Міністерство освіти і науки України
Національний технічний університет України
“Київський політехнічний інститут імені Ігоря Сікорського”
Факультет інформатики та обчислювальної техніки
Кафедра інформаційних систем та технологій

Лабораторна робота №8

Мова програмування Java

Тема: Multithreading

Виконав
студент групи ІА-32:
Костінський М. М.

Перевірив:
Лесик В. О.

2025

Тема: Multithreading.

Мета: ознайомлення з поняттям розпаралелювання та потоків виконання, практичне освоєння алгоритму запуску потоку виконання, синхронізації, блокування та спілкування між ними.

Хід роботи

Теорія:

Багатопотоковість у мові програмування Java є механізмом, що дозволяє виконувати кілька частин програми паралельно, ефективно використовуючи ресурси сучасних багатоядерних процесорів. Потік виконання представляє собою найменшу послідовність інструкцій, яка може бути запланована операційною системою, і всі потоки в межах одного процесу мають доступ до спільної області пам'яті. У Java створення нового потоку зазвичай реалізується через успадкування від класу Thread або імплементацію інтерфейсу Runnable, де метод run містить код, що має виконуватися паралельно. Запуск потоку здійснюється викликом методу start, який повідомляє віртуальній машині про необхідність виділення ресурсів та початку виконання в окремому стеку.

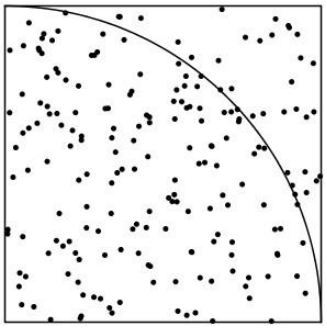
Ключовим аспектом розробки багатопотокових програм є синхронізація доступу до спільних ресурсів, оскільки одночасна модифікація даних кількома потоками може призвести до стану гонки та некоректних результатів. Для запобігання цьому використовуються механізми блокування, такі як монітори об'єктів та ключове слово synchronized, які гарантують, що в критичній секції коду одночасно може знаходитися лише один потік. Альтернативою є використання явних об'єктів Lock або атомарних змінних, які забезпечують потокобезпечні операції на апаратному рівні без необхідності повного блокування потоку.

Спілкування та координація між потоками є важливою складовою паралельних обчислень. Для того щоб головний потік програми міг отримати результати роботи дочірніх потоків, використовується метод join, який призупиняє виконання поточного потоку до моменту завершення роботи іншого. Це дозволяє гарантувати, що всі обчислення завершені перед тим, як програма перейде до агрегації фінальних даних. Ефективне використання цих інструментів дозволяє значно підвищити продуктивність програм при виконанні ресурсомістких завдань.

Завдання:

Обчислення наближеного значення числа Пі методом Монте-Карло

У цьому завданні вам належить написати паралельну програму, яка обчислює значення числа Пі. Метод обчислення дуже простий:



- Площа квадрата одиничної довжини дорівнює 1
- Площа сектора 90 ° для одиничного кола: π/4
- «Кидаємо» величезну кількість випадкових точок в одиничний квадрат
- Рахуємо кількість точок, що потрапили в межі кола, тобто відстань від яких до (0,0) менше або дорівнює 1 Частка точок, які потрапили в коло дорівнює

наближеному значенню π/4

Деталі реалізації

Ваше завдання написати паралельну реалізацію (ParallelMonteCarloPi.java). При написанні програми дотримуйтесь інструкцій:

- Першим і єдиним вхідним аргументом програми є кількість потоків
- В результаті програма виводить наступні дані:

  PI is 3.14221

  THREADS 8

  ITERATIONS 1,000,000,000

  TIME 12.83ms

Крім написання програми і виведення результату подумайте над наступними питаннями:

- Як впливає кількість ітерацій (кинутих точок) на кінцевий результат?
- При однаковій кількості точок і випадковому зерні, як впливає на результат різну кількість потоків?
- Як кількість потоків впливає на продуктивність вашої програми? (Для явних результатів вам ймовірно знадобиться набагато більша кількість семплів

(ітерацій).

