

**Проект**

**Разпределени Софтуерни**

**Архитектури**

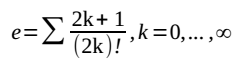
изготвил:

Андрей Ганчев Андреев

Ф.Н: 62173

**Тема на проекта**

Темата на проекта е паралелното смятане на числото „e“, използвайки следната формула:



**Алгоритъм за решение**

Задачата има за цел да пресметне числото «е» паралелно, за целта съществуват три основни проблема:

1. Прецизност (p):

*Избрах да работя с прецизност, а не с брой събираеми с цел по-сигурното тестване на резултата.*

За достигане на нужната прецизност(p), трябва да е сигурно, че сме пресметнали стойността за тези «k», които 1/(2k)! има поне «p+c» нули след десетичната запетая, където «c» е малка константа с цел точност при по-малки стойности на «p». За целта числото трябва да има точност «p+c», след което числото бива отрязано до точност «p». Тъй като «log10(n!) = Theta(nlog10n)»,то лесно се доказва, че «n!» има дължина поне n за не прекалено малки стойности на n. С това под предвид «2n !» има също над «n» цифри, за големи «n», което подсказва, че взимането на първите p елемента е достатъчно за алгоритъма. В реалността за по-големи стойностти на «p» не са ни нужни повече от «p/2» на брой елемента, но поради това, че целта на проекта е паралелната обработка съм опростил нещата, като условно съм взел първите «p+c» итерации за нужни.

2. Намиране на сумата.

2.1 Разбиване на множеството от събираеми

Това е основна стъпка в алгоритъма. Множеството <0...p> трябва да бъде разбито по подходящ начин, за че да могат всички нишки «t» да имат балансирана работа. За сега ще се абстрахирам от това как точно се разбива множеството, тъй като това зависи скоростта за обработка на един такъв подсет. Единственото, което мога да кажа до този момент е, че се разбива на подинтервали <0...x1> <x1+1...x2> …. <xv+1…p>.

2.2 Намиране на подсума.

Нека имаме интервал <a,b>(a<b). То трябва да можем да извършим максимално много операции паралелно за този интервал. Всяко едно действие, което не се извърши паралелно ще трябва да бъде извършено синхронизирано, което ще намали ефективността на паралелизма. Вместо за всеки интервал на събираеми да се смята нужният факториел от «0!» до «(2а)!»(което е най-бавната информация заедно с деленето), то за всеки интервал се смята сумата: «(2а+1)/(2а) + … + (2b+1)/[(2a)\*(2a+1)\*...\*(2b)]», като запазваме стойността «(2a)\*(2a+1)\*...\*(2b)» с цел преизползване. След като сме пресметнали тези стойности за всички интервали, можем да се възползваме от това, че

за един интервал преди <a,b> вече е пресметнат факториелът, или ако не е го пресмятаме индуктивно и така накрая след като умножим сумата от последният факториел «(2(а-1))!», можем да пресметнем цялата сума «(2а+1)/(2а)! + … + (2b+1)/(2b)!». Така върху една нишка няма да бъдат направени всики «2p» умножения, а само «v». Тъй като n е изключително голямо, а за една операция за умножение с бърз за алгоритъм на умножение не може да се избегне «O(nlogn)»(Без взимане под предвид апроксимиране на «n!», което поне на теория разбрах, че става за O(n)).

2.3 Намиране на сума

След като сме намерили всички суми, то после с «v» операции бихме могли да получим крайната сума, което е и числото «е».

3. Оптимално разбиване на интервала <0…p>

Има доста начини да се разбие интервал <0...p>, с цел оптимизацията написана в точка 2, интервалите трябва да са последователни.

<0...p> = <0...x1> + … + <xv+1...p>. Целта ми е да минимизирам чакането на нишките. За това трябва да подбера оптимално «x1...xv», така че да няма интервал чието време да е прекомерно по-голямо от друго(спрямо точка 2.2). За целта ако имаме функция «t(<a,b>)», която изчислява времето за намиране на интервал <a,b>, то търсим този вектор «x», за който всеки два интервала «U,V»: «max|t(U)-t(V)|» е минимална. Също така «v» не трябва да е прекалено голямо, защото скоростта на процесите, които изискват една нишка намалява. Най-лесният начин е да се избере такова «v», че «v=p/v0» и интервалите да са равни по големина. Колкото повече интервали има, толкова по-малка е разликата«max|t(U)-t(V)|», но и толкова по-бавна става работата за една нишка. За целта емпирично може да се намери оптимално v0, като «v0» не трябва да зависи от «t» . Ако «v» зависи от «t», то тогава биха се получили аномалии в ефективността спрямо «t». Тъй като това е оптимизация, то би се получило, че за някои «t» ефективността за една нишка ще е по голяма от 1ца. Методът който използвам аз е да оптимизирам «v0» спрямо работата с «tmax» нишки(тъй като е и най-бързо) и после да използвам резултата от наблюденията за останалите нишки.

**Използвани технологии**

За проекта съм използвал Java 8 и Мaven за dependency manager. Библиотеката за изчисления е ApFloat.

**Абстракции**

Разделям използваните абстракции на три вида:

1. Абстракции нужни за паралелност на изпълнението:

За паралелност използвам parallel streams с ForkJoinExecutorService с фиксирана големина на thread pool-а и custom ThreadFactory с цел по-добро дебъгване на нишките. Използвайки по-високо ниво на абстракция за паралелизма ми позволява да не ми се налага да менежирам повече самите нишки, тъй като всичко се случва отзад.

2. Абстракции нужни за решението на заданието:

* Context — Това е главната концепция свързана със съхранението и четенето на параметрите нужни на програмата
* Range — Това е основната абстракция, която позволява лесното разбиване на множестовот <0...n> на подмножества. С една функция мога да генерирам v последователни подмножества на <0...n>.
* Calculator — там седи решението.

3. Абстракции нужни за дебъгване:

* TestUtils — нужен за тестване на резултата спрямо посочен файл
* MonitorUtils — малък туул, който написах за мониторинг на скоростта и логинг на информация.

**Имплементаця**

Имплементацията на алгоритъма се състой от 4 фази:

1. Пресмятане паралелно на частичните суми(паралелно)

2. Допресмятане на факториелите (една нишка)

3. Допреясмятане на сумите (паралелно)

4. Събиране на сумите (една нишка)

*Имайки описанието на абстракциите и алгоритъма, не мисля да навлизам в имплементационни детайли. Тези точки биха били по-полезни накрая, като анализирам как се променя скоростта на алгоритъма при промяна на «v».*

**Използване**

Може да се борави със следните параметри на програмата:

-t <num> = брой на нишките

-p <num> = точност на калкулацията

-o <filename> = име на файл за изпечатване на e

-f <filename> = име на файл за тестване на вярност на решението

-r = още по-детайлна информация за създаването на нишки и коя нишка кой range обработва

-q = mute

-v <num> = броят на таскове, които да бъдат създадени. 1024 е сравнително оптимално за 1-32 нишки.

**Анализ на ефективността**

*Всички тестове са направени с v=1024, тъй като намерих тази стойност за най-оптимална при нарастване на t до 32.*

*Не успях да извлека адекватна информация за «p<10000», тъй като алгоритъмът работеше прекалено бързо за високи стойности на t и имаше прекалено голям шум в резултатите.*

*Резултати:*

*При p = 30,000*

**

*При p = 10,000*

**

*При p = 50,000*

