IN3030 – oblig 4

UiO : Institutt for informatikk
Det matematisk-naturvitenskapelige fakultet

Daniel Sharifi – danish

Contents

1. Introduction

This report is about parallelization of Right Radix Sort. It discusses and compares the sequential and parallel version of the algorithms, and aims to see if and what benefits there are to gain by parallelizing Radix Sort.

2. User guide

The main class is located in Main.java. To build the program compile all the java files with "javac *.java", and run it with one of the the following arguments:

(the first parameter specifies the mode)

Sequential Radix Sort:
java Main 0 {seed} {n}

Sequential Radix Sort:
java Main 1 {seed} {n} {threads}

Benchmarking (Sequential vs Parallel), this will run the parallel version with the specified thread count, and compare execution times with the sequential version for values of
n = {1000, 10 000, 100 000, 1 mill, 10 mill og 100 mill}
java Main 2 {seed} {threads}

3. Parallel Radix sort

The algorithm I'm trying to implement is inspired by the one presented on the lecture in week 12. Let a[] be the array we are going to sort. Create a new array b[] with the same size as a[]. This array is just going to be an auxiliary array for the sorting.
Find max value in a[].
Count the different digit values in a[]
To do this in parallel I have taken a look into the solution suggested by the Eric. Use the partitioning from step a) and make a local count array which is made available in a global nested array,, so all the other threads can access each others countings.

Sum the accumulated values in allcount[][] (these are the pointers for the digits). I have chosen to do this part sequentially. Each thread have their own array with digitpointers, so the thread with id = 0 will have the responsibility to set the pointers for all the other threads.

Move the elements back and forth from a and b using the same partitioning of a[] from part a). At last when the algorithm is finished If the number of swaps are odd, it means that the sorted array will end up in the axillary array. Thus, thread number 0 will have to copy it over to the main

array using System.arraycopy(). This part I did not parallelize because I used System.arraycopy(), but it is indeed to possible to parallelize this part as well using the partitioning of a[] each thread already have to copy values from b[].

4. Implementation

I have three classes. One for the main program, and one for the sequential and the parallel sorting algorithms. The sequential algorithm is implemented as an abstract class with a public static method for the sorting.

The parallel version requires to create an object of the class, and call sort on this object.

Both sequential and the parallel algorithm of the Radix Sort are “void” functions. It means that they don’t return a new array, but instead they sort the array given as argument.

The parallelization of step a is simple. Each thread finds a local max value in a partition of the unsorted array, and store it in a global variable array with their ‘id’ as the index.

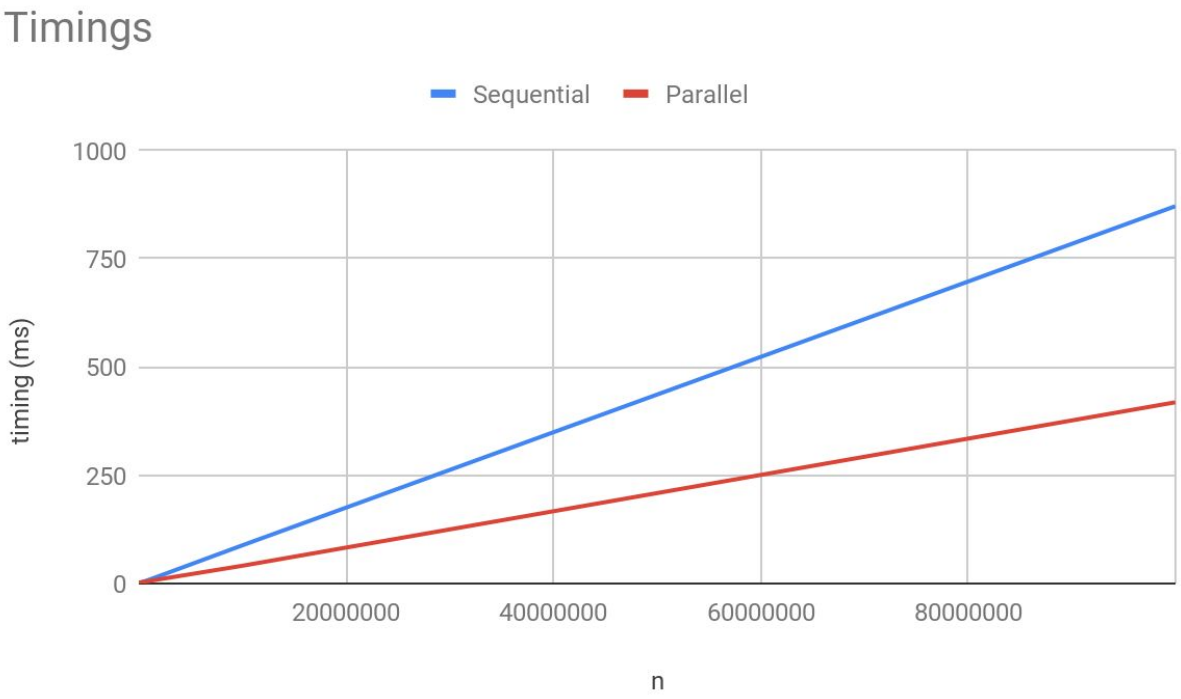
To minimize the number of synchronizations I’ve applied the algorithm in such a way that the threads do a lot of the same computation, instead of letting one do all of it and synchronize with the other threads. Except for part c), computing the new digit pointers for each thread with the offset of the previous threads.
For testing, the benchmark mode is the desired mode to use. It runs first the sequential version and stores all of its results. It asserts that the array is in increasing order. Afterwards it runs the parallel version and checks that its content is equal to the sequential array that was sorted.

