Софийски Университет „Св. Климент Охридски”

Факултет по математика и информатика

Ръководство на разработчика

“BetEuler”

1. Увод

1.1. Проблем който проекта решава:

Проекта представлява система за залагания, чрез която потребителя може да залага дали резултата от изчислението на числото на Ойлер ще е под или над средното време. Потребителя може да обособи начина на пресмятане на числото, сумата на залога, както и да специфицира типа на залога си както и да добави данни свързани с него. Използването на приложението става изцяло чрез конзолни параметри. 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:

$$e = \sum \frac{2k+1}{(2k)!}, k = 0, \ldots, \infty$$

1.1.2. Анализ на проблемите които изчисляването на числото на Ойлер поражда

Задачата поражда следните 2 проблема на които трябва да се намери решение:

1. Как се пресмята всеки факториел?

2. Кога трябва да спре пресмятането на числото до определена прецизност подадена от потребителя?

Следователно задачата има две подзадачи - пресмятането на факториел и пресмятането на дробите

От образците [1] и [2] примерни решения са:

На 1 въз жможните решения за пресмятана на факториел са:

1.1. Пресмятаме всеки (2k)! чрез стандартната библиотека в Java

1.2. Divide and conquer strategy за пресмятането на всеки (2k)!

1.3. Предварително пресмятане на всички факториели които ще ни са нужни при пресмятането

1.4. Пресмятане на даден факториел паралелно чрез ExecutorService(паралелно пресмятане)

За втория въпрос може да се разгледа като следната задача:

Да се намери такова положително число “p”, такова че сумата от [0;p] в формулата от условието да гарантира че сме пресметнали до “k”-тия знак. Решение:

Suppose that we define

$$a_n=\frac{2n+1}{(2n)!}\implies \frac{a_{n+1}}{a_n}=\frac{2n+3}{2(n+1)(2n+1)^2}$$

Writing

$$\sum_{n=0}^{\infty}a_n=\sum_{n=0}^{p}a_n+\sum_{n=p+1}^{\infty}a_n=\sum_{n=0}^{p}a_n+R_p$$

and *suppose* that we can *approximate*

$$R_p\sim a_p\sum_{n=p+1}^{\infty}\frac{2n+3}{2(n+1)(2n+1)^2}$$

So,

$$R_p=\frac{2p+1}{2(2p)!}\left(\psi^{(0)}\left(p+\frac{3}{2}\right)-\psi^{(0)}(p+2)+\psi^{(1)}\left(p+\frac{3}{2}\right)\right)$$

Taking logarithms and using very early truncated expressions,

$$\ln(R_p)\sim-\ln\left(4\sqrt{\pi}\right)-2p(\ln(p)-1+\ln(2))$$

and we want $R_p<10^{-k}$. This would give,as an *approximation*,

$$p=\frac{A}{2W\left(\frac{A}{e}\right)}\qquad\text{where}\qquad A=\ln\left(\frac{10^k}{4\sqrt{\pi}}\right)$$

$W(.)$ being Lambert function. For sure, you will need to use $\lceil p\rceil$.

1.2. Анализ на решаване на задачата

1.2.1. Терминологичен речник: Взаимствани са познания от различни статии изложени в края на документа.

1.2.2. Език за разработване: За имплементиране на задачата е избран

езикът Java тъй като поддържа имплементация на нишки, както и виртуални оптимизации "под капака" на JVM(Java Virtual Machine).

1.2.3. Използвани библиотеки

guava-11.0.2 - за точно измерване чрез Таймер.[1]

commons-cli-1.4.0 - за подаване на аргументи от конзолата [2]

junit-jupiter-engine.5.4.2.jar - за тестване [3]

junit-jupiter-commons-1.5.2.jar - за тестване [3]

log4j-api.2.11.2.jar - за логване [4]

log4j-core.2.11.jar - за логване [4]

lombok-1.18.4.jar - за логване [5]

Използваните библиотеки за тестване спомагат за писането на по кратък код за тестване.

Използвайки Log4j библиотеката ни дава възможност за различни опции на логване.Guava библиотеката ни дава прецизност при пресмятането на времето, което е отнело за изчисление.Използване Eclipse IDE за менажиране на проекта и добавяне на библиотеки.Използвани са вградените в Java Класове ThreadPoolExecutor.java и ForkedJoinPool.java за менажиране на нишки както и java 10+ lambda изрази за сравнение на производителността и по-кратък код.

1.2.4. Инфраструктурни изисквания

1.2.4.1. Модел на обслужване - Модела на обслужване е клиент сървър.Клиента подава параметри, чрез които нашата програма валидира неговия вход и в случай, че са

правилни изчислява стойността и му връща отговор както и допълнителна информация за времето протекло на цялостното изчисление

1.2.4.2. Анализ на софтуерните модели:

Алгоритъмът за пресмятане на реда включва разпределяне на работата между отделни нишки които работят паралелно и сумиране на резултата от всяка.Възможни решения са:

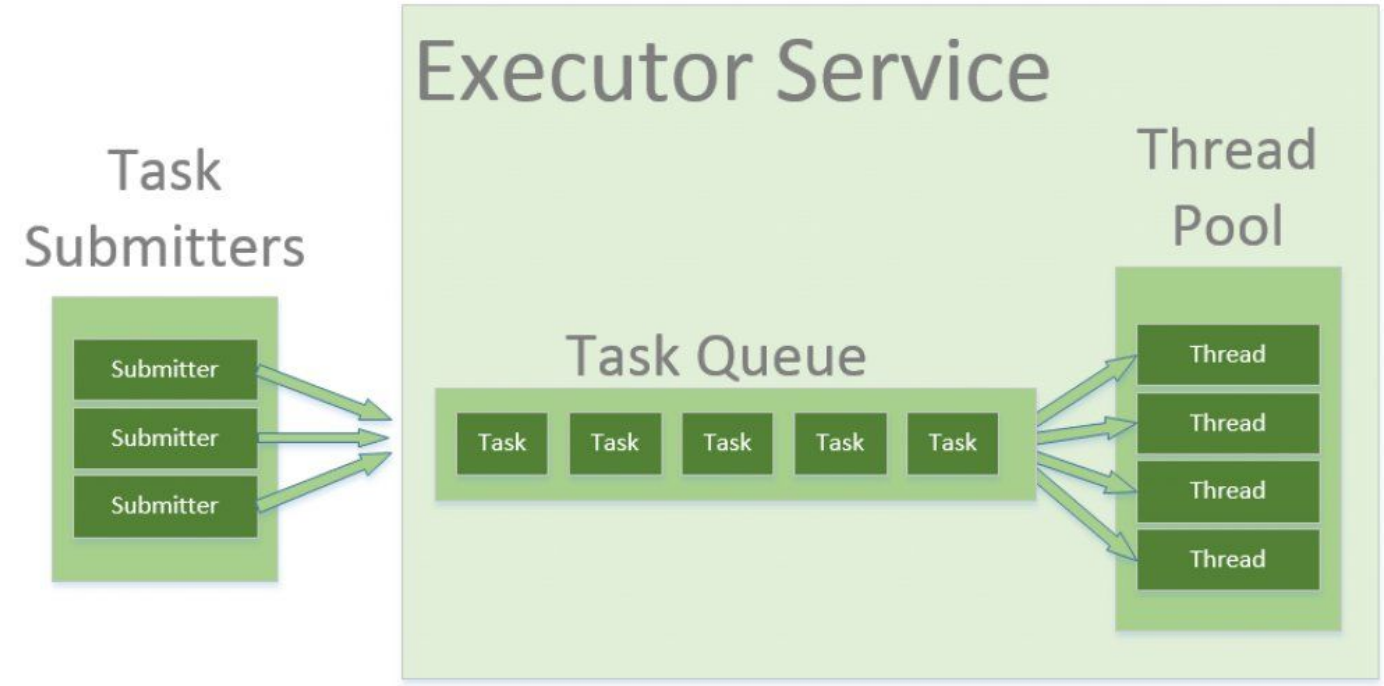
1.2.4.2.1. Решението чрез 1 нишка без оптимизация - не представлява оптимизация, има за цел да покаже резултати в най-лошия случай.

1.2.4.2.2. Решение чрез създаване и менажиране на собствени нишки - Използвания алгоритъм за пресмятане на реда включва разпределение на членовете от математическия израз между отделни нишки и сумиране на резултатите.Архитектурата използвана тук е Master - Slave [6]. Тук имаме Master нишка, която отговаря за останалите нишки които наричаме Slaves.Master нишката събира всички пресметнати отговори от всички Slaves за да генерира резултата, като изчислението става синхронно

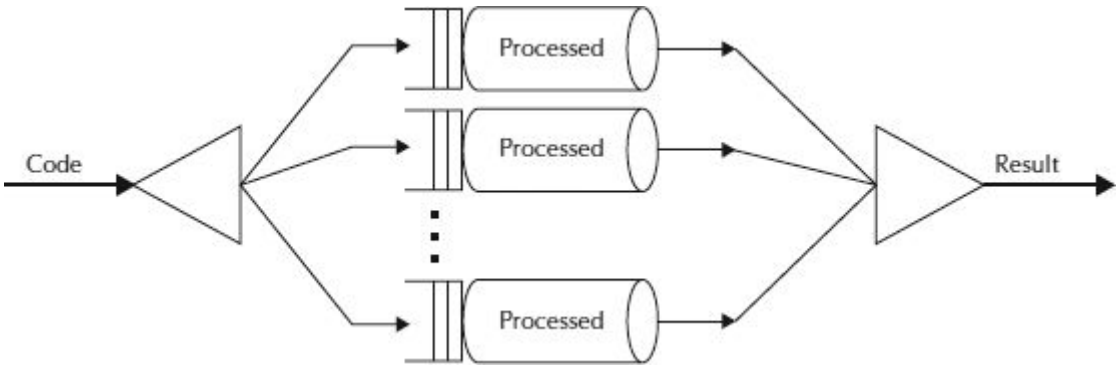
1.2.4.2.3. Решение чрез ExecutorService и ForkedJoinedPool

Какво представлява ExecutorService?

ExecutorService менажира собствено ръчно нишки.Чрез него може да регулираме какво ниво на паралелизъм искаме да има най-често според тестовата машина или машината на която се изпълнява кода..Могат да се задават определен брой нишки с които да работи, което спестява време от евентуално създаване на нишки които няма да се ползват, което не е бърза операция на операционната система.Пази ни от context-switching.ExecutorService преизползва нишки които са свършили вече своята работа.[7]



За всеки ExecutorService се използва имплементацията на ForkJoinPool [8]



ForkJoinPool е частна имплементация на ExecutorService. Дава начална точка от която да се изпълнят задачи. Работи на принципа “work-stealing”. Всички нишки се опитват да намерят и извършат публикува задача в ForkJoinPool пула или активна задача която други нишки са създали .Това позволява ефикасно изчисляване на под-задача

породена от задача,както и изчисление на малки задачи.

1.2.4.2.3. Решение чрез паралелни потоци: При използването на паралелни потоци, задачата се разбива на под-потоци като резултата от всеки под-поток се комбинира. и се дава на главния поток.Тази архитектура използва канали и филтри.При нея имаме поредица от подпрограми.При тази архитектура сумирането на резултатите се осъществяват в процеса на паралелната обработка.[9]

1.2.2.5. Оценки на натовареността при потребителска заявка.

Времето за отговор зависи от начина по който ще се пресмята на база на потребителските параметри.

1.2.2.6. Обосновка за избраното решение - избраното решение е комбинация от няколко решения на зададения проблем.Потребителя може да избере между различни решения подавайки потребителски параметри в зависимост от системата на която пуска приложението.

2. Проектиране

2.1. Описание на реализираните програмни единици.

Приложението е проектирано като е разделено на пакети. Всеки пакет съдържа специфична функционалност.

пакета “euler” съдържа помощни класове за изчисление

пакета “euler.factorial” съдържа класове, които по различен начин смятат факториел

пакета “euler.strategy” съдържа различна стратегия за изпълнението на програмата.

пакета “eulter.timer” съдържа Таймер за изчисление на времето което е отнело за пресмятане на Ойлеровото число

Пакет “euler”:

клас Application - има за цел да прочете потребителските данни и да стартира изчислението на база на тях.

private void run() - приема аргументи от потребителя и стартира чрез Таймер избраната от него стратегия за пресмятане на база на параметрите

private Timer createTimer(String mode, int numThreads, int precision, int numReps,String outputPath, String factMode) - създава таймер в който

генерира стратегия на базата на параметрите.

private boolean isOptionAvailable(CommandLine cmd, CmdOption optionCode) - проверка дали има съществуваща опция по код на опция “optionCode” в параметъра “cmd”.

protected Optional<CommandLine> parseCmdLineOfCode() - прочита и запазва подадени вход от конзолата от потребителя като връща обект от тип CommandLine. Ако са празни аргументите, връща празен CommandLine обект.

private String getValueForOption(CommandLine commandLine, CmdOption optionCode) - връща стойността на подаден параметър от потребителя на база на кода на опцията “optionCode” от “commandLine”

private Options getOptions() - Връща обект от тип Options, като се задават ключовете, на които потребителя може да зададе стойност

private enum CmdOption - генерира за всяка опция по 1 обект от тип CmdOption, който представлява дадена команда.Инициализира всяка опция с подразбираща се стойност.

public static void main(String[] args) - подадените параметри от потребителя се четат от класа “Application” който е входна точка на програмата

клас EulerFileWriter - записва резултата от изчислението в изходящ файл.

public static boolean writeToFile(BigDecimal eulerNumber, String path) - записва в файл който се намира в променливата “path”, резултата от изчислението “eulerNumber”. Връща истина при успех и лъжа при неуспех.

клас EulerStrategy- абстрактен клас който връща стратегия за изчисление

public static EulerStrategy parallelCachedFactStrategy(int threads,FactorialImpl factorialCache) - използва SharedForkJoinPool, със стратегия кеширане на факториел и задава брой нишки чрез параметъра “numThreads”. Всяка подадена задача на SharedForkJoinPool е паралелно изчисляване на дробите с кеширания факториел.

public static AbstractStrategy parallel(int numThreads) - използва SharedForkJoinPool, задава брой нишки чрез параметъра “numThreads” Всяка подадена задача на SharedForkJoinPool е паралелно изчисляване на дробите с изчисляване на факториел чрез паралелни потоци

public static AbstractStrategy parallelDC(int numThreads) -- връща SharedForkJoinPool стратегия с паралелно изчисление на всеки факториел чрез стратегия разделяй и владей и задава брой нишки чрез параметъра “numThreads”. Всяка подадена задача на ForkedJoin Executor-а е паралелно изчисляване на дробите с изчисляване на факториел чрез разделяй и владей стратегия

public static AbstractStrategy single() - връща стратегия с използването на 1 нишка, паралелни потоци изчисляват дробите и изчисляването на факториел се използват паралелни потоци

public static AbstractStrategy singleCached(FactorialImpl factorialCache) - връща стратегия с използването на 1 нишка, паралелни потоци изчисляват дробите и изчисляването на факториеле чрез предварително пресмятане

public static AbstractStrategy singleDC() - връща стратегия с използването на 1 нишка, паралелни потоци

изчисляват дробите и изчисляването на факториела чрез divide and conquer

public static AbstractStrategy commonForkJoin() - връща стратегия за използването на паралелни потоци като всеки изчислява факториела чрез паралелни потоци

public static AbstractStrategy sellfStrategy(int numThreads, FactorialImpl factorialCache) - връща стратегия за стартиране на собствено написани нишки като техния брой е “numThreads” , kato всяка изчислява факториел чрез кеширане.

