# IN3030, EKSAMEN V21, CANDIDATE: 15771

## Contents



## Introduction

IN3030 Exam spring 2021. Candidate: 15771.

I have added pictures of the source code in this document, but the source files are also added in the folder delivered and can be viewed from an editor to get a clearer picture.

## Q1: Caching and the Speed of light

### Q1.1: Speed of light

The speed of light in a vacuum is 299 792 458 m/s

### Q1.2: Caching

Access-time for memory is determined by the speed of the memory, and the distance between the memory and the CPU. Main memory is slow and “far” away from the CPU-core, and the CPU will waste cycles waiting for data to be fetched. Memory is expensive so its limits to how fast we can make the majority of memory, and the data also must travel, where limited to the speed-of-light through whatever medium the bus is made of. Therefore, latency hiding techniques such as small caches in a hierarchy, decreasingly slower and further away from the CPU, and pre-fetching is used.

## Q2 Variable Cyclic Barriers using Semaphores

### Q2.1: Variable Cyclic Buffer replacement

```
import java.util.concurrent.Semaphore;

public class semCyclicBarrier
{
    private int numOfThreads;
    private int numOfWaiting;
    private Semaphore lock;
    private Semaphore barrierOne;
    private Semaphore barrierTwo;

    public semCyclicBarrier(int numOfThreads)
    {
        this.numOfThreads = numOfThreads;
        this.numOfWaiting = 0;
        this.lock = new Semaphore( permits: 1);
        this.barrierOne = new Semaphore( permits: 0);
        this.barrierTwo = new Semaphore( permits: 1);
    }

    public void await()
    {
        try
        {
            lock.acquire();
            numOfWaiting++;
            System.out.println("Waiting threads increased to: " + numOfWaiting);

            if(numOfWaiting == numOfThreads)
            {
                System.out.println("All threads have reached the barrier.");
                barrierTwo.acquire();
                barrierOne.release();
            }

            lock.release();
            barrierOne.acquire();
            barrierOne.release();

            lock.acquire();
            numOfWaiting--;

            if(numOfWaiting == 0)
            {
                barrierOne.acquire();
                barrierTwo.release();
            }

            lock.release();

            barrierTwo.acquire();
            barrierTwo.release();
        }

        catch(Exception e)
        {
            e.printStackTrace();
        }
    }
}
```

The semCyclicBarrier is located in the file semCyclicBarrier.java and the test program is located in the Barrier.java, and the test-thread are implemented in the file testWorker.java. The semCyclicBarrier is an implementation of a CyclicBarrier using Semaphores. Its based on the implementation from the little book of semaphores: https://greenteapress.com/semaphores/LittleBookOfSemaphores.pdf. It contains the following variables, all set at initialization of the object:

- Int numOfThreads, the number of threads that will rendezvous at the barrier. For example, 6, if 5 threads and the main thread will rendezvous at the barrier.

- Int numOfWaiting, the number of already waiting threads, set to 0 at initialization. When this counter reaches the numOfThreads, all threads will be released from the barrier.

- Semaphore lock, used as a mutex/lock to guarantee mutual exclusion. I was unsure if I was allowed to use for example a ReentrantLock here, so I used a semaphore with 1 permit.

- Semaphore, barrierOne, and barrierTwo. These are used to make the threads block on the appropriate locations in the function. The reason we use two, is to safeguard against threads leaping around too fast. If this was not done, the reusable nature of a CyclicBarrier would be ruined, but it is another layer of synchronization, so it does make it slower. If the barrier would were not to be reusable, we could just have a mutex and a single semaphore.

The barrier functions as follows:

- When a thread arrives at the barrier, through calling await(), the thread will acquire the lock/mutex. The thread now has ensured mutual exclusion, and can safely increment the numOfWaiting counter, by 1. If the thread is not the n-th thread to arrive, it will simply release the mutex, and call acquire() on the barrierOne. Here it will be blocked until some other thread releases it sometime after the barrier has reached the threshold for waiting threads.

- When the n-th thread arrives, it will take the mutex, increment the numOfWaiting counter by 1. It will then acquire the barrierTwo semaphore, to ensure no threads can leap around and be released prematurely if the barrier is reused. It will then release the barrierOne, before releasing the lock.

- At this point, the threads are ready to be released from the barrier, so that they can continue their execution. The threads will release the barrierOne semaphore. Acquire the mutex, decrement the numOfWaiting counter by 1. When the counter reaches 0, when the last thread reaches the barrier, it acquires the barrierOne semaphore and releases the barrierTwo semaphore. All threads release the mutex at this point. Before exiting the threads acquire the barrierTwo semaphore, before releasing it.

