Oblig 4 – LSD Radix

Report about the runtimes and implementation of Right Radix, or Least Significant Digit radix

# Compiling and Running it

Compiling:

* javac \*.java

Running:

* java MultiRadix [n] [k]

N is the size of the array to sort, k is the number of threads to use.

# Parallel Radix Sort

The parallel radix sort implementation tries to do the same basic actions as the sequential version, but with each step parallelized. It still executes the algorithm step by step, it just does the steps differently.

My implementation uses a parallelized version of step a and b, similar to what we have seen and done before, except that it stores the local count array in a local class variable that is publicly accessible rather than in a global, two-dimensional array.

After each thread has counted the numbers in it’s own thread, thread 0 will access the local results and make a shared global list. Afterwards, each thread will copy that global list to it’s local memory, and it will calculate the accumulated values, before giving itself a value range and moving the elements in that range from a to be.

# Measurements

|  |  |  |  |
| --- | --- | --- | --- |
| N | Seq | Par | Speedup |
| 1000 | 0,102 | 1,190 | 0,086 |
| 10000 | 0,384 | 3,487 | 0,110 |
| 100000 | 1,770 | 5,883 | 0,301 |
| 1000000 | 12,146 | 16,873 | 0,720 |
| 10000000 | 216,812 | 150,093 | 1,445 |
| 100000000 | 2281,294 | 2098,035 | 1,087 |

All times are given in ms.

It’s notable that the speedup goes down again at 100 million. This is likely due to caching and memory issues at the machine used to test. The sequential time is also slightly above 2000 ms in that test, despite being a barely edited version of the pre-code.

If time is spent on it, both the sequential and the parallel implementations could probably perform significantly better, and be more cache friendly.

# Conclusion

This is not the fastest implementation of parallelized radix, but it is an implementation. Had I had more time, I could have polished it more (it’s in the middle of multiple home exams ++).

It does work however. It utilizes multiple ways of splitting a workload between threads, for example it splits both on the key of the elements in an array, and of the value of the elements in an array.

It also uses no locks for synchronization, it only has synchronization points at cyclic barriers.