

Міністерство освіти і науки України
Національний технічний університет України
«Київський політехнічний інститут ім. Ігоря Сікорського»
Інститут прикладного системного аналізу
Кафедра системного проектування

Лабораторна робота №1
з дисципліни «Паралельні обчислення»
на тему: «Дослідження базових операцій з потоками виконання»

Виконала:
студентка III курсу, групи ДА-01
Балуматкіна О. В.
Варіант 4
Прийняла(в):

Київ-2023

Зміст

Мета роботи	3
Завдання	3
Варіант 4	4
Хід роботи	4
<i>Базові характеристики ноутбуку</i>	4
<i>Механізм заміру часу</i>	5
<i>Вирішення завдання без паралелізації</i>	6
<i>Вирішення завдання з паралелізацією</i>	7
<i>Паралелізація з різною кількістю процесів</i>	9
<i>Паралелізація з різною кількістю процесів та даних</i>	10
Висновок	12
Додаток А	14
Додаток Б	16

Мета роботи

Розглянути основні операції з потоками виконання, навчитися використовувати неблокуючу паралелізацію для вирішення найпростіших математичних задач, використовуючи обрану мову програмування. Навчитися досліджувати та оцінювати ефективність паралелізації алгоритму.

Завдання

1. Визначити основні характеристики ПК, котрі впливають на ефективність виконання паралельних обчислень. Зафіксувати значення даних характеристик для ПК студента.
2. Створити механізм, котрий може бути використаний для заміру часу виконання програми, або інших параметрів, котрі студент вважає релевантними.
3. Вирішити обрану за варіантом задачу, не використовуючи паралелізацію. Заміряти час вирішення задачі, або інші параметри, котрі студент вважає релевантними.
4. Вирішити обрану за варіантом задачу, використовуючи паралелізацію. Заміряти час вирішення задачі, або інші параметри, котрі студент вважає релевантними. Обґрунтувати вибір алгоритму паралелізації.
5. Повторити пункт 4 з використання різної кількості процесів виконання. Обов'язково перевірити виконання задачі на фіксованих кількостях потоків: 2-рази меншій, ніж кількість фізичних ядер, на кількості рівній фізичним ядрам, на кількості рівній логічних ядрам, на кількості більшій в 2, 4, 8, 16 разів ніж кількість логічних ядер.
6. Повторити пункт 5 з використанням різної розмірності даних, в залежності від обраної задачі.
7. Заповнити таблицю й зробити графік часу виконання завдання від кількості потоків для різних розмірностей.

Варіант 4

Створити вектор з $N \geq 1000$ елементами випадкових чисел. Знайти моду та медіану цього вектору.

Хід роботи

Базові характеристики ноутбуку

На швидкість виконання паралельних завдань у будь-якому ПК впливають наступні характеристики:

1. Кількість процесорів у ПК;
2. Швидкість процесора(ів);
3. Кількість ядер у процесорі(ах);
4. Обсяг оперативної пам'яті ПК;
5. Швидкість обміну даними між процесорами;
6. Кількість КЕШ пам'яті;
7. Рівень оптимізації програмного забезпечення для виконання паралельних обчислень.

Оскільки під час виконання даної лабораторної роботи було використано ноутбук середньої цінової категорії, цілком логічно, що перший та п'ятий пункти можна відкинути, оскільки пристрій має у собі лише один процесор - Intel® Core™ i5-10210U. Загальні характеристики для процесору наведені нижче:

Total Cores ?	4
Total Threads ?	8
Max Turbo Frequency ?	4.20 GHz
Processor Base Frequency ?	1.60 GHz
Cache ?	6 MB Intel® Smart Cache

Рисунок 1 – Характеристики процесору пристрою

Оскільки сучасні моделі процесорів активно підтримують функцію `hyper-threading`, в характеристиках вказано як кількість фізичних ядер (4), так і кількість логічних ядер (8), тобто можна зробити висновки про те, що найбільш оптимальною кількістю потоків для виконання завдань з паралельних обчислень на даному пристрої, буде вісім потоків. Також у характеристиках зазначена максимальна та базова частота процесора, проте відомо, що процесор не завжди функціонує виключно на цих значеннях, тож перед виконанням програми та отриманням часу виконання буде зазначатися, яка швидкість процесору була до виконання коду, та до якого значення вона піднялась.

