Условие на задачата:

Да се пресметне числото pi (3.14159...) чрез формулата на Рамануджан :

$$\frac{4}{\pi} = \frac{1}{882} \sum_{n=0}^{\infty} \frac{(-1)^n (4n)!}{(4^n n!)^4} \frac{1123 + 21460n}{882^{2n}}$$

Пресмятането да се извърши чрез паралелни процеси и пресмята pi със зададена от потребителя точност.

Функционалности:

1. Точността е изразена чрез брой членове. Чрез командния параметър “-p XXXXX”.
2. Команден параметър да задава броя използвани нишки, на които разделяме задачата “-t XX” или “-task XX”.
3. Програмата да извежда лог на етапите от работата си, както и времето за работа.
4. Да се поддържа quietMode чрез “-q”.
5. Записване на резултата във файл чрез използването на команден параметър “-o result.txt” или само “-o”.

Преглед на алгоритмите и архитектура на приложението:

За пресмятането на π се налага да използваме BigDecimal понеже се получават доста големи числа. Програмата намира π с точност десет хиляди цифри след десетичната запетая и връща този резултат. В зависимост от броят членове на реда подадени при стартирането ѝ (подадени чрез командния параметър -p) тези цифри се променят.

Разделяме формулата на отделни части, за да я направим по-лесна и бърза за пресмятане.

Първо се фокусираме върху факториелите. Понеже не искаме да пресмятаме един и същ факториел във всяка нишка записваме резултатите в обща памет. Ако нужната стойност не е открита тя се пресмята - няма чакане на резултат от друга нишка.

Следва изчислението на 882^{2n} функцията за пресмятане на тази стойност работи на същия принцип като функцията за пресмятане на факториели.

Програмата е от един клас наречен MainApp. В него се изчислява сумата имплементира Runnable класът. Програмата също използва библиотеки като BigDecimal, MathContext(за закръгляне RoundingMode.*HALF_UP*, и за прецизност при делението на BigDecimal числа), Arrays(използва се при създаване на векторът, който взема String[] args и използваме Arrays, за да представим стринга като лист). След създаването на вектора се използва .contains, за откриването на командните параметри -p, -t, -q, -o. Ако открием даден параметър и очакваме той да ни зададе стойност за изпълнение на програмата - откриваме индекса на параметъра и вземаме следващия елемент във вектора.

При стартирането на програмата се стартира нишка която изчислява част от факториелите и ги записва в низа factorials.

След като имаме определен брой факториели пресметнати и запазени стартираме същинските нишки и ги join-ваме. В предна версия на задачата се използваше Busy waiting, но чрез join-ване програмата работи значително по-бързо при по-голям брой членове.

При стартирането на нишките подадени чрез команден параметър (-t или -task) или една зададена като default при липсата на този параметър се започва изпълнението на същинската част или функцията static Runnable appRun(final int start, final int end, int numThreads, boolean qm). Където start ∈ [0, numThreads] и numThreads е броят нишки за които изпълняваме програмата; end = sizeofN или броят членове, за които стартираме програмата. Изпълнява се цикъл намиращ даден елемент a_n на реда. Цикълът е със стъпка равна на броят на стартирани нишки - така избягваме постоянно пресмятане на едни и същи стойности от всяка нишка поотделно.
След това се създава BigDecimal pi = BigDecimal.*ZERO*;
Раздробяваме формулата на Рамануджан на top и bot (числител и знаменател).

При намирането им се изчислява a_n. То бива изпратено на функцията void assignResult(BigDecimal pi), която има за задача да прибави дадения член към result, който е равен на сумата до момента. Понеже assignResult() е synchronized това означава, че се предотвратява thread interference и греш в последователността на паметта.

Когато нишките свършат работата си - имаме result, равен на сумата от 0 до sizeofN-тия член. Остава изчислението на π, което се извършва в малко преди края на програмата. Програмата завършва като си засече времето си на работа чрез long start и long stop ,които имат стойност получена от System.*currentTimeMillis*(). Остава единствено записването на π в файл - *savetofile*(fileName,*result*).

Проведени тестове:

Проведени са тестове за 2 различни стойности на брой членове на реда (n=5000, n=10 000).

