Софийски университет „Св. Климент Охридски”

Факултет по математика и информатика

Пресмятане на Неперовото число *e*

Изготвил: Недко Недев, КН III курс ФН: 81650

Проверил: …….

(ас. Христо Христов)

Съдържание

Условие на задачата

Едно важно за математиката число е Неперовото число (Ойлеровото число), тоест числото **e**. Използвайки сходящи редове, можем да сметнем стойността на **e** с произволно висока точност. Един от сравнително бързо сходящите към **e** редове е:

$$\mathbf{e} = \sum_{k=0}^{\infty} \frac{(3k)^2 + 1}{(3k)!}$$

Вашата задача е да напишете програма за изчисление на числото **e**, използвайки цитирания ред, която използва паралелни процеси (нишки) и осигурява пресмятането на **e** със зададена от потребителя точност.

Функционалности

Програмите, които ще представим, реализират следните функционалности, зададени с параметри :

- *-p <precision>* : пресмята числото **e** с точност *<precision>* - това са броя на цифрите след десетичната запетая.

- *-t <num_threads>* : числото **e** се пресмята върху *<num_threads>* на брой нишки.

- *-o <output_file> :* стойността на числото **e**, заедно с допълнителна информация за процеса на работа на нишките, се записва във файл с име *<output_file>*. Ако такъв файл не се подаде, то информацията се записва във файл с име *output.txt.*

- *-q* : стартиране на програмата в *"quiet"* режим на работа, при който след приключване на програмата на екрана се изобразява времето на протичане.

Параметрите **–p** и **–t** за задължителни за нормална работа на програмите!

Анализ на методите за решаване

Алгоритмите за пресмятане на реда, които ще разгледаме, се базират на разпределяне на членовете му между отделни, паралелно изпълнявани подпрограми (нишки). Всяка подпрограма прави отделни изчисления и прибавя своя получен резултат към споделена памет, съдържаща отговора. След приключване на изпълнението на всички нишки този отговор се визуализира. Ще разгледаме два модела на йерархична архитектура.

Първият модел, на който ще се спрем, е **Master-Slaves [1]**. При него избираме една подпрограма, която влиза в ролята на Master – отговорна е за разпределянето на работата между останалите подпрограми - Slaves, тяхното стартиране и изчакване да завършат. Самите изчисления се извършват в Slave подпрограмите. Те също така актуализират отговора, пазен в споделената памет, който след приключването им се визуализира чрез Master подпрограмата.

Вторият модел, който ще разгледаме, е **Fork/Join [2]** модела за паралелна обработка. При него отново една подпрограма разпределя изчисленията между няколко паралелно изпълнявани подпрограми. Паралелните секции се раздробяват рекурсивно на все по-малки до достигане на определена грануларност (предварително зададена). В този момент започват да се правят отделните изчисления, които впоследствие се обединяват.

Технологии:

Език за програмиране - Java 8

Причина: удобство при конструиране на програми, използващи паралелна обработка, чрез класове Thread, RecursiveAction и **Fork/Join Framework [3]**.

Проектиране

Ще покажем три алгоритъма за решаване на посочената задача, представляващи паралелни програми от тип **Single Program Multiple Data**.

Първото и третото решение са базирани на Master-Slaves модела, като третото предлага съществена оптимизация по време, използвайки допълнителна памет. Второ решение прилага Fork/Join модела, използвайки сходен на първото решение алгоритъм. Идеята тук е да направим сравнение на времетраенето и съответно ускорението при двата различни модела на архитектура на програмите.

Първо решение

1. Идея на алгоритъма

Задачата се състои в това да реализираме пресмятане на сума от **p + 1** на брой събираеми, **k** се мени между 0 и **p**. Тъй като събираемите са независими едно от друго, то можем да разпределим изчислението по следния начин - нишка номер ***i*** (започвайки от 0 до **t – 1**) пресмята събираеми – **i, i + t, i + 2t, i + 3t** и така нататък до най-голямото **k**, ненадвишаващо **p**, което дава остатък **i** при деление на **t** . По този начин, изчисленията, изискващи най-много време (най-големите събираеми), ще са разпределени между всички нишки, така че времената на работа на нишките да са максимално близки.

Тази идея ще използваме и в останалите решения, заедно с други оптимизации.

Окончателно, пресмятаме всяко едно събираемо, което отнема линейно време спрямо **k**, заради пресмятането на факториела. Събираемите са **p + 1** на брой, тоест получаваме квадратична сложност на този алгоритъм.

2. Архитектура на програмата

За реализирането на споменатия по-горе алгоритъм използваме следните Java класове :

```
public class Main {
```

Main class, от който програмата започва изпълнение. Отговаря за обработката на входните параметри. Стартира изпълнение на нишките. Отваря файл за писане на информация за работата на програмата.

```
public class AtomicBigDecimal {

    private final AtomicReference<BigDecimal> valueHolder;
```

AtomicBigDecimal class [4] – споделена памет между нишките, отговаряща за пазенето на търсената сума. Осигурява атомарно актуализиране на сумата от всяка една нишка благодарение на класа **AtomicReference**<**BigDecimal**>. Класът **AtomicBigDecimal** е имплементиран със **Singleton pattern [5]**, за да имаме една споделена инстанция между всички нишки. След приключване на работата на нишките, във файла, който е подаден, се записва стойността на полето **valueHolder**.

```
public class PartialSumCalculator extends Thread {
```

PartialSumCalculator class, който е самата работеща нишка. Той отговаря за пресмятането на част от цялата сума като за целта при създаване получава начален индекс, от който да започва изчислението, стъпка на придвижване – броя нишки, точност на пресмятането, writer, чрез който да записва междинна информация като начало и край на работа на дадена нишка и флаг за тих режим на работа.