Оперативної пам'яті на даному пристрої складає 8 ГБ, з них доступні для використання 7.81 ГБ.

Механізм заміру часу

Для заміру часу виконання розробленої на мові програмування Java програми буде використовуватись `java.lang.System.nanoTime()` метод, що повертає поточне значення таймеру в наносекундах.

Було прийнято рішення проводити окремо заміри для функції пошуку моди в створеному векторі, та окремо для пошуку медіани. Загальна кількість витраченого на виконання цих двох найважливіших функцій все одно буде виведено сумою в консоль. Рішення про окремі заміри часу на кожну функцію були обрані з огляду на те, що завдання паралелізації буде виконано виключно на процесі пошуку медіани в створеному векторі, тому можна буде наочно побачити, як час виконання обчислення моди в обох розроблених програмах особливо не зміниться, а час підрахунку медіани буде змінюватись залежно, як буде потім видно по ходу виконання лабораторної, від обраної кількості потоків та того, наскільки це число більше або менше за оптимальну кількість.

Вирішення завдання без паралелізації

З точки вирішення без паралелізації, завдання доволі просте: створення вектору на тисячу чи більше елементів, заповнення вектору довільними значеннями, та пошук моди та медіани створеного вектору. Було обрано саме Vector, а не ArrayList, оскільки доступ до елементів вектору є синхронізованим та більш безпечним з точки зору спроби використання паралелізації обчислень. З точки зору швидкості виконання програми, в силу наявності у Vector спеціальних функцій по синхронізації, код з використанням Vector буде виконуватись повільніше, ніж код з ArrayList, проте міри безпеки коштують цього часу.

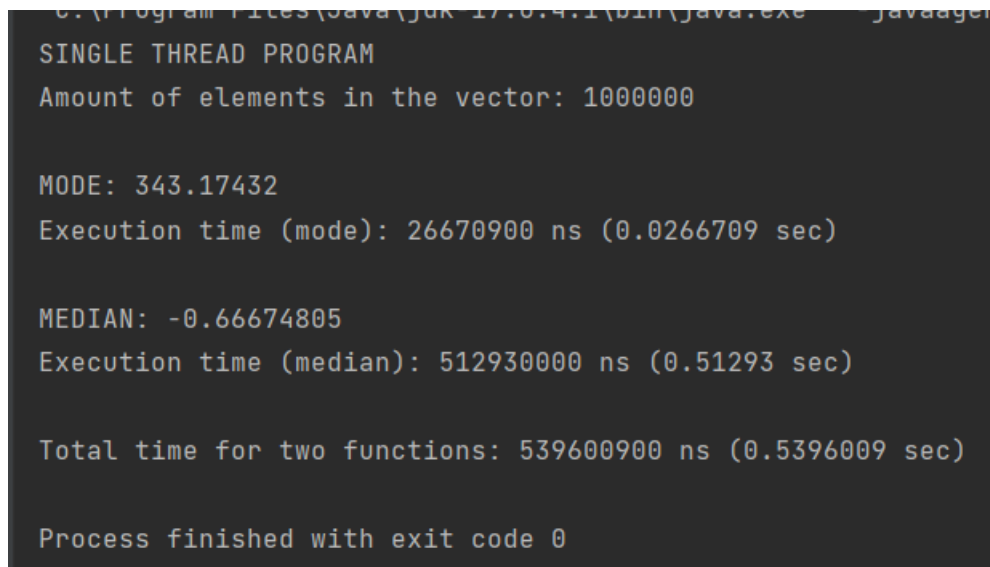
Для виконання завдання було вирішено створювати вектор значень float, заповнювати його довільними значеннями у діапазоні $[-1000;1000]$. Заповнення вектору відбувається через функцію generateRandomVector, що повертає вектор встановленого розміру, заповнений випадковими значеннями з плаваючою точкою.