клас MathUtils

public static BigInteger factorial(long k) - изчислява паралелно факториел за даден параметър “k”, като се използват паралелни потоци и връща резултат полученото число.

protected static int aprox(int precision) - Връща отговор на въпроса - ако е дадена точност след десетичната запетая за изчислението на Ойлеровата константа, колко дроби трябва да бъдат събрани минимално така че след като се съберат, знакът в резултата на позиция “precision” да съществува и независимо колко дроби се добавят той никога да не се промени.Решението може да се илюстрира:

Suppose that we define

$$a_n=\frac{2n+1}{(2n)!}\implies \frac{a_{n+1}}{a_n}=\frac{2n+3}{2(n+1)(2n+1)^2}$$

Writing

$$\sum_{n=0}^{\infty}a_n=\sum_{n=0}^{p}a_n+\sum_{n=p+1}^{\infty}a_n=\sum_{n=0}^{p}a_n+R_p$$

and *suppose* that we can *approximate*

$$R_p\sim a_p\sum_{n=p+1}^{\infty}\frac{2n+3}{2(n+1)(2n+1)^2}$$

So,

$$R_p=\frac{2p+1}{2(2p)!}\left(\psi^{(0)}\left(p+\frac{3}{2}\right)-\psi^{(0)}(p+2)+\psi^{(1)}\left(p+\frac{3}{2}\right)\right)$$

Taking logarithms and using very early truncated expressions,

$$\ln(R_p)\sim-\ln\left(4\sqrt{\pi}\right)-2p(\ln(p)-1+\ln(2))$$

and we want $R_p<10^{-k}$. This would give,as an *approximation*,

$$p=\frac{A}{2W\left(\frac{A}{e}\right)}\qquad\text{where}\qquad A=\ln\left(\frac{10^k}{4\sqrt{\pi}}\right)$$

$W(.)$ being Lambert function. For sure, you will need to use $\lceil p\rceil$.

В функцията е направена допълнителна оптимизация при изчисляването на константата А като се използва формулата ln(a/b) = ln(a) - ln(b)

Клас SharedForkJoinPool

synchronized static ExecutorService getExecutorService(int totalThreads) - връща обект от тип SharedForkJoinPoolInner със брой нишки равен на параметъра “totalThreads”.

private static class SHFJP - помощен клас за създаването на ForkJoinPool.

private SharedForkJoinPoolInner(int totalThreads) - инициализа свойството “executorService” от тип

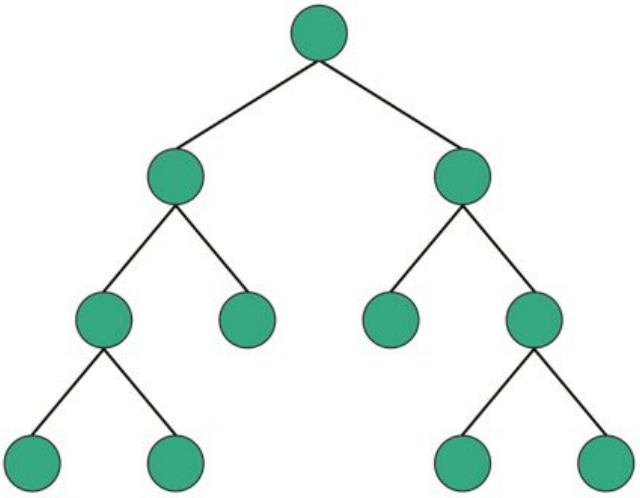
ForkJoinPool с б р о й н и ш к и р а в е н н а п а р а м е т ъ р а “totalThreads”

П а к е т euler.factorial:

к л а с DivideAndConquerFactorialSupplier

public BigInteger get(long k) - в р ъ щ а k!

public static BigInteger multiplyInRange(long a, long b, int split) - и з ч и с л я в а у м н о ж е н и е т о н а ч и с л а о т “a” д о “b” в к л ю ч и т е л н о.И д е я т а е у м н о ж е н и е т о д а с е р а з б и е н а п о д з а д а ч и к о и т о е д н а с д р у г а д а б ъ д а т у м н о ж е н и д о к а т о н е с е с т и г н е д о р а з л и к а м е ж д у п а р а м е т р и т е a-b<2, т о е с т е п р е к а л е н о м а л к а и л и п ъ к в е ч е н е м о ж е д а с е р а з д е л и у м н о ж е н и е т о н а п о д-у м н о ж е н и я.Ф у н к ц и я т а м о ж е д а б ъ д е п р е д с т а в е н а к а т о д ъ р в о :



Р е з у л т а т и т е с е к о м б и н и р а т о т г о р е н а д о л у и к о г а т о с е с т и г н е в ъ р х а п р и к л ю ч в а у м н о ж е н и е т о, т о е с т е и м п л е м е н т и р а н а р е к у р с и в н о.Р а з д е л я н е т о н а е д и н и н т е р в а л н а 2 п о д и н т е р в а л а з а п р е с м я т а н е н а з а д а ч и ч р е з н а м и р а н е н а с р е д а т а н а a и b.

private static BigInteger fastMultiply(long a, long b) - п о м о щ н а ф у н к ц и я з а public static BigInteger multiplyInRange(int split, long a, int b) .П р и б л и з к и с т о й н о с т и с е в р ъ щ а к а т о р е з у л т а т у м н о ж е н и е т о о т a д о (b-1) в к л ю ч и т е л н о.П а р а м е т ъ р а “split” с е и з п о л з в а з а и з ч и с л я в а н е н а д ъ л б о ч и н а т а н а д ъ р в о т о.

клас DummyFactorialImpl

public BigInteger get(long k) - връща резултата от k!, като използва паралелно изчисление чрез потоци

Клас CacheFact

private CacheFact(int k) - създава кеш който е от тип Map<Long, BigInteger> и го попълва с факториели до к-тия факториел

private void createCache(int k) - попълва кеша с факториели до к-тия факториел

public BigInteger get(long k) - взима факториел от кеша на базата на стойността на параметъра “к”

public static CacheFact of(int precision) - създава кеш с размер 2 * precision ;

public interface FactorialImpl - Интерфейс който всички кеш класове в пакета euler.factorial трябва да имплементират.

клас FastFactorialImpl

public FastFactorialImpl(ExecutorService executorService) - задава ExecutorService, чрез който ще се пресметне паралелно факториел.Всяка задача е паралелно пресмятане

public BigInteger get(long k) - взима к-тия факториел използвайки паралелно пресмятане чрез ExecutorService

пакет euler.strategy:

AbstractStrategy - абстрактен клас който всяка стратегия от euler.strategy наследява.

public BigDecimal compute(int precision) -- пресмята Ойлеровата константа чрез използвана стратегия и параметъра “precision”

static BigDecimal computeMember(int n, int scale, FactorialImpl FactorialImpl) - подразбираща се имплементация за изчисляване на дроб от числото на Ойлер.”n” определя кой номер на дроб ще се пресметне, “scale” каква прецизност, “FactorialImpl” - по какъв начин ще се сметне факториела.

ForkJoinStrategy - изчисляване чрез ForkJoinedPool

public ForkJoinStrategy(FactorialImpl FactorialImpl) - задава как ще се изчислява факториела

protected BigDecimal strategy(int precision) - имплементира изчислението на базата на “Precision”, като пресмята итерациите първо и започва да калкулира

public String toString() - съобщение за типа изчисление

клас LambertW - изчислява функцията на Ламберт

static public double branchNeg1(double x) - изчислява функцията на Ламбер за параметър “x”.

клас ParallelComputation - паралелна стратегия за изчисляване на Ойлеровата константа

public ParallelComputation(int numThreads, FactorialImpl fact) - задава брой нишки които ще се използват чрез параметъра “numThreads” както и какъв тип пресмята на факториела чрез “fact”. Използва SharedForkJoinPool в който публикува задачите и всяка задача паралелно изчислява дробите чрез паралелни потоци

protected BigDecimal strategy(int precision) - инициализира пресмятането Ойлеровата константа чрез използвана стратегия и параметъра “precision” . Връща намерената константа

protected Callable<BigDecimal> computeEuler(int precision) - пресмята паралелно чрез потоци Ойлеровата константа

public String toString() - съобщение за типа изчисление

клас OnEulerThread

public OnEulerThread(FactorialImpl fact) - задава как ще се изчислява факториела

protected BigDecimal strategy(int precision) - стартира изчислението на Ойлеровата константа.Използва цикъл за пресмятането на всяка дроб

public String toString() - съобщение за типа изчисление

клас CustomThreadStrategy

public CustomThreadStrategy(int numThreads, FactorialImpl FactorialImpl) - з а д а в а к а к щ е с е и з ч и с л я в а ф а к т о р и е л а

к а к т о и б р о я н а н и ш к и к о и т о т р я б в а д а б ъ д а т с т а р т и р а н и .

protected BigDecimal strategy(int precision) - с т р а т и р а о п р е д е л е н б р о й н и ш к и н а б а з а н а к о н с т р у к т о р а CustomThreadStrategy(int numThreads, FactorialImpl fact) и п р е с м я т а ц я л о с т н а т а с у м а о т с у м а т а н а в с я к а н и ш к а

к л а с EulerThread - р ъ ч н о с ъ з д а д е н а н и ш к а к о я т о с т а р т и р а м е

public EulerThread(int index, int numThreads, int totalIterations, int totalSymbols,FactorialImpl fact) - “index” з а д а в а к о й index ч л е н о т с у м а т а щ е п р е с м я т а , numThreads - ц я л о с т н и я б р о й н а в с и ч к и EulerThread к о и т о щ е с е с ъ з д а д а т , “totalIterations” б р о й ч л е н о в е д а с е п р е с м е т н а т . “totalSymbols” т о ч н о с т н а п р е с м я т а н е т о . “ FactorialImpl” - к а к щ е с е п р е с м я т а ф а к т о р и е л а

public void run() - м е т о д к о й т о и з ч и с л я в а ч и с л о т о “ e ” . П р и с т а р т и р а н е з а п о ч в а о т б р о я в а н е н а в р е м е т о к о е т о е о т н е л о н а Thread-a д а и з ч и с л и ч а с т о т ч и с л о т о “ e ” . З а п а з в а с е т е к у щ р е з у л т в п р о м е н л и в а т а “currentResult”. В з а в и с и м о с т о т н о м е р а н а Thread-a и с е з а д а в а к о я д р о б д а и з ч и с л и ч р е з и з р а з а

if (i % NUMBER_OF_THREADS != index) . П р е с м я т а с е п о м о д у л о т б р о я Threads. Т а к а р а в н о м е р н о с е р а з п р е д е л я р а б о т а т а . Н и ш к а т а п р и к л ю ч в а р а б о т а т а к о г а т о и н д е к с а н а т е к у щ а т а д р о б е п о - г о л я м о т ц я л о с т н и я б р о й д р о б и к о и т о с а н у ж н и .

п а к е т euler.Timer

Timer - П р е с м я т а в р е м е т о з а к о е т о е п р е с м е т н а т а О й л е р о в а т а к о н с т а н т а

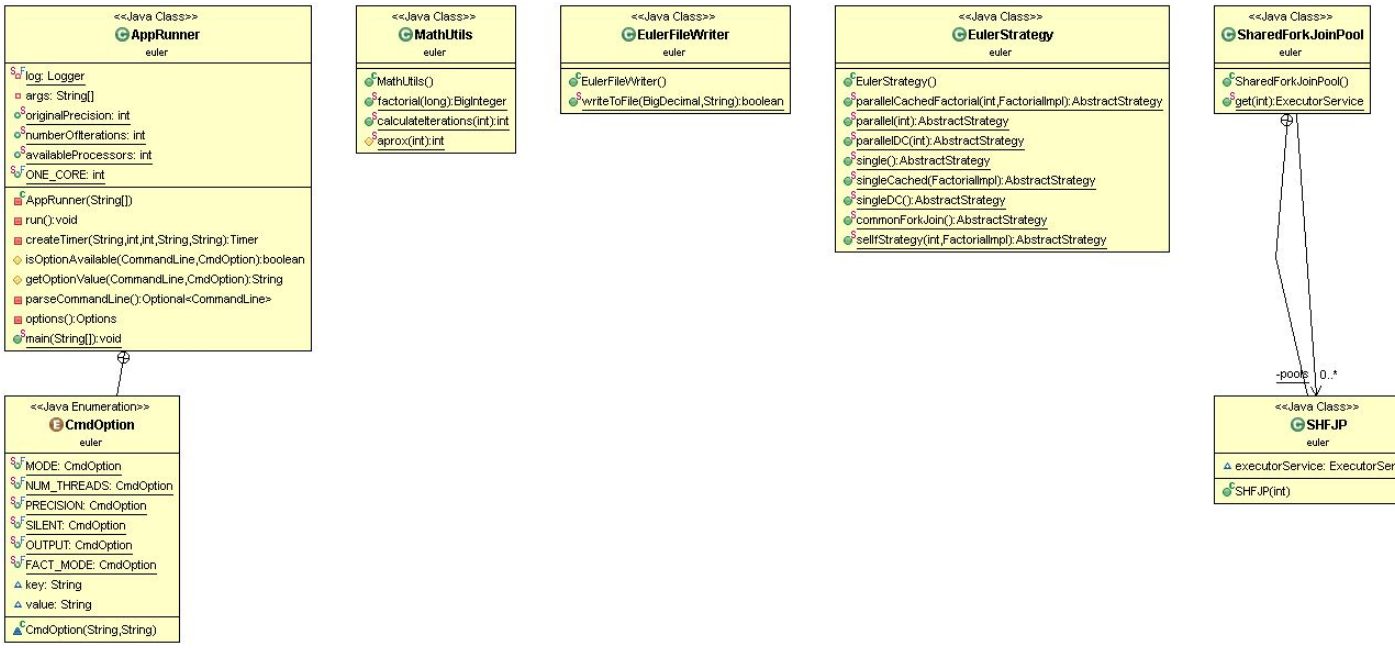
public Timer(int totalRetries, Runnable taskRunnable) - з а д а в а б р о я н а п о в т о р е н и я ч р е з “totalRetries” , к а к т о и з а д а ч а т а к о я т о т р я б в а д а с е и з п ъ л н и - “taskRunnable”

protected List<Long> time(boolean isSilent) - п р е с м я т а в р е м е т о з а к о е т о с е е и з п ъ л н и л о и з ч и с л я в а н е т о н а О й л е р о в а т а к о н с т а н т а , и з п о л з в а й к и к л а с ъ т StopWatch к а т о с е с т а р т и р а о т м е р в а н е т о п р е д и д а з а п о ч н е и с е с п и р а

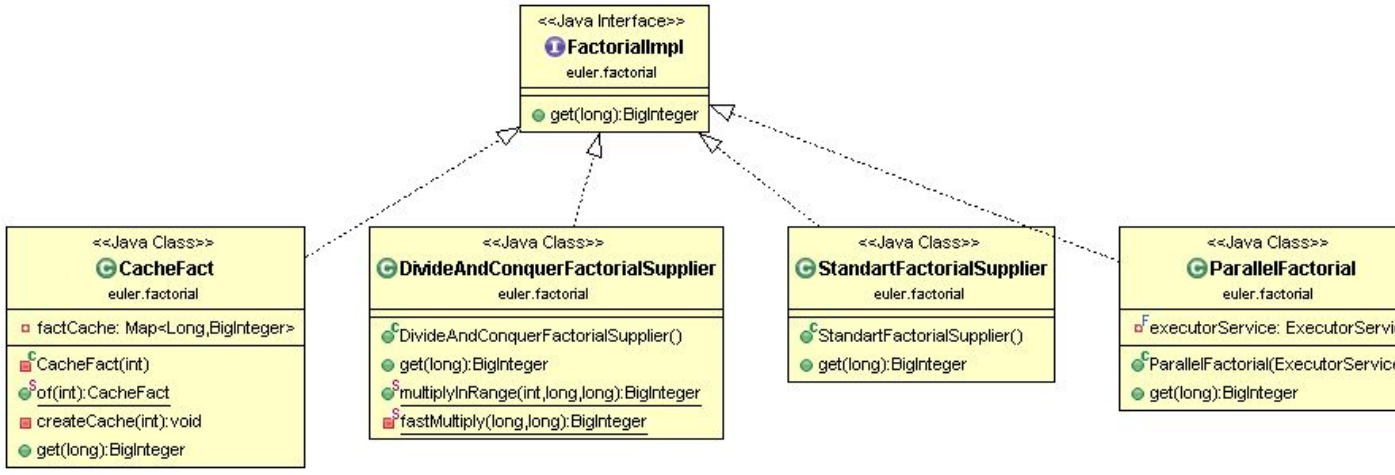
когато е приключило изчислението.В зависимост от параметъра “isSilent”, ако е истина, се показва времето за изчисление само, както и типа на избор за изчисление

