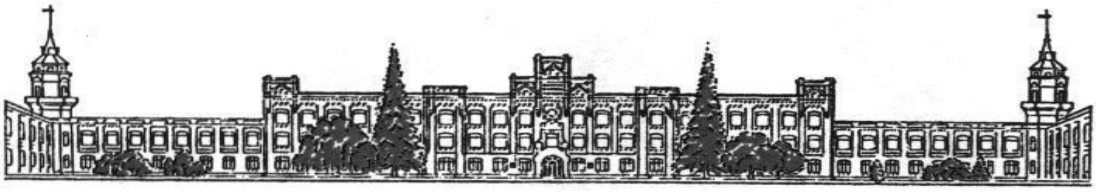


Міністерство освіти і науки України

Національний технічний університет України

«Київський політехнічний інститут ім. Ігоря Сікорського»

Інститут прикладного системного аналізу

Кафедра системного проектування

Лабораторна робота №3

з дисципліни “Паралельні обчислення”

на тему: “**Дослідження атомарних змінних та атомарних операцій**”

Виконала:

студентка III курсу, групи ДА-01

Балуматкіна О. В.

Варіант 4

Прийняла(в):

Київ-2023

Зміст

Мета роботи 3

Завдання 3

Варіант 4 3

Хід роботи 4

Теоретична складова 4

Рішення без використання потоків 6

Рішення з потоками та блокуючими примітивами 7

Рішення з потоками та атомарністю 9

Аналіз результатів роботи програм 11

Висновок 13

Додаток А 14

Додаток Б 15

Додаток В 18

Мета роботи

Розглянути поняття атомарності, навчитися працювати з атомарними змінними, а також ознайомитися з підходом написання паралельного коду без блокування.

Завдання

1. Ознайомитися з визначеннями: атомарна змінна, атомарна операція, неблокуючий алгоритм. Ознайомитися з деталями атомарності в обраній мові програмування.
2. Виконати завдання без використання потоків. Заміряти час виконання завдання.
3. Виконати завдання за варіантом з використанням блокуючих примітивів синхронізації. Заміряти час виконання завдання, а також кількість часу, яка витрачається на очікування розблокування примітивів.
4. Виконати завдання за варіантом з використанням атомарних змінних та CAS операцій. Заміряти час виконання завдання.
5. Повторити пункти 2, 3 та 4 з використанням різної розмірності даних. Навести результати у вигляді графіків.
6. Зробити висновки за отриманими результатами.

Варіант 4

Знайти кількість непарних елементів в масиві та найбільше непарне число.

Хід роботи

Теоретична складова

Відповідно до завдання лабораторної роботи, спочатку необхідно ознайомитись з визначеннями, що будуть використані під час виконання завдання.

Атомарність – властивість операцій, яка означає, що вони виконуються безперебійно в одному такті, не допускаючи втручання інших процесів або потоків всередину операції.

Атомарна змінна – спеціальні типи даних, призначені для коректного використання в паралельних середовищах. Дані типи даних гарантують атомарність операцій (виконання окремих операцій без переривань), забезпечуючи коректність результатів.

Атомарна операція – операція або їхня послідовність, що виконується як одна незалежна одиниця обробки, тобто дана операція або буде повністю виконана та не розбита на менші частини, або взагалі не виконана.

Неблокуючий алгоритм – алгоритм, що забезпечує синхронізацію між потоками без використання блокуючих примітивів типу м’ютексів. В основі таких алгоритмів лежать або відсутність використання синхронізації, або атомарні операції та змінні, що дозволяють виконувати операції над спільними ресурсами, не призупиняючи роботу інших потоків виконання та не приводячи до операції зміни контексту виконання.

Переваги неблокуючих алгоритмів:

1. Відсутність deadlock-ів;
2. Відсутність затримок на очікування розблокування;

3. Краща масштабованість (дозволяють паралельний доступ до спільних ресурсів без конфліктів блокування).