Можемо створити окремі класи для розрахунку моди та медіани – ModeCalculator та MedianCalculator відповідно. Після створення та заповнення вектору, створимо об'єкт класу для пошуку моди зі згенерованим вектором, після створення почнемо замір часу та викликаємо функцію вищезгаданого класу calculateMode(), яка, відповідно до алгоритму більшості голосів Бойєра-Мура, складність якого оцінюється в $O(n)$, поверне результат у вигляді останньої знайденої моди (або єдиної моди, на випадок, якщо у векторі відсутня ситуація з декількома модами). Після отримання результату у вигляді моди вектору, припиняємо вимірювання. Для реалізації пошуку моди можна було б скористатися HashMap або Array для зберігання значень та частоти їх виникнення у векторі, це б також відкрило можливість виводити у результат набір мод, на випадок, якщо у векторі все ж таки більше за одну моду, проте використання цих

структур робить виконання програми довшим, тому було прийнято рішення вважати, що в створюваному векторі існує лише одна унікальна мода.

Логічно аналогічні дії виконуємо з класом для пошуку медіани – MedianCalculator. Клас, при виклику його методу calculateMedian(), відсортовує отриманий в об'єкт вектор з використанням Merge Sort через Collections.sort(), а потім просто намагається отримати доступ до індексу та значення центрального елемента (у випадку, якщо довжина отриманого вектора є непарним числом), або отримує значення двох центральних елементів та повертає їхнє середнє значення (у випадку, якщо довжина вектора є парним числом).

Отже, перед виконанням створеної програми (без паралелізації) частота процесора складає 1.19ГГц, при виконанні програми швидкість піднялась до 3.73ГГц. Кількість елементів у векторі – 1 000 000.



```
0. (Program Files) java (jdk-17.0.4\bin) java.exe javaagen
SINGLE THREAD PROGRAM
Amount of elements in the vector: 1000000

MODE: 343.17432
Execution time (mode): 26670900 ns (0.0266709 sec)

MEDIAN: -0.66674805
Execution time (median): 512930000 ns (0.51293 sec)

Total time for two functions: 539600900 ns (0.5396009 sec)

Process finished with exit code 0
```

Рисунок 2 – Час виконання не паралелізованої програми

Вирішення завдання з паралелізацією

Для застосування паралелізації було обрано функцію пошуку медіани по створеному вектору, оскільки вона є більш оптимальною за пошук моди, та не вимагає поширення між потоками спільної інформації, що робить загальну

реалізацію багатопотоковості виконання легшим. Також можна було звернути увагу на можливість паралелізації заповнення вектору довільними значеннями з плаваючою точкою, проте, все ж таки, з самого початку виконання даної лабораторної роботи, цьому завданню не надається багато уваги, тому вибір спинився на введенні паралельних обчислень в пошук медіани вектору.

Для створення потоків, що будуть проводити обчислення над вектором, було використано спеціальний інтерфейс `ExecutorService`, що дозволяє в полегшеному форматі створювати потрібну кількість потоків для виконання програми, мінімізує шанси виникнення накладних витрат на самостійне створення та знищення потоків, дозволяє краще контролювати використання доступних ресурсів. Разом з `ExecutorService` було використано інтерфейс `Callable`, щоб визначити, які завдання можуть бути передані на виконання (було обрано саме `Callable`, оскільки в ньому реалізовано метод `call()`, що здатний повертати результати виконаного коду, на відміну від `run()` у `Runnable`), а також було використано інтерфейс `Future`, що й використовується для передачі від `Callable` до `ExecutorService` доступних до виконання завдань. В подальшому результати, що зберігаються в `Future`, будуть використані для об'єднання в єдиний масив медіан з подальшим пошуком медіани медіан з використанням алгоритму паралельного скорочення (`parallel reduction algorithm`).

Отже, перед виконанням створеної програми (з паралелізацією) частота процесора складає 1.81ГГц, при виконанні програми швидкість піднялась до 3.41ГГц. Кількість елементів у векторі – 1 000 000, обрана кількість потоків – 4.