В **Main** класа използваме масив от нишки **Thread[]**, в който записваме всичките създадени нишки спрямо подадения на входа на програмата параметър **–t**, стартираме ги и ги присъединяваме.

Второ решение

1. Идея на алгоритъма

При големи стойности на броя цифри след десетичната запетая времето на работа на всяка една нишка логично се увеличава. Това води до разминаване в работата на нишките, тоест времето на най-бързо приключилата да е доста по-малко от времето на най-бавната. За тази цел ще разбием на още по-малки части събираемите, които трябва да изчисли всяка нишка (по-фина грануларност). JDK предоставя за тази цел клас **ForkJoinPool**, чийто конструктор приема брой нишки, с които да работи.

```
forkJoinPool = new ForkJoinPool(NUM_THREADS);
```

Вече всяка нишка не изчислява на един път предвидената за нея работа. Вместо това, ако броят събираеми, които трябва да пресметне, е по-голям от дадена константа (обикновено се нарича **THRESHOLD**), то събираемите се разделят на групи. Вътрешно за всяка нишка се поддържа опашка с два края – **deque**, чиито елементи са разделените групи. Самата нишка изчислява събираемите от групите от единия край на опашката, а ако някоя друга нишка приключи работа по-рано, то тя взима група събираеми от другия край на deque-а на нашата нишка. По този начин се постига по-равномерно разпределение на работата.

Важно е да се уточни, че това разделяне на групи е задача на програмиста. В нашия случай избираме да разделим събираемите за дадена нишка на две групи – най-голямото в първата, второто вай-голямо във втората и така нататък редуваме. Ако новополучените групи продължават да са с повече елементи от **THRESHOLD**, то наново правим разделяне на половина по същия начин. Стойността на константата също е по преценка на програмиста и за жалост се проверява по-скоро експериментално коя би свършила най-добра работа.

2. Архитектура на програмата

По подобие на първото решение и тук използваме класа **AtomicBigDecimal** и няма да описваме отново неговото значение.
Main класът също е със сходна структура. Единствената разлика с първото решение е използването на **ForkJoinPool**, за който говорихме в първата точка. Той стартира работа на нишките чрез метода **execute()** и за да сме сигурни, че всички нишки са приключили, извикваме други два метода на класа – **shutdown()** и **awaitTermination().** Интересната част тук е изграждането на класа нишка. Този клас наследява **RecursiveAction**.

```
public class CustomRecursiveAction extends RecursiveAction {
```

CustomRecursiveAction class се характеризира с 2 конструктора. Първият се извиква от **Main** класа, за да имаме достъп до търсената точност, позицията на първото събираемо и броя нишки – отстоянието до следващото събираемо за тази нишка както и **THRESHOLD**. След като един път определим събираемите за дадена нишка, ги записваме в масив **requiredCalculations**[], който извикваме във втория конструктор при разделянето на групи от събираеми. По този начин се постига по-фина грануларност.

```
@Override
protected void compute() {
    if (requiredCalculations == null) {
        requiredCalculations = getRequiredCalculations();
    }
    if (requiredCalculations.length > THRESHOLD) {
        ForkJoinTask.invokeAll(createSubTasks());
    } else {
        BigDecimal partialResult = getPartialSum();
        AtomicBigDecimal result = AtomicBigDecimal.getInstance();
        result.setValue(result.addAndGet(partialResult));
    }
}
```

Този метод отговаря за стартиране на работата на нишката. Тук правим проверка за броя на събираемите, които трябва да пресметнем и ако този брой е по-голям от **THRESHOLD**, правим разделяне.

Трето решение

1. Идея на алгоритъма

Ключовото наблюдение, което правим тук, е, че не е необходимо за всяко събираемо наново да се изчислява факториелът в знаменателя. Вместо това поддържаме допълнителна памет, масив от тип **BigInteger**, в който ще пазим вече намерените факториели до даден момент и ще изчисляваме текущия, тръгвайки от най-близкия по-малък от него вече намерен. По този начин преизползваме междинните резултати по време на изчисленията.

Търсенето на най-близкия вече пресметнат факториел изчисляваме с двоично търсене. По този начин сложността на цялостния алгоритъм пада от **O(n^2)** до **O(nlogn)**, където **n** е търсената точност.

