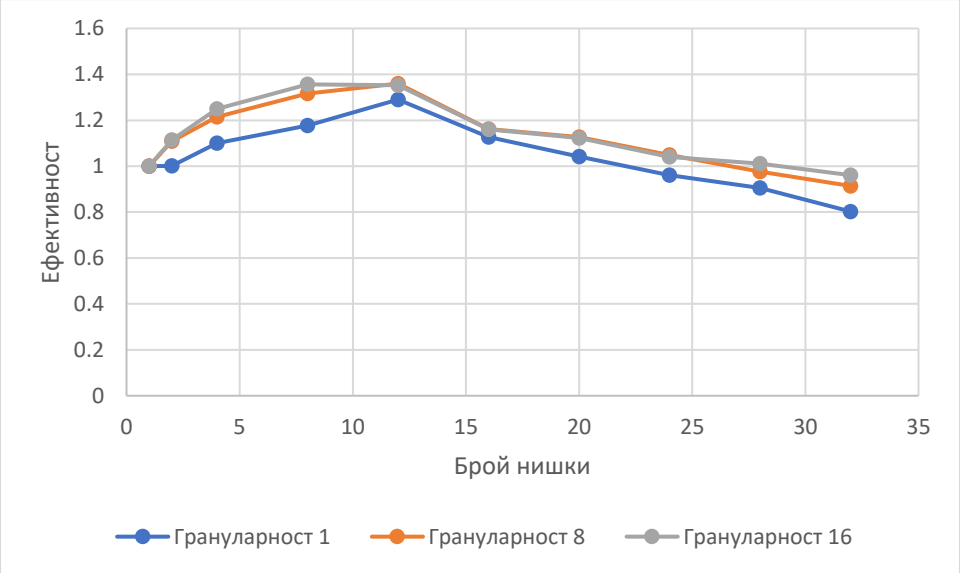
4.1.1. Тестване само на фаза 1 – изчисляване на частични резултати

#	p	g	$T_p^{(1)}$ / s	$T_p^{(2)}$ / s	$T_p^{(3)}$ / s	$\min\{T_p^{(i)}\}$ / s	$S_p = T_1/T_p$	$E_p = S_p/p$
1	1	1	71.975	71.243	72.466	71.243	1.000	1.000
2	2	1	35.595	35.918	35.837	35.595	2.001	1.001
3	4	1	16.473	16.194	16.399	16.194	4.399	1.100
4	8	1	7.597	7.564	7.677	7.564	9.419	1.177
5	12	1	4.601	4.617	4.638	4.601	15.484	1.290
6	16	1	3.949	4.422	4.366	3.949	18.041	1.128
7	20	1	3.522	3.420	3.475	3.420	20.831	1.042
8	24	1	3.185	3.089	3.132	3.089	23.063	0.961
9	28	1	2.876	2.809	2.824	2.809	25.362	0.906
10	32	1	2.774	2.780	2.789	2.774	25.682	0.803
11	1	8	52.687	52.498	53.603	52.498	1.000	1.000
12	2	8	24.201	23.663	23.790	23.663	2.219	1.109
13	4	8	10.803	10.896	10.921	10.803	4.860	1.215
14	8	8	5.158	5.082	4.985	4.985	10.531	1.316
15	12	8	3.217	3.399	3.428	3.217	16.319	1.360
16	16	8	2.907	2.826	2.878	2.826	18.577	1.161
17	20	8	2.438	2.465	2.330	2.330	22.531	1.127
18	24	8	2.086	2.162	2.126	2.086	25.167	1.049
19	28	8	1.945	1.969	1.920	1.920	27.343	0.977
20	32	8	1.795	1.807	1.814	1.795	29.247	0.914
21	1	16	47.257	49.710	49.198	47.257	1.000	1.000
22	2	16	21.822	21.554	21.211	21.211	2.228	1.114
23	4	16	9.646	9.450	9.644	9.450	5.001	1.250
24	8	16	4.355	4.387	4.441	4.355	10.851	1.356
25	12	16	2.911	3.120	2.996	2.911	16.234	1.353
26	16	16	2.545	2.561	2.610	2.545	18.569	1.161
27	20	16	2.104	2.183	2.127	2.104	22.461	1.123
28	24	16	1.893	1.896	1.894	1.893	24.964	1.040
29	28	16	1.716	1.680	1.670	1.670	28.298	1.011
30	32	16	1.573	1.582	1.537	1.537	30.746	0.961

Фигура 2. Таблица, в която са показани резултати за времената на изпълнение в секунди, ускорението и ефиктивността на програмата във фаза 1 при съответния брой нишки и съответната грануларност.