Як було зазначено у виборі методу заміру часу, можна чітко побачити, що для обох реалізованих програм час на виконання пошуку моди не дуже сильно відрізняється, оскільки обидві програми не використовують паралелізацію для виконання цього завдання. Відмінності стає одразу видно при замірі часу на

виконання завдання з підрахунку медіани вектору та, закономірно, при обрахунках загального часу, витраченого на виконання створеної програми.

```

MULTIPLE THREAD PROGRAM
Amount of threads: 4
Amount of elements in the vector: 1000000

MODE: -632.21313
Execution time (mode): 24939600 ns (0.0249396 sec)

MEDIAN: 1.1690979
Execution time (median): 358923700 ns (0.3589237 sec)

Total time for two functions: 383863300 ns (0.3838633 sec)

Process finished with exit code 0

```

Рисунок 3 – Час виконання паралелізованої програми

Паралелізація з різною кількістю процесів

Процесор пристрою, на якому було виконано дану лабораторну роботу, містить 4 фізичні ядра та 8 логічних, тож, відповідно до завдання, спробуємо виконати додаткові заміри виконання програми на фіксованих кількостях потоків та двох довільних, кількість елементів у векторі залишатиметься стабільною. Занесемо результати виконання програми до таблиці та зробимо висновки.

Таблиця 1

Результати виконання програми в наносекундах

Thread amount	Mode calculation, ns	Median calculation, ns	Total execution, ns
2	23514900	315489200	339004100
4	25737200	373756800	399494000
8	22904700	320052100	342956800
16	24151400	354747400	378898800
32	25640300	327892700	353533000
64	24481900	244873500	269355400
128	23069000	248898300	271967300
200	28547400	248744700	277292100
1000	24003000	267197300	291200300

Результати виконання програми в секундах

Thread amount	Mode calculation, s	Median calculation, s	Total execution, s
2	0,0235149	0,3154892	0,3390041
4	0,0257372	0,3737568	0,399494
8	0,0229047	0,3200521	0,3429568
16	0,0241514	0,3547474	0,3788988
32	0,0256403	0,3278927	0,353533
64	0,0244819	0,2448735	0,2693554
128	0,023069	0,2488983	0,2719673
200	0,0285474	0,2487447	0,2772921
1000	0,024003	0,2671973	0,2912003

Як можна помітити, було відмічено значення, що надають найкращі результати по швидкості обчислення медіани вектору та, як наслідок, найкращий час загального виконання обох важливих обчислень програми. Зі збільшенням кількості потоків, збільшується час виконання програми, що використовується на створення на роботу вже абсолютно непотрібних додаткових потоків. Так само з замалими значеннями кількості потоків, вони не дають програмі використати повний допустимий потенціал пристрою.

Паралелізація з різною кількістю процесів та даних

Повторимо обчислення з аналогічною варіацією кількості потоків, проте також будемо змінювати кількість елементів всередині вектора, який створюється програмою, що може потягнути за собою відповідні прискорення та сповільнення виконання. Результати для загального часу виконання програми (включаючи і пошук моди, і пошук медіани) занесемо до таблиці, зробимо висновки.

Таблиця 3

Результати виконання програми в наносекундах

Thread amount	1000	10000	100000	1000000	10000000
2	7389200	17859000	73138400	362700600	1540156800
4	5663800	15610400	54785100	327561700	1388406900
8	6207500	13109700	47005600	357480200	1282358900
16	8635300	13714700	44588700	365693300	2251358800
32	9986000	14603800	44152100	354156900	2426579100
64	15714200	16589200	42556200	305280300	2668756700
128	21984100	29615100	44145400	266131900	2274235600
200	30119600	39611300	55489900	273273100	3043796400
1000	108354000	120840500	171678900	299930400	1933664800