- By using two semaphores, I can ensure that no thread laps around and is released before where it is supposed to. For example, the main thread goes immediately back to waiting and may get released prematurely, if there was no mechanism in place to protect it from this. Before acquiring the first barrier semaphore a thread has to release the other.

## Q2.2: Test Case

To test my program, I have written a very simple test case in the main class Barrier. The main thread launches a couple of threads before waiting at the barrier. The threads will perform some sleep in a random interval (0-10sec), before waiting at the barrier. After all the threads reach the barrier, they will be released. Main will immaturely wait on the barrier again, while the threads perform another round of sleep, before waiting on the barrier again. The program prints the status of the threads, how many threads are waiting, and when they get released from the barrier. But really, the threads could have done any arbitrary work, the barrier works the same anyway. I just chose sleep, so the threads arrived at some random intervals. The program is run with Java Barrier <num of threads>. The program will then launch the number of threads specified and wait on the barrier, along with the main thread.

*Main class for semCyclicBarrierTest*

```
public class Barrier
{
    private static semCyclicBarrier testBarrier;

    public static void main(String[] args)
    {
        if(args.length != 1)
        {
            System.out.println("Run with <Num of threads>");
            System.exit( status: -1);
        }

        try
        {
            int numOfThreads = Integer.parseInt(args[0]);
            testBarrier = new semCyclicBarrier( numOfThreads: numOfThreads + 1);
            runTest(numOfThreads);
        }

        catch (Exception e)
        {
            e.printStackTrace();
        }
    }

    public static void runTest(int numOfThreads)
    {
        System.out.println("Main thread will now start " + numOfThreads +  " threads.");

        for(int i = 0; i < numOfThreads; i++)
        {
            new Thread(new testWorker(i, testBarrier)).start();
        }


        System.out.println("Main thread waiting at barrier test ONE.");
        testBarrier.await();
        System.out.println("Main thread released from barrier test ONE.");

        System.out.println("Main thread waiting at barrier test TWO.");
        testBarrier.await();
        System.out.println("Main thread released from barrier test TWO.");
    }
}
```

*TestWorker class for testing semCyclicBarrier*

```
import java.util.Random;

public class testWorker implements Runnable
{
    private int id;
    private semCyclicBarrier testBarrier;
    private Random rand;

    public testWorker(int id, semCyclicBarrier testBarrier)
    {
        this.id = id;
        this.testBarrier = testBarrier;
        this.rand = new Random();
    }

    public void run()
    {
        int randomInt = Math.abs(rand.nextInt( bound: 10000));

        try
        {
            System.out.println("Thread: " + id + " will sleep for: " + randomInt + " ms");
            Thread.sleep(randomInt);
        }

        catch (Exception e)
        {
            e.printStackTrace();
        }

        System.out.println("Thread: " + id + " waiting at barrier test ONE.");
        testBarrier.await();
        System.out.println("Thread: " + id + " released from barrier test ONE.");

        randomInt = Math.abs(rand.nextInt( bound: 10000));

        try
        {
            System.out.println("Thread: " + id + " will sleep for: " + randomInt + " ms");
            Thread.sleep(randomInt);
        }

        catch (Exception e)
        {
            e.printStackTrace();
        }

        System.out.println("Thread: " + id + " waiting at barrier test TWO.");
        testBarrier.await();
        System.out.println("Thread: " + id + " released from barrier test TWO.");
    }
}
```

*The terminal output of test of semCyclicBarrier with 5 threads + main thread.*

```
"C:\Program Files\Java\jdk1.8.0_201\bin\java.exe" ...
Main thread will now start 5 threads.
Thread: 0 will sleep for: 6229 ms
Thread: 2 will sleep for: 5676 ms
Main thread waiting at barrier test ONE.
Thread: 1 will sleep for: 5024 ms
Waiting threads increased to: 1
Thread: 3 will sleep for: 81 ms
Thread: 4 will sleep for: 7761 ms
Thread: 3 waiting at barrier test ONE.
Waiting threads increased to: 2
Thread: 1 waiting at barrier test ONE.
Waiting threads increased to: 3
Thread: 2 waiting at barrier test ONE.
Waiting threads increased to: 4
Thread: 0 waiting at barrier test ONE.
Waiting threads increased to: 5
Thread: 4 waiting at barrier test ONE.
Waiting threads increased to: 6
All threads have reached the barrier.
Thread: 0 released from barrier test ONE.
Thread: 0 will sleep for: 7042 ms
Thread: 4 released from barrier test ONE.
Thread: 4 will sleep for: 7302 ms
Main thread released from barrier test ONE.
Main thread waiting at barrier test TWO.
Waiting threads increased to: 1
Thread: 1 released from barrier test ONE.
Thread: 1 will sleep for: 9871 ms
Thread: 3 released from barrier test ONE.
Thread: 3 will sleep for: 2222 ms
Thread: 2 released from barrier test ONE.
Thread: 2 will sleep for: 7738 ms
Thread: 3 waiting at barrier test TWO.
Waiting threads increased to: 2
Thread: 0 waiting at barrier test TWO.
Waiting threads increased to: 3
Thread: 4 waiting at barrier test TWO.
Waiting threads increased to: 4
Thread: 2 waiting at barrier test TWO.
Waiting threads increased to: 5
Thread: 1 waiting at barrier test TWO.
Waiting threads increased to: 6
All threads have reached the barrier.
Thread: 2 released from barrier test TWO.
Main thread released from barrier test TWO.
Thread: 1 released from barrier test TWO.
Thread: 4 released from barrier test TWO.
Thread: 0 released from barrier test TWO.
Thread: 3 released from barrier test TWO.

Process finished with exit code 0
```