2.2. Клас диаграми

2.2.1. Класовете имплементиращи пресмятане на факториел:

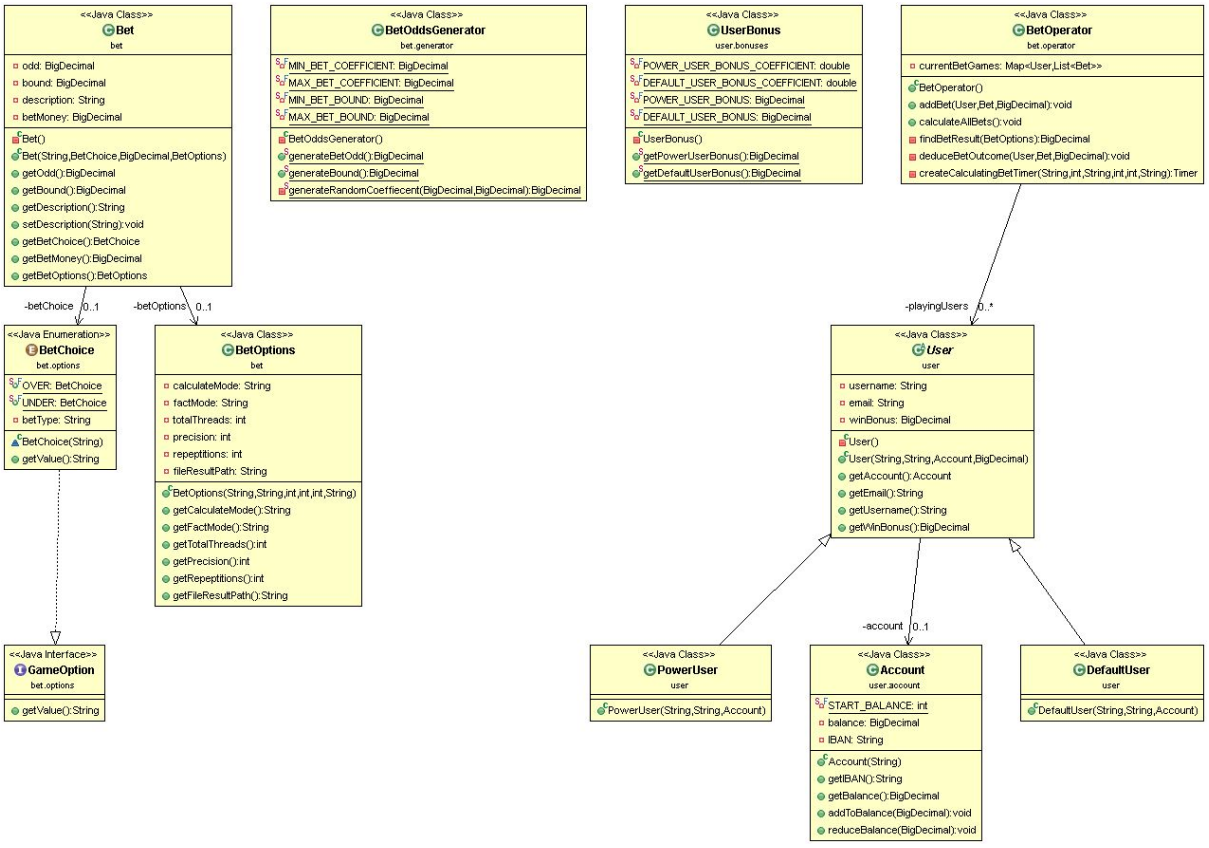


2.2.2. Класовете имплементиращи стратегия:



2.2.3. Класовете стартиращи и менажиращи приложението

2.4. Класове отговорни за залаганията



2.3. Входни параметри

Param Name	Possible values	Default value
-t	greater than zero values	16
-p	greater than zero values	1000
-mode	single,parallel,self,shared	parallel
-q	None	false
-output	Any file name	eulerResult.tmp
-fact_mode	cached dc	none

2.4. Изисквания за стартиране на проекта:

Нужно е да бъде инсталирана Java 11, както и поддръжка na multithreading за паралелни решения.

2.5. Стартиране на приложението

Примерно стартиране от конзолата чрез Windows е:

java -cp .\62125.jar euler.AppRunner

3. Моделиране

1. Решението изградено със създадени нишки които наследяват java класа Thread.java има SPMD архитектура с потокова архитектура за изчисляването на всеки факториел. Балансирането е статично за разпределянето на работата на нишките тъй като всяка нишка взима равен брой задачи по брой и сложност

2. Решение чрез паралелни потоци - MPMD архитектура

3. Решение чрез ForkJoinPool който има динамично постъпване на заданията.

Към всяко от решенията има възможност за избор на начин по който да се сметне факториела. Начините са чрез паралелни потоци или предварително кеширане.

4. Тестване

4.1. Тестови системи

Система 1:

- Канали на паметта: 2
- Видове памет: DDR4
- Набор на команди: 64 bit
- Предлага L3 кеш капацитет: 32 MB
- Предлага L2 кеш капацитет: 3 MB
- Цокъл на процесора: AM4
- Брой ядра: 6
- Име на ядро: Zen
- Вътрешна тактова честота: 3.60 GHz
- Turbo Clock Rate: 4.20 GHz
- total threads - 12
- Процесор: AMD Ryzen 5 3600

Система 2:

Модел: Ryzen 7 3700x 8-Core 3.6GHz AM4

<u>Основни характеристики:</u>

- Цокъл на процесора: AM4
- Работна честота: 3.6 GHz

- Физически ядра: 8
- Total L2 Cache 4MB
- Логически ядра: 16
- Кеш памет: 36 MB

4.2. Тестови сценарии

Тестовите сценарии се намират в пакета “test.tests” на проекта. Съдържа тестове за всеки тип стратегия, а за стратегията използваща ForkJoinPool има тестове за всеки 2к нишки като “к” е в интервала [1;16]. Всеки един тест проверява коректността на резултата за 100,1000 и 10,000 цифри.

4.2.1. Скорост на пресмятане на горната граница на итерациите - скоростта за пресмятане на граница до къде трябва да се пресмятат дробите е 16 милисекунди за вход прецизност 1,000,000. Това ограничение дава като примерен резултат за прецизност 10,000 отговор 1300.

4.3. Тестови резултати:

T1 - времето за изпълнение на серийната програма (използваща 1 нишка).

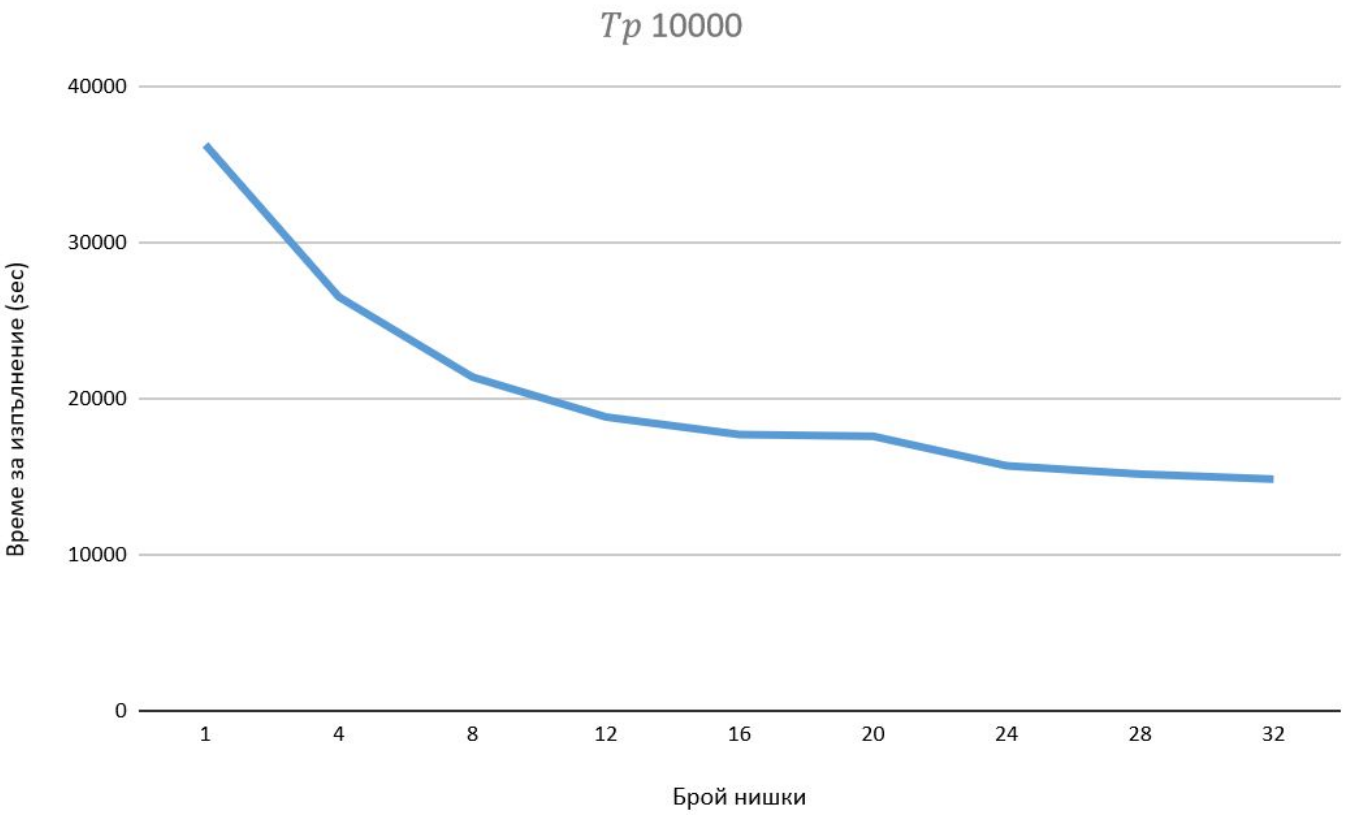
Tp - време за изпълнение на паралелната програма, използващ p нишки

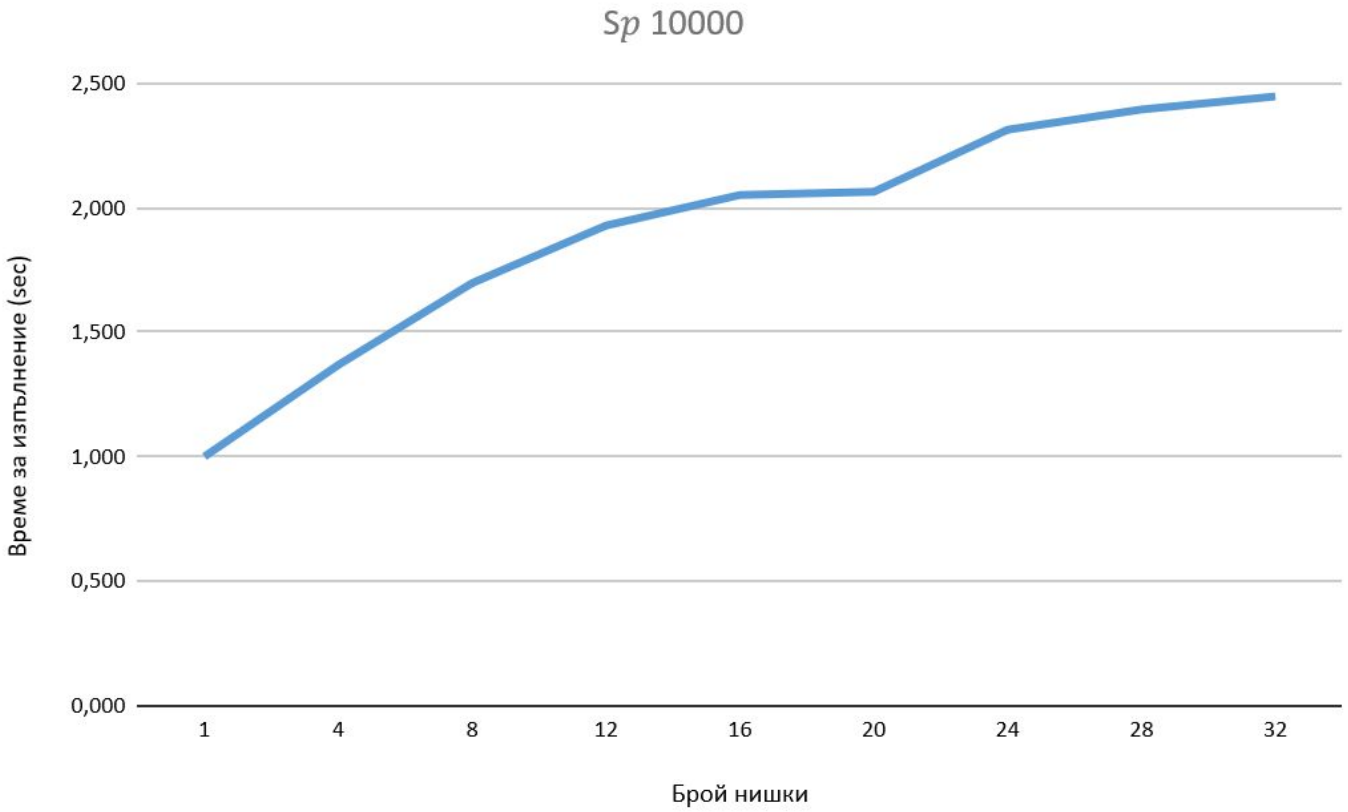
Sp = T1/Tp - ускорението, което програмата има при използването на p нишки.

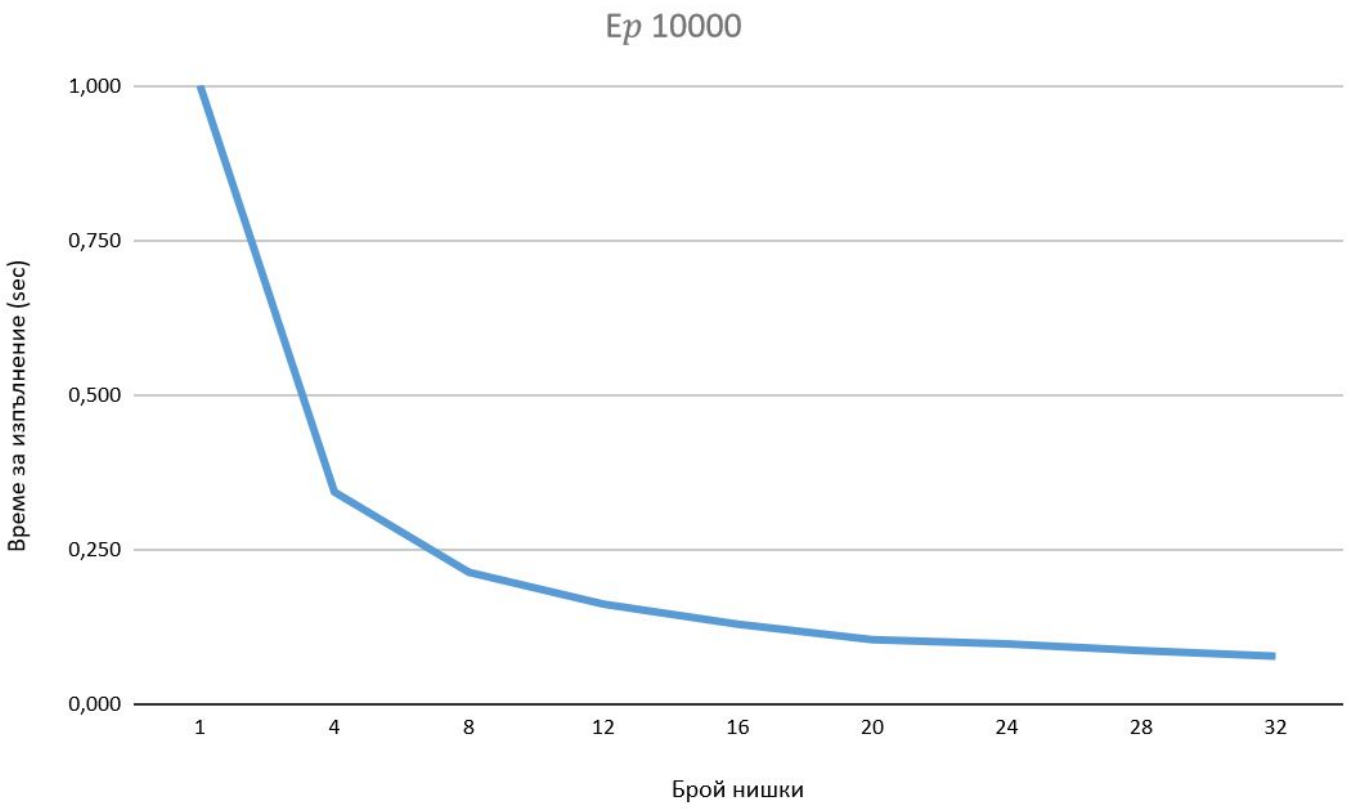
Ep = Sp/p - ефективността на програмата при използване на p нишки.