Правим и една малка оптимизация по памет, тъй като се интересува за **(3k)!** за всяко събираемо **k**. Вместо да използваме **3(p + 1)** индекса и да заделяме съответно толкова памет, всеки индекс **i** пази **(3i)!**, понеже ни интересуват само кратните на 3 факториели.

```
[u81650@t5600 ~]$ java -jar e-calculation-fast.jar -p 50000 -t 16
[u81650@t5600 ~]$ cat output.txt
pool-1-thread-1 started.
pool-1-thread-3 started.
pool-1-thread-2 started.
pool-1-thread-6 started.
pool-1-thread-4 started.
pool-1-thread-9 started.
pool-1-thread-11 started.
pool-1-thread-5 started.
pool-1-thread-7 started.
pool-1-thread-8 started.
pool-1-thread-10 started.
pool-1-thread-12 started.
pool-1-thread-13 started.
pool-1-thread-14 started.
pool-1-thread-15 started.
pool-1-thread-16 started.
pool-1-thread-2 stopped.
pool-1-thread-3 stopped.
pool-1-thread-5 stopped.
pool-1-thread-4 stopped.
pool-1-thread-6 stopped.
pool-1-thread-9 stopped.
pool-1-thread-7 stopped.
pool-1-thread-10 stopped.
pool-1-thread-10 execution time was (millis): 23441
pool-1-thread-9 execution time was (millis): 23442
pool-1-thread-5 execution time was (millis): 23440
pool-1-thread-3 execution time was (millis): 23445
pool-1-thread-8 stopped.
pool-1-thread-7 execution time was (millis): 23441
pool-1-thread-2 execution time was (millis): 23442
pool-1-thread-8 execution time was (millis): 23441
pool-1-thread-6 execution time was (millis): 23443
pool-1-thread-4 execution time was (millis): 23441
pool-1-thread-12 stopped.
pool-1-thread-12 execution time was (millis): 23442
pool-1-thread-13 stopped.
pool-1-thread-13 execution time was (millis): 23440
pool-1-thread-11 stopped.
pool-1-thread-11 execution time was (millis): 23445
pool-1-thread-14 stopped.
pool-1-thread-14 execution time was (millis): 23441
pool-1-thread-15 stopped.
pool-1-thread-15 execution time was (millis): 23440
pool-1-thread-1 stopped.
pool-1-thread-1 execution time was (millis): 23454
pool-1-thread-16 stopped.
pool-1-thread-16 execution time was (millis): 23441
Threads used in current run: 16
Total execution time for current run (millis): 23467
```

В тази програма няма да използваме **ForkJoinPool** класа тъй като опитите показват, че нишките работят с доста сходно време както може да се види от теста вляво.

2. Архитектура на програмата

Както вече споменахме, използваме допълнителна памет да кешираме вече намерените факториели.

```
        factorialsContainer = new BigInteger[PRECISION + 2];
```

Отново както в останалите решения ще използваме **AtomicBigDecimal** класа за съхранение на резултата. **Main** класът е със същото предназначение както в първото решение – обработва входа и стартира работата на нишките. Оптимизацията в класа **PartialSumCalculator** е следната :

```
private BigInteger getFactorial(int n) {
    int startPosition = 2;
    BigInteger result = BigInteger.ONE;

    int closestIndex = findClosestCalculatedFactorialIndex(n / 3);

    if (closestIndex != -1) {
        startPosition = 3*closestIndex + 1;
        result = factorialsContainer[closestIndex];
    }
```

Методът **findClosestCalculatedFactorialIndex**() намира индекса на най-близкия факториел, а чрез споделената памет **factorialsContainer**[] достъпваме самата стойност. Остава само линейно да пресметнем останалите малко на брой множители до желания **факториел**. Веднага след това актуализираме споделената памет с новата вече пресметната стойност.

```
private int findClosestCalculatedFactorialIndex(int n) {
```

Двоично търсене по масива с вече пресметнатите факториели.

Проведени тестове и измервания

Ще проведем тестове на машини със следните параметри:

Персонален компютър	Предоставен сървър
CPU(s): 12 Thread(s) per core: 2 Model name: Intel® Core ™ i7-8750H CPU @ 2.20 GHz	CPU(s): 32 Thread(s) per core: 2 Model name: Intel® Xeon® CPU E5-2660 @ 2.20 GHz

Таблиците по-долу представят измервания на бързодействието на трите решения при **5000** цифри точност след десетичната запетия на персонален компютър с посочените в таблицата по-горе параметри.

Първо решение

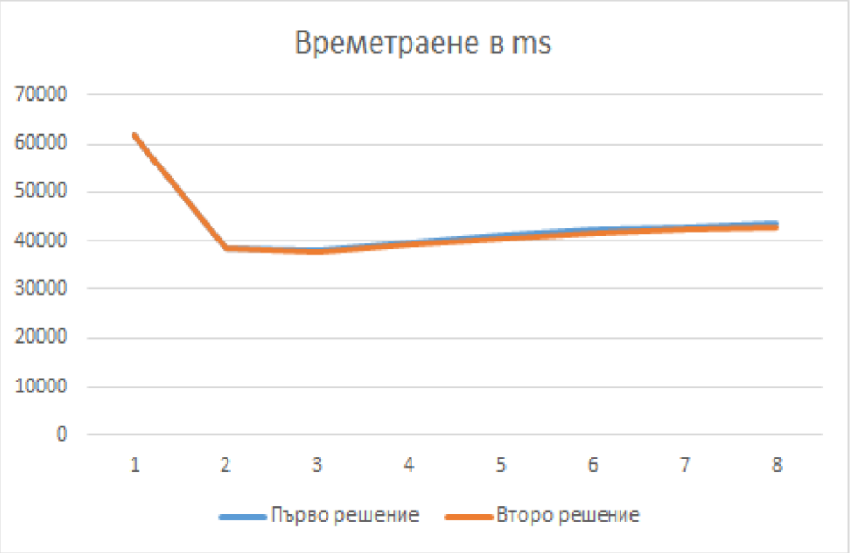
Брой нишки	***T_1(ms)***	***T_2(ms)***	***T_3(ms)***	***T = min()***	***Ускорение***	***Ефикасност***
1	61635	61680	61702	61635	1	1
2	38682	38674	38617	38617	1,596058731	0,798029365
3	37979	37925	37917	37917	1,625524171	0,54184139
4	41838	39979	39831	39831	1,547412819	0,386853205
5	41315	41303	41194	41194	1,496213041	0,299242608
6	42432	44192	42561	42432	1,452559389	0,242093232
7	43074	43001	42893	42893	1,436947754	0,205278251
8	43734	43671	43649	43649	1,412059841	0,17650748