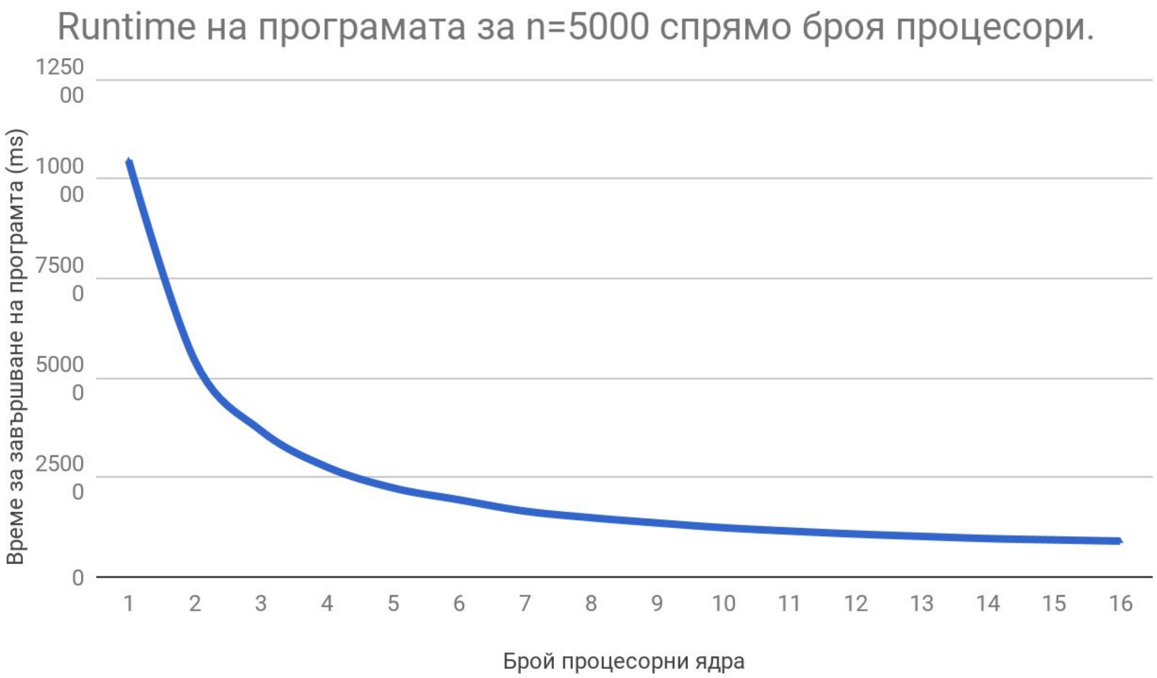
Изчислено е ускорението $S_p = \frac{T_1}{T_p}$, ефективността $E_p = \frac{S_p}{p}$,

Първият тест извършен е при n=5000.

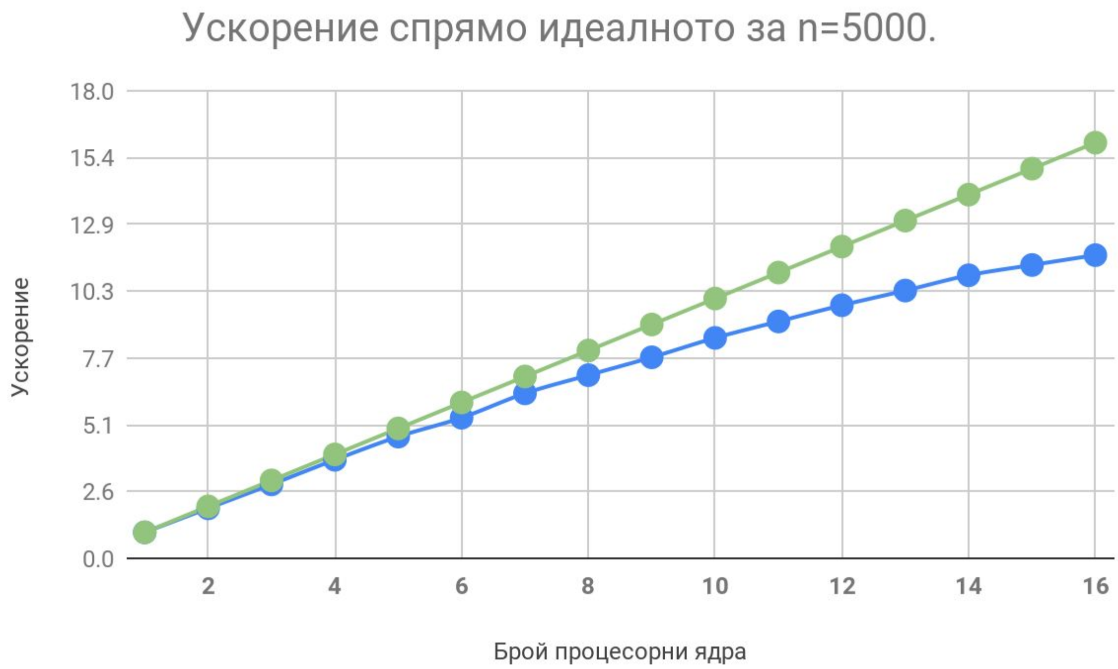
Брой ядра	Време в ms	Ускорение (Sp)	Ефективност (Ep)
1	104722	1	1
2	54383.6	1.925617282	0.9628086408
3	36781	2.847176531	0.9490588438
4	27607.8	3.79320337	0.9483008425
5	22351.2	4.685296539	0.9370593078
6	19348.8	5.412325312	0.9020542187
7	16492.6	6.349635594	0.9070907992
8	14850.8	7.051606647	0.8814508309
9	13532.8	7.738383779	0.8598204199
10	12342.4	8.484735546	0.8484735546
11	11485	9.118154114	0.8289231013
12	10750.4	9.741218931	0.8117682443
13	10160.6	10.3066748	0.7928211387
14	9611.2	10.89582987	0.7782735618
15	9274.6	11.29126863	0.7527512417
16	8972.6	11.67131043	0.7294569021

