Софийски университет „Св. Климент Охридски“

Факултет по математика и информатика
Специалност: “Компютърни науки ”
Курс: 3
Дисциплина: “Системи за паралелна обработка”

КУРСОВ ПРОЕКТ

Пресмятане на Pi - Ramanujan

Изготвил:

Десислава Димитрова Димитрова
(ФН: 81423)

Водещ лектор:

Проф. д-р Васил Цунижев

София
Летен семестър 2019/2020

Съдържание

1. Описание на поставената задача

Числото (стойността на) Pi може да бъде изчислено по различни начини. Използвайки сходящи редове, можем да сметнем стойността на Pi с произволно висока точност. Един от бързо сходящите към Pi редове е този, открит от индийския математик Srinivasa Ramanujan през 1910-1914 година. За стойността на Pi имаме:

$$\frac{1}{\pi} = \frac{\sqrt{8}}{99^2} \sum_{n=0}^{\infty} \frac{(4n)!}{(4^n\, n!)^4} \frac{1103 + 26390n}{99^{4n}}$$

1.1. Изисквания към реализацията

Параметрите за стартиране на програмата задават:

- Точност на изчисленията – това е число, което показва броят членове на реда. Параметърът се задава с -n. Стойността по подразбиране е 5000;
- Брой нишки – брой паралелни нишки, които обработват задачите. Параметърът се задава с -t. Стойността по подразбиране е броят на наличните ядра;
- Грануларност – брой членове на реда за една задача. Параметърът се задава със -g. Стойността по подразбиране се изчислява като се раздели броя на членовете на броя на нишките.
- Тих режим – параметър, отговарящ за текстово визуализиране на данни по отношение на работата на отделните нишки. Параметърът се задава с -q.

1.2. Цел и предназначение на проектираното приложение

Целта на проекта е да се анализират математическите особености на формулата и възможностите за програмна реализация чрез използване на паралелни процеси (многонишково програмиране).

Проектът трябва да реализира алгоритъма, следвайки принципите на паралелното програмиране и да разппредели изчислителния процес между множество от паралелно изпълняващи се нишки. Като резултат от направените тестове на приложението ще бъдат представени диаграми за ускорението и ефективността на алгоритъма.

Главната цел на проекта е да бъдат постигнати добри резултати по отношение на паралелизма при взимане предвид хардуерните възможности на тестовите машини.

2. Анализ - преглед на възможните решения и обосновка на избор

2.1. Обзор на възможни подходи за програмно изчисляване на π (функционален анализ)

2.1.1. Анализ на функционални източници

Основните функционални източници са [1] (Практическа имплементация на π алгоритми)), [2] (проекти по СПО / РСА от предишни години – документация - Янислав Василев), [3] (Pi калкулатор на Java с многонишкови технологии.

Анализ на функционалните източници и заимстваните от тях добри черти:

Източник	Добри черти	Недостатъци
[1]	Подробно описание на примерна оптимизация на формулата на Ramanujan – преизползване на предишни резултати чрез откриване на разликата между всеки два члена на реда	Липсва реализация
[2]	Добри идеи за оптимизация; добро ускорение и ефикасност на алгоритъма. Подробно описание на алгоритъма	Липса на таблични данни в документа. Липса на UML диаграми
[3]	Добра идея за разпределение на работата между нишките. Добре структуриран код	Липса на паралелизъм при обобщаване на резултатите. Липса на параметризация за грануларност.

2.1.2. Стандартен подход

При стандартен подход към решаване на задачата чрез паралелна обработка всички членове на редицата ще се поделят по равно между зададения брой нишки и всяка от тях ще приложи формулата на Рамануджан **[4].** Нишките ще работят паралелно и независимо една от друга. В този случай разпределянето на работата между паралелни процеси ще доведе до ускорение спрямо последователния процес (1 нишка), но времето за изпълнение ще си остане твърде дълго.

Проблемът е, че при този подход не се преизползват получените резултати от направените изчисления. Няма оптимизиране в разпределението на работата между нишките и не се постига добра скалируемост.

При този алгоритъм може да се постигне известно **подобрение**, ако се промени начинът на разпределение на работата между нишките и всяка нишка да взима не

последователни, а периодично разположени членове на реда, така че да не се наложи някои нишки да обработват само малки числа, а други – само големи.