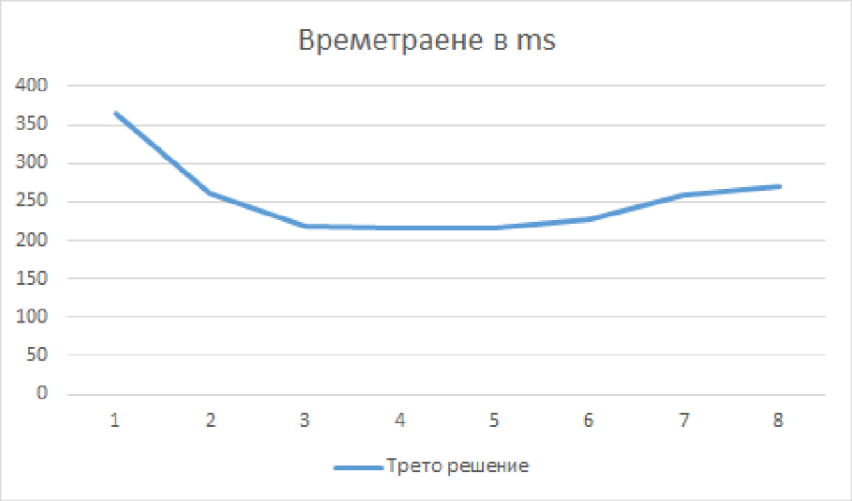
Второ решение

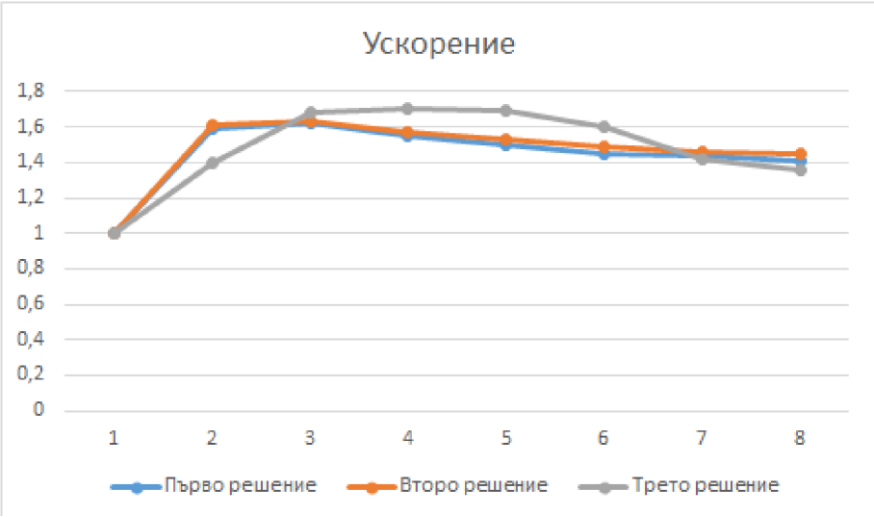
Брой нишки	***T_1(ms)***	***T_2(ms)***	***T_3(ms)***	***T = min()***	***Ускорение***	***Ефикасност***
1	62218	62127	61849	61849	1	1
2	38814	38443	38764	38443	1,608849465	0,804424733
3	37993	37812	37844	37812	1,635697662	0,545232554
4	39323	39307	39265	39265	1,575168725	0,393792181
5	40490	40549	40703	40490	1,527512966	0,305502593
6	41689	41494	41621	41494	1,490552851	0,248425475
7	42295	42507	42369	42295	1,462324152	0,20890345
8	42971	43007	42824	42824	1,444260228	0,180532528

Трето решение

Брой нишки	*T_1(ms)*	*T_2(ms)*	*T_3(ms)*	*T = min()*	*Ускорение*	*Ефикасност*
1	366	369	378	366	1	1
2	261	275	266	261	1,402298851	0,701149425
3	234	218	229	218	1,678899083	0,559633028
4	217	229	215	215	1,702325581	0,425581395
5	227	216	228	216	1,694444444	0,338888889
6	232	228	243	228	1,605263158	0,26754386
7	259	258	259	258	1,418604651	0,202657807
8	277	269	286	269	1,360594796	0,170074349