## Q3 Recursive MergeSort

### Q3.1: Parallelizing MergeSort

Merge sort is a problem that is very well suited for parallel computation. It works by splitting an array of integers into two halves recursively, then merges the two sorted halves. It's a divide and conquer algorithm, where the problem is divided into smaller problems recursively, before solving them and then merging the sub-results into the final result. Sorting the right and left subarrays are independent problems, meaning that they can easily be solved in parallel with minimal synchronization.

The way I pearlized merge-sort is very simple. I was considering using RecursiveAction and ForkJoin framework, but I was unsure if this was allowed. So, I did the implementation myself. My implementation is true to the algorithm and works identical to the sequential version supplied in the exam text.

The function is called through the method public static void parallelMergeSort(int[] inputArray, int len). The method sets a variable for the maximum number of threads to use, by acquiring this info from the Runtime-system. This number is passed to the merge sort, and will be divided by 2 for each recursive call to the method. I did try with an global atomic integer, but it hurt the performance about 25%.

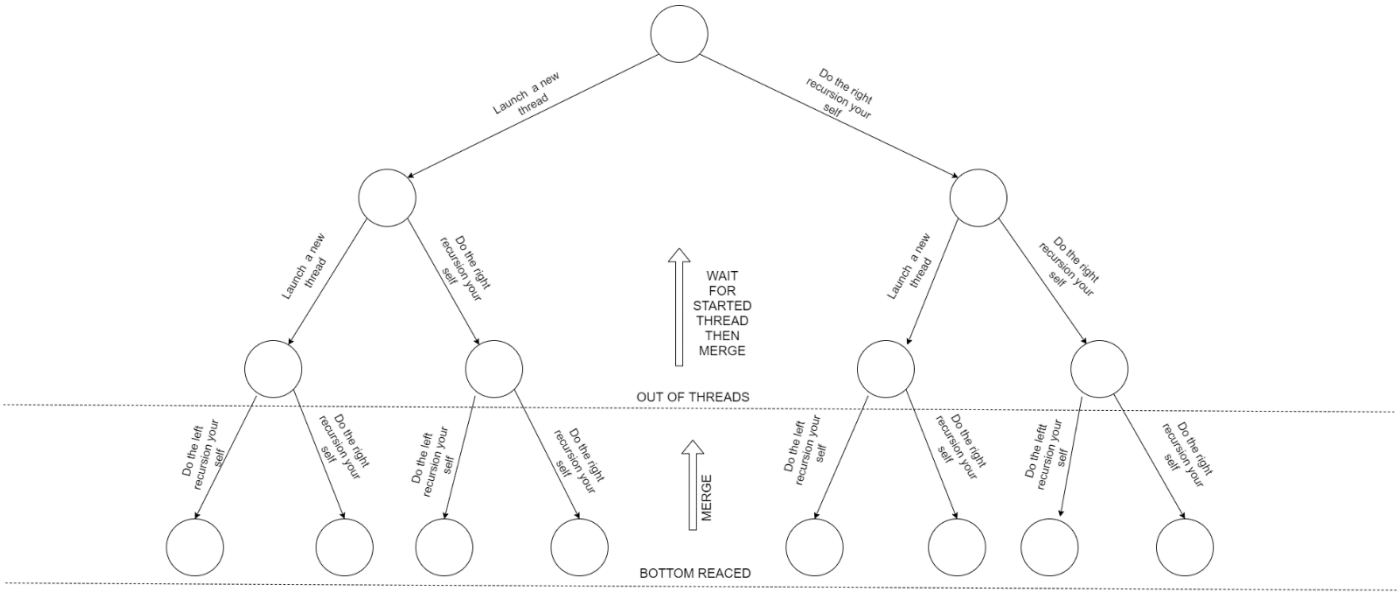
A reason for having this global AtomicInteger, instead of just assigning availableThreads/2 to each left and right side of the recursion is the dynamic aspect. If I did it this way, there would be no way of knowing how many threads are alive at some arbitrary moment in time, many threads might have exited. For example, the entire right recursion might be finished while the left has a bunch left, with a global variable the left side can utilize the unused threads (threads increment the threadCounter upon exiting, effectively meaning that another thread can be launched), instead of having a static number of threads available to each side. However, it does take some extra time to check this variable every time the method is called due to the synchronization needed. So for that reason I choose to just pass the number of threads and divide it as the recursion continues. Its also not likely that one side of the recursion is done way before the other side, because they do almost identical work, and a downside of the use of an AtomicInteger could be that the side that started first took all the threads before the other side could get any, so this solution may be more fair and efficient.

