Multi-Radix

Author: Pooria Mehregan

1. **Introduction – what this report is about**
   This is a report on how I parallelized the *Radix Sort* algorithm.

   Source code can be found inside the *src* folder, while extra materials like assignment text and graphs are located inside *materials* folder. Outputs of the program are saved inside the *out* directory. I

   In addition to the outputs required by assignment text, you will find a file named *timings.txt* inside the out directory which contains timings and speedups.

2. **User guide – how to run**
   Make sure you are inside the src folder.

   There are two options for running the program:
   1. User defined commands:
      javac Main.java && java Main <n> <seed> <useBits>

      where:
      - n: array size
      - seed: seed to create an random array
      - useBits: number of bits used to represent a single digit

   2. Automatic run:
      javac Main.java && java Main

3. **Parallel Radix sort – how did I parallelized the Radix Sort**
   There are in total 4 steps in radix sort:
   A) Find max value in array a
      a. Divide a into nrOfThreads elements
      b. Let each thread find the max of their own territory
      c. Save the local maxes to a globalMaxes array
      d. Take and return the max of globalMaxes array

   B) Count frequency of each radix
      a. Divide a into nrOfThreads elements
      b. Let each thread calculate digitFrequencies of their own territory
      c. Add local Frequencies to a globalFrequencies

   c) Calculate start position, in array b, for each digit
      a. Create array globaPointers, which is nrOfThreads long, and in each row there is another array with length mask + 1
      b. Go through globalFrequencies array

c. For each column (which represents a specific digit/useBits)
    i. If current column is not the first column
        1. Add values of previous column's last row and corresponding value of globalFrequncies and assign it to first row of the current column
    ii. Iterate throw each I'th row, starting from second row
    iii. Add the value of current and previous I'th rows in current column and assign it to the current row and column

d) Move elements from a to b based on calculated start positions. Since each thread have its own pointer, we now
a) Divide array a into nrOfThreads elements
b) Let each thread use its own local digitPointer to move values from a to be.

4. Implementation – a somewhat detailed description of how the Java program works & how tested

I have used different classes for each task, for instance Counting Sort is placed inside CountingSort, and Radix Sort is inside class ParallelRadixSort.

There is Main class which checks for inputs and whether they are valid, creates instances, times the algorithm, checks correctness of output and finally writes output and other results to file.

If your run the automated solution described in section 2, then each algorithm will be measured seven times for each value of
N = { 1000, 10 000, 100 000, 1000 000, 10 000 000, 100 000 000 },
and the time used in tables and graphs that you'll see in next section is the median of these seven runs.

Program also checks whether sorted array is actually sorted and contains the same elements as before sorting.

5. Measurements – tables, graphs of speedups, number of cores used

Specification of the machine that was used for measurement:
Name/Modell: MacBook Pro (16-inch, 2019)
CPU: 2,6 GHz 6-Core Intel Core i7
RAM: 16 GB 2667 MHz DDR4
L2 Cache (per Core): 256 KB
L3 Cache: 12 MB
Hyper-Threading Technology: Enabled (doubles the number of physicals cores)

Please see the file inside materials folder with name *Oblig4-table-and-graphs.pdf*.

What you'll see is when N becomes significantly large, the sequential algorithm becomes almost 2,5 times slower than the parallel algorithm. The parallel sort is also slower than sequential for N values less than ten million approximately.

6. Conclusion

In many cases such as this, it is possible to achieve faster algorithm for significantly big arrays. If you have a predefined goal that tells you what to do and where to parallelize, such as this assignment, you can simply follow the text and come up with a parallel solution. If not, you can divide the sequential algorithm into different parts, and measure each part to find the bottle neck and come up with a better solution for that part.