У мові програмування Java атомарності можна досягнути з використанням різних механізмів (наприклад, ключове слово synchronized, класи Lock, Semaphore, тощо), але в рамках даної лабораторної роботи нас цікавить саме клас Atomic для атомарності.

Клас Atomic у мові Java є частиною пакету *java.util.concurrent.atomic*, надає засоби для безпечної маніпуляції зі змінними, які можуть бути доступні з кількох потоків. Клас Atomic містить набір методів, що дозволяють здійснювати атомарні операції з числовими типами даних (int, long, double, float, тощо). Деякі з цих методів:

- *get()* - повертає поточне значення змінної;
- *set(value)* - встановлює нове значення змінної;
- *getAndSet(value)* - повертає старе значення змінної і встановлює нове значення;
- *compareAndSet(expectedValue, newValue)* - перевіряє, чи дорівнює поточне значення змінної очікуваному значенню, і якщо так, встановлює нове значення (також даний метод можна назвати операцією CAS).

Основна *перевага* використання класу Atomic полягає в тому, що він забезпечує атомарність операцій з даними, навіть якщо вони виконуються у багатопоточному середовищі, що означає, що доступ до змінної забезпечується тільки одним потоком в будь-який момент часу, що дозволяє уникнути змін змінної в той час, коли її використовує інший потік.

Рішення без використання потоків

Оскільки завдання варіанту – порахувати кількість непарних чисел у потоці та знайти найбільше непарне число, для реалізації програми без використання потоків необхідно просто створити масив на встановлену кількість значень, заповнити, до прикладу, довільними цілими числами, та, з використанням лічильника та змінної для збереження найбільшого елементу, «пройтися» по всьому створеному масиву, шукаючи непарні числа. Відповідно, коли непарне число знайдене – значення лічильника інкрементується, а значення максимального непарного числа порівнюється до знайденого і, якщо поточне значення більше за те, що міститься в змінній найбільшого непарного числа, значення змінної оновлюється. По закінченню виконання програми в консоль виводиться статистика. Для заміру часу було вирішено використовувати не лише секцію з пошуком найбільшого непарного значення та підрахунком кількості непарних чисел, а всю програму, починаючи зі створення та наповнення масиву, і закінчуючи рівно перед виводом текстової статистики в консоль. Для решти рішень буде використано такий самий підхід для заміру часу, оскільки, на мою думку, витрачений на створення потоків час, у випадку, коли ми порівнюємо до не багатопоточного рішення, має значення.

```
"C:\Program Files\Java\jdk-17.0.4.1\bin\java.exe" "-javaagent:D:\intelliJ_java
TASK: Find the number of odd elements in an array and the largest odd number
Array size: 1000000

Task completed in: 30272300 nanoseconds (30 milliseconds)

Number of odd elements: 499568
Largest odd number: 7999

Process finished with exit code 0
```

За аналізом скріншоту результату виконання програми можна побачити, що, у випадку рішення без використання потоків, було створено та оброблено

масив на 1.000.000 значень всього за 30 мілісекунд. Найбільшим значенням виявилось 7999 (за умови, що у кожному рішенні встановлено максимальне значення для генерування довільних значень у 8000), а загальна кількість непарних чисел дуже близька до половини масиву – 499,568.

Повний код рішення без потоків наведено у додатку А.

Рішення з потоками та блокуючими примітивами

Аналогічно до рішення без потоків, створюється масив довільних цілих чисел фіксованого розміру, після чого до обробки масиву (підрахунку кількості непарних чисел та пошуку найбільшого непарного елемента) приступають потоки з пулу потоків.

Кількість потоків самостійно розраховується програмою відповідно до кількості процесові, доступних для JVM (зазвичай це стабільне значення – 8 потоків), аналогічний підхід використано для реалізації рішення з використанням атомарності.