The function then starts the recursion process by calling the function private static void mergeSort(int[] inputArray, int len). This method works very similarly to the sequential one, it first constructs the right and left array. It then determines if there are any free threads, if this is the case it will fire up a thread. The new thread will be responsible for the left side of the recursion. Effectively meaning that a new thread will continue with the left recursion, while the thread that called the method will continue the right side of the recursion. To determine if there are any free threads the caller decrements the examines the value passed from above, if it's larger than 0 there is room to start up another thread. Here I would have checked the AtomicInteger if I opted for that, but I did not as I explained, due to performance.

If a new thread is started, the caller will only continue the right recursion. If not, it will simply do both the left and right recursion itself, in other words, the thread falls back to sequential merge-sort. If this is the case, this means that the program is already utilizing as many cores as it has available, and launching a new thread is ineffective. Java threads are mapped to operating system threads, so launching more than we have will just lead to time-slicing and less execution time for each thread. Once the caller returns to this point, it will call join on the thread it created (the one that worked on the left side), if it created such a thread. It will wait for it to finish its execution (it might be done already) and then start the merge process upwards. The merge method works identical to the sequential one. The merging process can not start before both the right and left recursion is finished, because they rely on each other. That's why I use a join to wait for the other thread to finish, but I could have used a CyclicBarrier or some other synchronization mechanism.

Other than that there is not much to the parallelization. When the bottom of the recursion is reached, the threads will start merging the subarrays back to where they started. Then they will exit, and the thread that created that thread will continue the merging upwards. The process will finally reach the thread that started the entire recursion, it will then finish by merging the left and right arrays, and the algorithm is finished. If we look at the entire process as a tree, the parallelization has very limited potential the closer to the root node we get. At the very beginning, only two threads will be active, one responsible for the left and one for the right array.

The illustration below shows how the problem is split between threads. Before the threads start merging upwards to their starting origin, before the thread that started that thread continues the merging. This is just an example of a very low depth recursion tree from the parallel merge-sort with a maximum of 3 threads. As you can see the program will launch new threads as long as it has enough headroom to do so. The tree can also be much, much deeper in a real use-case, so this is just an example. When the bottom is reached the merging starts, and the threads will merge their sub-arrays to the point where they were created. As you can also see, the multithreaded aspect is very low near the root, because of the nature of the problem.




To summarize. The merge-sort problem can be solved efficiently at a level of high parallelism, with a small amount of synchronization while remaining true to the sequential algorithm if the input array is large. The multi-threaded aspect is limited if the array is small. The entire process requires minimal synchronization, as the threads work on fully independent problems. They only have to synchronize once the thread returns to wherever it was started because at this point the problems become dependent on each other. Now the two solved sub-problems should be combined to a result and passed upwards. If the input array is very small, the sequential version solves the problem faster, due to the overhead and the synchronization that takes place in the parallel version.

The larger the problem becomes, the more efficient the parallel version becomes because the overhead of launching thread and the synchronization between these will be a much smaller part of the program's total execution time. Threads are expensive to launch, so when we first launch them we want them to perform as much work as possible to make up for this cost and to yield better results than just solving it sequentially.