5. Measurements

All the timings are in milliseconds.

Sequential	Parallel	n (input size)
0.24926	2.563544	1000
0.33208	1.188347	10000
1.598533	2.46197	100000
8.698121	5.88228	1000000
88.203228	41.001692	10000000
871.126544	418.370962	100000000

Speedup	n (input size)
0.09723258115	1000
0.2794469965	10000
0.6492902026	100000
1.478698906	1000000
2.151209467	10000000
2.082186918	100000000

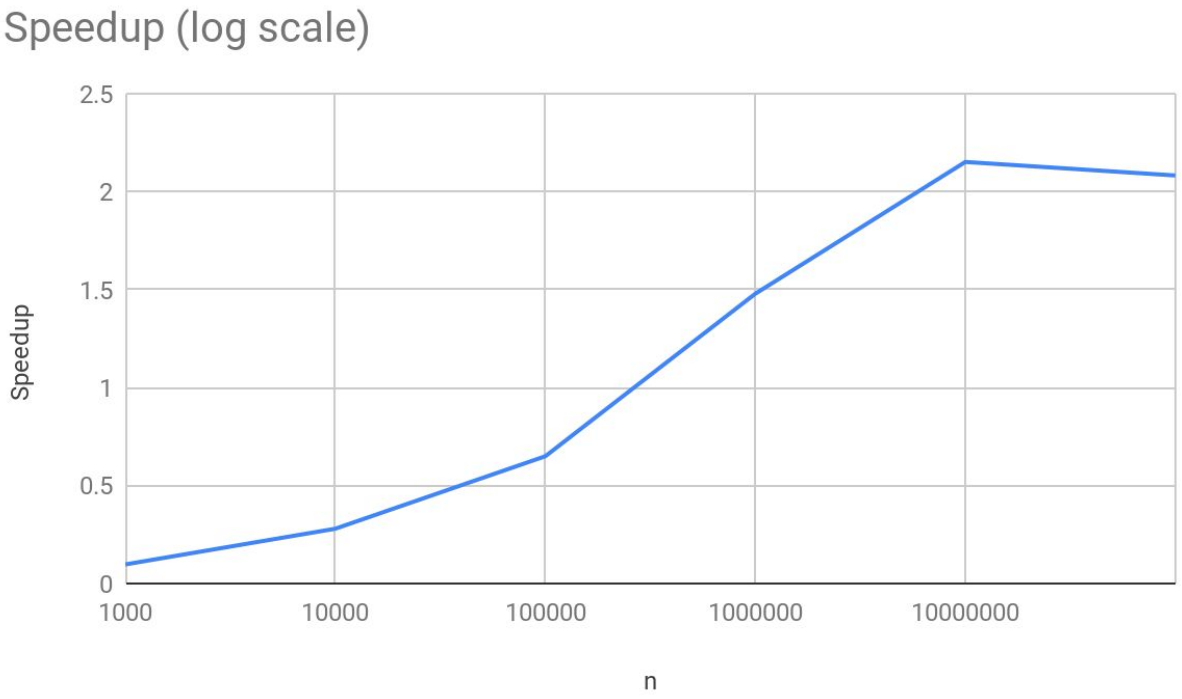


For the measurements I used my program's benchmarking mode using following arguments:

- java Main 2 10 10

Seed number is set to 10.
Usebits = 7.
Threads used = 10

This was run on an machine with 16gb ram and 4 cores (8 threads).