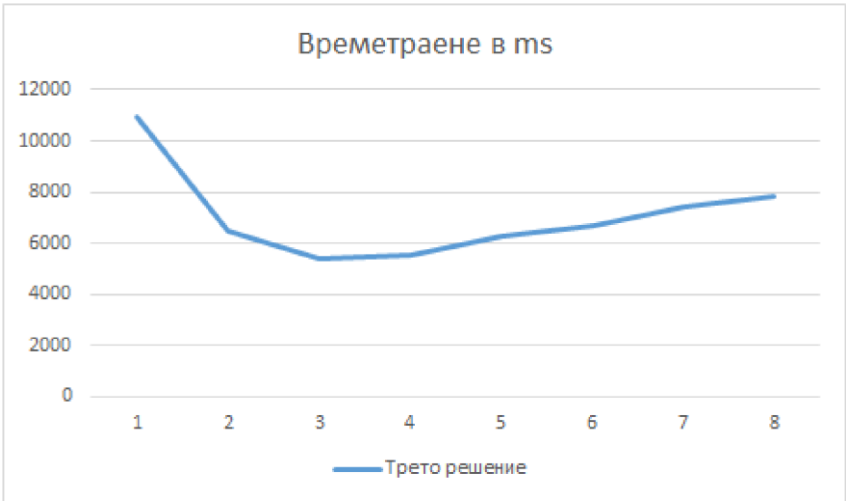
Анализ на данните от опитите

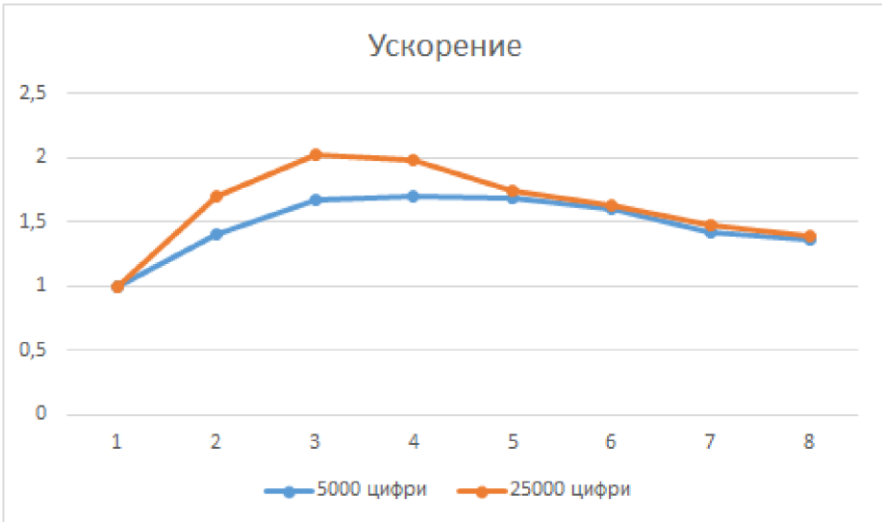
От таблиците и графиките се вижда, че използването на по-фина грануларност във второто решение повишава съвсем леко ускорението. Третото решение се справя за значително по-кратко време със задачата и поради характера на алгоритъма и именно по-ниската сложност не отчита почти никакво ускорение с нарастване на броя нишки.

За постигане на малко по-високо ускорение за третото решение увеличаваме **5** пъти точността – от **5000** на **25000** цифри след десетичната запетая. Тестването е отново на **персоналния компютър**. Таблицата по-долу представя резултатите от измерванията

Трето решение

Брой нишки	***T_1(ms)***	***T_2(ms)***	***T_3(ms)***	***T = min()***	***Ускорение***	***Ефикасност***
1	10998	10949	11003	10949	1	1
2	6496	6456	6646	6456	1,69594176	0,84797088
3	5493	5418	5527	5418	2,020856405	0,673618802
4	5805	5528	5669	5525	1,981719457	0,495429864
5	6589	6256	6982	6256	1,750159847	0,350031969
6	6903	6701	7105	6701	1,633935234	0,272322539
7	7492	7635	7435	7435	1,472629455	0,210375636
8	8139	8167	7834	7834	1,397625734	0,174703217





Заключение

С увеличаване на точността наблюдаваме леко повишаване на ускорението. Това показва, че при изискване на по-високата точност, съответно повече необходимо време за работа, наблюдаваме по-добро ускорение. За жалост личният компютър не позволява увеличаване на точността много повече от това, тъй като при последващо тестване класът **BigInteger** хвърля изключение.

Таблиците по-долу представят измервания на бързодействието на трите решения при **5000** цифри точност след десетичната запетия на **предоставения сървър**.