Фигура 3. Ускорение при различна грануларност.



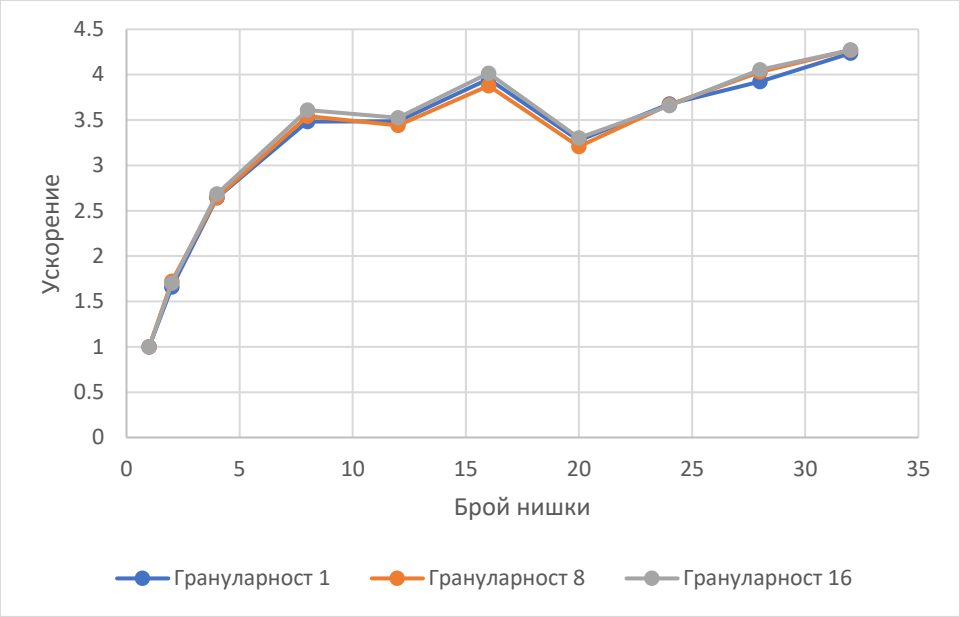
Фигура 4. Ефективност при различна грануларност.

В тази фаза от изпълнението на програмата забелязваме аномалията свръхлинейно ускорение. Това се дължи основно на факта, че броя подзадания зависи от броя на нишките, на които се изпълнява програмата. Колкото повече подзадания имаме, толкова повече сложността на работата на фаза 1 намалява – получаваме повече частични резултати, които да бъдат обединени във фаза 2. Няма голяма разлика спрямо различните грануларности, но все пак изглежда, че резултатите при грануларност 16 са най-добри, а при грануларност 1 – най-лоши.

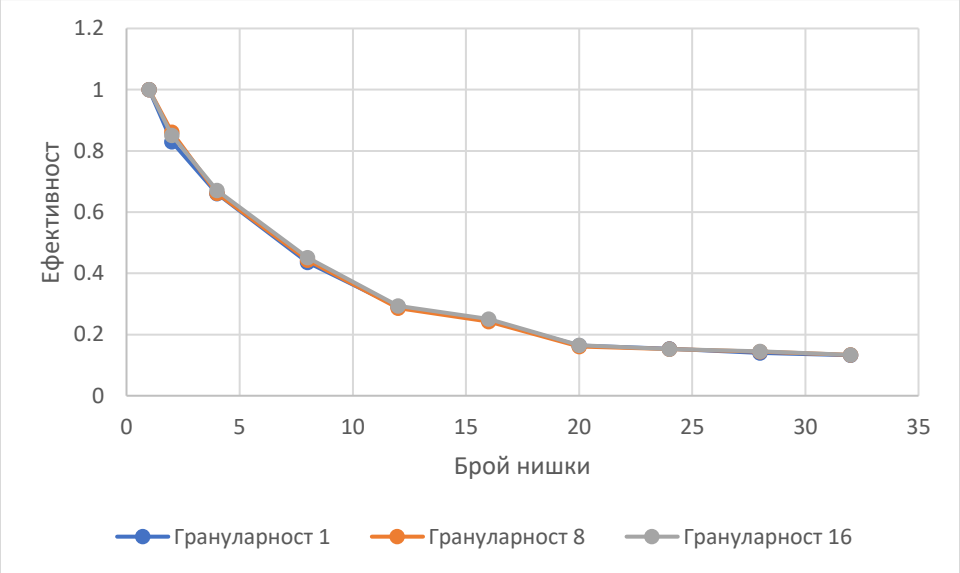
4.1.2. Тестване на фази 1 и 2 – изчисляване на частични резултати и обединяване на получените резултати в една обикновена дроб