2.1.3. Предварително пресмятане на факториели

Възможен подход за оптимизиране на изчисленията е предварителното пресмятане и мемоизиране на факториелите. При този подход е важно да се идентифицира кои факториели ще е достатъчно да осигурим. Ако пресметнем всички възможни факториели, ще загубим много памет. Тъй като всеки член на реда може да се получи от предходния инкрементално (без да се изчислява наново целият факториел), то за обработката на даден интервал от реда ще ни е необходим само стартовият за интервала факториел. Следователно трябва да осигурим толкова факториели, на колкото интервали (задачи) сме разбили реда (процеса). Изчисляването на факториелите от своя страна може да се оптимизира като се осъществи от паралелни нишки **[10], [11].**

При разработката на задачата не съм използвала подхода с предварително пресмятане на факториели, тъй като предпочетох друга идея за оптимизация и в моя случай не се налага пресмятане на факториели. При избрания от мен подход в етапа на паралелна обработка на задачите се получават началните за следващата задача коефициенти. И на следващия етап тези коефициенти се използват при обобщаването на резултатите. Логиката всъщност доста се доближава до подхода с факториелите. Този подход е описан подробно в следващата точка 2.1.4.

2.1.4. Преизползване на резултати от предходни изчисления

При изчисляването на членовете на реда на Ramanujan се налага да се пресметнат факториели на големи числа, зависещи от индекса на члена в редицата. Ако не се приложи някаква оптимизация, би трябвало всяка сума да се пресметне отделно и да се сумира с останалите. Така обаче ще се наложи много пъти да се повторят едни и същи сметки. Тъй като в множителите на членовете се открива рекурсивност, можем да опростим изчисленията като преизползваме предишни резултати.

Всеки член може да се представи като произведение на предишния член и някакъв инкрементален коефициент (постепенно увеличаващ се множител). Всяка нишка може да започне работа от произволен индекс в редицата като приеме за стартов множител единица и по-нататък работи по стандартната формула с инкрементален множител, показана по-надолу. Едва при комбинирането на резултатите на мястото на единицата ще се постави множителят, получен от предишната нишка, но при първоначалното обхождане на членовете на редицата не е необходимо изчакване между нишките.

Идеята за следващите опростявания и оптимизации беше взета от **[1]**.

Нека означим n-тия член с $a_n = F_n . S_n$, където

$$F_n = \frac{(4n)!}{(4^n . n!)^4 . 99^{4n}} \qquad S_n = (1103 + 26390n) \tag{1}$$

n-1-вият член е $a_{n-1} = F_{n-1} . S_{n-1}$

Нека изразим отношението на два последователни члена:

$$\frac{a_n}{a_{n-1}} = \frac{F_n . S_n}{F_{n-1} . S_{n-1}} \tag{2}$$

и оттук да изразим a_n чрез предходния член:

$$a_n = a_{n-1} . M_n . \frac{S_n}{S_{n-1}} \tag{3}$$

където

$$M_n = \frac{F_n}{F_{n-1}} = \frac{(4\mathrm{n} - 3).(4\mathrm{n} - 2).(4\mathrm{n} - 1)}{(4n)^3 . 99^4} \tag{4}$$

За a_0 $F_0 = 1$ и $\boldsymbol{a_0 = 1 . S_0}$

$$\boldsymbol{a_1} = a_0 . M_1 . \frac{S_1}{S_0} = S_0 . M_1 . \frac{S_1}{S_0} = \boldsymbol{M_1 . S_1}$$

$$\boldsymbol{a_2} = a_1 . M_2 . \frac{S_2}{S_1} = M_1 . S_1 . M_2 . \frac{S_2}{S_1} = \boldsymbol{M_1 . M_2 . S_2}$$

…

$$\boldsymbol{a_n} = a_{n-1} . M_n . \frac{S_n}{S_{n-1}} = M_1 \dots M_{n-1} . S_{n-1} . M_n . \frac{S_n}{S_{n-1}} = \boldsymbol{M_1 \dots M_n . S_n} \tag{5}$$

Разделяме броя на всички членове на редицата (numTerms) на броя на подадените нишки (numThreads) и получаваме колко члена трябва да сумира всяка нишка (numTermsByThread). Забелязваме, че има голяма повторяемост на множителите в членовете на редицата. Идеята за оптимизация е да се извадят пред скоби тези множители и нишката да сметне тази част от членовете, която не зависи от изваденитe пред скоби множители.

За яснота ще илюстрираме тази идея с простия случай от numTerms = 6 и numThreads = 2. Сумата от членовете в този случай изглежда така:

$$1 . S_0 + M_1 . S_1 + M_1 . M_2 . S_2 + M_1 . M_2 . M_3 . S_3 + M_1 . M_2 . M_3 . M_4 . S_4 + M_1 . M_2 . M_3 . M_4 . M_5 . S_5 =$$

$$= \underbrace{1 . S_0 + M_1 . S_1 + M_1 . M_2 . S_2}_{S_I} + \underbrace{M_1 . M_2 . M_3}_{M_{II}} . \underbrace{(S_3 + M_4 . S_4 + M_4 . M_5 . S_5)}_{S_{II}} = S_I + M_{II} . S_{II}$$

където S_I и M_{II} ще бъдат изчислени от първата нишка, а S_{II} – от втората.