## Q3.2 Parallel MergeSort

The parallel merge-sort is implemented in the file paraMergeSort.java, and the test program is written in the file Sort.java. There is also a sequential version of the sort (copied from the exam) in sequentialMergeSort.java

The program resides in the class paraMergeSort:

```
public class paraMergeSort
{
    private static AtomicInteger threadCounter;

    public static void parallelMergeSort(int[] inputArray, int len)
    {...}

    /*
        This method performs the upwards merging of two sub-arrays into the larger array one step up.
        Its called once the once the recursive division of the arrays reaches the bottom and starts to return upwards.

        @Params, the left, right and mother arrays and the left and right size.
    */
    private static void merge(int[] left_arr, int[] right_arr, int[] arr, int left_size, int right_size)
    {...}


    /*
        This method divides the array supplied into two sub-arrays.
        If there is any free threads, it allows a new thread to continue to split
        the left sub-array, while the caller continues on the right sub-array.
        If no threads free, the caller does both the sides.

        Once the function returns from the recursion, it starts to merge
        the two sub-arrays upwards.

        @Params, The array to divide and its length.
     */
    private static void mergeSort(int[] inputArray, int len)
    {...}

    /*
        This is the thread class that works in the merge-sort.
        It simply calls merge-sort on the the array it was supplied.
    */
    private static class worker implements Runnable
    {...}
}
```

The merge method:

```java
/*
    This method performs the upwards merging of two sub-arrays into the larger array one step up.
    Its called once the once the recursive division of the arrays reaches the bottom and starts to return upwards.

    @Params, the left, right and mother arrays and the left and right size.
*/
private static void merge(int[] left_arr, int[] right_arr, int[] arr, int left_size, int right_size)
{
    int i = 0;
    int l = 0;
    int r = 0;

    while(l < left_size && r < right_size)
    {
        if(left_arr[l] < right_arr[r])
        {
            arr[i++] = left_arr[l++];
        }

        else
        {
            arr[i++] = right_arr[r++];
        }
    }

    while(l < left_size)
    {
        arr[i++] = left_arr[l++];
    }

    while(r < right_size)
    {
        arr[i++] = right_arr[r++];
    }
}
```

The mergesort method:

```java
/*
    This method divides the array supplied into two sub-arrays.
    If there is any free threads, it allows a new thread to continue to split
    the left sub-array, while the caller continues on the right sub-array.
    If no threads free, the caller does both the sides.

    Once the function returns from the recursion, it starts to merge
    the two sub-arrays upwards.

    @Params, The array to divide and its length.
 */
private static void mergeSort(int[] inputArray, int len, int numOfThreads)
{
    if(len < 2)
        return;

    int mid = len/2;
    int[] left_arr = new int[mid];
    int[] right_arr = new int[len-mid];

    int r = 0;
    int l = 0;

    for(int i = 0; i < len; i++)
    {
        if(i < mid)
        {
            left_arr[l] = inputArray[i];
            l++;
        }

        else
        {
            right_arr[r] = inputArray[i];
            r++;
        }
    }

    if(numOfThreads > 1)
    {
        Thread leftThread = new Thread(new worker(left_arr, mid,  numOfThreads: numOfThreads/2));
        leftThread.start();
        mergeSort(right_arr,  len: len - mid,  numOfThreads: numOfThreads/2);

        try
        {
            leftThread.join();
        }

        catch (Exception e)
        {
            e.printStackTrace();
        }
    }

    else
    {
        mergeSort(left_arr, mid,  numOfThreads: 0);
        mergeSort(right_arr,  len: len-mid,  numOfThreads: 0);
    }

    merge(left_arr, right_arr, inputArray, mid,  right_size: len-mid);
}
```

The thread class that works in the parallel mergesort

```
/*
    This is the thread class that works in the merge-sort.
    It simply calls merge-sort on the the array it was supplied.
*/
private static class worker implements Runnable
{
    private int[] arr;
    private int len;
    private int numOfThreads;

    public worker(int[] arr, int len, int numOfThreads)
    {
        this.arr = arr;
        this.len = len;
        this.numOfThreads = numOfThreads;
    }

    public void run()
    {
        mergeSort(arr, len, numOfThreads);
    }
}
```

## Q3.3 Parallel Mergesort test case

To test my implementation, I have created a class called Sort with a main-method. The class takes in a seed, size, and the number of runs to execute for both the sequential and parallel solutions from the command line. To run the program use java Sort <Seed> <Size> <Num of runs>, for example, Java Sort 103 100000 7. This will use the seed and size to generate random arrays and run each test 7 times.

The program will run generate a new array for each at the start of the test, this is not included in the timings. This array is then copied over to another array for each run of the test (to not destroy the random array), but this is not included in timings either. First, the parallel tests are performed, and the timings are recorded. Once all the tests are finished the program will calculate the median timing. Then it will perform the same steps for the sequential version, before printing out the speedup of the parallel algorithm over the sequential one. I run many tests to make sure that the results are fair and accurate. The OS may interfere, other programs may run in the background and the JIT-compiler may kick in at different points, so we have to run a large number of tests, to remove any “fake” or “wrong” timings. I also do a simple test at the last run of each test, to make sure that the array is indeed correctly sorted.