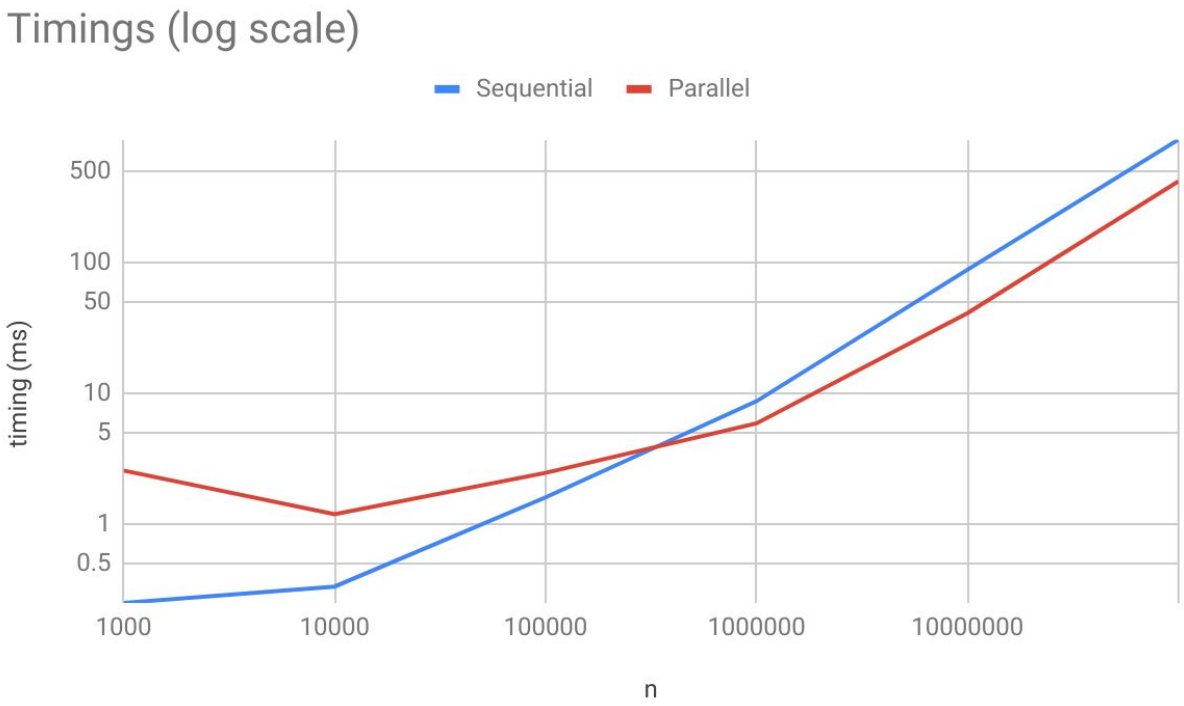


Looking at the speedup table we can conclude that using the parallelized algorithm is more efficient when the size of the unsorted array is equal to 1.000.000 or greater. The parallelization involves a lot of synchronization from the cyclic barrier, which gives a lot of overhead. Thus the parallelization will get a speedup only once n grows significantly large enough, to allow for the synchronization only being a small factor of the total time the algorithm is using.

Furthermore I believe the speedup could have reached even higher values, if I had managed to parallelize step c, as this would reduce one unnecessary synchronization as well. In addition the all count arrays are iterated in an non cache friendly way.

Neither of the algorithms use any heavy disposable objects, only the auxiliary array which the sorting is done back and forth from. So I believe the garbage collector should not take a heavy toll on the timings.

To compare the timing more easily I have included a graph of the timings with both axis using a log scale:



This shows that the parallel version is not increasing much in time for low values, because most of the time is spent on synchronizing instead of the “computation”, because of the high number of synchronizations.

6. Conclusion

In conclusion we can see that the parallelized radix sorting gives good speedups and should be used when the array to be sorted is larger than 1.000.000

It could be even more efficient for lower values if it was more optimized.
I have achieved to parallelize all steps except for step c), (calculating each threads’ digitpointers).