#	p	g	$T_p^{(1)}$ / s	$T_p^{(2)}$ / s	$T_p^{(3)}$ / s	$\min\{T_p^{(i)}\}$ / s	$S_p = T_1/T_p$	$E_p = S_p/p$
1	1	1	71.975	71.243	72.466	71.243	1.000	1.000
2	2	1	43.042	43.638	42.915	42.915	1.660	0.830
3	4	1	27.632	26.960	27.412	26.960	2.643	0.661
4	8	1	20.448	20.741	21.784	20.448	3.484	0.436
5	12	1	21.301	20.444	21.054	20.444	3.485	0.290
6	16	1	18.032	18.265	18.643	18.032	3.951	0.247
7	20	1	22.286	21.746	23.203	21.746	3.276	0.164
8	24	1	19.622	19.383	20.733	19.383	3.676	0.153
9	28	1	18.158	18.601	18.391	18.158	3.924	0.140
10	32	1	16.818	17.049	17.606	16.818	4.236	0.132
11	1	8	72.745	72.302	73.343	72.302	1.000	1.000
12	2	8	42.615	42.198	41.985	41.985	1.722	0.861
13	4	8	27.489	27.321	27.492	27.321	2.646	0.662
14	8	8	20.481	20.584	20.404	20.404	3.544	0.443
15	12	8	21.025	21.319	21.537	21.025	3.439	0.287
16	16	8	18.639	18.978	18.761	18.639	3.879	0.242
17	20	8	22.655	23.199	22.545	22.545	3.207	0.160
18	24	8	20.126	19.701	20.024	19.701	3.670	0.153
19	28	8	18.320	17.948	18.561	17.948	4.028	0.144
20	32	8	16.928	17.490	17.332	16.928	4.271	0.133
21	1	16	73.193	75.561	76.016	73.193	1.000	1.000
22	2	16	43.125	43.195	43.005	43.005	1.702	0.851
23	4	16	27.268	27.438	27.591	27.268	2.684	0.671
24	8	16	20.603	20.272	20.949	20.272	3.611	0.451
25	12	16	21.179	21.835	20.757	20.757	3.526	0.294
26	16	16	18.641	18.233	18.962	18.233	4.014	0.251
27	20	16	23.183	23.145	22.166	22.166	3.302	0.165
28	24	16	20.334	19.996	20.602	19.996	3.660	0.153
29	28	16	19.012	18.060	18.507	18.060	4.053	0.145
30	32	16	17.134	17.559	17.559	17.134	4.272	0.133

Фигура 5. Таблица, в която са показани резултати за времената на изпълнение в секунди, ускорението и ефиктивността на програмата във фази 1 и 2 при съответния брой нишки и съответната грануларност.



Фигура 6. Ускорение при различна грануларност.



Фигура 7. Ефективност при различна грануларност.

Забелязваме, че след фази 1 и 2 ускорението на програмата е далеч от линейно. Това е, защото във фаза 2 има значително повече работа като сложност отколото във фаза 1 и голяма част от тази работа не може да бъде ефективно разпределена на p нишки. Например когато останат 4 члена да бъдат обединени, тази работа не може да бъде вършена от повече от 2 нишки. Когато останат последните 2 члена да бъдат обединени, тази работа не може да бъде вършена от повече от 1 нишка. И тази работа не е никак малка, в сравнение с останалата. Обединението на последните 2 члена включва умножение на две цели числа, като всяко от тях има почти 80 000 000 десетични цифри! С този проблем не можем да се справим, освен ако не напишем паралелно умножение на

две цели числа. Но както казахме по-горе за делението – това не е никак лесна задача и освен това ще е почти невъзможно да запазим скоростта на умножението от GMP библиотеката.