В общия случай сумата ще изглежда така:

$S_I + M_{II}.S_{II} + \cdots + M_N.S_N$, където *N* е броят нишки.

Разделени по този начин, операциите не зависят една от друга и могат да бъдат обработени от отделни нишки. Първата нишка обхожда своите членове, изчислява ги и ги сумира като паралелно с това изчислява с натрупване и множителя M_{II}. Втората нишка работи асинхронно за изчислението на S_{II}, тъй като то не зависи от резултатите, получени от първата нишка, т.е. голяма част от работата по изчисление на членовете на редицата се извършва паралелно.

След като всички нишки приключат работа, резултатите трябва да бъдат комбинирани като е важно да се спазва редът, тъй като множителите "пред скоби" се изчисляват от предходната нишка. За да се гарантира тази подреденост, резултатите от работата на нишките се записват в списък от обекти TaskResult. За да не се налага изчакване между нишките, за резултатите използваме класа Future.

При голям брой резултати от задачи за обобщаване (т.е. при фина грануларност) е смислено процесът по обобщаването също да се извърши паралелно. Главният процес разделя броя на задачите на броя на нишките и всяка нишка получава комплект задачи за обобщаване, като полученият резултат е от същия тип. На последния етап главният процес обобщава крайния резултат.

С добавянето на всеки следващ член от редицата печелим допълнителни log10(96059601) = 7.98254… точни цифри след десетичната запетая. От справка с **[7], [8]**

2.2. Нефункционален анализ

Executor Service – екзекуторите се явяват като мениджъри на множество задачи с конкурента обработка. Те са удобни, защото регулират нивото на конкурентност на дадено приложение например спрямо това с каква мощ разполага машината, върху която се изпълнява приложението. В приложението е използван **FixedThreadPool** с фиксиран брой нишки - задават му се максимален брой нишки, с които да работи и той оптимизира тяхното управление като решава кога да създава нова нишка и кога да използва свободна **[12]**.

Използвана е външна библиотека **Apfloat**, предназначена за прецизни аритметични действия с висока производителност. С нейна помощ се извършват изчисления точност до милиони цифри **[9]**. С тази библиотека е разработено приложение за Pi калкулатор **[10]**.

Future – това е интерфейс, който позволява асинхронно получаване и записване на резултати. Използваме го при извикването на задачите, които трябва да върнат резултат, който да бъде достъпен за главната програма, но без да се налага изчакване края на работата на задачата **[11], [12]**.

3. Проектиране

Приложението е разработено на Java 11.

Моделът на паралелизма е SPMD (Single Program, Multiple Data), като работата на нишките е асинхронна. Използваме **програмен паралелизъм по данни**. Работата между нишките се поделя като всяка от тях извършва еднотипна обработка върху съответната ѝ част от данните.

Модел на софтуерната архитектура е Master-Slaves – имаме една основна нишка (main), която получава входните параметри, стартира зададения от потребителя брой нишки и след приключване на работата им обединява получените от тях стойности и дава крайния резултат. Master програмата управлява бързодействието като използва като използва Executor Service за разпределяне на заявките.

3.1. Модел на декомпозиция на данните

Реализация 1 - едра грануларност / статично балансиране

При този подход използваме едра грануларност: разделяме броя на членовете на броя на нишките и получаваме колко члена ще обработи всяка нишка, т.е. броят на задачите е равен на броя на нишките. Предимството на този подход е, че броят на задачите се запазва сравнително малък и работата за обобщаването на резултатите не е голяма. Недостатък е, че е възможно някои от нишките да приключат работата си по-бързо от други нишки и следователно да не се натоварват пълноценно. Затова използваме декомпозиция на данни с по-фина грануларност:

Реализация 2 – управляема грануларност / динамично балансиране

При този подход въвеждаме параметър, с който управляваме броя на членовете на редицата, които се подават в една задача.

Забележка: Трябва да се съобрази какви стойности за грануларност са смислени в съответствие с броя на нишките – например, ако броят на членовете е 5000, а броят на нишките е 10, няма смисъл да се подава грануларност по-голяма от 500, защото в този случай броят на задачите ще стане по-малък от броя на нишките и някои от нишките ще останат без работа. Следователно най-едрата грануларност е различна при различния брой нишки. Например за случая с 5000 члена - за 2 нишки най-едрата грануларност е 2500, за 3 – 1666, за 4 – 1250 и т.н. Именно заради това във всяка от таблиците с резултати от тестовете в точка 3 е добавена колона с грануларност.