## *The test program*

```java
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collections;
import java.util.Random;

public class Sort {
    private static int[] masterArray;
    public static void main(String[] args)
    {
        if(args.length != 3)
        {
            System.out.println("Run with <Seed> <N> <Trials>");
            System.exit( status: -1);
        }

        try
        {
            int seed = Integer.parseInt(args[0]);
            int size = Integer.parseInt(args[1]);
            int trials = Integer.parseInt(args[2]);
            masterArray = new int[size];
            System.out.println("\nProgram starting testing of seq and para merge-sort\nWith seed: " + seed + ", size: " + size + ", trails: " + trials + "\n");
            runSortTest(seed, size, trials);
        }

        catch (Exception e)
        {
            e.printStackTrace();
        }

    }

    public static void runSortTest(int seed, int size, int trials)
    {
        masterArray = generateArray(seed, size);
        int[] array = new int[size];

        ArrayList<Long> timings = new ArrayList<>();
        long start, end, resultPara, resultSeq;

        System.out.println("Starting tests of parallel merge-sort, the program will run: " + trials + " tests.");
        for(int i = 0; i < trials; i++)
        {
            System.arraycopy(masterArray, srcPos: 0, array, destPos: 0, masterArray.length);
            start = System.nanoTime();
            paraMergeSort.parallelMergeSort(array, array.length);
            end = System.nanoTime();
            System.out.println("Parallel merge-sort used: " + (end-start)/1000000 + " ms on run: " + (i + 1) + "/" + trials);
            timings.add((end-start)/1000000);

            //just a simple check to make sure its correct.
            if(i == trials-1)
                validateResults(array);
        }

        resultPara = calculateMedian(timings);
        System.out.println("\nParallel merge-sort used a median time of " + resultPara + " ms over " + trials + " runs\n");
        timings.clear();

        System.out.println("Starting tests of sequential merge-sort, the program will run: " + trials + " tests.");
        for(int i = 0; i < trials; i++)
        {
            System.arraycopy(masterArray, srcPos: 0, array, destPos: 0, masterArray.length);
            start = System.nanoTime();
            sequentialMergeSort.mergeSort(array, array.length);
            end = System.nanoTime();
            System.out.println("Sequential merge-sort used: " + (end-start)/1000000 + " ms on run: " + (i + 1) + "/" + trials);
            timings.add((end-start)/1000000);

            //just a simple check to make sure its correct.
            if(i == trials-1)
                validateResults(array);
        }

        resultSeq = calculateMedian(timings);
        System.out.println("\nSequential merge-sort used a median time of " + resultSeq + " ms over " + trials + " runs\n");

        double speedup = (double)resultSeq/resultPara;
        System.out.format("\nSPEEDUP: %.2f", speedup);
        timings.clear();
    }
```