Резултати при Система 1:

1. Паралелно с ExecutorService, без оптимизация за итерации и факториел изчисляващ чрез паралелни потоци. Резултати за прецизност 10,000:

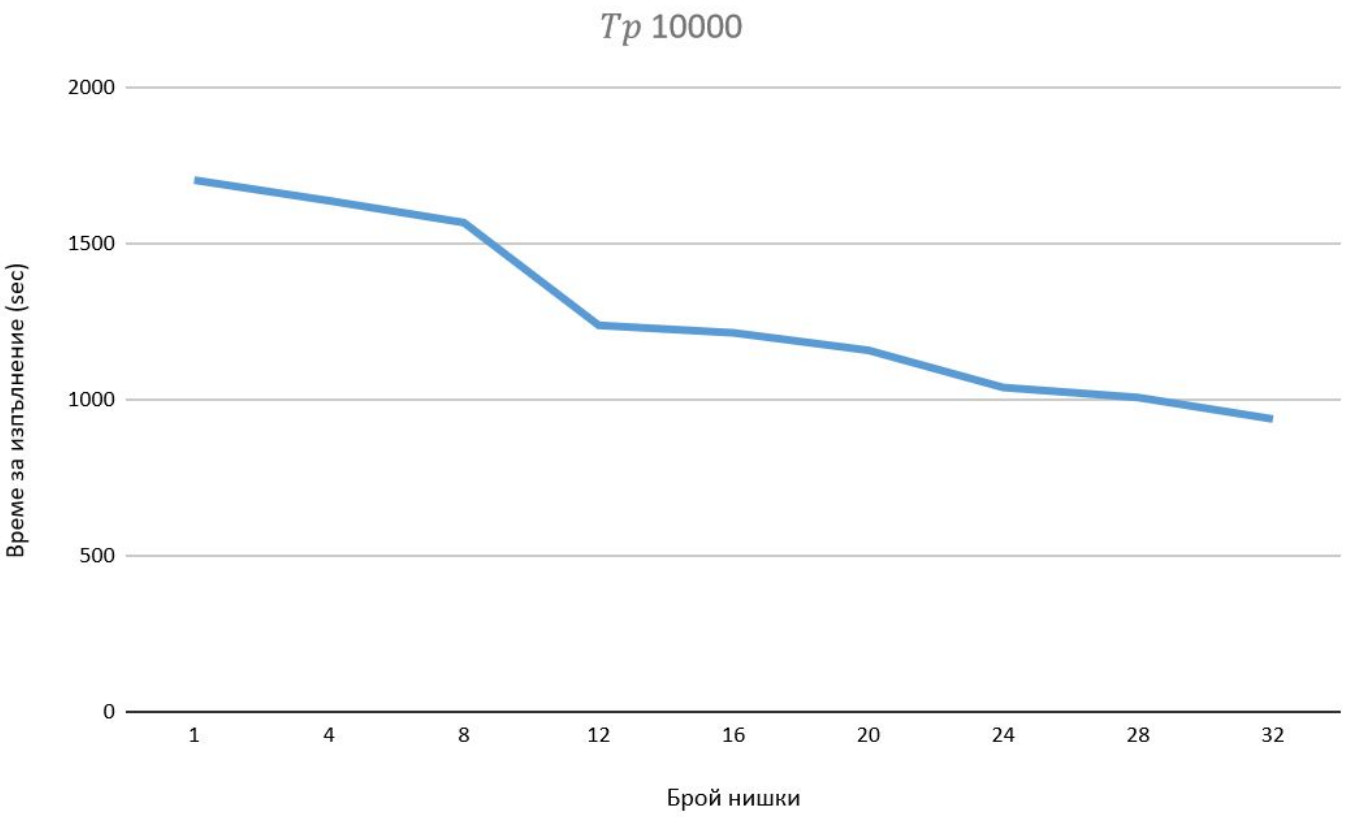

Tp 10000
40000
30000
20000
10000
0
Време за изпълнение (sec)
1
4
8
12
16
20
24
28
32
Брой нишки

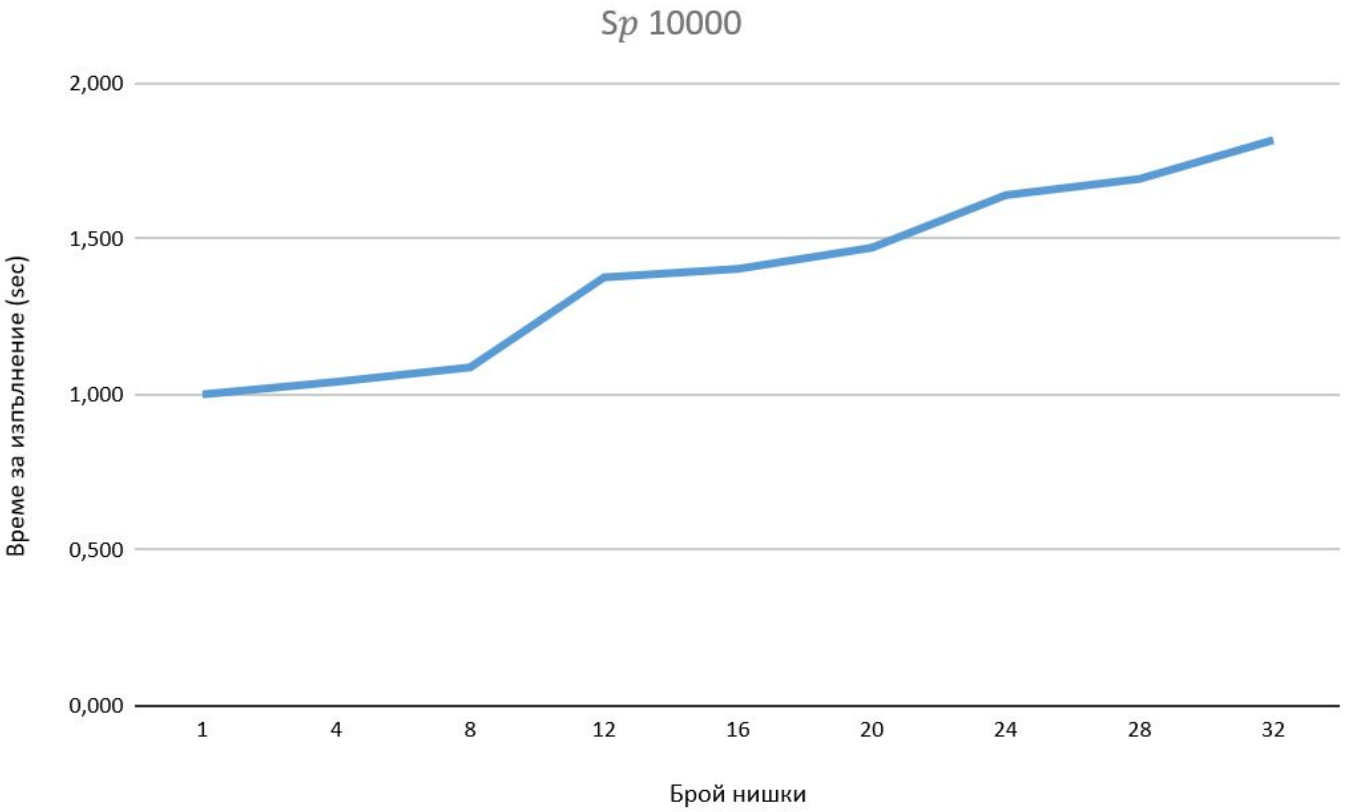

Sp 10000
2,500
2,000
1,500
1,000
0,500
0,000
Време за изпълнение (sec)
1
4
8
12
16
20
24
28
32
Брой нишки

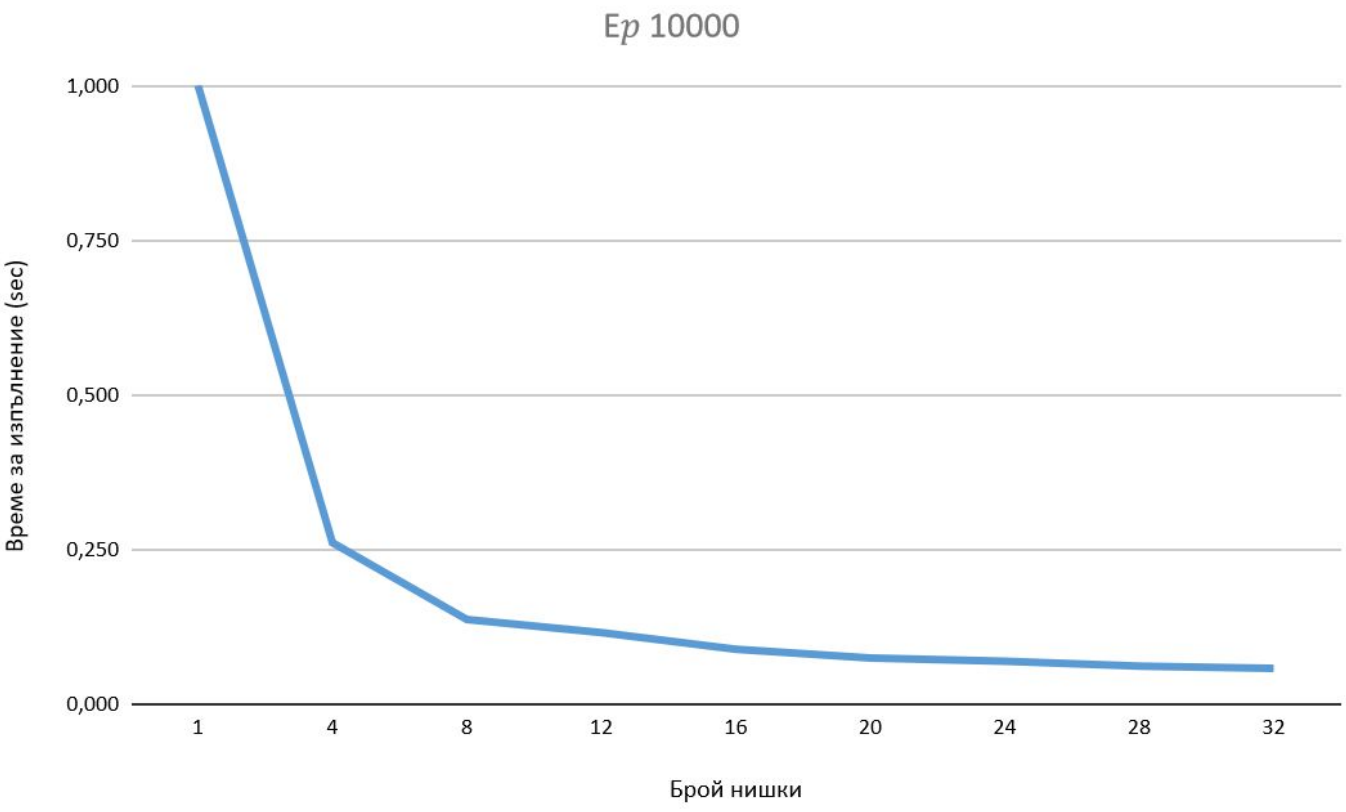

Ep 10000
1,000
0,750
0,500
0,250
0,000
Време за изпълнение (sec)
1
4
8
12
16
20
24
28
32
Брой нишки

Precision	Number of threads	Time	Speed-up	Efficiency
100	1	78	1,000	1,000
100	4	75	1,040	0,260
100	8	71	1,099	0,137
100	12	67	1,164	0,097
100	16	63	1,238	0,077
100	20	61	1,279	0,064
100	24	58	1,345	0,056
100	28	56	1,393	0,050
100	32	51	1,529	0,048
1000	1	567	1,000	1,000
1000	4	563	1,007	0,252
1000	8	523	1,084	0,136
1000	12	478	1,186	0,099
1000	16	451	1,257	0,079
1000	20	424	1,337	0,067
1000	24	368	1,541	0,064
1000	28	341	1,663	0,059
1000	32	326	1,739	0,054
10000	1	9200	1,000	1,000
10000	4	8753	1,051	0,263
10000	8	7563	1,216	0,152
10000	12	7100	1,296	0,108
10000	16	6754	1,362	0,085
10000	20	6543	1,406	0,070
10000	24	5326	1,727	0,072
10000	28	4876	1,887	0,067
10000	32	4600	2,000	0,063
50000	1	141807	1,000	1,000
50000	4	104534	1,357	0,339
50000	8	93563	1,516	0,189
50000	12	78563	1,805	0,150
50000	16	61785	2,295	0,143
50000	20	60004	2,363	0,118
50000	24	58135	2,439	0,102
50000	28	57113	2,483	0,089
50000	32	49545	2,862	0,089

2.Паралелно с ExecutorService, с пресмятане на итерациите без изчисляване на факториел.