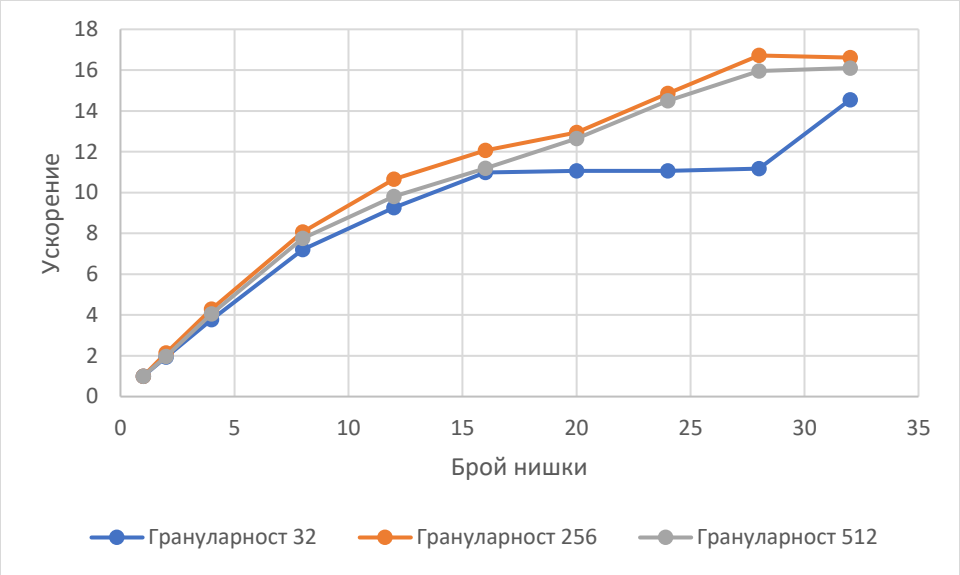
4.2. Тестване на версията с фиксиран брой подзадания (независещ от броя нишки)

В тази версия на програмата броят подзадания е $M = g$.

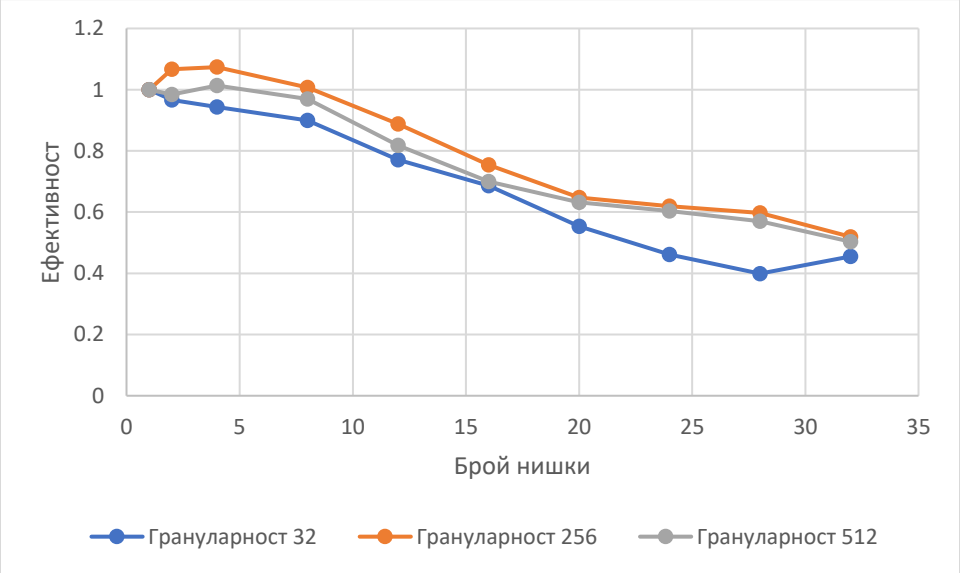
4.2.1. Тестване само на фаза 1 – изчисляване на частични резултати