Таким чином, в класі Main створюється масив довільних цілих значень фіксованої довжини, після чого створюється об’єкт counter класу OddNumberCounter, в який передається створений масив. Після створення об’єкту викликається метод класу OddNumberCounter - countOdds(). Даний метод створює пул потоків (який також використовується в рішенні з атомарністю) на фіксовану кількість потоків, що відповідає кількості вільних процесорів для JVM (у статистиці по кінцю виконання програми зазначається кількість використаних потоків).

Наступним кроком метод розділяє отриманий масив довільних цілих чисел на частини відповідно до кількості потоків в пулі потоків та передає кожну частину на опрацювання потоку, а циклом for метод запевнюється, що потоки не оброблюють одну й ту ж частину масиву. Всередині циклу for викликається

метод submit() з інтерфейсу ExecutorService для надсилання об'єктів класу OddNumberCounterTask до екземпляру вищезгаданого пулу потоків. Кожен об’єкт класу OddNumberCounterTask по суті представляє частину масиву, який обробляється для пошуку непарних чисел з використанням такого блокуючого примітиву, як м’ютекс класу Semaphore для забезпечення взаємного виключення для спільних для усіх потоків змінних oddCount та maxOdd. Також всередині переписаного методу run() з приватного класу OddNumberCounterTask є таймер для заміру часу, витраченого потоком на очікування розблокування м’ютексу, сумарне значення часу в подальшому виводиться разом з іншою статистикою у консоль.

```
TASK: Find the number of odd elements in an array and the largest odd number
Array size: 1000000

Amount of used threads: 8

Task completed in: 109954600 nanoseconds (109 milliseconds)
Waiting lock time: 517653275 nanoseconds (517 milliseconds)

Number of odd elements: 499823
Largest odd number: 7999

Process finished with exit code 0
```

За аналізом скріншоту результату виконання програми можна побачити, що, у випадку поточного рішення з використанням потоків та блокуючих примітивів, програма самостійно визначила, що необхідно використовувати 8 доступних потоків для обробки масиву на аналогічну до попереднього кількість елементів в масиві – 1,000,000. З врахуванням часу, витраченого на створення пулу потоків, розділу масиву на частини, делегування завдань потокам, програма завершила своє виконання за 109 мілісекунд та визначила, що найбільшим непарним числом знов було 7999 (що не дивно, оскільки в нас генерується аж мільйон різних елементів), а загальна кількість непарних елементів знову дуже близька до половини масиву – 499,823. Сумарна кількість

часу, витрачена потоками на очікування розблокування м’ютексу, також наведена – 517 мілісекунд (що в 5 разів більше, ніж загальний час виконання програми, але цьому дивуватись не варто, оскільки вже було зазначено, що це сумарний час очікування на розблокування м’ютексу від усіх восьми створених потоків в пулі потоків).

Повний код рішення з використанням потоків та блокуючих примітивів наведено у додатку Б.

Рішення з потоками та атомарністю

Для рішення з атомарністю було використано атомарний integer та масив атомарних чисельних значень. В першому випадку, атомарна змінна використовується для збереження в собі значення найбільшого непарного елементу, яке, відповідно до факту використання багатьох потоків для вирішення завдання, має бути захищеним від доступу з декількох потоків водночас. Атомарний масив чисельних значень, у свою чергу, використовується для збереження підрахунків по кількості непарних значень з частин масиву, які відводяться кожному з потоків в пулі потоків (даний атомарний масив надає доступ до певного свого елементу лише одному потоку, тобто теж зберігає значення від зміни декількома потоками водночас, що може призводити до некоректних результатів).

