Зад. 22 (Пресмятане на Pi – Monte Carlo)

Числото (стойността на) Pi може да бъде изчислено по различни начини. Разглеждаме следния метод за приближено пресмятане на Pi:

1) Разглеждаме окръжност, вписана в квадрат;

2) Генерираме по произволен начин точки в рамките на квадрата;

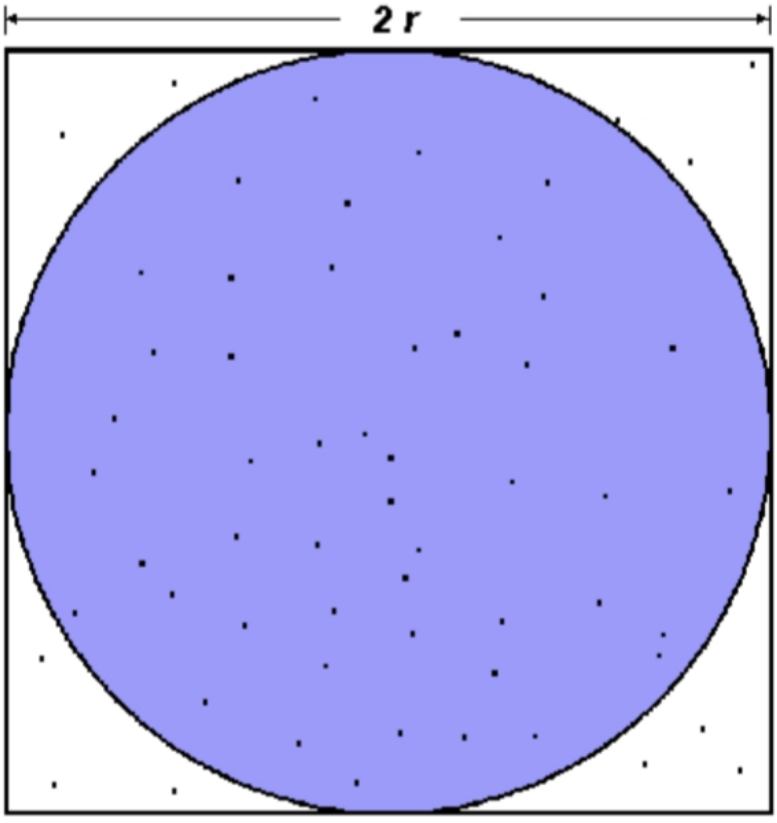
3) Определяме броя на точките, които се намират в окръжността;

4) Нека k е число равно на броя на точките в окръжността, разделен на броя на точките в квадрата;

5) Тогава: Pi ~ 4.0 * k;

Бележка: Колкото повече точки генерираме толкова по-добра ще бъде точността с която пресмятаме Pi, въпреки че трудно можем да я сравним с тази постигана при използването на сходящи редове;

Визуално, можем да представим метода със следната картинка:



$$A_S = (2r)^2 = 4r^2$$

$$A_C = \pi r^2$$

$$\pi = 4 \times \frac{A_C}{A_S}$$

Разглеждаме сериен (последователен) псевдокод реализиращ този метод:

=== cut ===
```
npoints = 10240
circle_count = 0

do j = 1, npoints
       generate 2 random numbers between 0 and 1
       xcoordinate = random1
       ycoordinate = random2
       if (xcoordinate, ycoordinate) inside circle
       then circle_count = circle_count + 1
end do

Pi = 4.0*circle_count/npoints
```
=== cut ===

Както се вижда от кода, по-голямата част от времето за изпълнение на програмата ще премине в изпълнение на операциите от цикъла.

Вашата задача е да напишете програма за изчисляване на числото Pi, по описания метод, която използва паралелни процеси (нишки). Изискванията към програмата са следните:

(o) Размерността на квадрата се задава в точки (пиксели) от подходящо избран команден параметър – например "**-s 10240**";

(o) Точките в квадрата, генерираме произволно с помощта на **Math.random()**, (т.е. класа **java.util.Random**) или **java.util.concurrent.ThreadLocalRandom**;
Разликата между двата начина - **Math.random()** е достъпен във всички версии на Java и ужасно бавен. Не е проектиран за работа в много-нишкова (multi-threaded) среда; В Java 7 и 8, разполагаме с **ThreadLocalRandom**, който е специално проектиран за работа в рамките на отделен thread (респективно multi-threaded среда);
Ще бъде много интересно да реализирате програма, използваща и двата начина и съответно получим два вида резултати от работата на програмата;

(o) Друг команден параметър задава максималния брой нишки (задачи) на които разделяме работата по пресмятането на Pi – например "**–t 1**" или "**–tasks 3**";