#	p	g	$T_p^{(1)}$ / s	$T_p^{(2)}$ / s	$T_p^{(3)}$ / s	$\min\{T_p^{(i)}\}$ / s	$S_p = T_1/T_p$	$E_p = S_p/p$
1	1	32	40.648	43.967	43.895	40.648	1.000	1.000
2	2	32	21.022	21.569	21.395	21.022	1.934	0.967
3	4	32	10.973	10.775	11.045	10.775	3.772	0.943
4	8	32	5.742	5.886	5.648	5.648	7.197	0.900
5	12	32	4.392	4.440	4.480	4.392	9.255	0.771
6	16	32	3.703	3.856	3.821	3.703	10.977	0.686
7	20	32	3.921	3.673	3.944	3.673	11.067	0.553
8	24	32	3.958	3.674	3.746	3.674	11.064	0.461
9	28	32	4.285	3.638	3.638	3.638	11.173	0.399
10	32	32	2.794	2.814	2.835	2.794	14.548	0.455
11	1	256	35.413	30.578	34.666	30.578	1.000	1.000
12	2	256	14.433	14.466	14.331	14.331	2.134	1.067
13	4	256	7.133	7.120	7.304	7.120	4.295	1.074
14	8	256	3.794	3.820	3.859	3.794	8.060	1.007
15	12	256	2.889	2.869	2.924	2.869	10.658	0.888
16	16	256	2.696	2.535	2.612	2.535	12.062	0.754
17	20	256	2.361	2.411	2.379	2.361	12.951	0.648
18	24	256	2.110	2.068	2.057	2.057	14.865	0.619
19	28	256	1.829	1.935	1.851	1.829	16.718	0.597
20	32	256	2.189	1.841	1.840	1.840	16.618	0.519
21	1	512	27.669	25.510	28.137	25.510	1.000	1.000
22	2	512	13.788	12.961	12.995	12.961	1.968	0.984
23	4	512	6.318	6.292	6.551	6.292	4.054	1.014
24	8	512	3.553	3.288	3.412	3.288	7.759	0.970
25	12	512	2.608	2.711	2.600	2.600	9.812	0.818
26	16	512	2.323	2.287	2.280	2.280	11.189	0.699
27	20	512	2.048	2.051	2.018	2.018	12.641	0.632
28	24	512	1.771	1.760	1.786	1.760	14.494	0.604
29	28	512	1.598	1.635	1.623	1.598	15.964	0.570
30	32	512	1.609	1.590	1.584	1.584	16.105	0.503

Фигура 8. Таблица, в която са показани резултати за времената на изпълнение в секунди, ускорението и ефиктивността на програмата във фаза 1 при съответния брой нишки и съответната грануларност.