З атомарних методів було використано наступне:

- *getAndIncrement()* – отримує значення поточної змінної (повертає, до чері, аналогічне значення) та інкрементує отримане значення й записує до змінної нове значення так, щоб при наступному, до прикладу, виклику методу get() вивелось інкрементоване значення. В даному випадку використано для збільшення значення знайдених непарних чисел в атомарному масиві;

- *get()* – просто отримує значення змінної, в цьому випадку використано для присвоєння тимчасовій змінній наявного значення максимального непарного числа;
- *compareAndSet(),* або CAS – використовується для порівняння значення всередині змінної з першим (очікуваним) значенням, переданим до цього методу. У випадку, якщо значення всередині змінної аналогічне до очікуваного значення, це значення всередині змінної змінюється на друге значення (наступне), передане всередину даного методу. У випадку, якщо значення змінної та очікуване значення все ж таки різні, метод повертає false та нічого не змінює в змінній. В даному випадку метод використовується для пошуку найбільшого непарного значення.

```
"C:\Program Files\Java\jdk-17.0.4.1\bin\java.exe" "-javaagent:D:\intelliJ_java\
TASK: Find the number of odd elements in an array and the largest odd number
Array size: 1000000
Threads amount: 8

Task completed in: 50724100 nanoseconds (50 milliseconds)
Number of odd elements: 500144
Largest odd number: 7999

Process finished with exit code 0
```

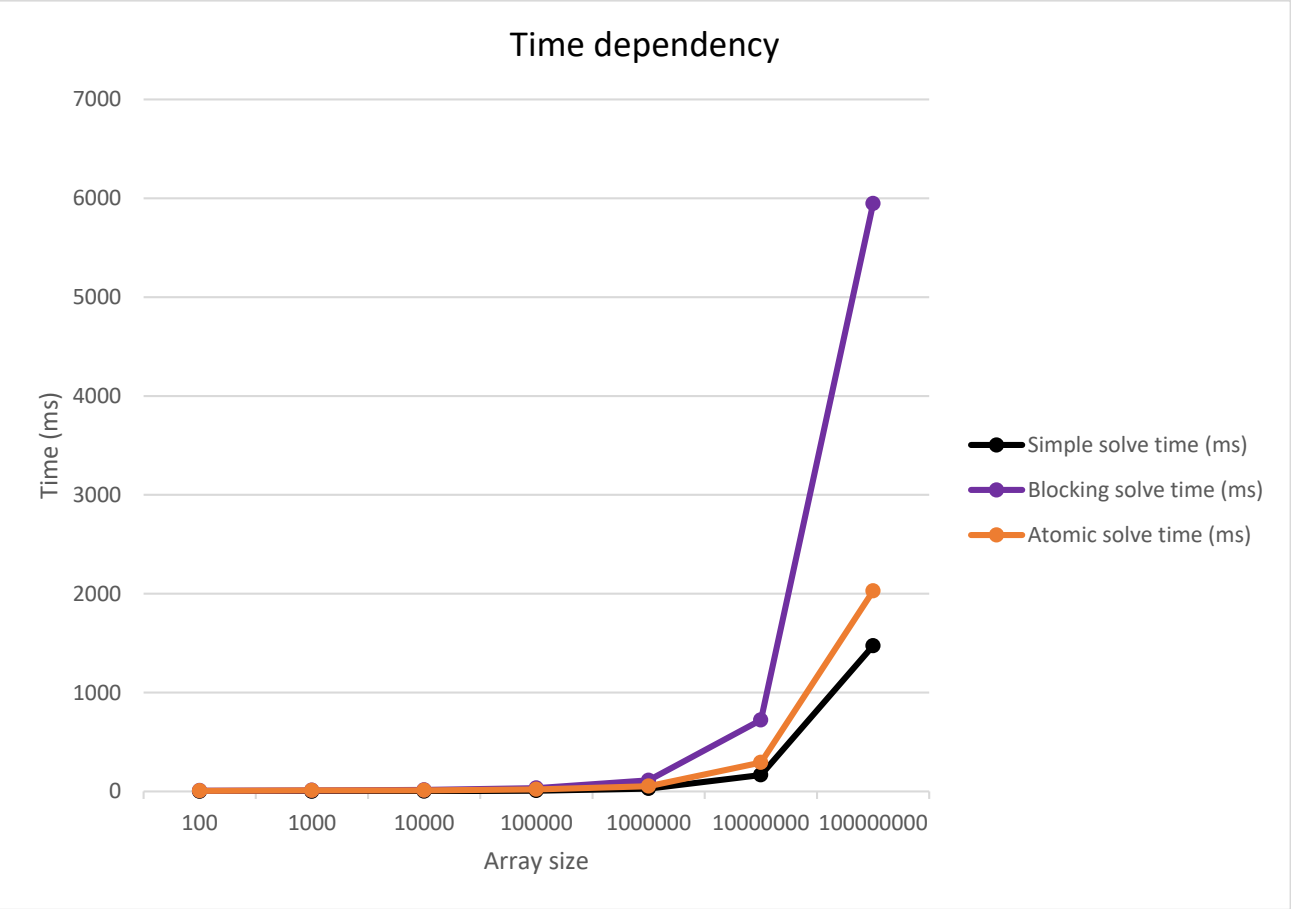
За аналізом скріншоту результату виконання програми можна побачити, що, у випадку поточного рішення з використанням потоків та атомарних змінних і самивів, програма аналогічно самостійно визначила, що може використовувати 8 вільних потоків для обробки створеного масиву на мільйон довільних цілих чисел. Програма завершила своє виконання за 50 мілісекунд та визначила, що найбільшим непарним елементом знову є 7999, а загальна кількість непарних чисел в масиві тепер більша за половину – 500,144.