С увеличаването на броя на задачите е възможно по-добре да се балансира работата на нишките. Executor Service динамично разпределя поредената задача към нишката, която е свободна. При този подход е възможно броят на задачите да нарасне твърде много и съответно да нарасне и работата за обобщаването на резултатите.

4. Тестване и апробация

Всички тестове и измервания, въз основа на които са представени графиките и таблиците, са направени на сървъра t5600.rmi.yaht.net на ФМИ. Характеристиките на машината са изведени чрез изпълнението на командата –lscpu:

Architecture: x86_64 CPU op-mode(s): 32-bit, 64-bit CPU(s): 32 Thread(s) per core: 2 Core(s) per socket: 8 Socket(s): 2	Model name: Intel(R) Xeon(R) CPU E5-2660 0 @ 2.20GHz CPU MHz: 2199.865 L1d cache: 32K L1i cache: 32K L2 cache: 256K L3 cache: 20480K

Сървърът разполага с 32 ядра (2 sockets * 8 cores * 2 threads), но на практика **апаратният паралелизъм** е 16 (2 sockets * 8 cores).

Провеждаме тестове по пресмятания на 5000 члена на реда (-n 5000) и изследваме ускорението **Sp** и ефективността **Ep** като меним броя на нишките (-t <1,2,…>).

Използваме следните означения:

- **p** – брой нишки
- **T1** – времето за изпълнение на серийната програма (при използване на една нишка)
- **Tp** – времето за изпълнение на паралелната програма, използваща p нишки
- **Sp = T1 / Tp** – ускорението на програмата при използване на p нишки
- **Ep = Sp / p** – ефективността на програмата при изпозлване на p нишки
- **g** – грануларност, измерваме я с броя членове на редицата, които се обработват от всяка задача.

Измерваме времената на работа на програмата като провеждаме по три теста за всеки брой нишки и за различните видове грануларност, и измежду тези стойности взимаме най-добрата. Резултатите от тестовете са обобщени и визуализирани в следващите таблици.

Първоначално започнах тестовете с еднакви стойности за грануларност, но в процеса на тестването установих, че една и съща стойност на грануларността не е с аналогично качество за различния брой нишки. При различен брой нишки горната граница за грануларност е много различна (при 5000 члена за 2 нишки – тя е 2500, а за 20 - 250), а долната не се променя – т.е. интервалите силно варират.

Построени са графики при еднаква стойност на грануларността, а също и за аналогични по качество грануларности, за да се демонстрира поведението на програмата.

Графиките при еднаква стойност на грануларността не са равномерни, защото, за да се получи равномерна графика, трябва да се изследват аналогични по качество грануларности за различен брой нишки.

Най-лесният пример е за едра грануларност. За 2 нишки това е 2500, а за 10 нишки - 500. Друг пример: нека разгледаме грануларност 500. За 2 нишки това ще означава 10 задачи, които ще се разпределят по 5 за двете нишки и резултатът ще е приблизително същият като при грануларност 2500, но за 8 нишки тези 10 задачи ще се разпределят твърде неравномерно и резултатът ще е по-лош, отколкото едрата грануларност за 8 нишка, т.е. грануларност 500 не е аналогична по качество за 2 и 10 нишки.

4.1. Тестове с най-едра грануларност

В този случай броят на задачите е равен на броя на нишките. С увеличаване на броя на нишките се намалява и времето за изпълнение на програмата. В колона ***g*** стойностите са различни в зависимост от броя нишки, защото е съобразено броят на задачите да не е по-малък от броя на нишките.

p	g	$T_p^{(1)}$ (ms)	$T_p^{(2)}$ (ms)	$T_p^{(3)}$ (ms)	$T_p = \min$ (ms)	$S_p = T_1/T_p$	$E_p = S_p/p$
1	5000	30366	30809	30825	**30366**	**0.9285**	**0.9285**
2	2500	14824	14610	14832	**14610**	**1.9299**	**0.9650**
4	1250	7431	7386	7336	**7336**	**3.8435**	**0.9609**
6	833	5180	5126	5113	**5113**	**5.5146**	**0.9191**
8	625	4199	4170	4284	**4170**	**6.7616**	**0.8452**
10	500	3980	3683	3507	**3507**	**8.0399**	**0.8040**
12	416	3369	3419	3400	**3369**	**8.3692**	**0.6974**
14	357	3338	3202	3361	**3202**	**8.8057**	**0.6290**
16	312	3169	3284	3110	**3110**	**9.0662**	**0.5666**
18	277	2964	3095	3017	**2964**	**9.5128**	**0.5285**
20	250	3071	3016	3098	**3016**	**9.3488**	**0.4674**
22	227	3059	2930	3102	**2930**	**9.6232**	**0.4374**

Таблица 1 – едра грануларност

4.2. Тестове с недостатъчно фина грануларност

В този случай броят на задачите е по-голям от броя на нишките, но не е достатъчно голям, за да се балансират добре задачите между нишките. При тази грануларност резултатите очаквано не са добри, тъй като задачите са с голям размер и когато броят на задачите е неравномерно разпределен между нишките, се получава голяма разлика в натовареността им. Ако обемът на задачите е по-малък, тогава те ще могат по-добре да се разпределят между нишките.