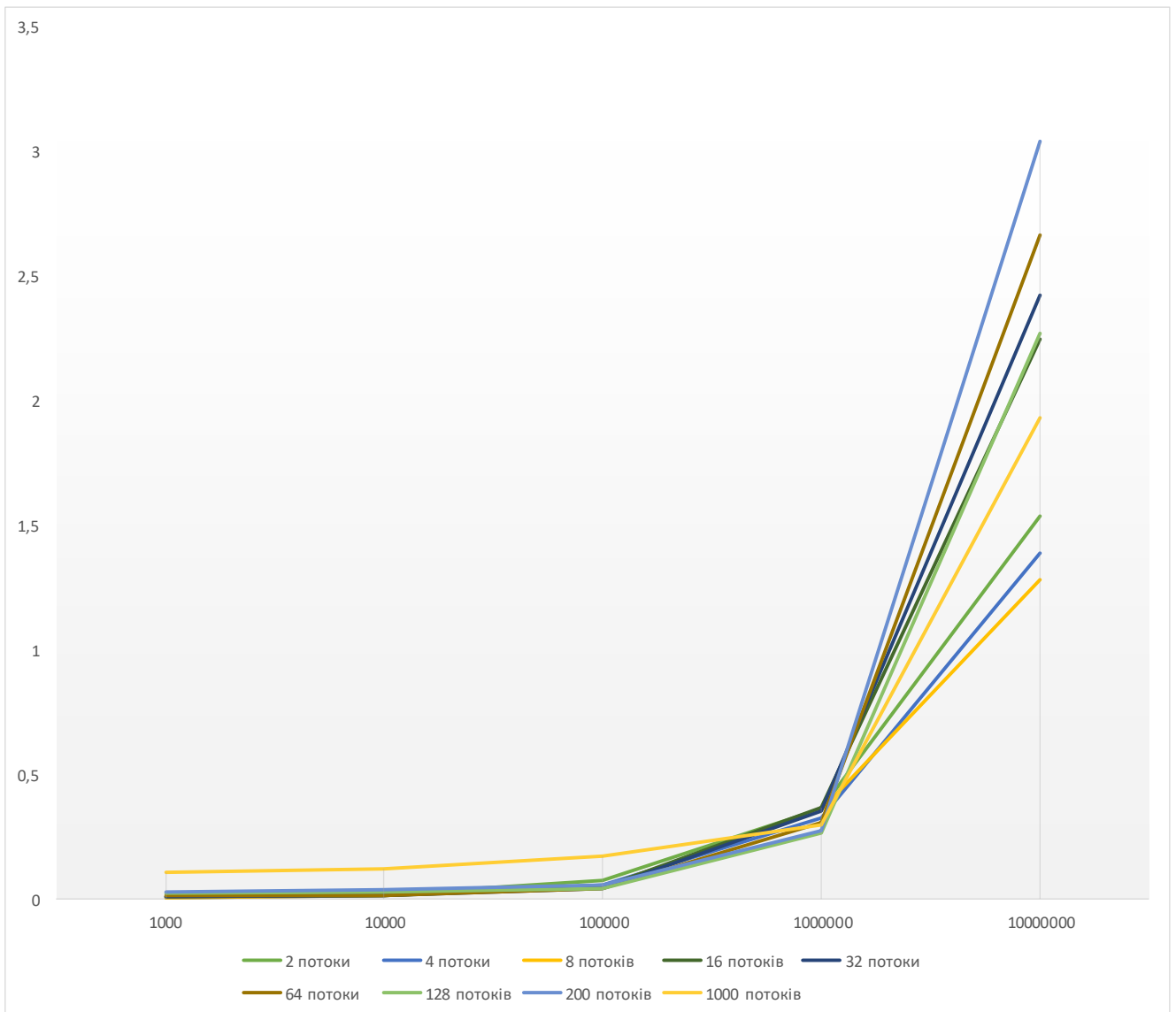
Таблиця 4

Результати виконання програми в секундах

Thread amount	1000	10000	100000	1000000	10000000
2	0,0073892	0,017859	0,0731384	0,3627006	1,5401568
4	0,0056638	0,0156104	0,0547851	0,3275617	1,3884069
8	0,0062075	0,0131097	0,0470056	0,3574802	1,2823589
16	0,0086353	0,0137147	0,0445887	0,3656933	2,2513588
32	0,009986	0,0146038	0,0441521	0,3541569	2,4265791
64	0,0157142	0,0165892	0,0425562	0,3052803	2,6687567
128	0,0219841	0,0296151	0,0441454	0,2661319	2,2742356
200	0,0301196	0,0396113	0,0554899	0,2732731	3,0437964
1000	0,108354	0,1208405	0,1716789	0,2999304	1,9336648

Кольором відмічено найкращі значення, отримані для встановленої n кількості елементів у векторі.

Створимо графік на основі отриманих таблиць.



