Oblig 4 – IN 3030

Marcusti@student.matnat.uio.no

Introduction

In this Oblig we were to implement a parallel version of the Radix Sort Algorithm. The 4 major steps required for this solution are as follows:

1) Find max value in a[].
2) Count the different occurrences of different ciphers in a[].
3) Sum the accumulated values of count[].
4) Move elements from a[] into b[] based on the accumulated values of count[].

User Guide

To compile the program, run the command 'javac *.java'. To run the program, run the command "java Main <N> <K> <DIGIT_SIZE> <Seed>. So if you want to sort an array of 2 million integers with the seed 123, and use all available cores of your current machine, sorting with 7 digits at a time, you would write –

Java Main 2000000 0 7 123.

Parallel Radix Sort

The file Radix.java includes my implementation the parallel algorithm. In a short description, the algorithm works by spawning 2 types of Threads. The first type is for step 1) – finding the max value, and the second type is for the repetitive steps 2) 3) and 4).

Step 1)

I've implemented step 1 by spawning K amount of threads, let them have each their part of the a[] array, find their local max, and update a final global array to report what max this thread has found. I continue by having a single thread (the main thread) go through each of these values and calculate the overall max value.

Step 2 – 4)

I spawn K new threads of type 2 mentioned above. These threads are waiting in Cyclic Barriers until the main thread tells them to do some work.

I let the master thread go through the radix loop, which calls the radixSort function. Inside of the radix sort function, the threads are told when they should "wake up" and begin their calculation.

Step 2) Each thread gets each their portion of the a[] array. Each thread has their own local_count, which they count the occurrences of a[] array. After a thread is done, it is to upload their result to a global array, allCount[k][..]. After this the threads are told to wait in a barrier.

Step 3) I now chose to let the master continue with calculating the accumulated occurrences, in sumCount[]. I also let the master calculate the pointers for the different threads to use in step 4). The pointer[k][..] array is created by letting each thread have their own definition of where different digits counted should be placed in b[], with regards to the other thread's occurences.

Step 4) Wake up the threads again, and let them go through their respective parts of the a[] array, and place them correctly into the b[] array, with regards to occurences found previously. This way all threads can work simultaneously because they all have each their own locations in the b[]-array where elements should be placed.

Implementation

I've implemented the parallel solution step by step, inspired by the sequential algorithm found in the TA's github. I've done a lot of testing to make sure each individual step of my algorithm gave equal results as for the sequential solution. Once step 1 gave the same result as step 1 for the sequential solution, I moved on with step 2 etc...

I initially ran into some RAM problems, which and my computer was running out of memory every time I ran for large values of N. The problem was caused because of a simple mistake – I parsed wrong arguments from the main program.. So that the DIGIT_BITS was read into the Seed, and vice versa. This means instead of creating arrays of size 2^10 – 2^11, (1024 and 2048), my arrays was of size 2^100+! Small mistake, and everything seems to be running fine now.

Measurements.

I've run the parallel solution on my 8GB ram, 2-core/4-thread cpu.

NUMBER_DIGITS is sat to 7, and I'm running with all 4 cores.

N = 1000

```
mvrcus@Mvrcus:~/Documents/IN3030/Oblig4$ java -Xmx7500m Main 1000 0 7 100
Creation Precode: 0.920943ms
---------Sequential Sorting------------
0.384703ms
0.138455ms
0.132841ms
0.132207ms
0.132698ms
0.132099ms
0.132446ms
-------------------------------------
      Median Time Sequential
           0.132698ms
-------------------------------------
---------Parallell Sorting------------
3.372012ms
3.476332ms
2.032088ms
7.211124ms
10.721488ms
1.584992ms
1.545358ms
-------------------------------------
      Median Time Parallel
           3.372012ms
-------------------------------------
-------------------------------------
      Speed Up S/P
          0.0394
-------------------------------------
Sequential and Parallell solution generated same result.
```

N = 10 000

```
mvrcus@Mvrcus:~/Documents/IN3030/Oblig4$ java -Xmx7500m Main 10000 0 7 100
Creation Precode: 1.702026ms
---------Sequential Sorting------------
1.846439ms
0.408488ms
0.527386ms
0.628026ms
0.632782ms
0.63562ms
0.6427ms
-------------------------------------
      Median Time Sequential
           0.632782ms
-------------------------------------
---------Parallell Sorting------------
4.731675ms
1.811051ms
2.244457ms
2.805654ms
2.776974ms
12.340966ms
8.059869ms
-------------------------------------
      Median Time Parallel
           2.805654ms
-------------------------------------
-------------------------------------
      Speed Up S/P
          0.2255
-------------------------------------
Sequential and Parallell solution generated same result.
```

N = 100 000

```
mvrcus@Mvrcus:~/Documents/IN3030/Oblig4$ java -Xmx7500m Main 100000 0 7 100
Creation Precode: 4.442937ms
---------Sequential Sorting------------
7.793493ms
3.559527ms
4.181654ms
3.29264ms
2.318237ms
1.825589ms
1.874482ms
-------------------------------------
      Median Time Sequential
           3.29264ms
-------------------------------------
---------Parallell Sorting------------
11.298123ms
16.388317ms
8.121962ms
8.191631ms
11.214561ms
2.228618ms
1.922351ms
-------------------------------------
      Median Time Parallel
           8.191631ms
-------------------------------------
-------------------------------------
      Speed Up S/P
          0.4020
-------------------------------------
Sequential and Parallell solution generated same result.
```