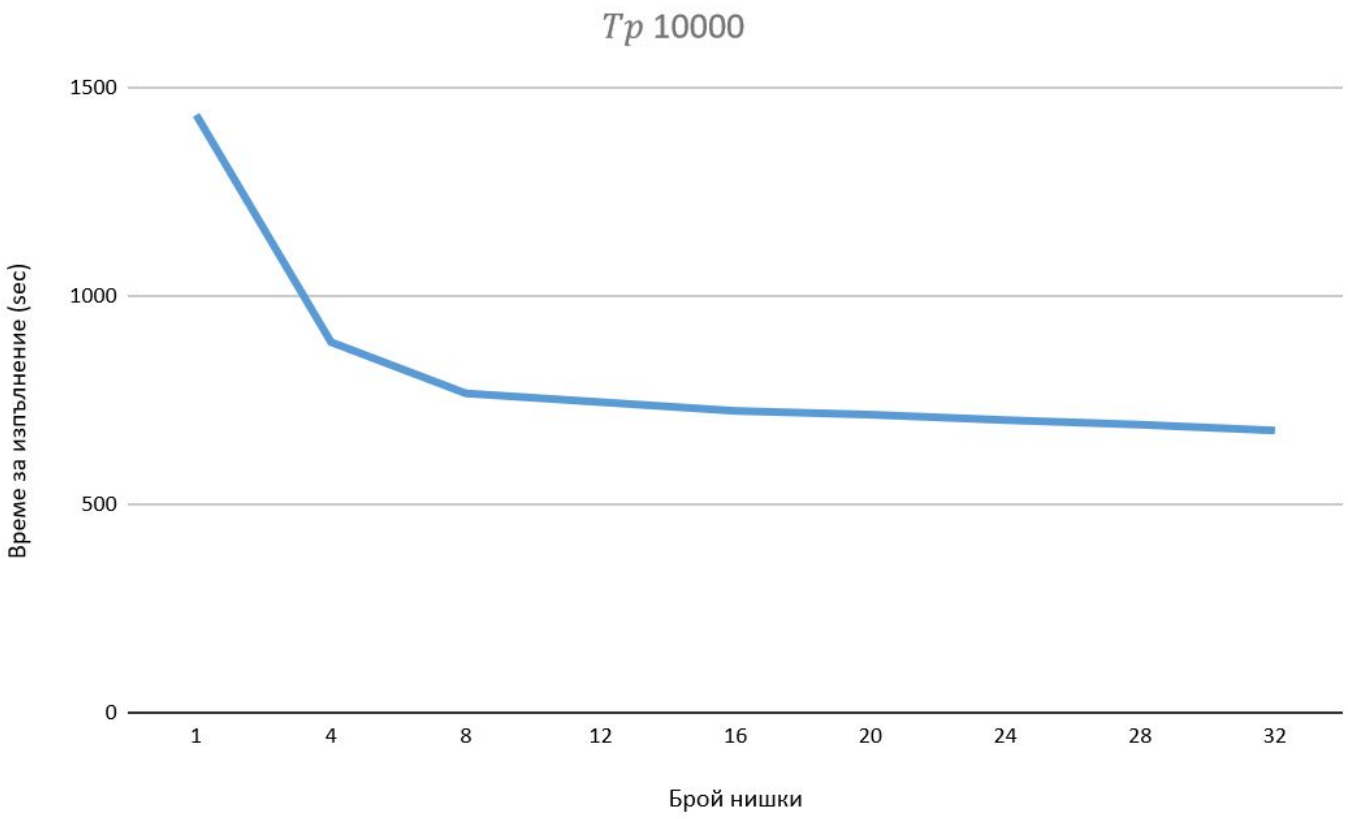

Tp 10000
2000
1500
1000
500
0
Време за изпълнение (sec)
1
4
8
12
16
20
24
28
32
Брой нишки

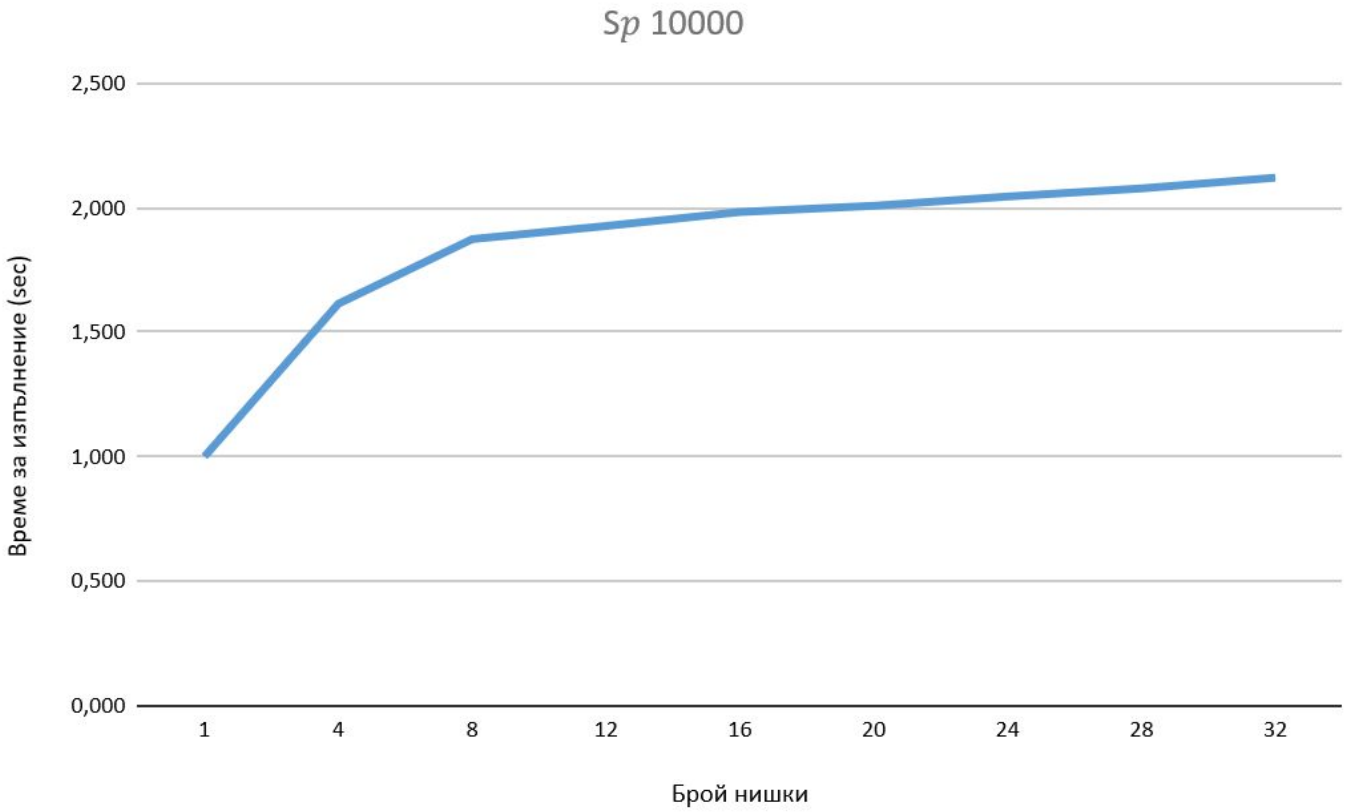

Sp 10000
2,000
1,500
1,000
0,500
0,000
Време за изпълнение (sec)
1
4
8
12
16
20
24
28
32
Брой нишки

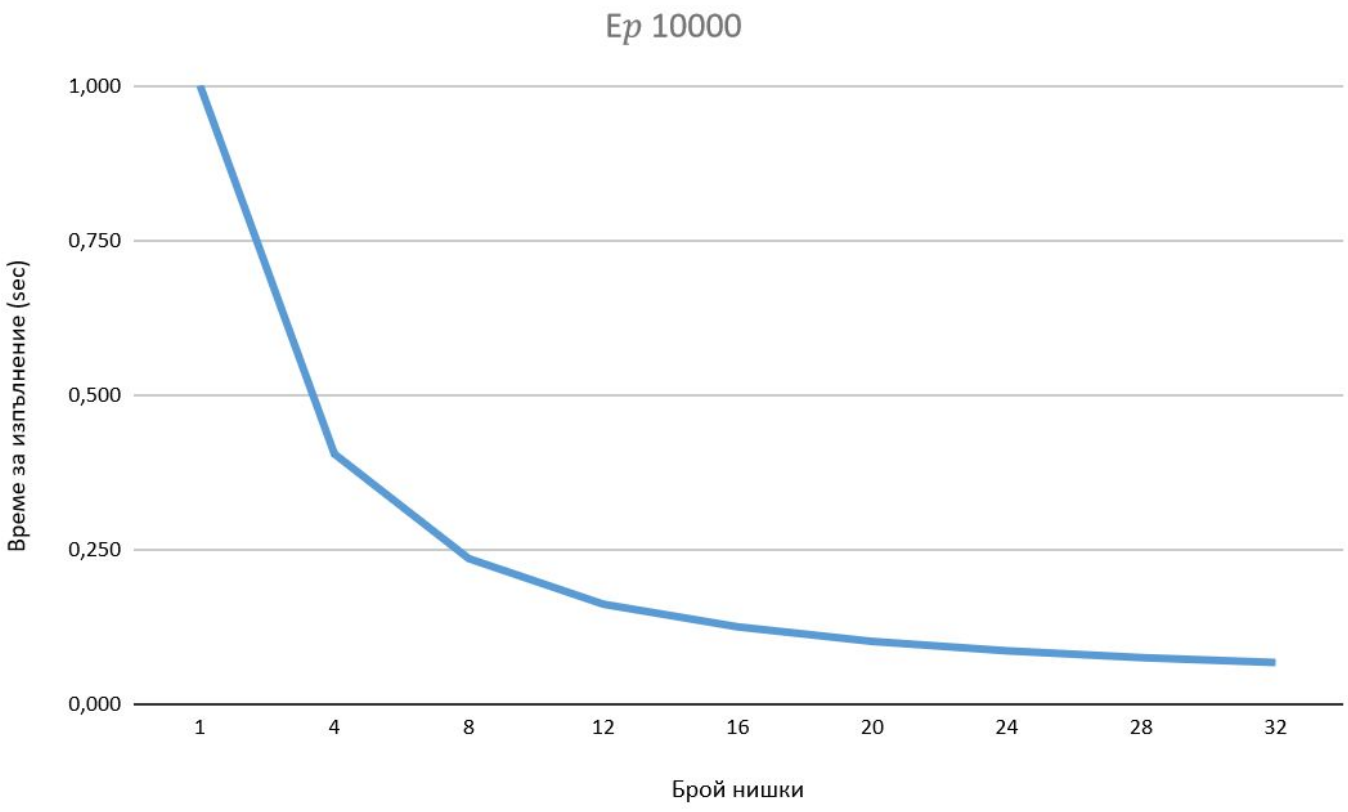

Ep 10000
1,000
0,750
0,500
0,250
0,000
Време за изпълнение (sec)
1
4
8
12
16
20
24
28
32
Брой нишки

Precision	Number of threads	Time	Speed up	Efficiency
100	1	43	1,000	1,000
100	4	42	1,024	0,256
100	8	42	1,024	0,128
100	12	42	1,024	0,085
100	16	41	1,049	0,066
100	20	41	1,049	0,052
100	24	41	1,049	0,044
100	28	39	1,103	0,039
100	32	36	1,194	0,037
1000	1	225	1,000	1,000
1000	4	214	1,051	0,263
1000	8	205	1,098	0,137
1000	12	185	1,216	0,101
1000	16	165	1,364	0,085
1000	20	144	1,563	0,078
1000	24	126	1,786	0,074
1000	28	113	1,991	0,071
1000	32	101	2,228	0,070
10000	1	1700	1,000	1,000
10000	4	1634	1,040	0,260
10000	8	1564	1,087	0,136
10000	12	1235	1,377	0,115
10000	16	1211	1,404	0,088
10000	20	1155	1,472	0,074
10000	24	1036	1,641	0,068
10000	28	1004	1,693	0,060
10000	32	935	1,818	0,057
50000	1	57643	1,000	1,000
50000	4	47574	1,212	0,303
50000	8	34267	1,682	0,210
50000	12	24646	2,339	0,195
50000	16	17853	3,229	0,202
50000	20	15211	3,790	0,189
50000	24	15116	3,813	0,159
50000	28	14888	3,872	0,138
50000	32	14793	3,897	0,122

3.Паралелно с ExecutorService, с пресмятане на броя на итерации,с паралелно изчисляване на дробите, оптимизиран с изчисляване на факториел чрез метода divide and conquer.

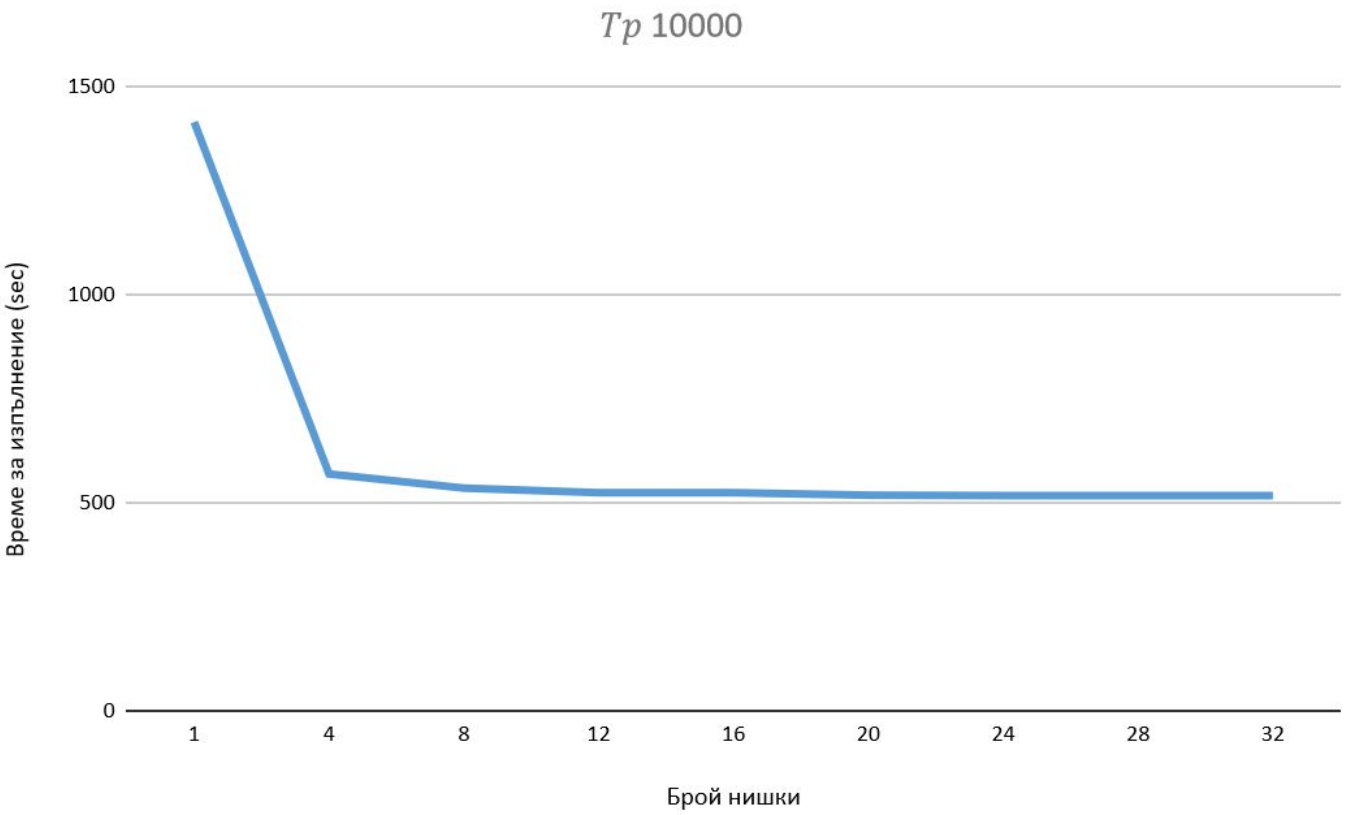


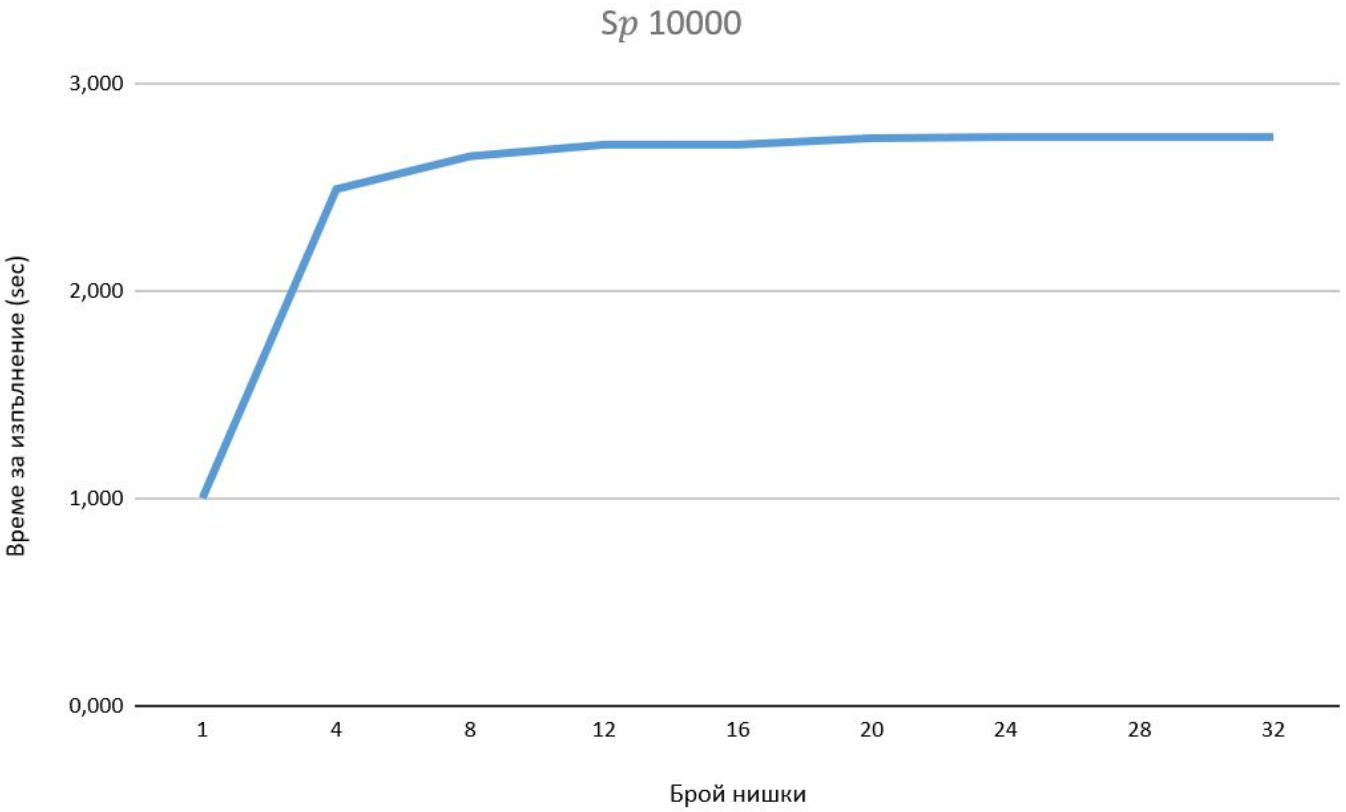

Sp 10000
2,500
2,000
1,500
1,000
0,500
0,000
Време за изпълнение (sec)
1
4
8
12
16
20
24
28
32
Брой нишки