Висновок

У ході виконання першої лабораторної роботи було досліджено поведінку потоків та вивчено методи роботи з потоками на мові програмування Java. Було реалізовано дві програми для виконання завдань лабораторної роботи: одна виконує поставлене завдання без використання паралелізму, інша використовує легкозмінну кількість потоків для виконання обчислень по пошуку медіани створеного вектора, заповненого довільними значеннями з плаваючою точкою.

По часовим результатам, занесеним до таблиць та виведеним на графік, можна зробити наступні висновки:

1. Кількість потоків, в оптимальному варіанті, має бути виваженою до кількості опрацьовуваних даних;
2. Посилаючись на попередній пункт, замала чи зовелика кількість потоків не гарантуватиме швидке виконання програми, оскільки, за умови замалої кількості потоків, програма не використовує увесь потенціал пристрою, а за умови зовеликої кількості потоків – здійснюється створення та виконання вже не потрібних ніяким чином для обчислень потоків, що доволі сильно загалом впливають на час виконання програми.

```
import java.util.Random;
import java.util.Vector;

public class SingleThread {
    private static final int VECTOR_SIZE = 1000000;
    private static final float MIN_VALUE = -1000;
    private static final float MAX_VALUE = 1000;

    public static void main(String[] args) {
        Vector<Float> vector = generateRandomVector(VECTOR_SIZE, MIN_VALUE,
MAX_VALUE);

        ModeCalculator modeCalculator = new ModeCalculator(vector);
        long startTime = System.nanoTime();
        float mode = modeCalculator.calculateMode();
        long modeTime = System.nanoTime() - startTime;

        MedianCalculator medianCalculator = new MedianCalculator(vector);
        startTime = System.nanoTime();
        float median = medianCalculator.calculateMedian();
        long medianTime = System.nanoTime() - startTime;

        System.out.println("SINGLE THREAD PROGRAM");
        System.out.println("Amount of elements in the vector: " + VECTOR_SIZE +
"\n");

        System.out.println("MODE: " + mode);
        System.out.println("Execution time (mode): " + modeTime + " ns (" +
modeTime / 1e9 + " sec)\n");

        System.out.println("MEDIAN: " + median);
        System.out.println("Execution time (median): " + medianTime + " ns (" +
medianTime / 1e9 + " sec)\n");

        long totalTime = (modeTime + medianTime);
        System.out.println("Total time for two functions: " + totalTime + " ns
(" + totalTime / 1e9 + " sec)");
    }

    private static Vector<Float> generateRandomVector(int size, float min, float
max) {
        Random random = new Random();
        Vector<Float> vector = new Vector<>(size);
        for (int i = 0; i < size; i++) {
            vector.add(min + random.nextFloat() * (max - min));
        }
        return vector;
    }
}
```

```
import java.util.Collections;
import java.util.Vector;

public class MedianCalculator {
```

```

private Vector<Float> vector;

public MedianCalculator(Vector<Float> vector) {
    this.vector = vector;
}

public float calculateMedian() {
    Collections.sort(vector);
    int vectorSize = vector.size();
    int middle = vectorSize / 2;
    if (vectorSize % 2 == 0) {
        return (vector.get(middle - 1) + vector.get(middle)) / 2;
    } else {
        return vector.get(middle);
    }
}
}

```

```

import java.util.Vector;

public class ModeCalculator {

    private Vector<Float> vector;

    public ModeCalculator(Vector<Float> vector) {
        this.vector = vector;
    }

    public float calculateMode() {
        float candidate = 0;
        int count = 0;
        for (float value : vector) {
            if (count == 0) {
                candidate = value;
                count = 1;
            } else if (value == candidate) {
                count++;
            } else {
                count--;
            }
        }
        return candidate;
    }
}