N = 1 Million

```
mvrcus@Mvrcus:~/Documents/IN3030/Oblig4$ java -Xmx7500m Main 1000000 0 7 100
Creation Precode: 18.255463ms
---------Sequential Sorting------------
36.191477ms
19.908944ms
20.159686ms
17.195031ms
22.961637ms
12.446193ms
12.826606ms
-------------------------------------
      Median Time Sequential
           19.908944ms
-------------------------------------
---------Parallell Sorting------------
57.496645ms
16.426999ms
11.384568ms
20.359818ms
7.721556ms
15.090108ms
9.862434ms
-------------------------------------
      Median Time Parallel
           15.090108ms
-------------------------------------
-------------------------------------
      Speed Up S/P
          1.3193
-------------------------------------
Sequential and Parallell solution generated same result.
```

N = 10 Million

```
mvrcus@Mvrcus:~/Documents/IN3030/Oblig4$ java -Xmx7500m Main 10000000 0 7 100
Creation Precode: 151.863126ms
---------Sequential Sorting-------------
154.893818ms
148.463881ms
138.804677ms
138.83149ms
139.046485ms
138.661551ms
138.930517ms
-------------------------------------
      Median Time Sequential
           138.930517ms
-------------------------------------
---------Parallell Sorting-------------
107.652981ms
74.764943ms
74.150687ms
60.711792ms
61.854629ms
61.925ms
57.85537ms
-------------------------------------
      Median Time Parallel
           61.925ms
-------------------------------------
-------------------------------------
      Speed Up S/P
          2.2435
-------------------------------------
Sequential and Parallell solution generated same result.
```

N = 100 million.

```
mvrcus@Mvrcus:~/Documents/IN3030/Oblig4$ java -Xmx7500m Main 100000000 0 7 100
Creation Precode: 1602.511723ms
---------Sequential Sorting-------------
1671.819162ms
1638.026222ms
1628.100308ms
1631.334191ms
1648.372329ms
1625.485745ms
1625.659712ms
-------------------------------------
      Median Time Sequential
           1631.334191ms
-------------------------------------
---------Parallell Sorting-------------
739.949388ms
702.092062ms
782.763352ms
910.41062ms
667.406936ms
653.53536ms
615.516341ms
-------------------------------------
      Median Time Parallel
           702.092062ms
-------------------------------------
-------------------------------------
      Speed Up S/P
          2.3235
-------------------------------------
Sequential and Parallell solution generated same result.
```

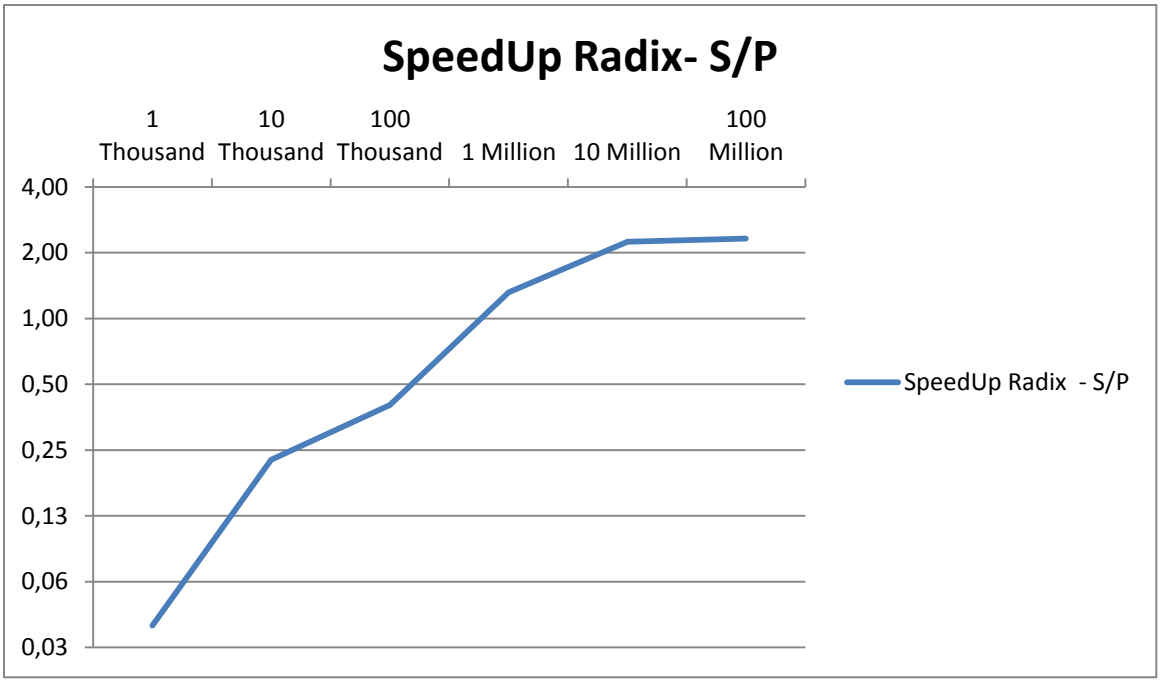

Figure 1: Y axis shows the speedup, S/P for the Radix sort in log2 scale.

Conclusion

As we can see from the graph, it is clear that the parallel version is the clear winner when it comes to larger values of N. We can also see how the speedup seems to stop increasing at around 2.3, this is because of Amdahls Law. There will always be some work that has to be serialized. This in addition to the synchronization and overhead of communication and creation of threads proves that even with 4 cores we do cannot realize a speedup of 4.0.

Appendix

The output of my program is the result of running each algorithm 7 times and comparing the median of these runs. I've also included a check to see if the parallel version creates an equal solution to that of the serial implementation.