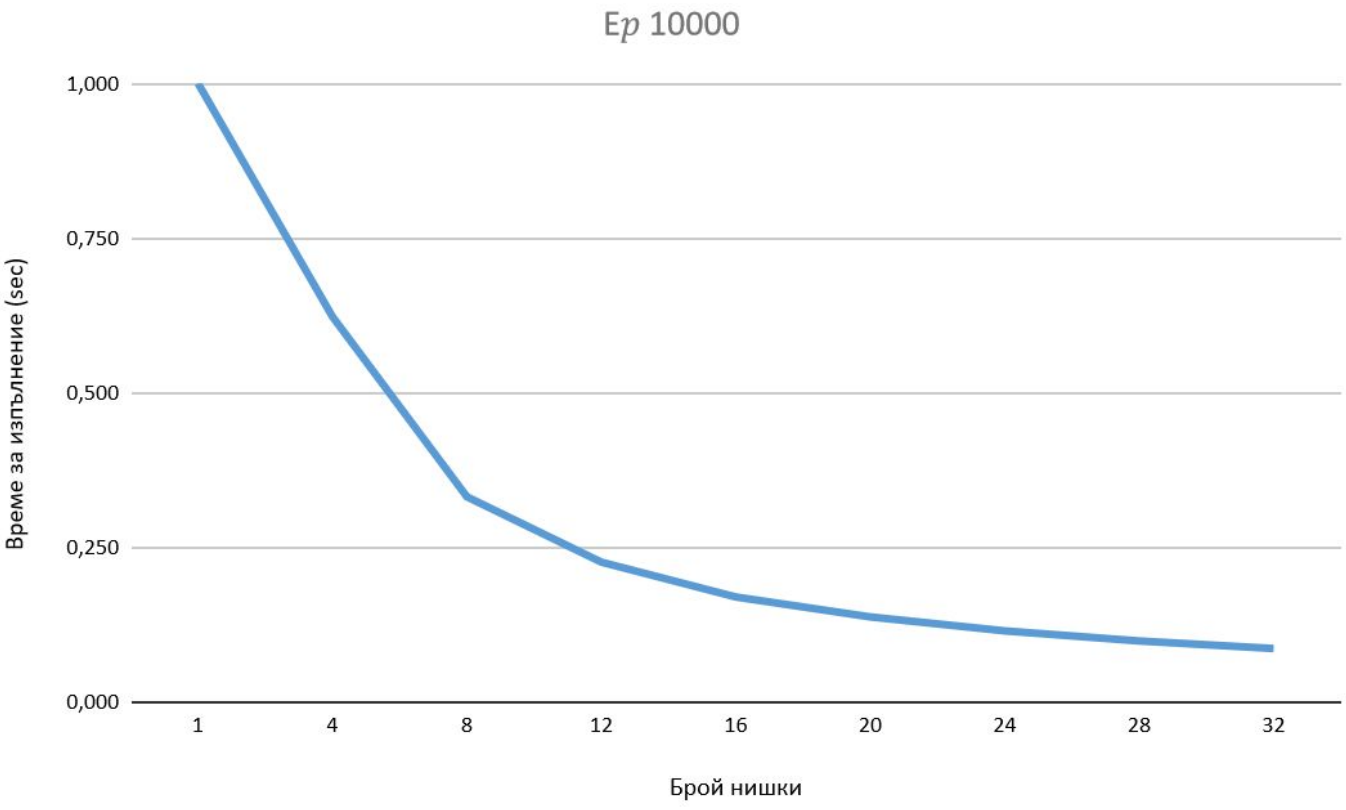

Ep 10000
1,000
0,750
0,500
0,250
0,000
Време за изпълнение (sec)
1
4
8
12
16
20
24
28
32
Брой нишки

Precision	Number of threads	Time	Speed up	Efficiency
100	1	32	1,000	1,000
100	4	32	1,000	0,250
100	8	32	1,000	0,125
100	12	32	1,000	0,083
100	16	32	1,000	0,063
100	20	31	1,032	0,052
100	24	31	1,032	0,043
100	28	29	1,103	0,039
100	32	29	1,103	0,034
1000	1	120	1,000	1,000
1000	4	104	1,154	0,288
1000	8	99	1,212	0,152
1000	12	95	1,263	0,105
1000	16	91	1,319	0,082
1000	20	91	1,319	0,066
1000	24	87	1,379	0,057
1000	28	83	1,446	0,052
1000	32	78	1,538	0,048
10000	1	1432	1,000	1,000
10000	4	887	1,614	0,404
10000	8	764	1,874	0,234
10000	12	743	1,927	0,161
10000	16	722	1,983	0,124
10000	20	713	2,008	0,100
10000	24	700	2,046	0,085
10000	28	689	2,078	0,074
10000	32	675	2,121	0,066
50000	1	33237	1,000	1,000
50000	4	27895	1,192	0,298
50000	8	24321	1,367	0,171
50000	12	23123	1,437	0,120
50000	16	21563	1,541	0,096
50000	20	18765	1,771	0,089
50000	24	16543	2,009	0,084
50000	28	15563	2,136	0,076
50000	32	11213	2,964	0,093

4.Паралелно с ExecutorService, с пресмятане на броя на итерации, с паралелно изчисляване на дробите ,оптимизиран с изчисляване на факториел чрез кеширане на нужните факториели.

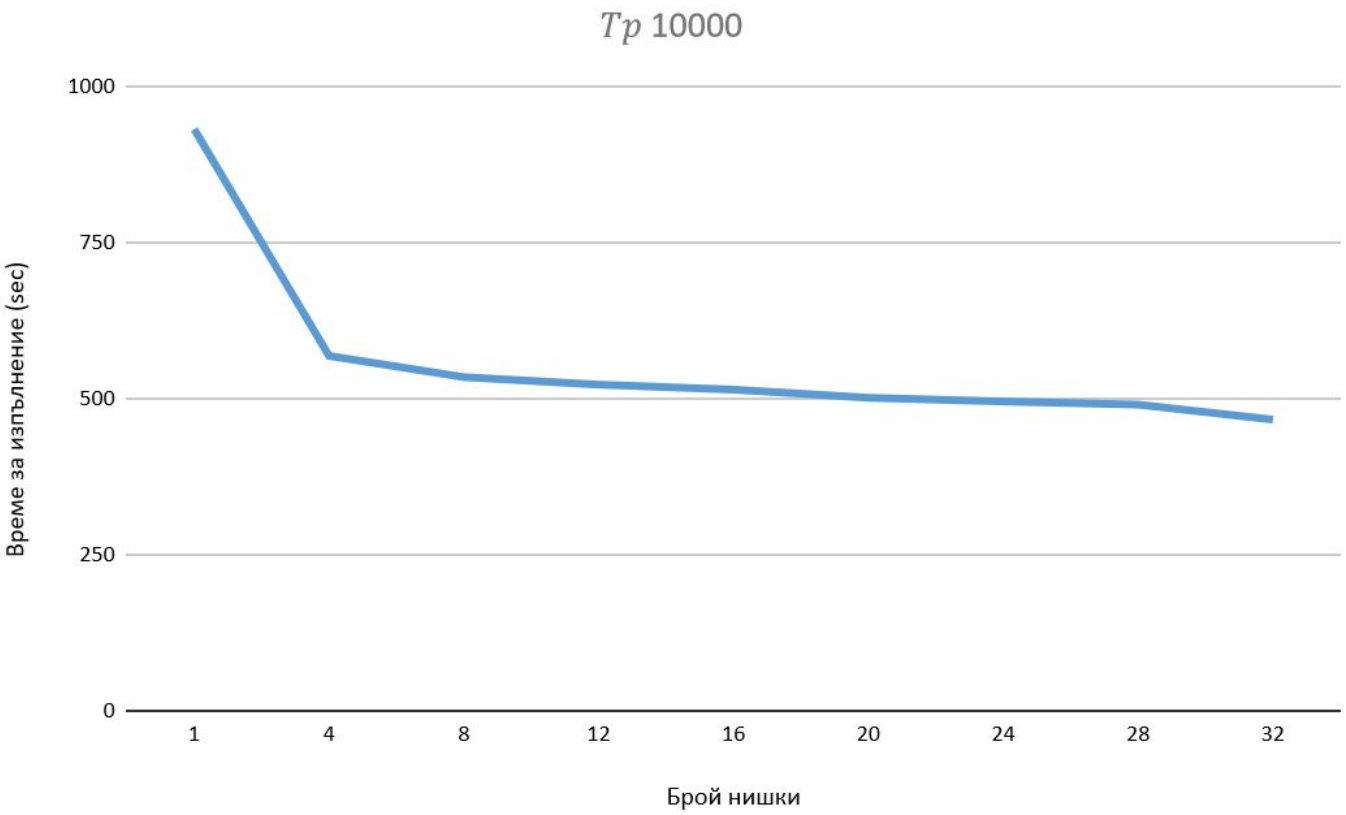

Tp 10000
1500
1000
500
0
Време за изпълнение (sec)
1
4
8
12
16
20
24
28
32
Брой нишки

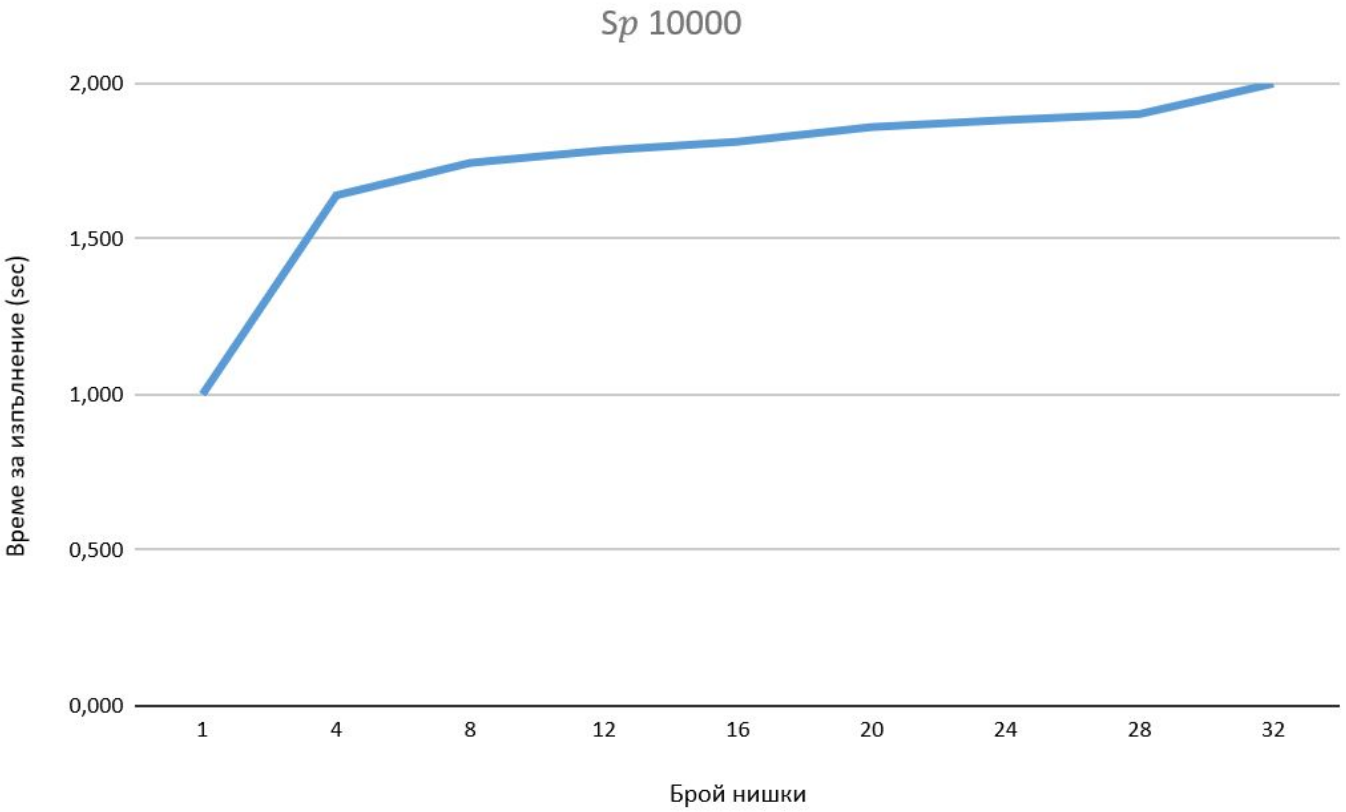

Sp 10000
3,000
2,000
1,000
0,000
Време за изпълнение (sec)
1
4
8
12
16
20
24
28
32
Брой нишки