```

```

import java.util.*;

public class MultipleThread {
    private static final int VECTOR_SIZE = 10000000;
    private static final int THREADS_AMOUNT = 1000;
    private static final float MIN_VALUE = -1000;
    private static final float MAX_VALUE = 1000;

    public static void main(String[] args) {
        Vector<Float> vector = generateRandomVector(VECTOR_SIZE, MIN_VALUE,
MAX_VALUE);

        ModeCalculator modeCalculator = new ModeCalculator(vector);
        long startTime = System.nanoTime();
        float mode = modeCalculator.calculateMode();
        long modeTime = System.nanoTime() - startTime;

        MedianCalculator medianCalculator = new MedianCalculator(vector,
THREADS_AMOUNT);
        startTime = System.nanoTime();
        float median = medianCalculator.calculateMedianParallel();
        long medianTime = System.nanoTime() - startTime;

        System.out.println("MULTIPLE THREAD PROGRAM");
        System.out.println("Amount of threads: " + THREADS_AMOUNT);
        System.out.println("Amount of elements in the vector: " + VECTOR_SIZE +
"\n");

        System.out.println("MODE: " + mode);
        System.out.println("Execution time (mode): " + modeTime + " ns (" +
modeTime / 1e9 + " sec)\n");

        System.out.println("MEDIAN: " + median);
        System.out.println("Execution time (median): " + medianTime + " ns (" +
medianTime / 1e9 + " sec)\n");

        long totalTime = modeTime + medianTime;
        System.out.println("Total time for two functions: " + totalTime + " ns
(" + totalTime/1e9 + " sec)");
    }

    private static Vector<Float> generateRandomVector(int size, float min, float
max) {
        Random random = new Random();
        Vector<Float> vector = new Vector<>(size);
        for (int i = 0; i < size; i++) {
            vector.add(min + random.nextFloat() * (max - min));
        }
        return vector;
    }
}

```

```

import java.util.Arrays;
import java.util.List;
import java.util.Vector;

```



```

import java.util.concurrent.*;

public class MedianCalculator {
    private final Vector<Float> vector;
    private final int nCores;

    public MedianCalculator(Vector<Float> vector, int nCores) {
        this.vector = vector;
        this.nCores = nCores;
    }

    public float calculateMedianParallel() {
        int vectorSize = vector.size();
        int chunkSize = vectorSize / nCores;
        int i = 0;

        //chunks aka parts of vector, minecraft hello
        Vector<List<Float>> chunks = new Vector<>();
        while (i < vectorSize) {
            int j = i + chunkSize;
            if (j > vectorSize) {
                j = vectorSize;
            }
            chunks.add(vector.subList(i, j));
            //take parts from sublist, copy to chunks
            i = j;
        }

        ExecutorService executor = Executors.newFixedThreadPool(nCores);
        Vector<Future<Float>> futures = new Vector<>();
        for (final List<Float> chunk : chunks) {
            Callable<Float> task = () -> {
                float[] chunkArray = new float[chunk.size()];
                for (int i1 = 0; i1 < chunk.size(); i1++) {
                    chunkArray[i1] = chunk.get(i1);
                }
                Arrays.sort(chunkArray);
                int middle = chunkArray.length / 2;
                if (chunkArray.length % 2 == 0) {
                    return (chunkArray[middle - 1] + chunkArray[middle]) / 2;
                } else {
                    return chunkArray[middle];
                }
            };
            futures.add(executor.submit(task));
        }

        executor.shutdown();

        //future.get() instead of join() + all the exceptions from future and
        //callable
        Vector<Float> medians = new Vector<>();
        for (Future<Float> future : futures) {
            try {
                medians.add(future.get());
            } catch (InterruptedException | ExecutionException e) {
                e.printStackTrace();
            }
        }
    }
}

```

```

        //searching for median of medians with parallel reduction algorithm
        while (medians.size() > 1) {
            int n = medians.size();
            int m = (n + 1) / 2;
            for (int j = 0; j < m; j++) {
                medians.set(j, medians.get(j * 2));
            }
            medians.setSize(m);
        }
        return medians.get(0);
    }
}

```

```

import java.util.Vector;

public class ModeCalculator {
    private Vector<Float> vector;

    public ModeCalculator(Vector<Float> vector) {
        this.vector = vector;
    }

    public float calculateMode() {
        float candidate = 0;
        int count = 0;
        for (float value : vector) {
            if (count == 0) {
                candidate = value;
                count = 1;
            } else if (value == candidate) {
                count++;
            } else {
                count--;
            }
        }
        return candidate;
    }
}

```