IN3030, Oblig 4.

Introduction

This report is about the paralizaiton of the radix sort algorithm. It will contain the description of how its done, and what speedups the parallel solution had over the sequential.

User guide

Compile as normal with javac *java. Run with java Main <n> <seed> , where n is the of arrays to work with and seed is used to generate the random arrays.

Parallel radix sort

Firstly, I divide/partition the input array into subparts. Meaning that each thread that is launched will be responsible for one part of the array. When threads are launched, they will start finding the maximum value in their part of the array. Once they have found the max value, they compare it with the global maximum value, and swap it out if their value is larger. This is done with a lock, to make sure no race conditions occur. Then the threads will wait at a barrier, until all threads have found and potentially swapped the global max value. After all threads reach the barrier, thread 0 will calculate the number of digits needed in the largest number, before all threads call count sort on their part of the array. All the threads perform a normal count sort on their part of the array and places the value in their local count array. Once the thread is finished, it will add its local count values to a global count array, again protected by a lock. They will then wait for all threads to do the same. Then thread 0 will fill the output array, before all threads moves their part of the output array to the input array, before returning. This process is repeated until there are no more digits to work on, once this is done the array is sorted completely. The threads then call wait on the main barrier, to indicate to the main thread that they are finished. The main thread simply returns.

I did not manage to parallize this part of the solution where the output array is built, so thread 0 will execute this code alone:

```
for (int i = inputArray.length - 1; i >= 0; i--)
{
    int digit = (inputArray[i] / placeVal) % 10;
    outputArray[sumCount[digit] - 1] = inputArray[i];
    sumCount[digit]--;
}
```

Measurements

I started seeing speeds ups for n values greater then 100 000. Before this the serial code is faster, because of overhead and synchronization delays. However, after the 100k is passed, the parallel code achieves speedups around 2x of the serial code for every n value up to 100m. I could not test any further, because my jvm machine ran out of heap space. A 2x speedup is significant, but I think it's a little low for a 6-core machine. It's probably because I did not manage to parallize the code pasted above. Everything is tested on a 6 core 2.3 ghz machine with 16gb of ram.

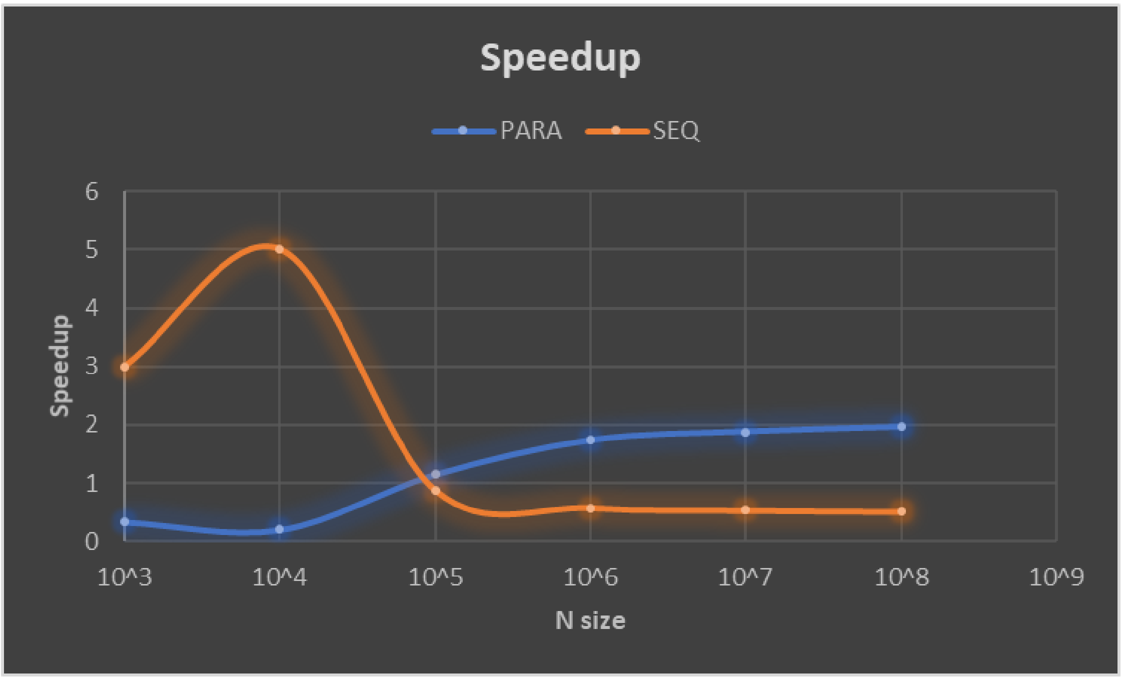
Execution time.

