Министерство науки и высшего образования Российской Федерации

**Федеральное государственное бюджетное образовательное учреждение высшего образования**

«Владимирский государственный университет

имени Александра Григорьевича и Николая Григорьевича Столетовых»

**(ВлГУ)**

**Кафедра информационных систем и программной инженерии**

Лабораторная работа №5

по дисциплине «Платформонезависимое

программирование»

Тема: «Работа в многопоточной среде. Метод Монте-Карло.»

Выполнил:

студент гр. ПРИ-123

А.Ц. Нямаа

Приняла:

Ст. пр. кафедры ИСПИ

О.Н. Шамышева

Владимир, 2025 г.

ЦЕЛЬ РАБОТЫ:

Реализовать параллельное и последовательное вычисление числа ПИ методом Монте-Карло на языке программирования Java.

ЗАДАНИЕ 1 (вариант 12).

Реализовать последовательный алгоритм вычисление числа PI методом Монте-Карло с машинной точностью (вычислительная погрешность = 0). Реализовать программное измерение времени работы.

ВЫПОЛНЕНИЕ РАБОТЫ:

В начале выполнения данного задания я определил физическую структуру приложения.

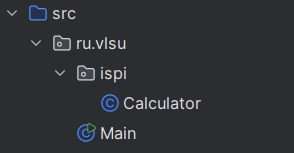


Рисунок 1. Физическая структура программы

Далее написал последовательный алгоритм вычисления PI методом Монте-Карло, соответствуя всем требованиям. В программе создал следующие классы: исполнительный класс Main и класс Calculator, где происходят необходимые вычисления.

Листинг Main.java:

package ru.vlsu;  
  
import ru.vlsu.ispi.Calculator;  
  
public class Main {  
 public static void main(String[] args) {  
 long startTime = System.currentTimeMillis();  
  
 System.out.println("Число PI, вычисленное написанным алгоритмом: ");  
 System.out.println(Calculator.calculator(Integer.MAX\_VALUE));  
 System.out.println("Число PI, вычисленное встроенным методом: ");  
 System.out.println(Math.PI);  
  
 long endTime = System.currentTimeMillis();  
 System.out.println("Время работы программы: " + (endTime - startTime) / 1000 + " секунд");  
 }  
}

Листинг Calculator.java:

package ru.vlsu.ispi;  
  
public class Calculator {  
 private static final int radius = 1;  
  
 public static double calculator(int countPoints){  
 int pointsIn = 0;  
  
 for (int i = 0; i < countPoints; i++){  
 double x = Math.random() \* (radius \* 2) - 1;  
 double y = Math.random() \* (radius \* 2) - 1;  
  
 if (Math.pow(x, 2) + Math.pow(y, 2) <= Math.pow(radius, 2)){  
 pointsIn++;  
 }  
 }  
  
 return (4.0 \* pointsIn) / countPoints;  
 }  
}

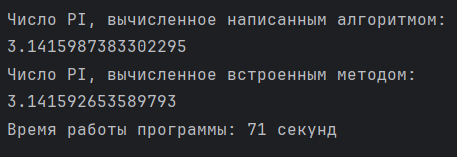


Рисунок 2. Выполнение программы с максимальной значением Integer точек.

ЗАДАНИЕ 2 (вариант 12).

Реализовать параллельную версию той же программы на основе многопоточности. Построить график зависимостей точности вычисления от числа точек и скорости работы программы от числа потоков.

ВЫПОЛНЕНИЕ РАБОТЫ:

Для выполнения данного задания сначала я определил физическую структуру приложения.

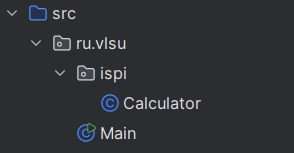


Рисунок 3. Физическая структура программы

Следующим шагом реализовал параллельную версию написанного в первом задании алгоритма на основе многопоточности. Также построил график зависимостей точности вычисления от числа точек и скорости работы от числа потоков.

Листинг Main.java:

package ru.vlsu;  
  
import ru.vlsu.ispi.Calculator;  
  
public class Main {  
 public static void main(String[] args) throws InterruptedException {  
 long startTime = System.currentTimeMillis();  
  
 System.out.println("Число PI, вычисленное многопоточным алгоритмом: ");  
 System.out.println(Calculator.calculator(2\_147\_483\_647, 6));  
 System.out.println("Число PI, вычисленное встроенным методом: ");  
 System.out.println(Math.PI);  
  
 long endTime = System.currentTimeMillis();  
 System.out.println("Время работы программы: " + (endTime - startTime) / 1000 + " секунд");  
 }  
}

Листинг Calculator.java:

package ru.vlsu.ispi;  
  
import java.util.concurrent.ThreadLocalRandom;  
  
public class Calculator {  
 private static final int radius = 1;  
  
 public static double calculator(long countPoints, int countThreads) throws InterruptedException {  
 long pointsInTotal = 0;  
 Thread[] threads = new Thread[countThreads];  
 long[] singleThreadsResults = new long[countThreads];  
  
 for (int i = 0; i < countThreads; i++) {  
 final int threadIndex = i;  
 threads[threadIndex] = new Thread(() -> {  
 long pointsIn = 0;  
 long threadStart = (countPoints / countThreads) \* threadIndex;  
 long threadEnd = (countPoints / countThreads) \* (threadIndex + 1);  
  
 for (long j = threadStart; j < threadEnd; j++) {  
 double x = ThreadLocalRandom.current().nextDouble(radius);  
 double y = ThreadLocalRandom.current().nextDouble(radius);  
 if (Math.pow(x, 2) + Math.pow(y, 2) <= Math.pow(radius, 2)) {  
 pointsIn++;  
 }  
 }  
 singleThreadsResults[threadIndex] = pointsIn;  
 });  
 threads[threadIndex].start();  
 }  
  
 for (Thread thread : threads) {  
 thread.join();  
 }  
  
 for (long singleResult : singleThreadsResults) {  
 pointsInTotal += singleResult;  
 }  
  
 return (4.0 \* pointsInTotal) / countPoints;  
 }  
}

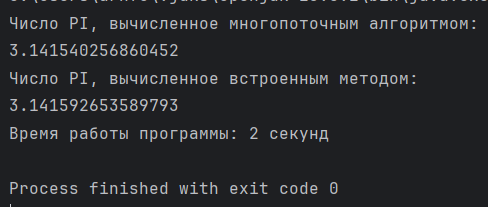


Рисунок 4. Выполнение программы с максимальной значением Integer точек.

ВЫВОД К РАБОТЕ:

Я реализовал параллельное и последовательное вычисление числа ПИ методом Монте-Карло на языке программирования Java.