Първо решение

Брой нишки	***T_1(ms)***	***T_2(ms)***	***T_3(ms)***	***T = min()***	***Ускорение***	***Ефикасност***
1	157670	155768	159112	155768	1	1
2	88419	89045	88789	88419	1,761702801	0,880851401
3	63049	61780	61147	61147	2,54743487	0,849144957
4	45590	45505	46101	45505	3,423096363	0,855774091
5	38374	39014	39560	38374	4,059206755	0,811841351
6	34056	33460	32138	32138	4,846847968	0,807807995
7	29613	29978	30014	29613	5,260122244	0,751446035
8	26082	26573	26942	26082	5,972241393	0,746530174
9	24934	24319	25135	24319	6,405197582	0,71168862
10	23124	22865	22550	22550	6,90767184	0,690767184
11	21937	22800	22456	21937	7,100697452	0,64551795
12	21314	21079	21876	21079	7,38972437	0,615810364
13	21810	22412	22785	21810	7,142044934	0,549388072
14	20347	19832	20189	19832	7,854376765	0,561026912
15	19683	19716	19302	19302	8,070044555	0,53800297
16	19201	18728	18984	18728	8,317385733	0,519836608
17	19128	19387	18904	18904	8,239949217	0,484702895
18	19365	19613	19994	19365	8,052106487	0,447339249
19	20018	20731	19890	19890	7,831473102	0,412182795
20	20438	20875	21019	20438	7,621489383	0,381074469

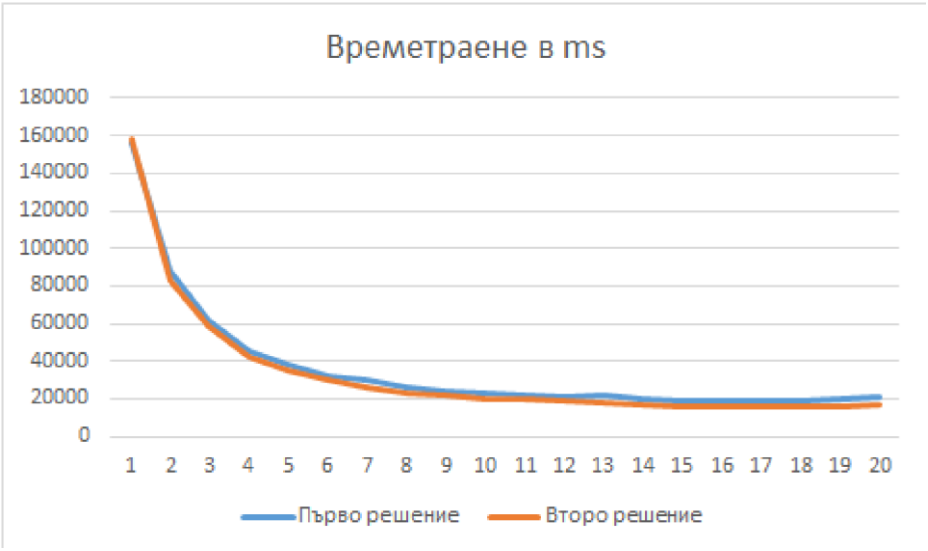
Второ решение

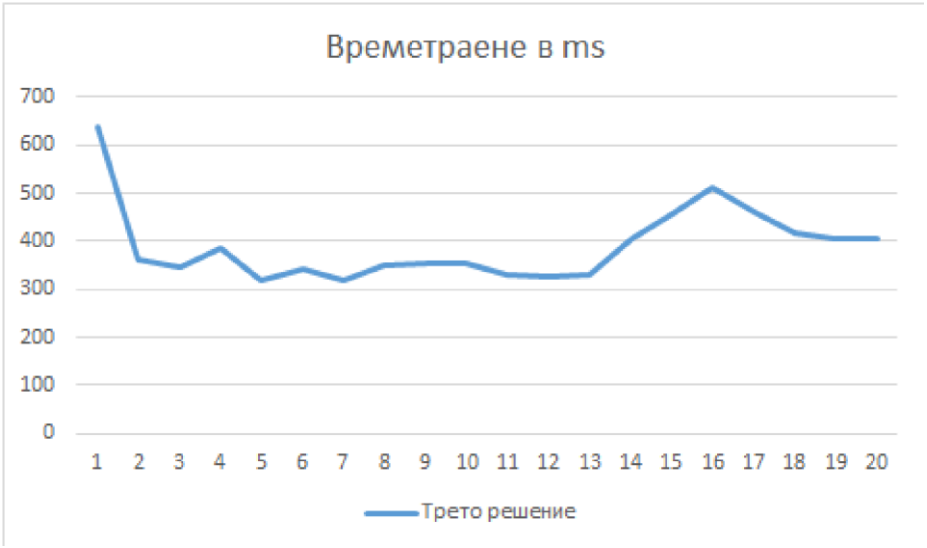
Брой нишки	***T_1(ms)***	***T_2(ms)***	***T_3(ms)***	***T = min()***	***Ускорение***	***Ефикасност***
1	159914	158198	161359	158198	1	1
2	82793	83467	83930	82793	1,910765403	0,955382701
3	58314	58938	59382	58314	2,712864835	0,904288278
4	43084	42994	42776	42776	3,69828876	0,92457219
5	35781	35072	35497	35072	4,510663777	0,902132755
6	30679	30186	30791	30186	5,240773869	0,873462311
7	26539	26318	26024	26024	6,078927144	0,868418163
8	23279	23811	23908	23279	6,795738649	0,849467331
9	21957	22016	21615	21615	7,318898913	0,81321099