Повний код рішення з використанням потоків та атомарності наведено у додатку В.

Аналіз результатів роботи програм

Результати роботи програм по часу залежно від розміру масивів представлено у таблиці нижче та на графіці.

Array size	100	1000	10000	100000	1000000	10000000	100000000
Simple solve time (ms)	0	0	1	7	28	166	1473
Blocking solve time (ms)	7	10	14	33	111	722	5946
Atomic solve time (ms)	5	9	10	20	52	292	2027



Як можна побачити, для такого типу завдання найкращим варіантом навіть для дуже великих масивів є рішення без використання потоків, друге місце посідає рішення з використанням потоків та атомарності. Найгірші результати продемонструвало рішення з використанням потоків та блокуючих

примітивів. Варто нагадати, що такі результати є дещо очікуваними, оскільки впродовж виконання програм замірювалась не лише частина з підрахунком кількості непарних значень та пошуком найбільшого з них, а й створення потоків, розподілення масиву на опрацювання потоками, зібрання результатів роботи потоків разом, тощо.

Висновок

У ході виконання третьої лабораторної роботи було розглянуто поняття атомарності, атомарних змінних та операцій, неблокуючого алгоритму. В рамках необхідності реалізувати програму підрахунку кількості непарних чисел та найбільшого непарного числа в масиві було здобуто навички роботи з атомарними змінними та методами.

За результатами роботи можна сказати, що атомарність дійсно переважно варто використовувати в моментах, коли для програми необхідна не правильна послідовність операцій потоків, а правильність обчислень, оскільки атомарні змінні не дають доступ до себе для багатьох потоків водночас, що забезпечує неможливість зміни змінної якимось потоком у той час, поки інший потік теж намагається провести з цією змінною операції. Безперечно, використання атомарності для створення потокобезпечних змінних є більш привабливим рішенням, аніж використання блокуючих примітивів, причому не тільки через покращений час роботи програми, а й через відносну простоту реалізації атомарності та її методів у Java. Єдине, необхідно бути доволі уважними в процесі написання коду з використанням атомарності, оскільки не абсолютно усі операції з атомарними змінними також є атомарними.

Підсумовуючи результати замірів часу в залежності від розміру створюваного масиву, можна сказати, що для даного завдання найбільш оптимальним рішенням є використання безпотокової реалізації, але також варто зазначити, що заміри починались з ініціалізації масиву та його заповнення довільними цілими числами, а закінчувались лише безпосередньо перед виводом статистики до консолі користувача, тому результати з використанням потоків більш-менш очевидні, адже витрачається час на створення пулу потоків.