(o) Програмата извежда подходящи съобщения на различните етапи от работата си, както и времето отделено за изчисление и резултата от изчислението;
Примери за подходящи съобщения:
„**Thread-<num> started.**",
„**Thread-<num> stopped.**",
„**Thread-<num> execution time was (millis): <num>**",
„**Threads used in current run: <num>**",
„**Total execution time for current run (millis): <num>**" и т.н.;

(o) Да се осигури възможност за „**quiet**" режим на работа на програмата, при който се извежда само времето отделено за изчисление на Pi (и самото число), отново чрез подходящо избран друг команден параметър – например "**-q**";

Не е задължително „сляпо" да следвате логиката на цитирания последователен код; Възможно е първо да решите задачата с генерирането на точки, след което задачата с определянето на това кои от тях принадлежат на окръжността. И въпреки че подобен подход не е много ефективен от гледна точка на памет, няма ограничения за количеството, което използва програмата Ви ;);

ЗАБЕЛЕЖКА:

(о) При желание за направата на подходящ графичен потребителски интерфейс (**GUI**) с помощта на класовете от пакета **javax.swing** задачата може да се изпълни от **двама души**; Разработването на графичен интерфейс не отменя изискването Вашата програма да поддържа изредените командни параметри. В този случай към функцията на параметъра параметъра **„-q“** се добавя изискването **да не пуска** графичният интерфейс. Причината за това е, че Вашата програма трябва да позволява отдалечено тестване, а то ще се извършва в **terminal**.

Уточнения (|| hints) към задачата:

(о) В условието на задачата се говори за разделянето на работата на две или повече нишки. Работата върху съответната задача, в случаят в който е зададен **„-t 1“** (т.е. цялата задача се решава от една нишка) ще служи за еталон, по който да измерваме евентуално ускорение (т.е. това е **T1**). В кода реализиращ решението на задачата трябва да се предвиди и тази възможност – задачата да бъде решавана от единствена нишка (процес); Пускайки програмата да работи върху задачата с помощта на единствена нишка, ще считаме че използваме серийното решение на задачата; Измервайки времето за работа на програмата при използването на **„p“** нишки – намираме **Tp** и съответно можем да изчислим **Sp**. Представените на защитата данни за работата на програмата, трябва да отразят и ефективността от работата и, тоест да се изчисли и покаже **Ep**.

Като обобщение - данните събрани при тестването на програмата Ви, трябва да отразяват **Tp**, **Sp** и **Ep**. Желателно е освен табличен вид, да добавите и графичен вид на **Tp**, **Sp**, **Ep**, в три отделни графики.

(о) Не се очаква от Вас да реализирате библиотека, осигуряваща математически операции със комплексни числа. Подходяща за тази цел е например **Apache Commons Math3 (http://commons.apache.org/proper/commons-math/userguide/complex.html).** При изчисленията, свързани с генерирането на множеството на Манделброт (задачите за фрактали), определено ще имате нужда от нея.

(о) Не се очаква от вас да реализирате библиотека, осигуряваща математически операции със голяма точност. Подходяща за тази цел библиотека е например **Apfloat (http://www.apfloat.org)**. Ако програмата Ви има нужда от работа с големи числа, можете да използвате нея.

Разбира се **BigInteger** и **BigDecimal** класовете в **java.math** са също възможно решение – въпрос на избор и вкус.

Преди да направите избора, проверете дали избраната библиотека не използва също нишки – това може да доведе до неочаквани и доста интересни резултати; ;)

(о) Не се очаква от Вас да търсите (пишете) библиотека за генериране на **.png** изображения, в случай че задачата Ви го изисква или просто сте решили да добавите малко „цвят“ в решението. Java има прекрасна за нашите цели вградена библиотека, която може да се ползва.

Примерен проект, показващ генерирането на чернобялата и цветната версия на фрактала на Манделброт /множество на Манделброт за формула (2)/, цитирани в задачите за фрактали, е качена на **http://rmi.yaht.net/docs/example.projects/ - pfg.zip**.

(о) Командните аргументи (параметри) на терминална Java програма, получаваме във масива **String args[]** на **main()** метода, намиращ се в стартовият клас. За „разбирането“ им (анализирането им) може да ползвате и външни библиотеки писани специално за тази цел . Един добър пример за това е: **Apache Commons CLI (http://commons.apache.org/cli/)**.

(о) Интересен е въпросът, кога достигаме зададената точност на изчисленията? Тоест кога сме сметнали Pi със зададените от потребителя брой цифри след десетичната точка. Едно добро ограничение за серийната (последователната) програма е разликата между две поредно изчислени стойности на Pi да е произволно малка. (в случай че използваме сходящ ред за Pi);

В случаят с Monte Carlo Pi, точността ни се определя от точността, с която извършваме последната операция в алгоритъма;