Например при 6 нишки сме посочили грануларност 250 – в този случай се получават 5000 : 250 = 20 задачи. 20 : 6 = 3 с остатък 2 - това означава, че 4 от нишките

ще изпълнят по 3 задачи, а 2 от тях – по 4, тъй като задачите са твърде обемисти, двете нишки ще се забавят повече от останалите.

В таблица 2 са посочени примерни стойности на грануларността, при които се получава лошо балансиране на задачите. Показали сме примери за грануларност по-висока от оптималната (оптималната грануларност е различна спрямо броя на нишките).

p	g	$T_p^{(1)}$ (ms)	$T_p^{(2)}$ (ms)	$T_p^{(3)}$ (ms)	T_p = min (ms)	$S_p = T_1/T_p$	$E_p = S_p/p$
1	5000	30366	30809	30825	**30366**	**0.9285**	**0.9285**
2	1000	16293	16294	16611	**16293**	**1.7306**	**0.8653**
4	500	8791	8750	8836	**8750**	**3.2224**	**0.8056**
6	250	6090	6145	6083	**6083**	**4.6352**	**0.7725**
8	250	4795	4876	4795	**4795**	**5.8803**	**0.7350**
10	200	4426	4280	3975	**3975**	**7.0933**	**0.7093**
12	120	4170	4242	4191	**4170**	**6.7616**	**0.5635**
14	180	4179	4296	4339	**4179**	**6.7471**	**0.4819**
16	100	3992	3780	3891	**3780**	**7.4593**	**0.4662**
18	100	3729	3563	3684	**3563**	**7.9136**	**0.4396**
20	100	4127	3306	4222	**3306**	**8.5287**	**0.4264**
22	100	3836	3584	3853	**3584**	**7.8672**	**0.3576**

Таблица 2 – недостатъчно фина грануларност

4.3. Тестове с оптимална грануларност

В този случай сме показали резултати за оптималната грануларност, при която броят на задачите се разпределя най-добре между нишките. Тя е различна при различен брой нишки.

С увеличаване на броя нишки оптималната грануларност се увеличава. Когато броят нишки е голям, те бързо свършват първоначалната обработка на членовете на редицата и оставащата работа от процеса спрямо извършената става относително по-голяма. Затова е по-добре да се намали втората част от работата, а това се постига като се намали броят на задачите.

p	g	$T_p^{(1)}$ (ms)	$T_p^{(2)}$ (ms)	$T_p^{(3)}$ (ms)	T_p = min (ms)	$S_p = T_1/T_p$	$E_p = S_p/p$
1	5000	30366	30809	30825	**30366**	**0.9285**	**0.9285**
2	5	11325	11334	11450	**11325**	**2.4897**	**1.2449**
4	5	5896	5870	5942	**5870**	**4.8034**	**1.2009**
6	10	4738	4590	4746	**4590**	**6.1429**	**1.0238**
8	10	4051	3824	3794	**3794**	**7.4317**	**0.9290**
10	25	3343	3390	3344	**3343**	**8.4343**	**0.8434**
12	30	3158	3480	3469	**3158**	**8.9284**	**0.7440**
14	45	3414	3245	3175	**3175**	**8.8806**	**0.6343**
16	100	3291	3065	3188	**3065**	**9.1993**	**0.5750**
18	277	3017	2964	3095	**2964**	**9.5128**	**0.5285**

20	250	3016	3098	3071	**3016**	**9.3488**	**0.4674**
22	227	3059	2930	3102	**2930**	**9.6232**	**0.4374**

Таблица 3 – оптимална грануларност

4.4. Тестове с твърде фина грануларност

В този случай броят задачи е достатъчно голям, за да се балансират добре задачите между нишките. Недостатъкът е, че се появява твърде много работа за обобщаващия процес. При повече нишки тази граница настъпва при по-висока грануларност.

Резултатите от тестовете илюстрират, че с намаляване на грануларността под дадена граница, нашият алгоритъм дава по-лоши резултати.