Посилання на гіт: https://github.com/balumatkina/parallel-computing-kpi.git

Додаток А

```
import java.util.Random;

public class Main {

    public static final String ANSI_RESET = "\u001B[0m";
    public static final String ANSI_PURPLE = "\u001B[35m";

    public static final int ARRAY_SIZE = 1000000;
    public static final int MAX_ARRAY_ELEMENT_VALUE = 8000;

    public static void main(String[] args) {
        System.out.println(ANSI_PURPLE +
                "TASK: Find the number of odd elements in an array and the
largest odd number"
                + ANSI_RESET);
        System.out.println("Array size: " + ARRAY_SIZE);

        Random random = new Random();
        long startTime = System.nanoTime();
        int[] array = new int[ARRAY_SIZE];
        for (int i = 0; i < ARRAY_SIZE; i++) {
            array[i] = random.nextInt(MAX_ARRAY_ELEMENT_VALUE + 1);
        }

        int count = 0;
        int maxOdd = Integer.MIN_VALUE;
        for (int i = 0; i < ARRAY_SIZE; i++) {
            if (array[i] % 2 != 0) {
                count++;
                if (array[i] > maxOdd) {
                    maxOdd = array[i];
                }
            }
        }
        long endTime = System.nanoTime();
        long duration = endTime - startTime;

        System.out.println();
        System.out.println("Task completed in: " + duration +
                " nanoseconds (" + duration / 1000000 + " milliseconds)");
        System.out.println();

        System.out.println("Number of odd elements: " + count);
        if (maxOdd != Integer.MIN_VALUE) {
            System.out.println("Largest odd number: " + maxOdd);
        } else {
            System.out.println("No odd numbers found in array");
        }
    }
}
```

Додаток Б

```
import java.util.Random;

public class Main {

    public static final String ANSI_RESET = "\u001B[0m";
    public static final String ANSI_PURPLE = "\u001B[35m";

    public static final int ARRAY_SIZE = 1000000;
    public static final int MAX_ARRAY_ELEMENT_VALUE = 8000;

    public static void main(String[] args) {
        Random random = new Random();
        System.out.println(ANSI_PURPLE +
                "TASK: Find the number of odd elements in an array and the
largest odd number"
                + ANSI_RESET);
        System.out.println("Array size: " + ARRAY_SIZE);
        System.out.println();

        long startTime = System.nanoTime();
        int[] array = new int[ARRAY_SIZE];
        for (int i = 0; i < ARRAY_SIZE; i++) {
            array[i] = random.nextInt(MAX_ARRAY_ELEMENT_VALUE + 1);
        }

        OddNumberCounter counter = new OddNumberCounter(array);
        counter.countOdds();

        long endTime = System.nanoTime();
        long duration = endTime - startTime;

        System.out.println("Amount of used threads: " + counter.nThreads);
        System.out.println();
        System.out.println("Task completed in: " + duration +
                " nanoseconds (" + duration / 1000000 + " milliseconds)");
        System.out.println("Waiting lock time: " + counter.mutexWaitTime+ "
nanoseconds ("
                + counter.mutexWaitTime / 1000000 + " milliseconds)");
        System.out.println();
        System.out.println("Number of odd elements: " + counter.getOddCount());
        System.out.println("Largest odd number: " + counter.getMaxOdd());
    }
}
```

```
import java.util.concurrent.*;

public class OddNumberCounter {
    private int[] array;
    private int oddCount;
    private int maxOdd;
    private Semaphore mutex;
    public long mutexWaitTime;
    public int nThreads;

    public OddNumberCounter(int[] array) {
        this.array = array;
```

```
        this.oddCount = 0;
        this.maxOdd = 0;
        this.mutex = new Semaphore(1);
        this.nThreads = Runtime.getRuntime().availableProcessors();
    }

    public int getOddCount() {
        return oddCount;
    }

    public int getMaxOdd() {
        return maxOdd;
    }

    public void countOdds() {
        ExecutorService executor = Executors.newFixedThreadPool(nThreads);

        int chunkSize = array.length / nThreads;
        for (int i = 0; i < nThreads; i++) {
            int startIndex = i * chunkSize;
            int endIndex = (i == nThreads - 1) ? array.length : (i + 1) *
chunkSize;
            executor.submit(new OddNumberCounterTask(startIndex, endIndex));
        }

        executor.shutdown();

        try {
            executor.awaitTermination(Long.MAX_VALUE, TimeUnit.NANOSECONDS);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }



    private class OddNumberCounterTask implements Runnable {
        private int startIndex;
        private int endIndex;

        public OddNumberCounterTask(int startIndex, int endIndex) {
            this.startIndex = startIndex;
            this.endIndex = endIndex;
        }

        @Override
        public void run() {
            for (int i = startIndex; i < endIndex; i++) {
                if (array[i] % 2 == 1) {
                    long startWaitTime = System.nanoTime();
                    try {
                        mutex.acquire();
                        long endWaitTime = System.nanoTime();
                        mutexWaitTime += endWaitTime - startWaitTime;
                        oddCount++;
                        if (array[i] > maxOdd) {
                            maxOdd = array[i];
                        }
                    } catch (InterruptedException e) {
```

```
                        e.printStackTrace();
                    } finally {
                        mutex.release();
                    }
                }
            }
        }
    }
}
```