Код:

```
import java.util.concurrent.ThreadLocalRandom;
import java.util.concurrent.atomic.AtomicLong;

public class ParallelMonteCarloPi {

    public static void main(String[] args) {
        final long TOTAL_ITERATIONS = 1_000_000_000L;

        int numThreads = 1;
        if (args.length > 0) {
            try {
                numThreads = Integer.parseInt(args[0]);
            } catch (NumberFormatException e) {
                System.err.println("Аргумент має бути цілим числом.");
                return;
            }
        }

        long startTime = System.currentTimeMillis();
```

```
        AtomicLong totalPointsInCircle = new AtomicLong(0);

        Thread[] threads = new Thread[numThreads];

        long iterationsPerThread = TOTAL_ITERATIONS / numThreads;

        for (int i = 0; i < numThreads; i++) {
            threads[i] = new Thread(() -> {
                long pointsInCircle = 0;

                ThreadLocalRandom random = ThreadLocalRandom.current();

                for (long j = 0; j < iterationsPerThread; j++) {
                    double x = random.nextDouble();
                    double y = random.nextDouble();

                    if (x * x + y * y <= 1) {
                        pointsInCircle++;
                    }
                }

                totalPointsInCircle.addAndGet(pointsInCircle);
            });

            threads[i].start();
        }

        for (Thread thread : threads) {
            try {
                thread.join();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }

        long endTime = System.currentTimeMillis();
        long executionTime = endTime - startTime;

        double pi = 4.0 * totalPointsInCircle.get() / TOTAL_ITERATIONS;

        System.out.println("PI is " + pi);
        System.out.println("THREADS " + numThreads);
        System.out.println("ITERATIONS " + TOTAL_ITERATIONS);
        System.out.println("TIME " + executionTime + "ms");
    }
}
```

Приклади виконання коду:

```
PI is 3.141500716
THREADS 1
ITERATIONS 1000000000
TIME 6496ms
```

```
PI is 3.141601092
THREADS 2
ITERATIONS 1000000000
TIME 3682ms
```

```
PI is 3.141605172
THREADS 4
ITERATIONS 1000000000
TIME 2270ms
```

```
PI is 3.1416253
THREADS 8
ITERATIONS 1000000000
TIME 1720ms
```

```
PI is 3.141630644
THREADS 10
ITERATIONS 1000000000
TIME 1753ms
```

```
PI is 3.141589692
THREADS 20
ITERATIONS 1000000000
TIME 1749ms
```

Відповіді на запитання:

1. Як впливає кількість ітерацій (кинутих точок) на кінцевий результат?
Відповідь: Збільшення кількості ітерацій підвищує точність наближеного значення числа Пі, оскільки метод Монте-Карло ґрунтується на законі великих чисел.

2. При однаковій кількості точок і випадковому зерні, як впливає на результат різну кількість потоків?
Відповідь: Кількість потоків не впливає на кінцевий математичний результат, оскільки загальна кількість точок залишається сталою, а обчислення для кожної точки є незалежним.

3. Як кількість потоків впливає на продуктивність вашої програми?
Відповідь: Збільшення кількості потоків зазвичай підвищує продуктивність (зменшує час виконання) до певного моменту, залежно від кількості фізичних ядер процесора (у моєму випадку 8), оскільки обчислювальна робота розподіляється та виконується паралельно.

Висновок: У ході виконання лабораторної роботи було створено програмне забезпечення для паралельного обчислення наближеного значення числа Пі методом Монте-Карло. Було практично засвоєно роботу з класом Thread для створення та запуску потоків, а також реалізовано механізм розподілу великої кількості ітерацій між ними. Для забезпечення цілісності даних при паралельному записі результатів було використано атомарні змінні, а для координації завершення роботи всіх потоків застосовано метод join. Отримані результати підтвердили ефективність розпаралелювання обчислювальних задач на багатоядерних системах.