*Example test program output with seed = 100, size = 10 000 000 and trials = 7*

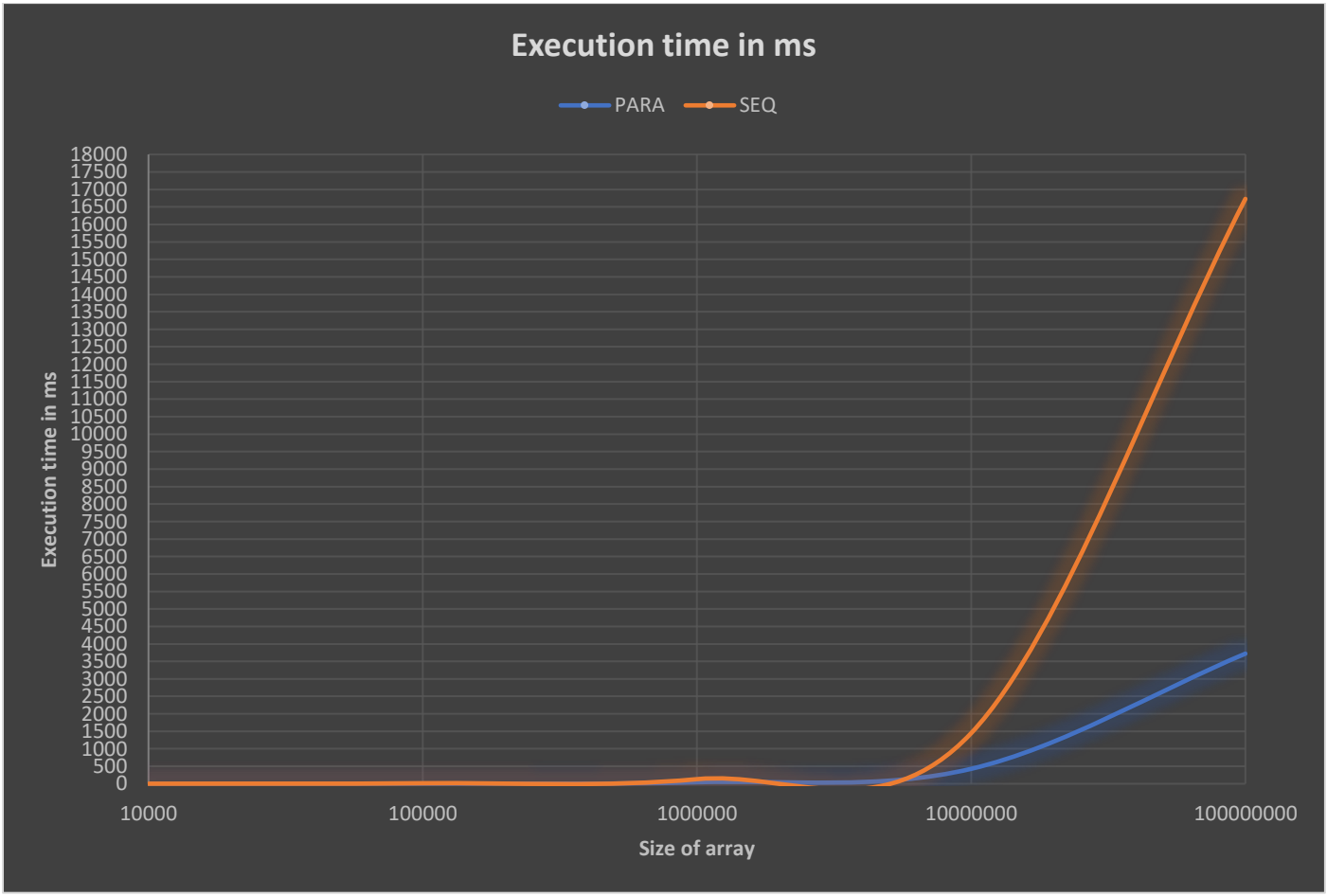
```
"C:\Program Files\Java\jdk1.8.0_201\bin\java.exe" ...

Program starting testing of seq and para merge-sort

Starting tests of parallel merge-sort, the program will run: 7 tests.
Parallel merge-sort used: 482 ms on run: 1/7
Parallel merge-sort used: 514 ms on run: 2/7
Parallel merge-sort used: 371 ms on run: 3/7
Parallel merge-sort used: 450 ms on run: 4/7
Parallel merge-sort used: 370 ms on run: 5/7
Parallel merge-sort used: 377 ms on run: 6/7
Parallel merge-sort used: 323 ms on run: 7/7
The array is correctly sorted.

Parallel merge-sort used a median time of 377 ms over 7 runs

Starting tests of sequential merge-sort, the program will run: 7 tests.
Sequential merge-sort used: 1468 ms on run: 1/7
Sequential merge-sort used: 1461 ms on run: 2/7
Sequential merge-sort used: 1439 ms on run: 3/7
Sequential merge-sort used: 1454 ms on run: 4/7
Sequential merge-sort used: 1461 ms on run: 5/7
Sequential merge-sort used: 1449 ms on run: 6/7
Sequential merge-sort used: 1461 ms on run: 7/7
The array is correctly sorted.

Sequential merge-sort used a median time of 1461 ms over 7 runs


SPEEDUP: 3.88
Process finished with exit code 0
```

### *Test results*

I have tested my program with arrays of sizes from 10 000 – 100 000 000, with a 10x increase in size in each test. The test is performed on an i9-9900k, with a base frequency of 3.60 GHz and a boost clock of 5.00 GHz. The seed used was 103 and the number of tests performed for each algorithm was 7. The arrays are randomly generated, but identical for each test within one execution of the program, to get accurate results.