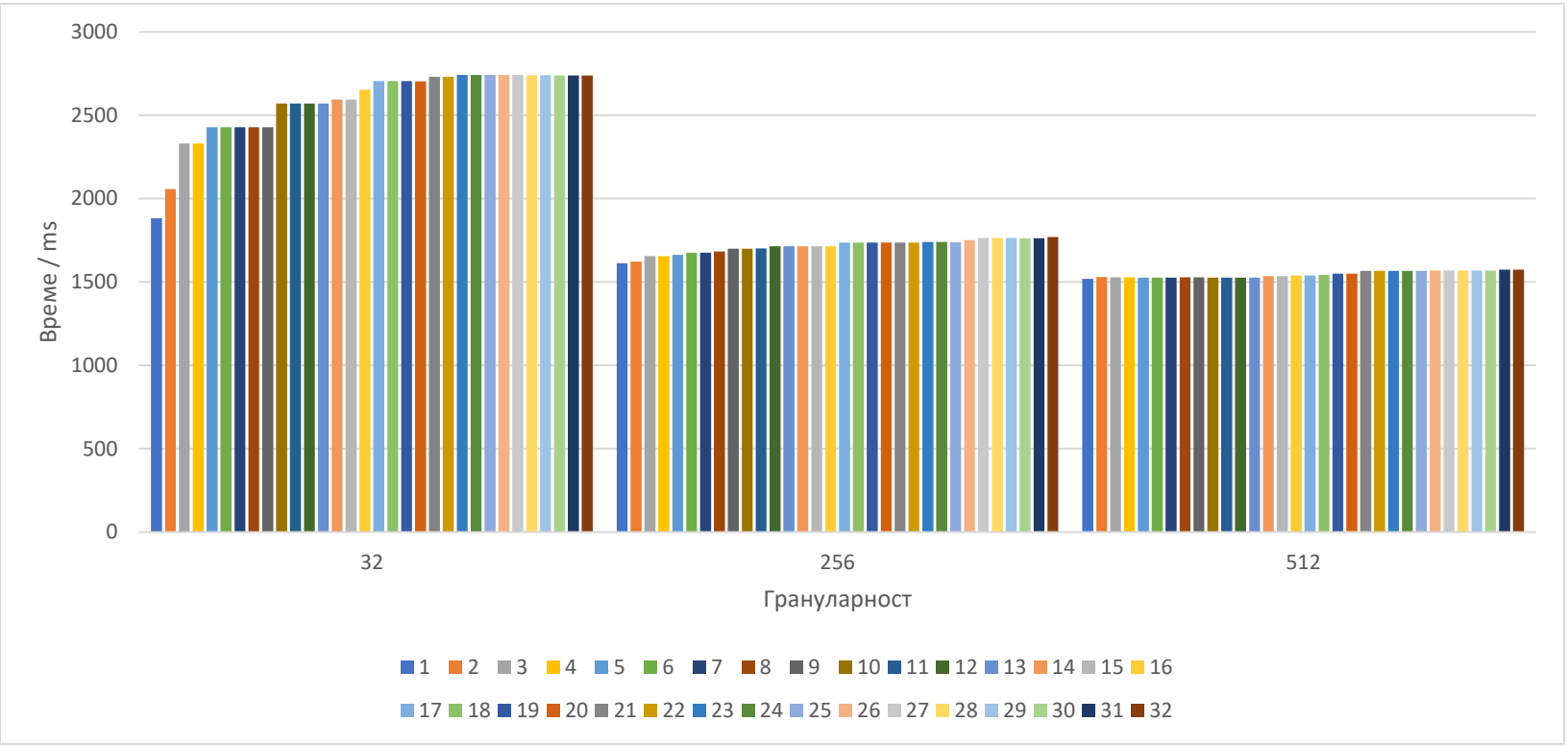
Фигура 9. Ускорение при различна грануларност.



Фигура 10. Ефективност при различна грануларност.

След като вече броят подзадания не зависи от броя на нишките, аномалията от точка 3.1.1. вече не се наблюдава. Все още има леко свръхлинейно ускорение при изпълнение на 2, 4 и 8 нишки и грануларност 256, което може да се дължи на ефективно използване на кеша. То обаче, може да се дължи и на неточности в измерванията на времето (забележете почти 5 s разлика при различни

изпълнения с едни и същи параметри). Забелязваме също, че при грануларност 32, ускорението е доста по-малко в сравнение с грануларности 256 и 512. Нека видим на какво се дължи това:



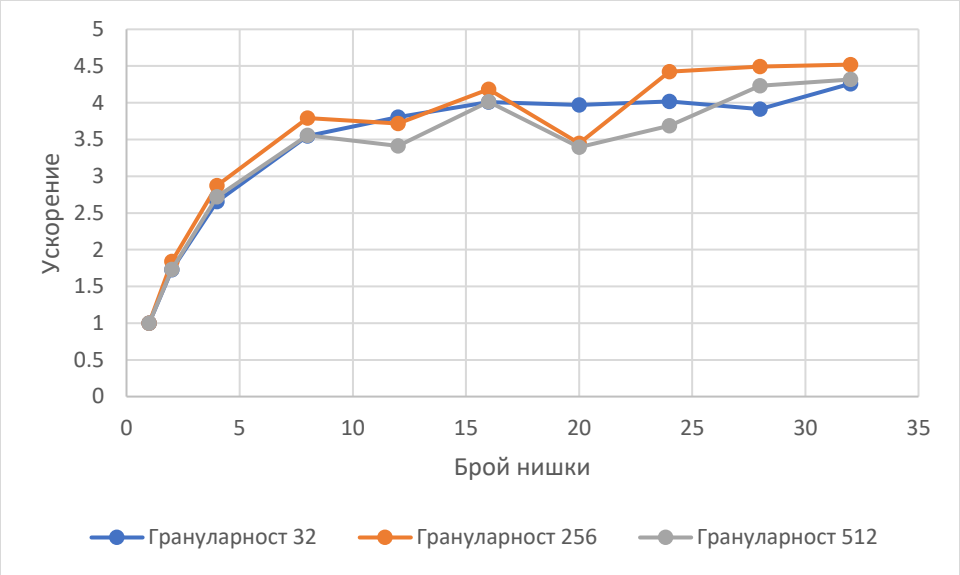
Фигура 11. Показва баланса между 32 нишки при грануларности 32, 256 и 512.