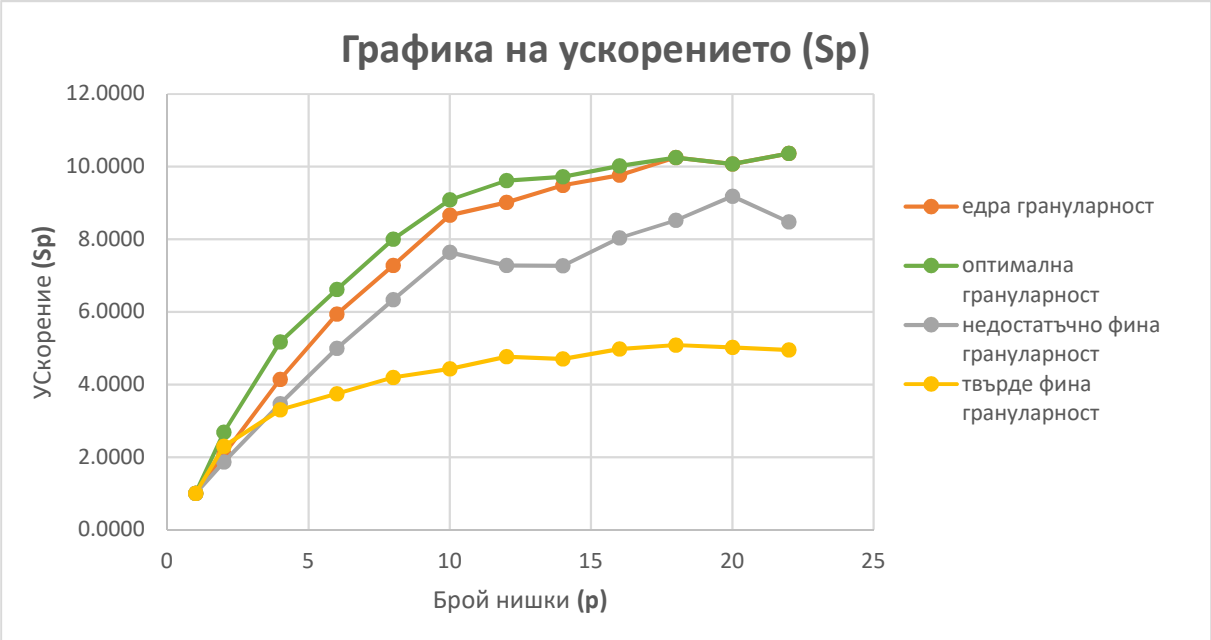
Забелязва се аномалията, че при твърде ниска грануларност започва да намалява бързодействието, затова е намерен начин да се компенсира увеличаването на броя на задачите, които трябва да сумира последователният процес и е пусната тази работа да бъде свършена също от паралелни нишки.

p	g	$T_p^{(1)}$ (ms)	$T_p^{(2)}$ (ms)	$T_p^{(3)}$ (ms)	T_p = min (ms)	$S_p = T_1/T_p$	$E_p = S_p/p$
1	5000	30366	30809	30825	**30366**	**0.9285**	**0.9285**
2	2	13231	13249	16277	**13231**	**2.1311**	**1.0655**
4	3	9213	9279	9193	**9193**	**3.0671**	**0.7668**
6	4	8103	8224	8302	**8103**	**3.4797**	**0.5799**
8	5	7311	7298	7243	**7243**	**3.8929**	**0.4866**
10	5	6878	6858	6993	**6858**	**4.1114**	**0.4111**
12	5	6370	6528	6777	**6370**	**4.4264**	**0.3689**
14	10	6784	6578	6453	**6453**	**4.3694**	**0.3121**
16	10	6231	6133	6100	**6100**	**4.6223**	**0.2889**
18	15	5998	6003	5971	**5971**	**4.7222**	**0.2623**
20	15	6211	6043	6123	**6043**	**4.6659**	**0.2333**
22	20	6279	6132	6243	**6132**	**4.5982**	**0.2090**