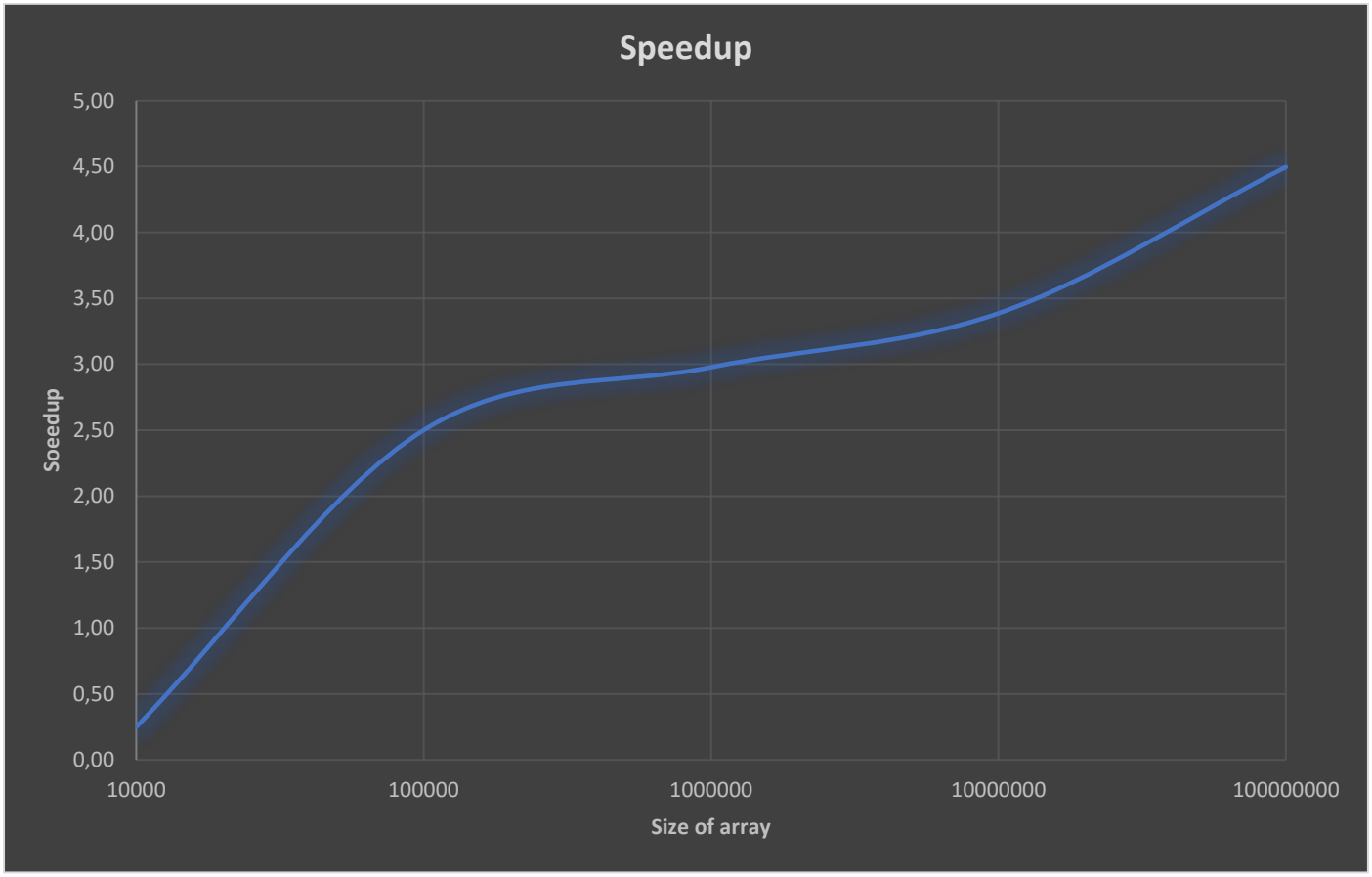
### Execution times

| N | PARA | SEQ |
|---|---|---|
| 10000 | 4 ms | 1 ms |
| 100000 | 6 ms | 15 ms |
| 1000000 | 44 ms | 131 ms |
| 10000000 | 427 ms | 1446 ms |
| 100000000 | 3721 ms | 16729 ms |



Speedup

| N | Speedup |
|---|---|
| 10000 | 0,25 |
| 100000 | 2,50 |
| 1000000 | 2,98 |
| 10000000 | 3,39 |
| 100000000 | 4,50 |



## Observations

As you can see from the graphs and tables, the parallel algorithm overtakes the sequential one somewhere between 10.000 and around 30.000 – 40.000 elements and maxes out at around a speedup of 4.5. I have not tested any larger values because the tests ran for very long and in the middle of tests I suddenly ran out of memory on some occasions. However, the data I collected draws a pretty clear picture, there are large gains in using a multi-threaded merge-sort, and not only for super large arrays. I am unsure if we would see larger speedups for larger N's, but from the way that the speedup increases with N, I do believe we might see some small improvements for even larger N's. However, at some point determined by the CPU cores and speed of these, the growth will stagger as the machine will utilize its entire multicore capabilities... From my testing, I can also observe that the parallel timings span more from run to run, while the sequential ones are rather stable. This may be the fact that the machine/operating system does not allow the program to occupy all resources.