Паралелно изчисление на **π** с произволно висока точност

# Проект по СПО Георги Ангелов, ф.н.80699 Компютърни науки, 4. група

# Въведение

π е една от най-известните математически константи. Пресмятането на π с висока точност се е превърнало в нещо подобно на спорт или предизвикателство. Текущият рекорд е 12,100,000,000,050 изчислени позиции след десетичната запетая.  
От практическа гледна точка, точност от 39 десетични цифри би била достатъчна, за да се определи дължината на вселената с точност до атом.

# Инструкции за употреба

Програмата позволява задаването на следните опции:

* **-o <file>, --out <file>** – Файл, в който да се запише резултатът.
* **-p <int>, --terms <int>** – Брой членове на сумата.
* **-t <int>, --threads <int>** - Брой нишки, на които да се смята. Ако не е зададено се използва броя нишки/ядра на процесора (до които JVM има достъп).
* **--sum (Ramanujan|Chudonovsky)** – Сумата, която да се използва.
* **-q** – Тих режим – не показва прогрес.
* **--server** – Стартира изчислителен сървър (включително и RMI registry).
* **-r <ip:port>, --remote <ip:port>** - Може да има много такива опции. Всяка една задава адрес на сървър, който може да се използва за смятане.
* **--registry-port <int>** - Порт за RMI registry-то на сървъра.

# Формула

За бързо пресмятане на π най-често се използват сходящи редове, които имат граница въпросната константа. Има десетки такива редове, като основната разликата е в бързината на сходимост. В този проект се използва ред открит от индийския математик Srinivasa Ramanujan:

Това е един от по-бързо сходящите редове, като за всяко следващ член на сумата се получават от 4 до 6 десетични цифри от π. Средният брой цифри на член за голям брой членове е 5,78. Има и по-бързо сходящи редове, като този на братята Chudnovsky, който пресмята 14 цифри на събираемо.

Всяка подобна формула може да бъде използвана в този проект.

# Последователен метод

В класическия, последователен, вариант за пресмятане на горната формула би било добра идея да се пази временната стойност на частта от формулата, която може да се преизползва от следващата итерация на сумата. Тази част за сумата на Ramanujan е: . Така, за получаване на (n+1)-вия член от n-тия тази запомнена стойност от предишната итерация се умножава с

Тук **fact** е следната функция: , която може да се имплементира рекурсивно като   
Това позволява тя да се пресметне с O(log(b-a)) на брой необходими умножения, което е по-добре при големи n заради времето, което се изисква за умножение на големи числа.

# Паралелен метод

Голям принос за бързодействието на последователния метод има “натрупването” на преизползваемата част от сумата във всеки следващ член. Това е и частта от серийната имплементация, която не е желателно да се жертва в паралелната, защото това би довело до доста по-големи времена за смятане.

Основната идея за паралелизация, която използва програмата е, че една сума може да се раздели на произволен брой суми.

За описание на алгоритъма нека разгледаме сума в по-общ вид:

Всеки член е съставен от произведение на две функции, зависещи от индекса, като **f** има свойството: , където **d** е известна функция. Функцията f е частта от членовете на сумата, която може да се преизползва в следващия член и е добре (от гледна точка на ефективност) да се пресмята чрез горното свойство, а не директно. Функцията **g** не притежава такова свойство – това е частта от членовете, която не може да се натрупва и следователно не е възможно да е част от **f**.

Нека **t** е броят на нишките, на които трябва да се разпредели сумата, а **n** е броят членове, които трябва да се сметнат.

Тогава най-добре би било да разделим сумата така, че да няма дублиране на работата. Тоест **f(k)** винаги трябва да се смята чрез горепосоченото свойство, и то приложено веднъж за всеки член от сумата. С това изискване разпределянето на индексите на четни/нечетни (или **mod t** в случая с **t** нишки) не би било най-оптималното решение, защото за пресмятане на **k**-тия член ще трябва да се приложи горното правило **t-1** пъти.

Оптималното решение на този проблем е **i**-тата нишка да смята следната част от сумата:

Така всяка нишка има налична стойността на **f(k-1)**, когато се наложи да пресметне **f(k)**. Проблем, обаче, все още има, защото първият член на **Si** трябва да бъде пресметнат в началото. За конкретният проект не са правени тестове с този вид на решението, затова не е ясно колко това забавя изчислението в началото.  
Решението на проблема следва от факта, че при премахване на определен брой членове от началото на сумата, то останалите членове имат общ множител, който може да бъде изваден извън нея:

Където .  
Тъй като нишката с индекс **i-1** ще пресметне при изчисление на последния й зададен член, то тази стойност може да се използва, като се умножи директно с получената от **i**-тата нишка сума. Така всеки от членовете се пресмята оптимално без повторни пресмятания на едни и същи неща. В случая когато **i** = 0, тогава , следователно твърдението е вярно и в този случай.

След пресмятането на Si от всяка нишка паралелно, резултатите се комбинират по гореописания начин и полученият резултат се “финализира”. Финализиране наричаме получаването на стойността на π от стойността на сумата. Това включва умножение/деление с определена константа и намиране на реципрочна стойност. Времето за комбиниране на сумите и финализиране зависи от броя на “парчетата”, на които е разделена сумата, като това е от порядъка на 0,2% от общото време за пресмятане (измерено за 4 нишки при изчисление на 10 000 члена). Това процентно съотношение намалява при увеличение на броя на членовете.

# Изчисление на брой получени правилни цифри

Тъй като при стартиране на програмата опцията –p задава броя членове на сумата, то при завършване на изчисленията трябва да се изчисли с каква точност сме получили π. Това се прави като се пресмятат две поредни стойности за π (за n и n+1 члена на сумата) и се намира броя на общите цифри (броя нули в тяхната разлика). Това се счита за полученият точен брой на цифри от π, като след това полученото π се закръглява до толкова цифри след запетаята.