Таблица 4 – твърде фина грануларност

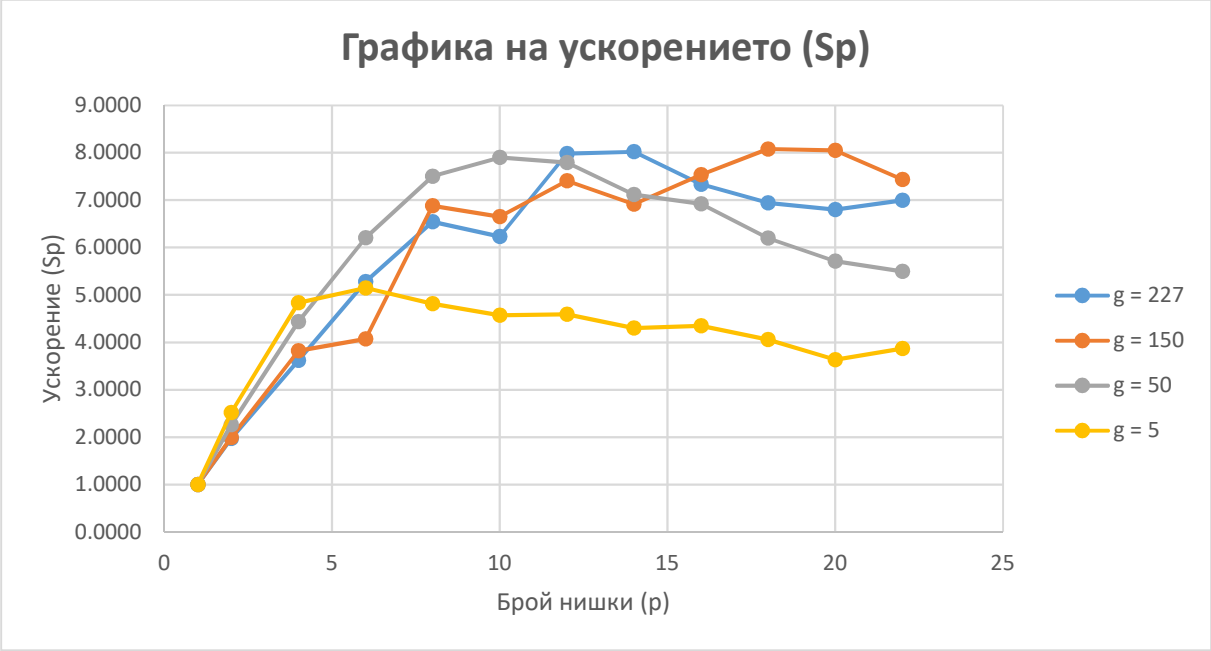
4.5. Графики

Графика на ускорението при грануларностите, еднакви по качество - подходящата грануларност е избрана тестово според броя нишки.



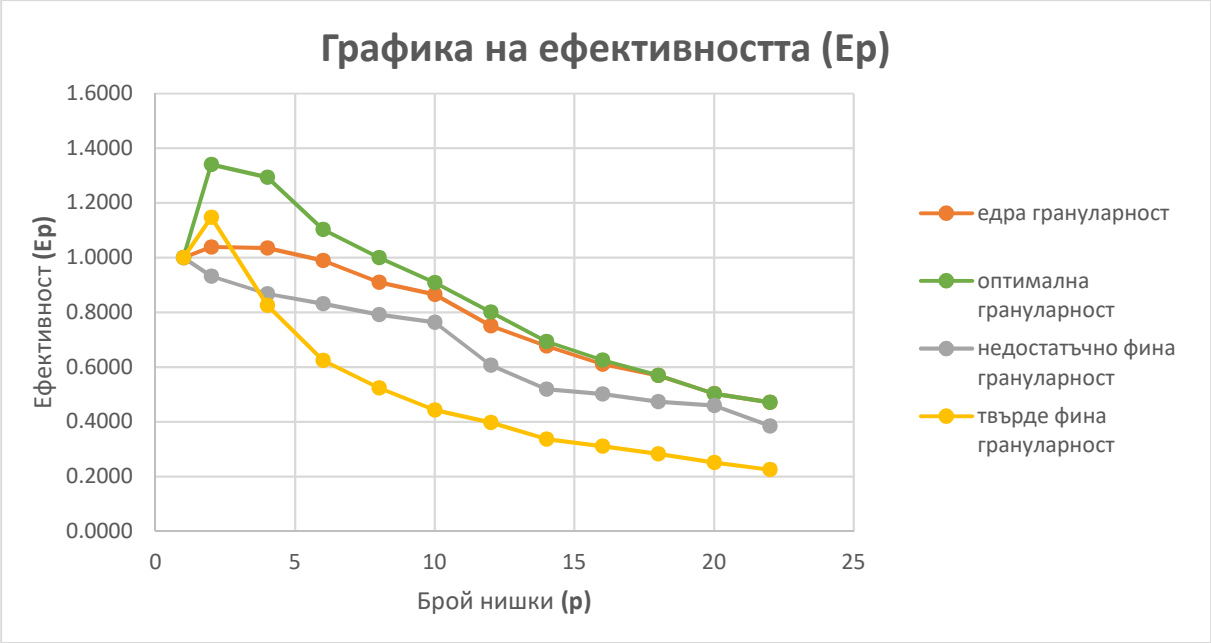
Графика 1

Графика на ускорението при грануларностите, еднакви по стойности.