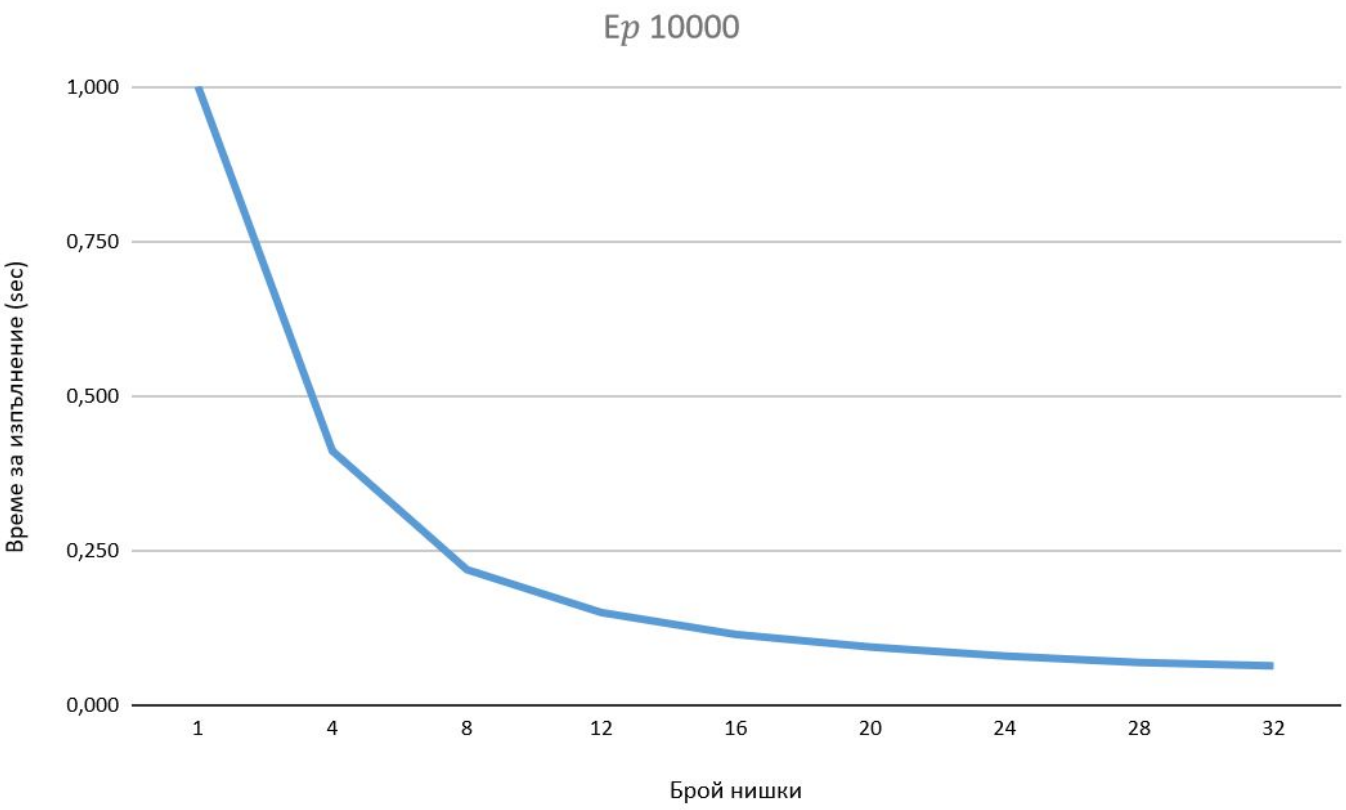

Ep 10000
1,000
0,750
0,500
0,250
0,000
Време за изпълнение (sec)
1
4
8
12
16
20
24
28
32
Брой нишки

Precision	Number of threads	Time	Speed up	Efficiency
100	1	34	1,000	1,000
100	4	33	1,030	0,258
100	8	33	1,030	0,129
100	12	33	1,030	0,086
100	16	32	1,063	0,066
100	20	32	1,063	0,053
100	24	32	1,063	0,044
100	28	32	1,063	0,038
100	32	29	1,172	0,037
1000	1	70	1,000	1,000
1000	4	64	1,094	0,273
1000	8	61	1,148	0,143
1000	12	54	1,296	0,108
1000	16	54	1,296	0,081
1000	20	54	1,296	0,065
1000	24	53	1,321	0,055
1000	28	53	1,321	0,047
1000	32	51	1,373	0,043
10000	1	1412	1,000	1,000
10000	4	567	2,490	0,623
10000	8	533	2,649	0,331
10000	12	522	2,705	0,225
10000	16	522	2,705	0,169
10000	20	516	2,736	0,137
10000	24	515	2,742	0,114
10000	28	515	2,742	0,098
10000	32	515	2,742	0,086
50000	1	25342	1,000	1,000
50000	4	24123	1,051	0,263
50000	8	22666	1,118	0,140
50000	12	20645	1,228	0,102
50000	16	18535	1,367	0,085
50000	20	15634	1,621	0,081
50000	24	14214	1,783	0,074
50000	28	11453	2,213	0,079
50000	32	9654	2,625	0,082

5.Паралелни потоци, които изчисляват дробите и всеки факториел се изчислява паралелно.

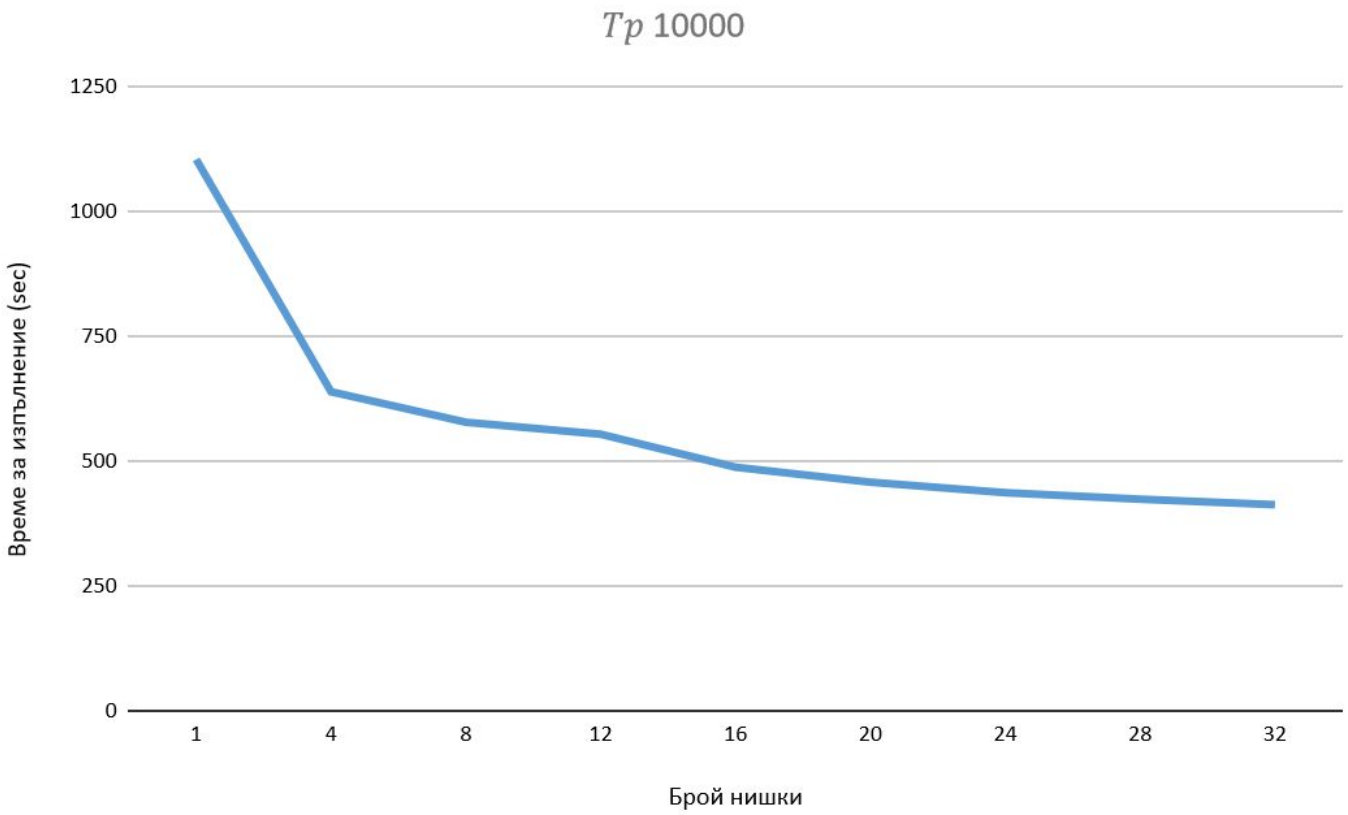

Tp 10000
1000
750
500
250
0
Време за изпълнение (sec)
1
4
8
12
16
20
24
28
32
Брой нишки

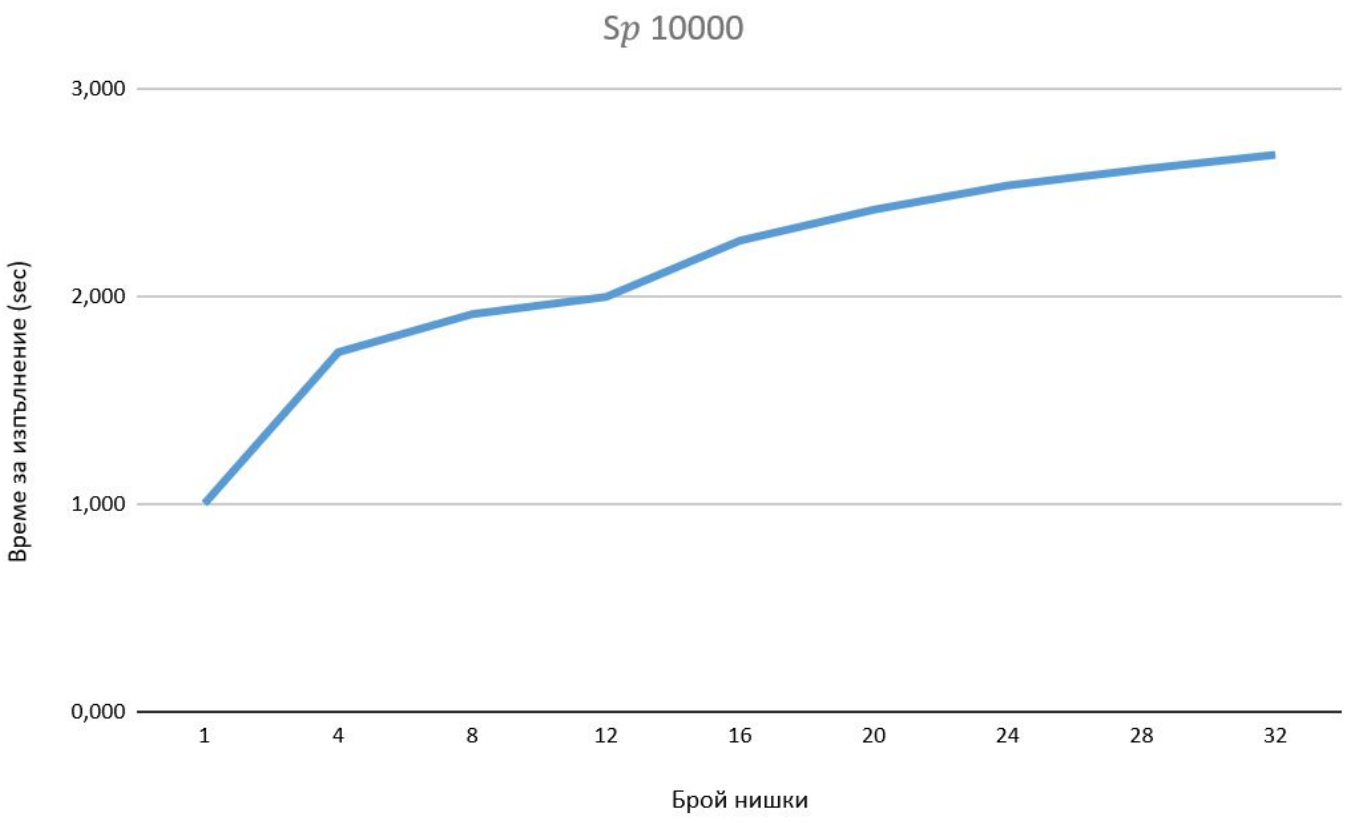

Sp 10000
2,000
1,500
1,000
0,500
0,000
Време за изпълнение (sec)
1
4
8
12
16
20
24
28
32
Брой нишки

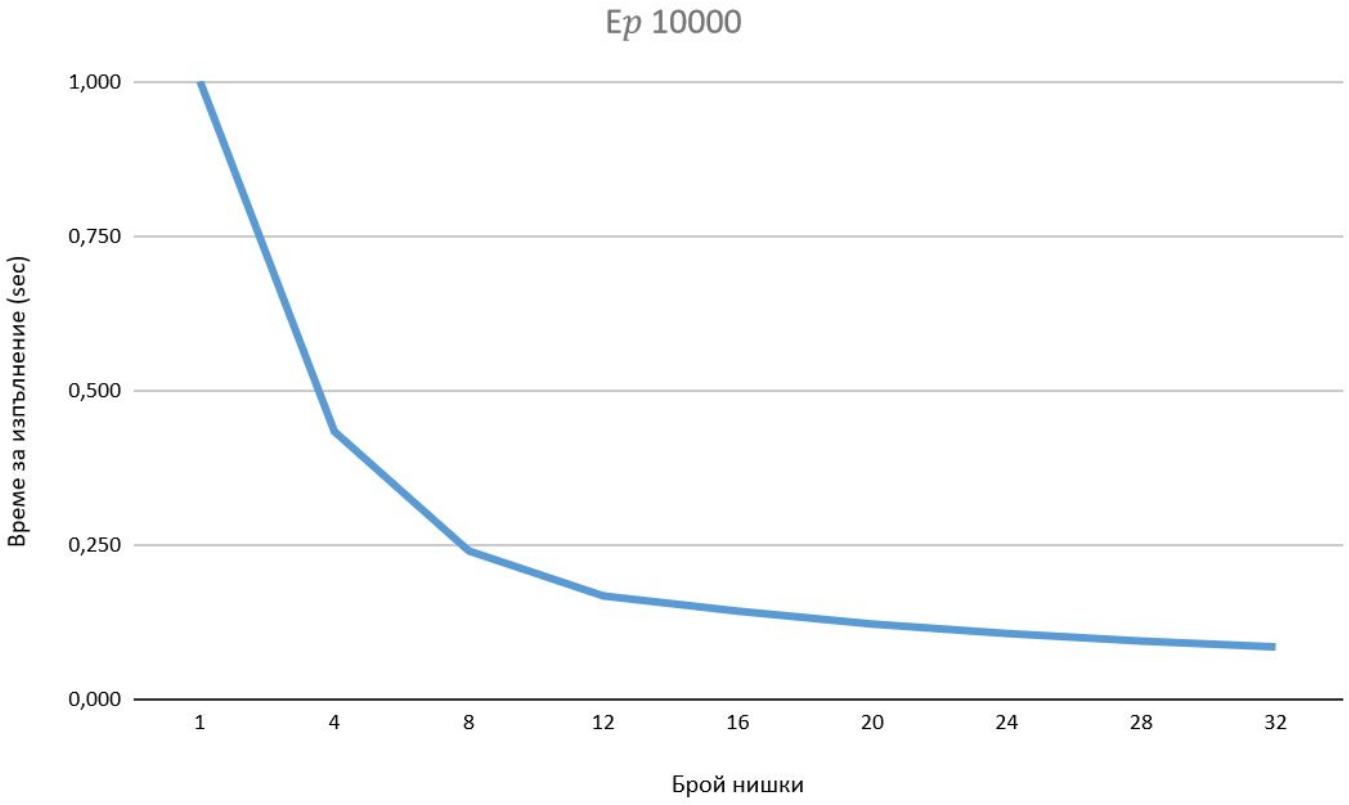

Ep 10000
1,000
0,750
0,500
0,250
0,000
Време за изпълнение (sec)
1
4
8
12
16
20
24
28
32
Брой нишки

Precision	Number of threads	Time	Speed up	Efficiency
100	1	38	1,000	1,000
100	4	33	1,152	0,288
100	8	17	2,235	0,279
100	12	16	2,375	0,198
100	16	35	1,086	0,068
100	20	13	2,923	0,146
100	24	11	3,455	0,144
100	28	9	4,222	0,151
100	32	29	1,310	0,041
1000	1	102	1,000	1,000
1000	4	33	3,091	0,773
1000	8	32	3,188	0,398
1000	12	30	3,400	0,283
1000	16	75	1,360	0,085
1000	20	27	3,778	0,189
1000	24	26	3,923	0,163
1000	28	25	4,080	0,146
1000	32	65	1,569	0,049
10000	1	1060	1,000	1,000
10000	4	936	1,132	0,283
10000	8	921	1,151	0,144
10000	12	906	1,170	0,097
10000	16	891	1,190	0,074
10000	20	876	1,210	0,061
10000	24	866	1,224	0,051
10000	28	854	1,241	0,044
10000	32	839	1,263	0,039
50000	1	20889	1,000	1,000
50000	4	19543	1,069	0,267
50000	8	18645	1,120	0,140
50000	12	17345	1,204	0,100
50000	16	15786	1,323	0,083
50000	20	14443	1,446	0,072
50000	24	16557	1,262	0,053
50000	28	14222	1,469	0,052
50000	32	12341	1,693	0,053

