1. INTRODUCTION

This program uses least significant digit Radix sort, which sorts N random integers from 0 to 4N-1.

Two arrays are generated; A and B, where A is the initial numbers and B is the array to insert the radixvalues.

The objective of this program is to create a radix sort solution in parallel, where we achieve a speedup > 1 if it was done sequentially.

Upon completion, two files are created, one for each solution, to check if is correct.

2. USER GUIDE

Run this program by using the following command:
java Oblig4 <N> <SEED>

‘N’ is the amount of numbers to be sorted
‘SEED’ is the random number generator seed.

The program will automatically choose how many threads to run, which is the amount available on your processor.

The global variable NUM_LOOPS defines how many times the program is run (default 7), and can be changed to whatever number >= 1.

File: Oblig4.java

3. TESTS

The program utilizes several tests to make sure everything works as intended.

First the program checks if the correct arguments are given. This includes correct amount of arguments (2), and correct type of input (both needs to be integers >= 0)

After both arrays are sorted, the program will check that the array is correctly sorted, where every number x is smaller than x+1.

Both arrays are also checked if they are equal.

If all tests pass, the message *‘ALL TESTS PASSED’* is printed, meaning the program works as intended. If not, a red text explaining the error is printed.

File: Oblig4Test.java

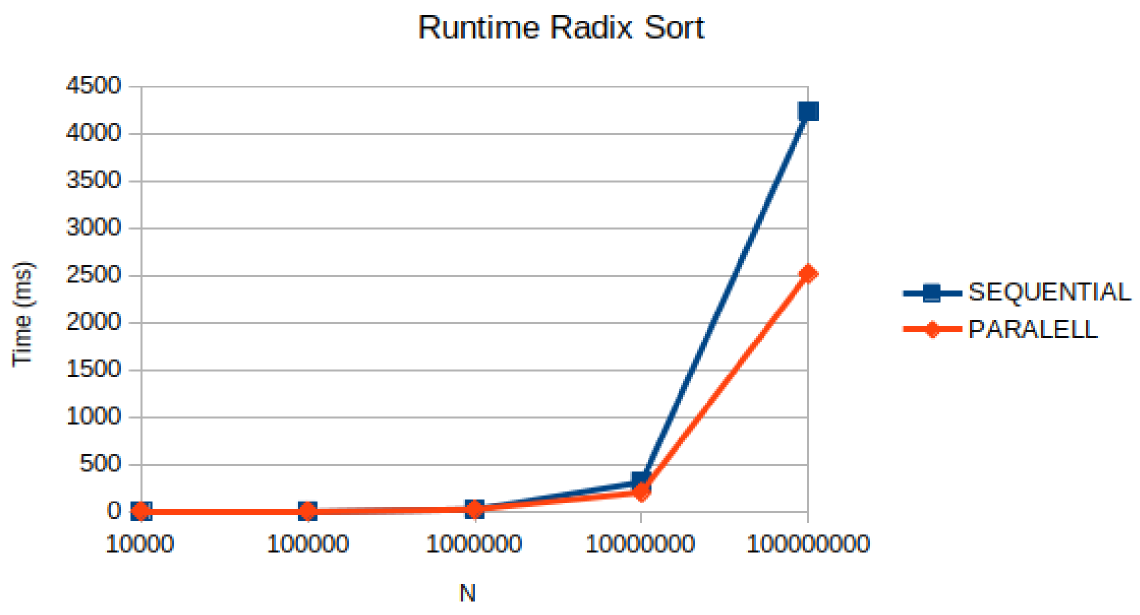
4. RUNTIMES AND SPEEDUP

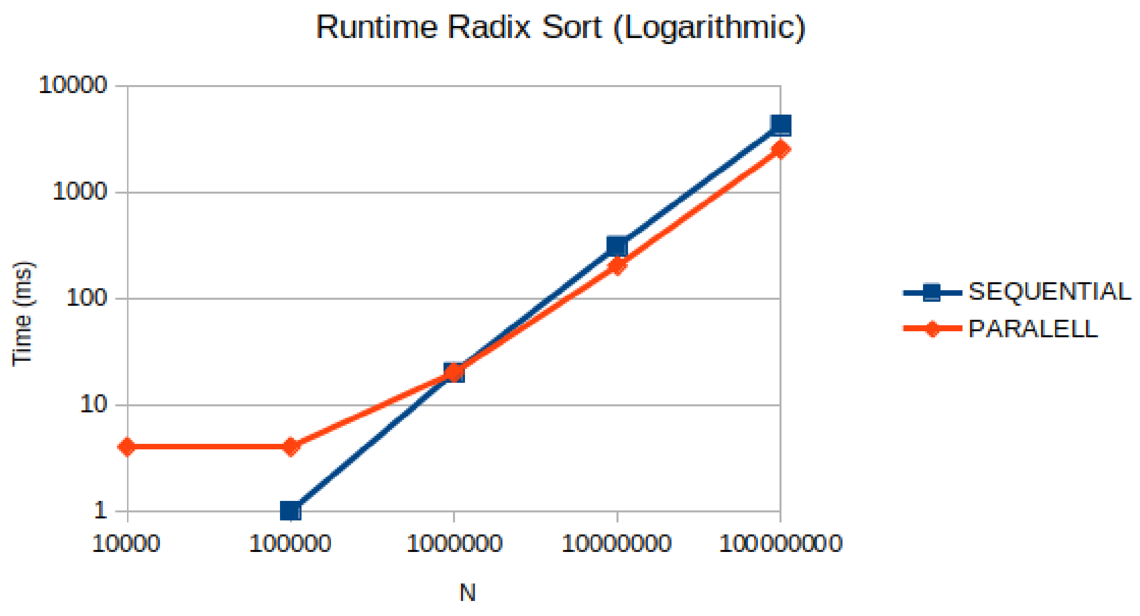
All runs have been tested on a Intel i5-6200U @ 2.30 GHz * 4 processor, running Ubuntu 18.04.1 and java version 1.8.0_201-b09.