Додаток В

```
import java.util.Random;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.atomic.AtomicInteger;
import java.util.concurrent.atomic.AtomicIntegerArray;

public class Main {

    public static final String ANSI_RESET = "\u001B[0m";
    public static final String ANSI_PURPLE = "\u001B[35m";

    public static final int ARRAY_SIZE = 1000000;
    public static final int MAX_ARRAY_ELEMENT_VALUE = 8000;

    public static void main(String[] args) throws InterruptedException {
        System.out.println(ANSI_PURPLE +
                "TASK: Find the number of odd elements in an array and the
largest odd number"
                + ANSI_RESET);
        System.out.println("Array size: " + ARRAY_SIZE);

        Random random = new Random();

        int numThreads = Runtime.getRuntime().availableProcessors(); // number
of threads to use
        int totalOddCount = 0;

        AtomicIntegerArray oddNumbCount = new AtomicIntegerArray(numThreads);
        AtomicInteger maxOddElement = new AtomicInteger(0);

        long startTime = System.nanoTime();
        ExecutorService threadPool = Executors.newFixedThreadPool(numThreads);
        int[] array = new int[ARRAY_SIZE];
        for (int i = 0; i < ARRAY_SIZE; i++) {
            array[i] = random.nextInt(MAX_ARRAY_ELEMENT_VALUE + 1);
        }

        for (int i = 0; i < numThreads; i++) {
            final int threadID = i;
            threadPool.execute(() -> {
//                int localMax = 0;
                for (int j = threadID; j < array.length; j += numThreads) {
                    int num = array[j];
                    if (num % 2 != 0) {
                        // increment count for this thread
                        oddNumbCount.getAndIncrement(threadID);
                        while (true) {
                            int currMax = maxOddElement.get();
                            if (num > currMax) {
                                if (maxOddElement.compareAndSet(currMax, num)) {
//                                    localMax = num;
                                    break;
                                }
                            } else {
                                break;
```

```
                            }
                        }
                    }
                }
//                  System.out.println("Thread " + threadID + " finished with max
odd number: " + localMax);
            });
        }

        threadPool.shutdown();
        threadPool.awaitTermination(Long.MAX_VALUE, TimeUnit.MILLISECONDS);

        for (int i = 0; i < numThreads; i++) {
            totalOddCount += oddNumbCount.get(i);
        }

        long endTime = System.nanoTime();
        long duration = endTime - startTime;

        System.out.println("Threads amount: " + numThreads);
        System.out.println();
        System.out.println("Task completed in: " + duration +
                " nanoseconds (" + duration / 1000000 + " milliseconds)");
        System.out.println("Number of odd elements: " + totalOddCount);
        System.out.println("Largest odd number: " + maxOddElement.get());
    }
}
```