6.Собствено стартирани нишки, които използват граница на итерациите както и пресмятане чрез кеширан факториел


Tp 10000
1250
1000
750
500
250
0
Време за изпълнение (sec)
1
4
8
12
16
20
24
28
32
Брой нишки


Sp 10000
3,000
2,000
1,000
0,000
Време за изпълнение (sec)
1
4
8
12
16
20
24
28
32
Брой нишки



Precision	Number of threads	Time	Speed up	Efficiency
100	1	28	1,000	1,000
100	4	27	1,037	0,259
100	8	26	1,077	0,135
100	12	26	1,077	0,090
100	16	26	1,077	0,067
100	20	25	1,120	0,056
100	24	24	1,167	0,049
100	28	24	1,167	0,042
100	32	23	1,217	0,038
1000	1	48	1,000	1,000
1000	4	24	2,000	0,500
1000	8	28	1,714	0,214
1000	12	28	1,714	0,143
1000	16	28	1,714	0,107
1000	20	28	1,714	0,086
1000	24	25	1,920	0,080
1000	28	25	1,920	0,069
1000	32	24	2,000	0,063
10000	1	1102	1,000	1,000

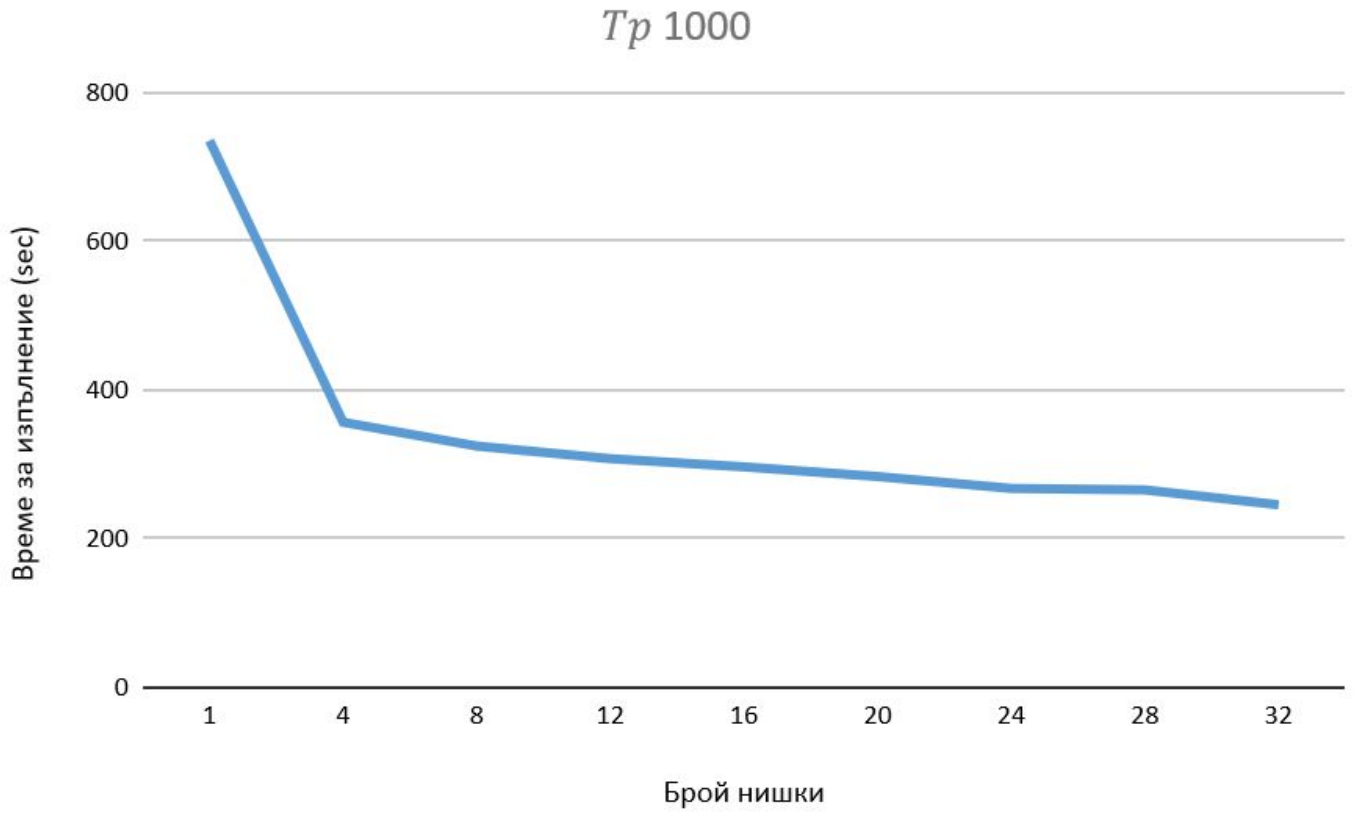
10000	4	1050	1,050	0,262
10000	8	926	1,190	0,149
10000	12	877	1,257	0,105
10000	16	836	1,318	0,082
10000	20	821	1,342	0,067
10000	24	753	1,463	0,061
10000	28	653	1,688	0,060
10000	32	604	1,825	0,057
50000	1	27564	1,000	1,000
50000	4	23252	1,185	0,296
50000	8	18644	1,478	0,185
50000	12	16453	1,675	0,140
50000	16	14325	1,924	0,120
50000	20	13531	2,037	0,102
50000	24	12555	2,195	0,091
50000	28	11467	2,404	0,086
50000	32	10888	2,532	0,079

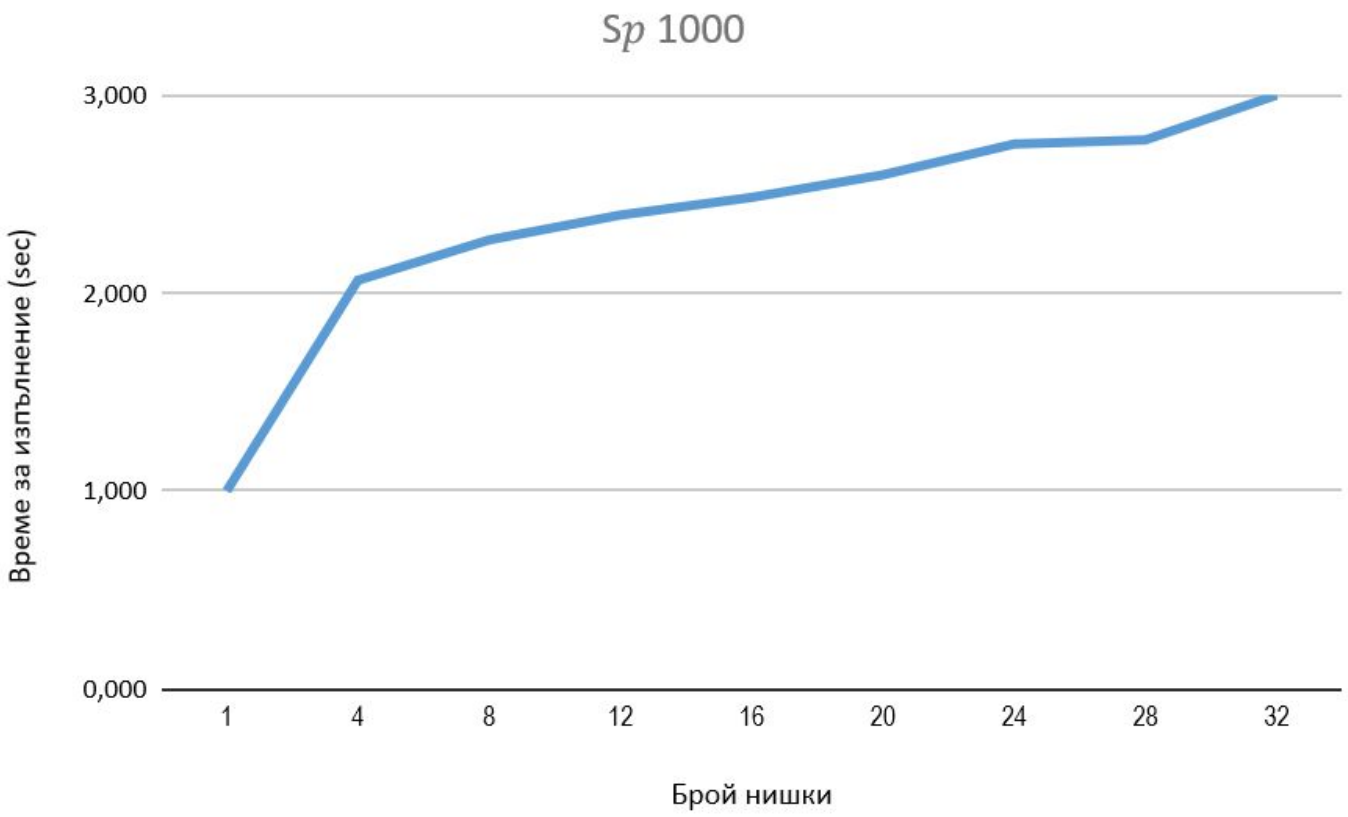
Резултати при система2

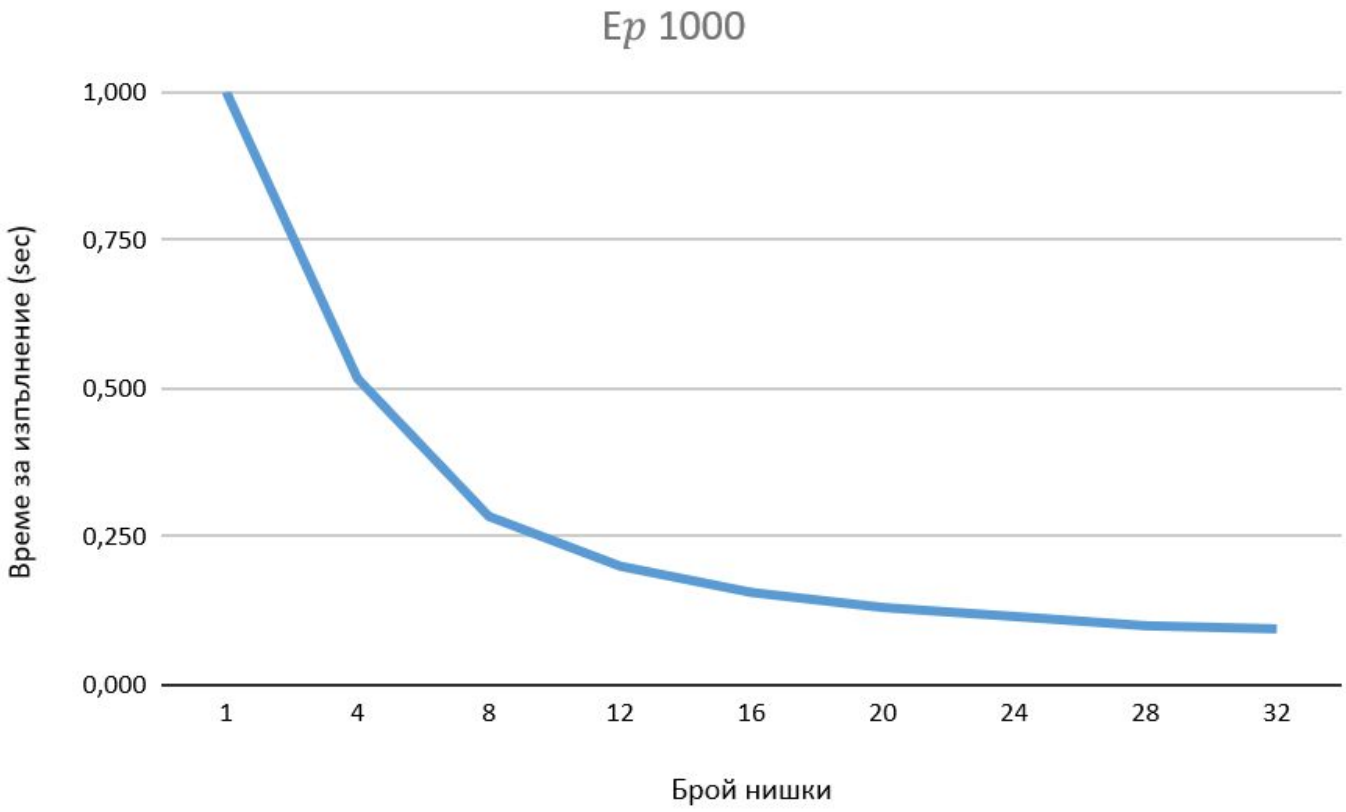
Резултати при използване на ExecutorService, с пресмятане на броя на итерации, с паралелно изчисляване на дробите,оптимизиран с изчисляване на факториел чрез кеширане на нужните факториели.

Precision	Number of threads	Time	Speed up	Efficiency
100	1	16	1,000	1,000
100	4	15	1,067	0,267
100	8	14	1,143	0,143
100	12	14	1,143	0,095
100	16	14	1,143	0,071
100	20	13	1,231	0,062
100	24	12	1,333	0,056
100	28	11	1,455	0,052
100	32	8	2,000	0,063
1000	1	36	1,000	1,000
1000	4	18	2,000	0,500
1000	8	17	2,118	0,265
1000	12	16	2,250	0,188
1000	16	16	2,250	0,141
1000	20	14	2,571	0,129
1000	24	13	2,769	0,115
1000	28	13	2,769	0,099

1000	32	11	3,273	0,102
10000	1	735	1,000	1,000
10000	4	356	2,065	0,516
10000	8	324	2,269	0,284
10000	12	307	2,394	0,200
10000	16	296	2,483	0,155
10000	20	283	2,597	0,130
10000	24	267	2,753	0,115
10000	28	265	2,774	0,099
10000	32	245	3,000	0,094
50000	1	20564	1,000	1,000
50000	4	16875	1,219	0,305
50000	8	14578	1,411	0,176
50000	12	12347	1,666	0,139
50000	16	11567	1,778	0,111
50000	20	10675	1,926	0,096
50000	24	9876	2,082	0,087
50000	28	9304	2,210	0,079
50000	32	8542	2,407	0,075




Sp 1000
3,000
2,000
1,000
0,000
Време за изпълнение (sec)
1
4
8
12
16
20
24
28
32
Брой нишки



4.4. Анализ на тестовите резултати:

Резултати при система 1:

Анализ на резултатите при сравнение с прецизност 50,000 и 32 нишки

Коментар от резултата: Най-оптимален начин е използването на ForkJoinPool в който се публикуват задачи които паралелно пресмятат дробите като се пресмятат нужните факториели предварително.

- **Резултати при Система 2**

Анализ на резултатите при сравнение с прецизност 50,000 и 32 нишки : При 1 нишка времето спада сравнение със Система 1 от 150,000 на 110,000 милисекунди.Най-добър резултат дава в двата случая - “ ForkJoinPool с ограничение на брой итерации и изчисляване на факториел предварително” като времето е подобрено при тестване на втората система от 9654 на 8542 милисекунди за 32 нишки.

5.Източници

[1] Research Directions in Parallel Functional Programming , page 234 , authors Грег Майкълсън, Кевин Хамънд

[2] Image Processing and Pattern Recognition Based on Parallel Shift Technology , page 190 , authors Stepan Bilan, Sergey Yuzhakov

[3] https://github.com/google/guava

[4] http://commons.apache.org/proper/commons-cli/

[5] https://junit.org/junit5/docs/current/user-guide/

[6] https://logging.apache.org/log4j/2.x/

[7] https://projectlombok.org/

[8] https://subscription.packtpub.com/book/big_data_and_business_intelligence/9781783989126/1/ch01lvl1sec10/the-master-slave-architecture, Cassandra high availability ,Robbie Strickland , December 29, 2014 , 186 pages

[9] https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ExecutorService.html

[10] https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ForkJoinPool.html

[11] https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html