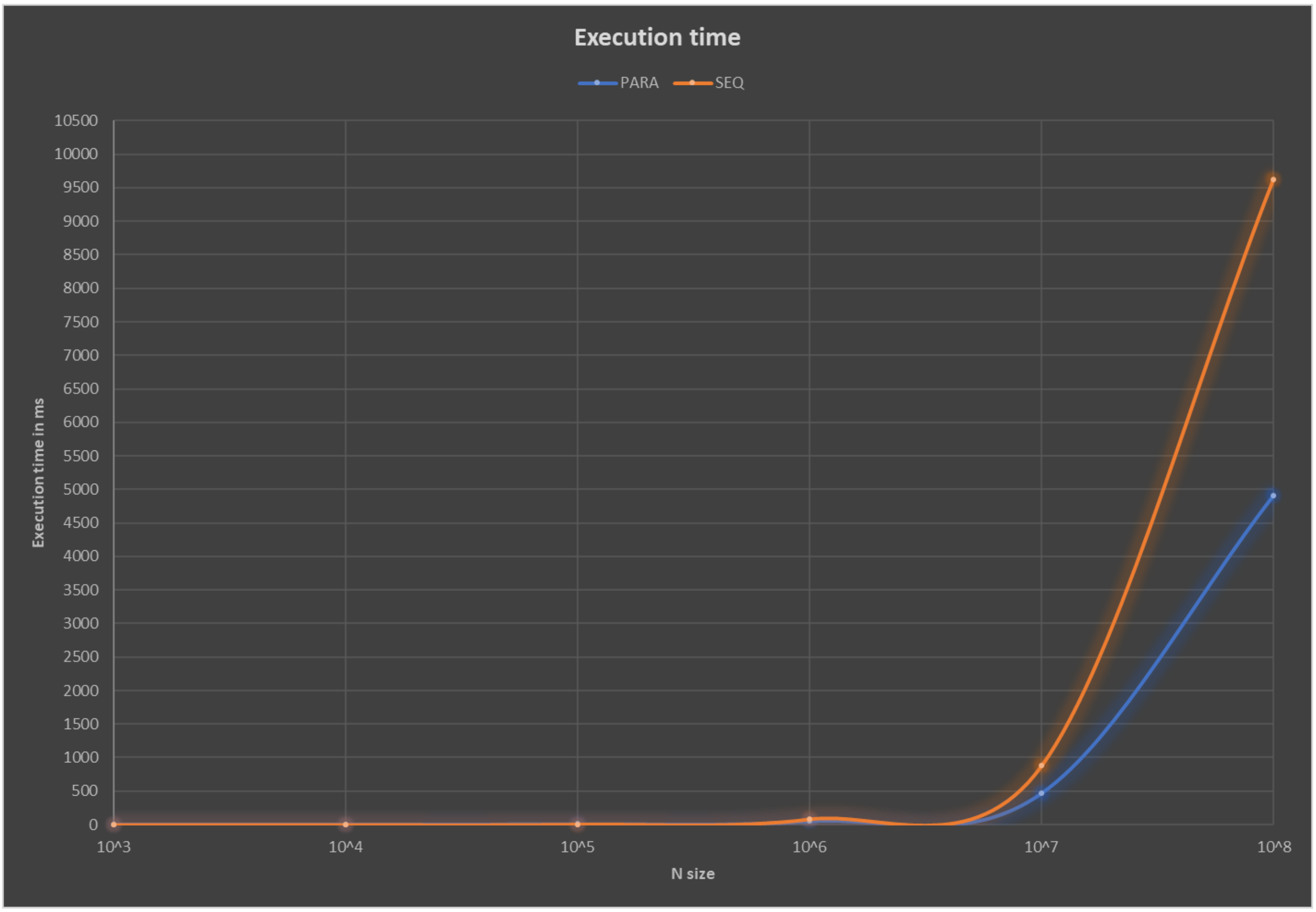
N	PARA	SEQ
1000	3	1
10000	5	1
100000	8	7
1000000	49	85
10000000	467	878
100000000	4907	9625

Speedup

N	PARA	SEQ
1000	0,33333333	3
10000	0,2	5
100000	1,14285714	0,875
1000000	1,73469388	0,57647059
10000000	1,88008565	0,53189066
100000000	1,96148359	0,50981818

The n values in these graphs are represented as powers of 10, to fit it all. And as you can see the speedups stabilize at around 2x the speed of the serial code when we pass the n = 100k.





Conclusion

I have achieved a 2x speedup and have been able to parallize almost all the code. I'm happy with the results but wonder if my solution is suboptimal. Because im testing on a 6 core, and I do not see any speedsups larger then 2. However, I did not find many resources online, except for high level research papers on radix sort, so I found it rather difficult.