Report – oblig 4 IN3030

1. Introduction

In this report I will explain my solution to the parallel version of the Radix Sort algorithm. This includes discussion of how I did the implementation, in addition to measurements, tables and graphs of runtime and speedup for different values of N.

2. User guide

Compilation:

- javac *.java

Run program:

- java Main <N> <number of threads>
- E. g `java Main 1000000 8`
- If you choose 0 threads the program will use the number of threads equal to the number of cores on the machine, same as in Oblig 3.

The user will then be presented with a menu:

```
**** MENU ****
---------------

1. Get measurements for each run
2. Get median measurements for all values of N
3. Run program tests

Option:
```

1. **Get measurements for each run**
   - Let the user decide number of runs for timing the algorithms, and prints out measurements for each run. Uses number of threads chosen by the user for parallel version.

2. **Get median measurements for all values of N**
   - Gets median measurements for all values of N, including 1000, 10 000, 100 000, 1 mill, 10 mill, 100 mill. Uses number of threads chosen by the user for the parallel version.

3. **Run program tests**
   - Run all program tests to check that the program works properly and gives feedback to the user. Uses input value N and number of threads for testing.

3. Parallel Radix sort

The parallel radix sort is done in the parallelRadixSort class. The constructor takes three arguments: Int a[] (array to be sorted), Int k (number of threads used) and Int useBits.
To parallelize the sort, each thread is assigned a segment of a []. The max value in a [] is found by each thread iterating through its segment and checking if the values are greater than the global variable maxValue. In the next step the counting of different digits in a [] is done in the method getCount (), where each thread counts the different digit values in its segment. In the next step, the accumulated values are summed in count and stored continuously as pointers in the digPointList array. In the last step, each thread moves numbers from a [] to b [] within its segment, and then synchronizes. This is done by the moveNumbers() method.

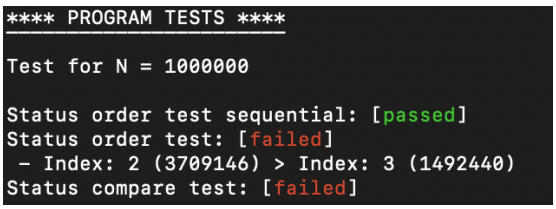
4. Implementation

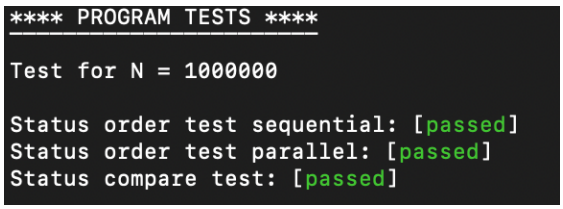
As presented in the user guide, the program contains a menu with 3 possible options.

Time measurements for each run:
Let the user decide number of runs for timing algorithms and prints out measurements for each run. It uses the number of threads chosen by the user for the parallel version. An unsorted array is generated through the precode, and the two versions of Radix Sort then sort their own array, while the runtime is measured.

Time measurements for all N:
Gets median measurements of a total of 8 runs for all values of N, including 1000, 10 000, 100 000, 1 mill, 10 mill, 100 mill. The results are then printed out to the terminal. It uses the number of threads chosen by the user for the parallel version.

Testing:
The program has two tests that check that the code works properly. The first test "check order" checks that the sorting is done correctly. This means that the array is sorted in ascending order and that a low index cannot be greater than a higher index. The method takes the array, generated from the Radix Sort algorithm, as an argument.
The second test "compare output" checks that the output generated from the sequential and parallel versions are equal using the Arrays.equals () method. Finally, the program will use the saveResults () method in Oblig4Precode to store some of the results from the sorted arrays in a text file. To provide feedback to the user, the program prints out status of each test. For example:



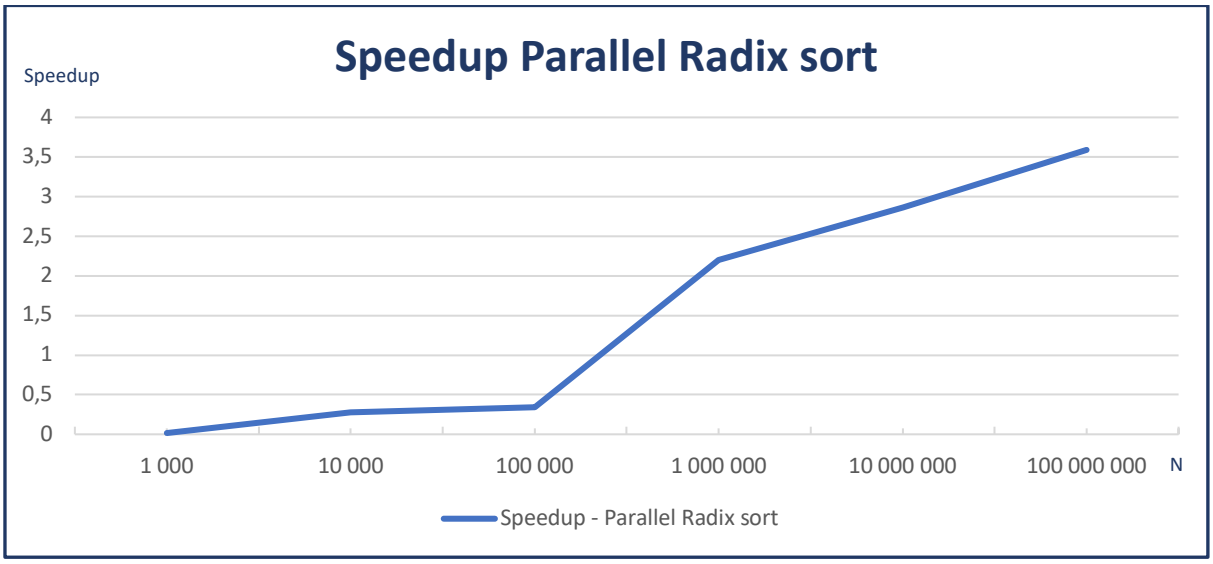


5. Measurements

Speedup Parallel Radix sort

N	Sequential runtime (ms.)	Parallel runtime (ms.)*	Speedup
1 000	0,17	10,75	0,016
10 000	0,55	1,93	0,28
100 000	1,18	3,45	0,34
1 000 000	12,26	5,55	2,20
10 000 000	102,17	35,63	2,86
100 000 000	1546,41	430,56	3,59

***Number of cores used in parallel version: 8**



We can see from the graph, which shows speedup for the parallel Radix Sort algorithm, that the parallel version gives speedup > 1 for values > 1 000 000. The parallel version is not very efficient for values < 100 000, where the sequential version is the best performing. The speedup increases a lot from N = 100 000 to N = 1 000 000 and keeps increasing for N = 100 000 000.

6. Conclusion

From the measurements that have been made, we have seen that the parallel version of Radix Sort can be very efficient compared to the sequential version, but that it requires a large enough value of N to be so. I think the reason for the low speedup for smaller values of N is related to the time it takes for the threads to start up, combined with the fact that the parallel version contains many Cyclic Barriers and that it therefore takes a lot of time to synchronize the threads. At the same time, the number of synchronizations will not change when N becomes larger, which means that the relative time it takes for the threads to synchronize becomes smaller when N becomes larger. I think this is one of the reasons why we only see the effect of the parallelization when N is greater than 100,000, which is when the real effect of the parallelization is made visible by the measurements taken.