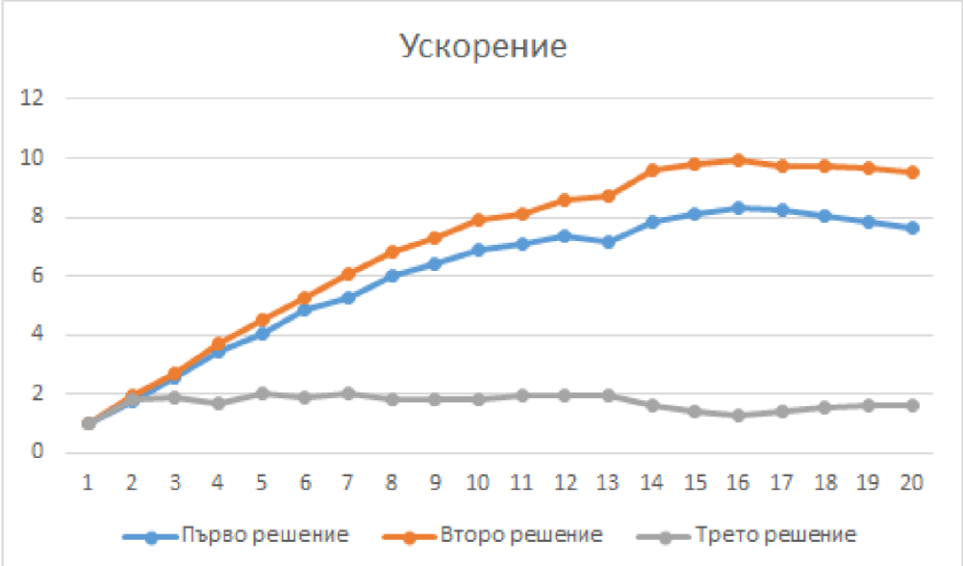
10	20493	20068	20918	20068	7,883097469	0,788309747
11	19516	20104	19718	19516	8,106066817	0,736915165
12	18678	18458	18795	18458	8,570701051	0,714225088
13	18108	18206	18387	18108	8,73635962	0,672027663
14	16781	16523	16815	16523	9,574411426	0,68388653
15	16258	16629	16120	16120	9,813771712	0,654251447
16	15906	15889	16018	15889	9,956447857	0,622277991
17	16066	16334	16208	16066	9,695506038	0,570323885
18	16060	16354	16072	16060	9,699128269	0,538840459
19	16395	16178	16439	16178	9,628384225	0,506757064
20	16376	16492	16613	16376	9,511968735	0,475598437

Трето решение

Брой нишки	***T_1(ms)***	***T_2(ms)***	***T_3(ms)***	***T = min()***	***Ускорение***	***Ефикасност***
1	674	639	672	639	1	1
2	448	361	404	361	1,770083102	0,885041551
3	346	385	413	346	1,846820809	0,615606936
4	405	387	388	387	1,651162791	0,412790698
5	473	378	320	320	1,996875	0,399375
6	724	681	343	343	1,862973761	0,310495627
7	342	363	320	320	1,996875	0,285267857
8	351	387	353	351	1,820512821	0,227564103
9	353	365	377	353	1,8101983	0,201133144
10	353	356	385	353	1,8101983	0,18101983
11	332	407	378	332	1,924698795	0,174972618
12	358	327	362	327	1,95412844	0,162844037
13	371	434	331	331	1,930513595	0,148501046
14	430	408	406	406	1,573891626	0,11242083
15	456	457	467	456	1,401315789	0,093421053
16	511	702	579	511	1,250489237	0,078155577
17	963	461	777	461	1,386117137	0,081536302
18	465	434	418	418	1,528708134	0,08492823
19	405	409	453	405	1,577777778	0,083040936
20	425	499	407	407	1,57002457	0,078501229


Времетраене в ms
180000
160000
140000
120000
100000
80000
60000
40000
20000
0
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
Първо решение
Второро решение


Времетраене в ms
700
600
500
400
300
200
100
0
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
Трето решение


Ускорение
12
10
8
6
4
2
0
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
Първо решение
Второро решение
Трето решение