Очаквано, при 32 нишки и 32 подзадания, всяка нишка има да свърши по едно подзадание във фаза 1. Тъй като нишките с по-малък номер изчисляват по-малките членове на реда, те приключват изпълнението си по-рано. Този проблем се отстранява при грануларности 256 и 512.

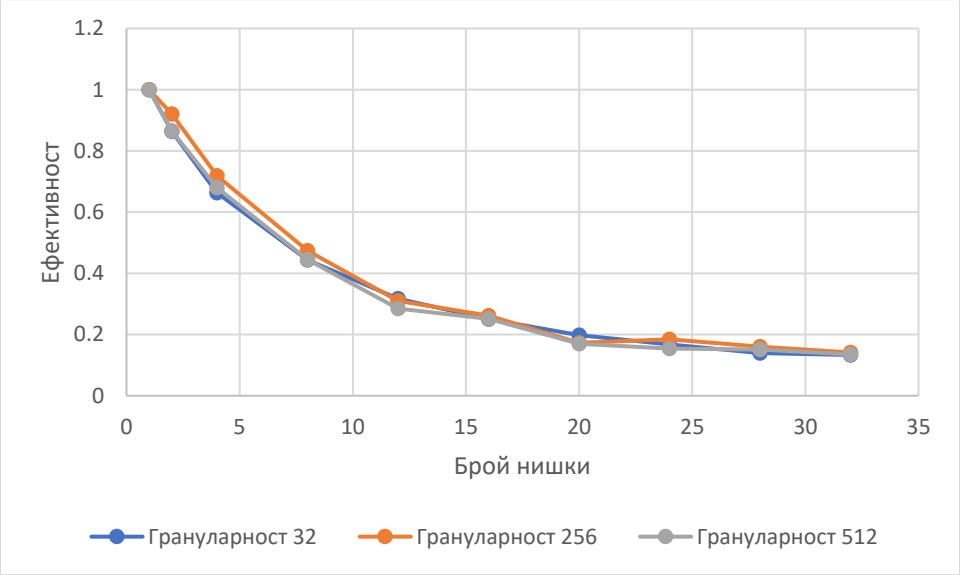
4.2.2. Тестване на фази 1 и 2 – изчисляване на частични резултати и обединяване на получените резултати в една обикновена дроб

#	p	g	$T_p^{(1)}$ / s	$T_p^{(2)}$ / s	$T_p^{(3)}$ / s	$\min\{T_p^{(i)}\}$ / s	$S_p = T_1/T_p$	$E_p = S_p/p$
1	1	32	72.027	77.087	77.298	72.027	1.000	1.000
2	2	32	41.670	43.014	42.537	41.670	1.729	0.864
3	4	32	27.372	27.142	27.476	27.142	2.654	0.663
4	8	32	20.295	20.393	20.312	20.295	3.549	0.444
5	12	32	18.923	19.199	19.119	18.923	3.806	0.317
6	16	32	17.967	18.042	18.719	17.967	4.009	0.251
7	20	32	18.289	18.151	18.298	18.151	3.968	0.198
8	24	32	18.624	17.922	18.463	17.922	4.019	0.167
9	28	32	18.411	18.427	18.415	18.411	3.912	0.140
10	32	32	17.257	16.908	17.544	16.908	4.260	0.133
11	1	256	83.091	77.412	82.321	77.412	1.000	1.000
12	2	256	42.047	42.391	42.446	42.047	1.841	0.921
13	4	256	27.799	26.946	28.028	26.946	2.873	0.718
14	8	256	20.421	20.678	20.770	20.421	3.791	0.474
15	12	256	21.369	20.826	21.484	20.826	3.717	0.310
16	16	256	18.930	18.638	18.512	18.512	4.182	0.261
17	20	256	23.088	23.077	22.453	22.453	3.448	0.172
18	24	256	17.505	17.834	17.713	17.505	4.422	0.184
19	28	256	17.588	17.225	17.540	17.225	4.494	0.161
20	32	256	20.255	17.583	17.126	17.126	4.520	0.141
21	1	512	78.529	73.959	78.237	73.959	1.000	1.000
22	2	512	44.002	42.902	42.757	42.757	1.730	0.865
23	4	512	27.180	27.713	28.387	27.180	2.721	0.680
24	8	512	20.919	20.799	20.829	20.799	3.556	0.444
25	12	512	21.696	22.099	21.673	21.673	3.412	0.284
26	16	512	18.432	19.181	18.435	18.432	4.013	0.251
27	20	512	21.792	21.833	22.845	21.792	3.394	0.170
28	24	512	20.072	20.048	20.621	20.048	3.689	0.154
29	28	512	17.558	17.714	17.479	17.479	4.231	0.151
30	32	512	17.532	17.132	17.320	17.132	4.317	0.135

