1. How long it took your app to finish the job when using 1 thread? - 0.093 s

```
/Users/v/Library/Java/JavaVirtualMachines/openjdk-19.0.2/Contents/Home/bin/java
You have 12 cores avaiable. Test with (A) 1 core or (B) 12 cores?
A

Calculating divisor of 10000 ints with 1 threads
---------- Results ----------------
10000 numbers were tested with 1 threads
The number 7560 has the largest divisor of 64
Total time: 0.093 seconds


Process finished with exit code 0
```

2. How many threads your machine can run simultaneously?

My CPU has 6 cores enhanced with hyper-threading and therefore my OS recognize them as 12 (virtual/logical) threads, another words hyper-threading virtually doubles amount of cores that on a CPU (2 threads per core), so CPU can perform more tasks in the same amount of time. Hyper-threading not double physical cores but only double virtual/logical cores. In a nutshell my pc can run many applications while maintaining performance. In other words pc not going to slow down. Each core is like a worker, each worker do what operating system (like boss) tells them to do and threads like sequence of commands (1's and 0's) given to a cores. Its like conveyer belts of products sent to worker.

When 1 core does only 1 thread is less efficient then 1 core does 2 threads (multithreading) For example when one worker works on one conveyer belt and this conveyer belt is run out of products, worker switch to second conveyer belt.

3. How long it took your app to finish the job when using all available threads? -0.033

```
/Users/v/Library/Java/JavaVirtualMachines/openjdk-19.0.2/Contents/Home/bin/java
You have 12 cores avaiable. Test with (A) 1 core or (B) 12 cores?
B

Calculating divisor of 10000 ints with 12 threads
---------- Results ----------------
10000 numbers were tested with 12 threads
The number 7560 has the largest divisor of 64
Total time: 0.033 seconds


Process finished with exit code 0
```

4. Did you use task or data parallelization for this program? Why?

All the 6 cores work simultaneously (it is call parallel operation) but the cores themself can not work on multiple threads so cores themself going to need a switch between single threads at a time, so switching between threads is call concurrent execution execution. It mimic || execution.

```java
package com.company;

import java.util.Scanner;

public class Main {

    /*
     * Variables to track the largest number of divisors and its corresponding number
     * These are set as volatile because updates of one thread is not visible by
     * another thread in memory.
     * Volatile guarantees visibility immediately to other threads
     */
    public volatile static int maxDivisorsCount = 0;
    public volatile static int numberWithMaxDivisors = 0;

    /*
     * The main method will retrieve the number of cores that is available to the JCM
     * The user is given an option to utilize 1 core or all cores provided
     * Note: The range is hardcoded for the purpose of this assignment
```

```
     */
    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);

        int cores = Runtime.getRuntime().availableProcessors();
        int preferredCores = 1;
        int range = 10000;

        System.out.println("You have " + cores + " cores avaiable. Test
with (A) 1 core or (B) " +cores +" cores?");
        String result = input.next().toUpperCase();

        if(result.equals("A")){
            preferredCores = 1;
        } else if(result.equals("B")){
            preferredCores = 12;
        } else {
            System.out.println("Unknown input. Testing with " +cores+ "
core");
        }

        System.out.println("\nCalculating divisor of " +range+ " ints
with " +preferredCores+ " threads");
        countDivisorNumbers(range, preferredCores);
    }


    /*
```

```
     * (Note: The Static keyword is used to provide class level access to a method)
     * ---------------------------------------------
     * This method is responsible for creating and running the threads based on inputted range
     * and thread size by the user
     */
    public static void countDivisorNumbers(int range, int numOfThreads){

        long startTime = System.currentTimeMillis();

        /*
         * In this part, MultithreadingDivisorCount objects are created in an array that is the size of
         * the thread. Each object represents a group of numbers from the total range based on threads used
         * Check below for the implementation of the MultithreadingDivisorCount class
         */
        MultithreadingDivisorCount[] multithreadingDivisorCount = new MultithreadingDivisorCount[numOfThreads];
        int rangePerThread = range / numOfThreads; // Group sizes
        int start = 1;  // Initially, start at number 1
        int end = rangePerThread;

        for(int i=0; i<numOfThreads; i++) {
            multithreadingDivisorCount[i] = new MultithreadingDivisorCount(start, end);
            start = end + 1; // Start with the previous group size
```

```
            end =  start + (rangePerThread - 1); // calculate next ending group size
        }


        /*
         * Once we are done grouping the MultithreadingDivisorCount objects with different ranges,
         * we can begin to start the thread.
         */
        for(int i=0; i<numOfThreads; i++){
            multithreadingDivisorCount[i].start();
        }

        /*
         * After starting all the threads, we must wait for the threads to die. The join() methods
         * allow this by waiting for each thread to complete its execution. We can also ensure that
         * the join methods execute in order
         */
        for (int i = 0; i < numOfThreads; i++) {
            try {
                multithreadingDivisorCount[i].join();
            }
            catch (InterruptedException e) {
            }
        }
```

```
        /*
         * At this point, the threads have all successfully executed and we will now get the amount
         * of time it took to complete. The results are also displayed to the user
         */
        long endTime = System.currentTimeMillis() - startTime;
        System.out.println("----------- Results ----------------");
        System.out.println(range + " numbers were tested with " +numOfThreads+ " threads");
        System.out.println("The number " + numberWithMaxDivisors + " has the largest divisor of " + maxDivisorsCount);
        System.out.println("Total time: " + (endTime/1000.0) + " seconds\n");


    }


    /**
     * (Note: A Nested class is used to logically group classes used for similar purposes.
     * The Static keyword means that the class belongs specifically to outer class. It can be
     * called without instantiating the outer class)
     * ------------------------------------------
     * This class begins with start and end integers for a range of numbers declared in its constructor
```

```
    * The class extends Thread which allows it to be called using
start() and begins execution
    * of the run() method. Once the highest divisor and number is
retrieved, the results are set to the
    * static variables declared in the beginning of the Main class
    */
    public static class MultithreadingDivisorCount extends Thread {
        int start, end;

        public MultithreadingDivisorCount(int start, int end){
            this.start = start;
            this.end = end;
        }

        /*
         * Count the number of divisors for each integer
         * If the number evenly divides, then it has a divisor
         */
        public int countLargestDivisors(int number) {
            int divisorCount = 0;
            for (int i = 1; i <= number ; i++) {
                if ( number % i == 0 )
                    divisorCount ++;
            }
            return divisorCount;
        }
```

```java
        /**
         * This is the thread. It compares every number that is passed through the constructors start/end
         * range and passes it to countDivisors() which will get the divisors for that number. Similar to
         * finding min/max operation of two numbers, this finds the max number with the most divisor and sets
         * it to the static variable
         */
        public void run() {
            int largestDivisor = 0;
            int largestDivisorValue = 0;
            for (int i = start; i < end; i++) {
                int divisors = countLargestDivisors(i);
                if (divisors > largestDivisor) {
                    largestDivisor = divisors;
                    largestDivisorValue = i;
                }
            }
            // Compares the largest
            if (largestDivisor > maxDivisorsCount) {
                maxDivisorsCount = largestDivisor;
                numberWithMaxDivisors = largestDivisorValue;
            }
        }

    }
```