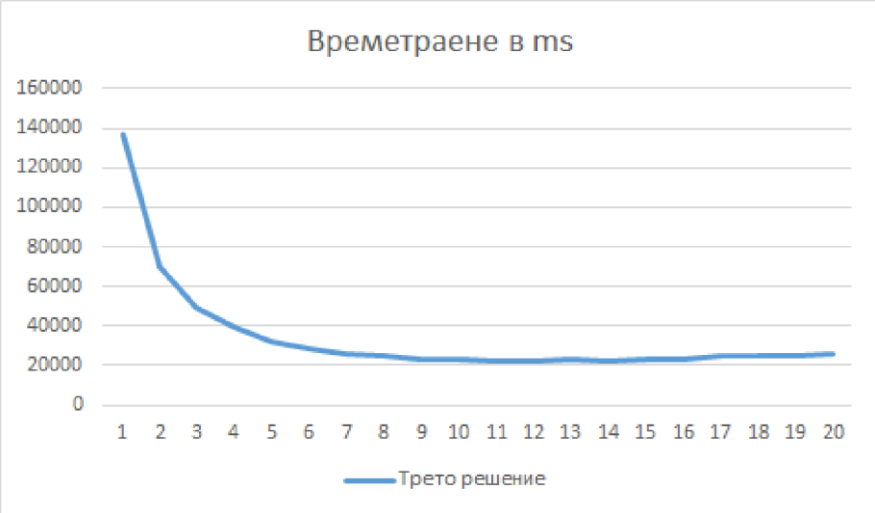
Анализ на данните от опитите

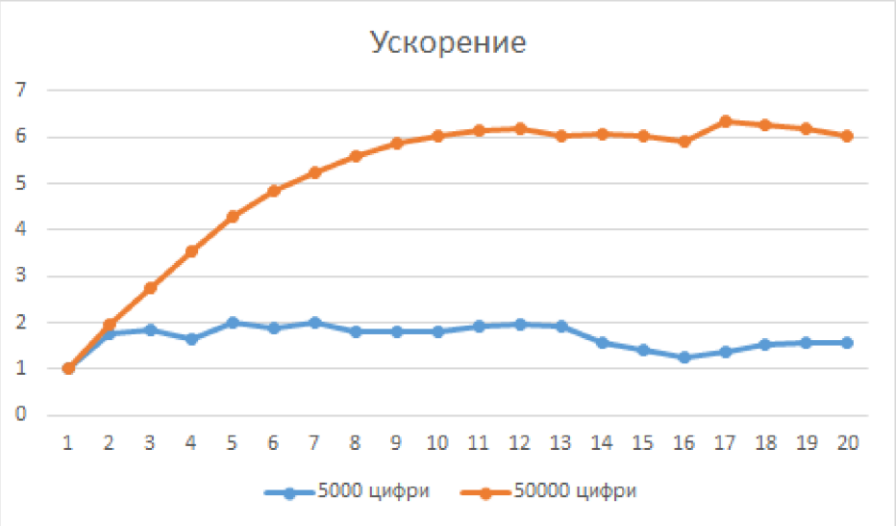
От таблиците и графиките се вижда, че използването на по-фина грануларност във второто решение повишава най-доброто ускорението от 8.31 на 9.95 пъти. Третото решение се справя за значително по-кратко време със задачата и поради характера на алгоритъма и именно по-ниската сложност не отчита почти никакво ускорение с нарастване на броя нишки.

За постигане на значително по-високо ускорение за третото решение увеличаваме **10** пъти точността – от **5000** на **50000** цифри след десетичната запетая. Тестването е отново на **предоставения сървър**. Таблицата по-долу представя резултатите от измерванията

Трето решение

Брой нишки	***T_1(ms)***	***T_2(ms)***	***T_3(ms)***	***T = min()***	***Ускорение***	***Ефикасност***
1	138906	137362	138611	137362	1	1
2	70340	71049	70917	70340	1,952829116	0,976414558
3	49714	50318	50018	49714	2,763044615	0,921014872
4	39305	39517	38983	38983	3,523638509	0,880909627
5	32243	31947	32439	31947	4,299683851	0,85993677
6	28718	28372	28945	28372	4,841463415	0,806910569
7	26398	26874	26144	26144	5,254054468	0,75057921
8	24518	25147	25398	24518	5,602496125	0,700312016
9	23941	23466	23876	23466	5,853660615	0,650406735
10	22956	23048	22838	22838	6,014624748	0,601462475
11	22365	22712	22891	22365	6,14182875	0,558348068
12	22587	22261	22911	22261	6,170522438	0,514210203
13	23058	23846	22727	22727	6,044000528	0,464923118
14	22669	24309	23171	22669	6,059464467	0,43281889
15	22983	22770	23514	22770	6,032586737	0,402172449
16	23709	23804	23202	23202	5,920265494	0,370016593
17	24500	24838	25954	24500	6,357877551	0,373992797
18	31170	25484	24796	24796	6,281980965	0,348998942
19	25135	25886	25521	25135	6,197254824	0,326171307
20	30151	35696	25773	25773	6,043844333	0,302192217





Заключение

Това решение е доста по-бързо по време от предишните две заради по-ниската сложност. Освен това нишките работят с до няколко милисекунди разлика при положение, че времето на работа на програмата е повече от 20 секунди, тоест изчисленията, които трябва да направят, са добре разпределени между нишките.

Недостатъкът на решението е липсата на високо ускорение, съответно и по-ниската ефикасност. По-високо ускорение би се получило при подаване на по-голяма точност, съответно повече време на работа, при което паралелното смятане би дало по-добър резултат. Тук обаче се сблъскваме и с проблема, че с нарастване на точността, пресмятането на факториела става непосилно за класа **BigInteger**. Все пак, дори и тази точност е доста задоволителна, имайки предвид и времето необходимо за нейното изчисляване.

Използвана литература

[1]Master-Slaves
https://www.tutorialspoint.com/software_architecture_design/hierarchical_architecture.htm

[2] Fork/Join model
https://en.wikipedia.org/wiki/Fork%E2%80%93join_model
Автор: Wikipedia
Последна промяна: 01.2020

[3] Fork/Join Framework
https://www.baeldung.com/java-fork-join
Автори: Baeldung Team
Последна промяна: 04.2020

[4] AtomicBigDecimal
https://github.com/qbit-for-money/commons/blob/master/src/main/java/com/qbit/commons/model/AtomicBigDecimal.java
Автор: Alexander Alexandrov
Дата: 2014

[5] Singleton pattern
https://dzone.com/articles/singleton-bill-pugh-solution-or-enum
Автор: Harinath Kuntamukkala
Дата: 2017