Фигура 12. Таблица, в която са показани резултати за времената на изпълнение в секунди, ускорението и ефиктивността на програмата във фази 1 и 2 при съответния брой нишки и съответната грануларност.



Фигура 13. Ускорение при различна грануларност.



Фигура 14. Ефективност при различна грануларност.

Отново ускорението след фази 1 и 2 е далеч от линейно. Проблемът е същият както е описано в точка 3.1.2. и не можем да го отстраним.

4.3. Сравнение на времената за изпълнение на програмата върху процесори Intel® Core™ i7-8086K и 2x Intel® Xeon® Processor E5-2660.

*Забележка: при тестване на **ускорението** върху процесор Intel® Core™ i7-8086K няма за забележим нищо неочаквано и интересно, затова то не е представено в този документ.*

И на двете машини програмата е изпълнена по един и същи начин:

```
./a.out -t 32 -g 512 -o test.txt -p 23000000
```

Процесор	**Intel® Core™ i7-8086K**	**2x Intel® Xeon® Processor E5-2660**
Брой ядра	**6**	**16**
L1d cache	**32 KB**	**32 KB**
Базова тактова честота / Максимална Turbo честота	**4.00 GHz / 5.00 GHz**	**2.20 GHz / 3.00 GHz**
Фаза 1: изчисляване на частичните резултати	1.390 s	1.584 s
Фаза 2: обединяване на частичните резултати	10.870 s	15.736 s
Фаза 3: преобразуване на резултата във floating-point число (деление)	11.955 s	20.661 s
Фаза 4: записване на резултата във файл	42.344 s	82.427 s
Общо време за изпълнение (159 330 955 цифри след десетичната точка)	**66.559 s**	**120.408 s**

Забелязваме, че на **Intel® Core™ i7-8086K** непаралелно изпълняваните фази от програмата се изпълняват почти 2 пъти по-бързо. Причината за това най-вероятно е по-високата тактова честота на този процесор.

5. Източници

[1] y-cruncher - A Multi-Threaded Pi-Program, Alexander J. Yee
http://numberworld.org/y-cruncher/

[2] Notable Large Computations: 50 000 000 000 decimal digits of Pi, WaiKin Wong & Rickie Chang, 2009
http://numberworld.org/y-cruncher/runs/50b_12_21_2009.txt

[3] Y-CRUNCHER - PI-1B, benchmark on Intel® Core™ i9-10980XE
https://hwbot.org/submission/4657665_splave_y_cruncher___pi_1b_core_i9_10980xe_14sec_387ms

[4] Документация на GNU Multiple Precision Arithmetic Library, версия 6.2.1
https://gmplib.org/gmp-man-6.2.1.pdf

[5] FASTER INTEGER MULTIPLICATION, MARTIN FURER
https://web.archive.org/web/20130425232048/http://www.cse.psu.edu/~furer/Papers/mult.pdf

[6] Binary splitting method, Xavier Gourdon and Pascal Sebah, 2001
http://www.brotherstechnology.com/docs/binarysplit.pdf

[7] Mathematical Constants - Millions of Digits, Alexander J. Yee
http://www.numberworld.org/constants.html