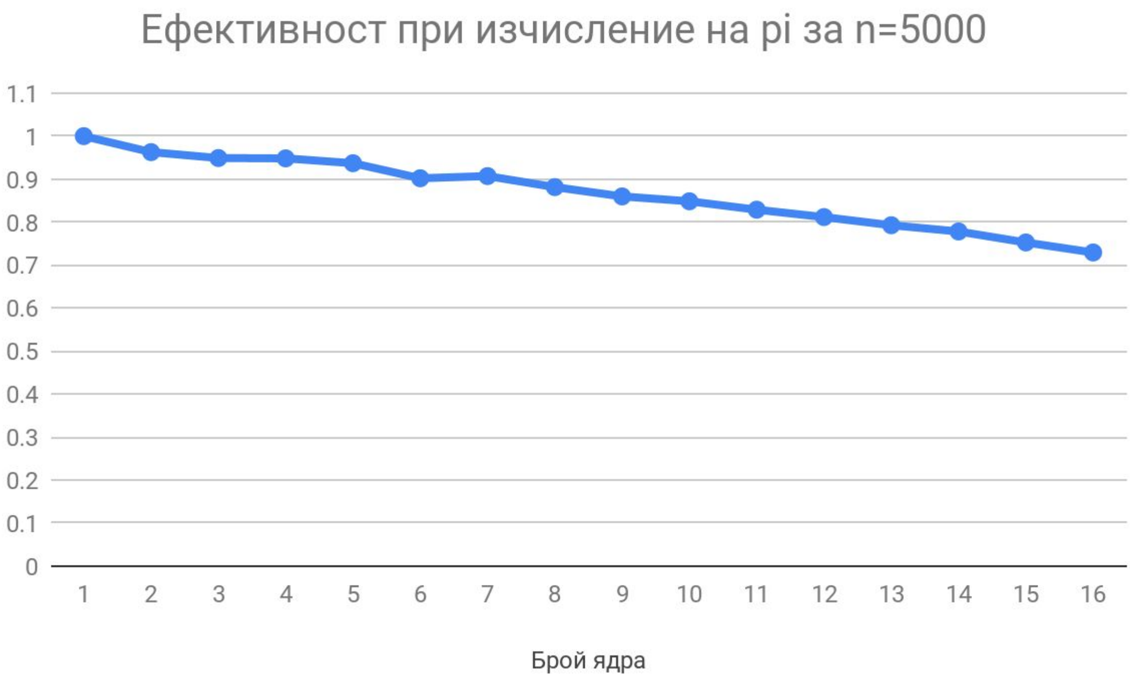
С увеличаване на броя процесорни ядра използвани се намалява времето на изпълнение на задачата.

Следната диаграма показва показва в синьо е показано времето необходимо за намирането на π за 5000 члена на реда на Рамануджан.



На диаграма 2 е изобразено ускорението при изчисляване на pi при 5000 броя членове на реда на Рамануджан и различен брой използвани процесорни ядра.





Горната диаграма показва ефективността (Ep) при намирането на pi при n=5000 при употреба на различен брой ядра, започвайки от 1 (серийна версия на програмата) до 16 ядра.