# Абстракция на сума

Тъй като гореописаният метод решава проблема за разпределянето на сумата в общ случай, то той може да бъде използван за паралелно изчисление на всяка сума. Така, чрез абстрахиране на сумата в интерфейс в кода може да се пресмятат различни константи с различни сходящи към тях суми, без да се ограничава до изчисление на π със сумата на Ramanujan.

Това е и целта на интерфейса **InfiniteSum**, който задава начина, по който конкретни суми биват написани, така че да работят с тази програма.

# Абстракция на процес

Още повече – чрез тази дефиниция на метода може много добре да се абстрахира и начина, по който се правят самите изчисления. Това следва от факта, че всяка част от голямата сума е също, сама по себе си, сума. Това означава, че може да се направи йерархия от такива паралелизирания на суми без да има значение дали те се пресмятат последователно, паралелно чрез нишки/процеси или паралелно на различни физически компютри. Общото наименувание в проекта на тези изчислителни единици е **Calculator**.

Интерфейса **Calculator** има метод **calculate**, на който се задава сумата, която се смята, начален и краен индекс и **ProgressHandler** обект (callback за съобщаване на progress). Извикващите този метод не се интересуват от това как се смята зададената част от сумата. Една възможна имплементация разпределя сумата на множество компютри. Това зависи изцяло от класа, който имплементира интерфейса.

В програмата има две имплементации на **Calculator**:

* **SequentialCalculator** – това е всъщност последователната версия на сумирането. Не се правят разделяния или паралелни изчисления. Използва се за листата (сивите правоъгълници в горната схема). Една нишка използва един такъв обект, за да пресметне нейната част от сумата.
* **ConcurrentCalculator** – това е калкулаторът, който разделя сумата на множество части. Тук се намира основната логика за разпаралеляването. Инстанции на този обект са всички калкулатори в схемата, с изключение на листата. При създаването на този обект се подават **Calculator** обектите, които той трябва да използва за изчислението. Това може да са други **SequentialCalculator** или **ConcurrentCalculator** обекти. Всеки такъв обект бива изпълнен в отделна нишка. В схемата по-горе зелените са **ConcurrentCalculator** обекти, на които са зададени 1 или 2 **SequentialCalculator**-a, а на синият са зададени два **ConcurrentCalculator**-a.

Интересен проблем тук е, че при такава йерархия от Calculator обекти не би било ефективно във всеки вътрешен връх работата да се разделя точно на две. Това би довело до резултати в една част на дървото много по-бързо, отколкото в другите. Схемата по-горе е добър пример за това.

За да се реши този проблем към всеки **Calculator** клас се добавя и метод, връщащ **performance score**. Това е число, което приблизително определя бързодействието на конкретния калкулатор спрямо другите. В текущата имплементация това число е броят на нишките, които калкулаторът използва. Според тези оценки се разпределя работата (отговорен за това е **ConcurrentCalculator**). Оценката на **SequentialCalculator** е 1, а тази на **ConcurrentCalculator** автоматично се пресмята като сума на оценките на калкулаторите, които той използва. В горната схема числата в скобите са точно тези оценки.

По този начин може да се направи произволно дълбока и широка йерархия от изчислителни единици, като работата ще бъде разделена максимално оптимално.

Разбира се, броят на нишките не е единственият показател, затова е възможно и други показатели (като тактова честота и benchmark резултати на процесорите) да бъдат добавени към оценката за бързодействието.

# Имплементационни детайли

За изчисления с произволно висока точност се използва библиотеката **Apfloat**, чиято възможност за паралелни изчисления на няколко нишки е изключена, тъй като паралелизацията се прави на ниво сума.

За комуникация на повече от един компютър се използва **Java RMI**, като се предоставя за отдалечено извикване инстанция на **ConcurrentCalculator** и **ProgressHandler** обект за докладване на текущото състояние на изчислението.

За комбиниране на progress event-и се използва клас **MultiProgressHandler**, чиято задача е да докладва общия progress по начин, който е thread-safe. Това е и ProgressHandler-ът, който се предоставя за отдалечено извикване чрез RMI. Прогресите се докладват с 1% гранулярност, така че да не се натоварва мрежата и да не е необходимо често да се синхронизират нишките.

Освен при докладване на прогрес (което става до 100t пъти) не се изисква синхронизиране на работата на нишките, което позволява процесорите да се натоварят максимално. Действително при следене на натовареността на процесорите при изпълнение на програмата те се натоварват 100%.

# Бързодействие

Както се очаква, при гореописаният метод, ускорението е почти линейно. Разбира се, непаралелната част (комбинирането на отделните суми и финализацията) също оказва влияние на резултатите.

Забележка: Горните измервания са направени за 10 000 члена на сумата. Интересен факт е, че при горният сървър, когато 2k на брой нишки вършат работата си едновременно, то всяка от тях завършва за повече време, отколкото k нишки, които вършат същата работа. Например при 8 нишки времето за завършване на всяка една е почти 30% повече от времето за завършване на всяка ако работят по 4 едновременно. Тоест при по-голяма натовареност се наблюдава повече време за пресмятане, въпреки че работят паралелно. С това е свързано и наблюдаваното намаляване на ефективността при по-голям брой нишки. Трябва да се отчете и факта, че тестваните 10 000 члена от сумата е възможно да са под оптималният брой за 24 нишки.

Забележка: Горните измервания са направени съответно върху 4, 9 и 18 четириядрени компютъра за 100 000 члена на сумата (589 096 цифри от π след десетичната запетая).