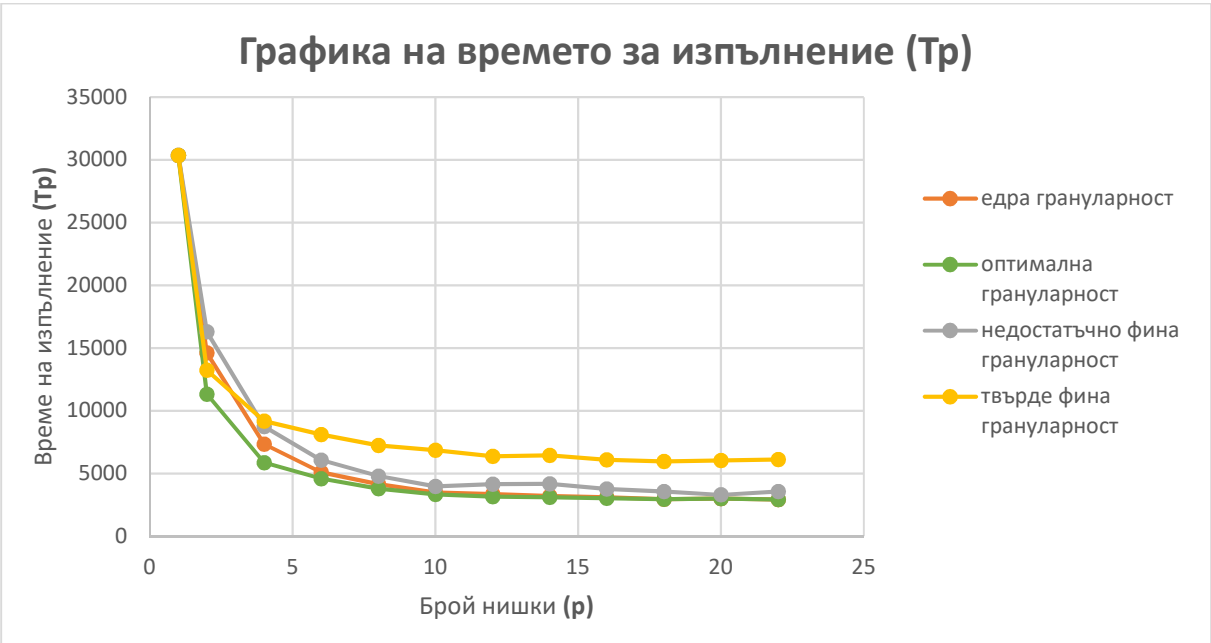
Графика 2

Графика на ефективността при грануларностите, еднакви по качество - подходящата грануларност е избрана тестово според броя нишки.



Графика 3

Графика на ефективността при грануларностите, еднакви по качество - подходящата грануларност е избрана тестово според броя нишки.



Графика 4

5. Източници

[1] Practical implementation of π Algorithms
By Henrik Vestermark – 4 ноември, 2016
(http://www.hvks.com/Numerical/Downloads/HVE%20Practical%20implementation%20of%20PI%20Algorithms.pdf)

[2] Пресмятане на Pi – Ramanujan
Янислав Василев – юни, 2019
(https://learn.fmi.uni-sofia.bg/pluginfile.php/260286/mod_resource/content/2/SPOprojects.pdf)

[3] Pi калкулатор на Java с многонишкови технологии
Светослав Кръстев
(https://github.com/skrustev/java-pi-calculation)

[4] Pi Ramanujan Java Formula
(https://stackoverflow.com/questions/21692610/how-would-i-enter-the-ramanujan-series-formula-into-java/21692929#21692929)

[5] Parallelizing Factorial Calculation
(https://stackoverflow.com/questions/34978236/parallelizing-factorial-calculation)

[6] Factorial Function Memoization
(https://stackoverflow.com/questions/55538806/javascript-factorial-function-memoization/60027632#60027632)

[7] Approximations of π
(https://en.wikipedia.org/wiki/Approximations_of_%CF%80#20th_and_21st_centuries)

[8] A catalogue of mathematical formulas involving π, with analysis
David H. Bailey - April 22, 2020
(https://www.davidhbailey.com/dhbpapers/pi-formulas.pdf)

[9] Apfloat
(http://www.apfloat.org/apfloat_java/docs/org/apfloat/samples/package-summary.html)

[10] Pi calculator with Apfloat
(http://www.apfloat.org/apfloat_java/applet/pi.html)

[11] Future Interface
(https://www.baeldung.com/java-future)

[12] Executor Service
(https://www.baeldung.com/java-executor-service-tutorial)