Every runtime refers to the median value after 7 runs.

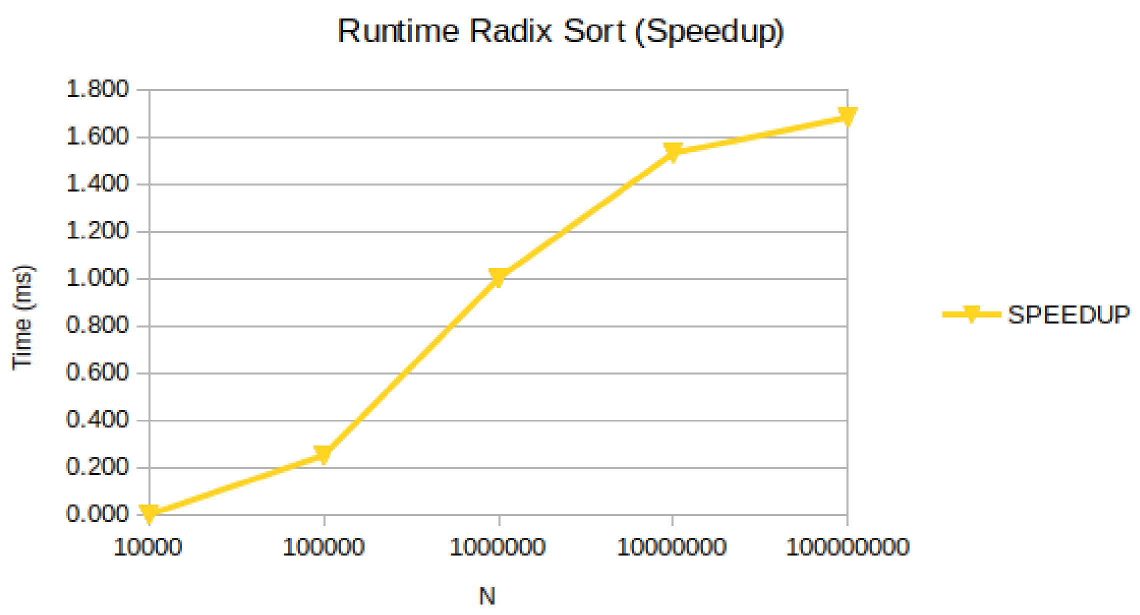
N	10000	100000	1000000	10000000	100000000
SEQUENTIAL	0	1	20	309	4228
PARALLEL	4	4	20	202	2515
SPEEDUP	0.000	0.250	1.000	1.530	1.681

Here we can see the runtimes for N values 10 000 up to 100 000 000. Take note that the sequential solution is faster than the parallel one for N up to 100 000. However, for N = 1 000 000 they are the same speed. Any values higher than this will make the parallel solution faster.





Once again, here we have the same values mapped onto a chart, showing how the parallel solution intersects the sequential one.



And here we have the speedup. Note how as N gets higher, the speedup increases, but it seems to converge close to 2x speed as N approaches infinity.

5. CONCLUSION

The program works as expected as of 11th of April 2019. No bugs are detected and the output is as expected.

Our objective is complete, as we manage to get a speedup > 1.

We could improve the solution a little bit though. In the moveNumbers() method I've decided to use the most obvious solution, where every thread has a copy of the sumCount[] array. This allows us to move from A to B without overwriting any data.

Every thread then keeps increasing the indexed values until 'i' reaches 'low'. A downside to this is that the last thread will do this N-low times, making it almost as slow as the sequential one. We save some time without having to access the array from 0...low, but as this is just one operation, the time-save is not significant.

6. APPENDIX

The program is run NUM_LOOPS amount of times (default: 7). Every run creates an output like this:

```
Run i
Sequential runtime: x ms
Parallel runtime:   y ms
Speedup:            x/y ms
```

ALL TESTS PASSED!