Proposed Optimization of MSD Radix Sort by Manipulating Radix

James Shah, Gaurav Thorat, and Akash Bhimani

College of Engineering, Northeastern University

December 6, 2021

Abstract

Sorting is the most fundamental task in computer science as well as in a day to day life. There has been many research work in this field. As a result, we have nearly reached the saturation level in sorting as the fastest sorting algorithms gives nearly linear performance in the average case. But, there is always some space for improvements and there will be a better performing sorting algorithm in future. But the fastest algorithms also have some caveats for some specific cases. MSD Radix sort algorithm is one such algorithm that uses conting sort as base to sort Strings. The problem with MSD Radix sort algorithm is that it can get really slow for large alphabets, and creates a huge number of small subarrays due to recursion. Many optimizations have been done on it to improve it's performance. In this paper, we discuss two papers that discusses optimizations in MSD Radix sort, and also propose an optimization by manipulating the number of radices.

Keywords
MSD Radix Sort, String Sorting, Sorting Algorithm

1 Introduction

Sorting natural language which uses Unicode (i.e. Chinese, Devanagari, etc.) is a desired task that has many use cases across various fields. A short description of how MSD Radix sort algorithm achieves it is as follows:

1. Partition array into R pieces(radices) according to first character using key-indexed counting.

2.Recursively sort all strings that start with each character (key-indexed counts delineate sub-arrays to sort)

The problem arises due to the large number of R pieces(65536) for UNICODE16 sorting. The time and space taken by algorithm increases rapidly. Thus, we propose a solution to use R = 119, to sort the Devanagari script strings. The ASCII value range corresponding to the Devanagari script UNICODE blocks is from 2304 to 2423 (i.e. 119). We subtract 2304 from the ASCII value of each character (i.e., the ASCII value of 2304 becomes 0), and store them accordingly in the counting array. This range is critical for optimization of MSD Radix sort as it is even less than 256 which is for ASCII values. Other approaches that we analyse in this paper uses strategies such as maintaining inversions after each pass, and parallel implementation of the algorithm for in-place sorting.

2 Pseudo Code

```
private static void sort(String[] a, int lo, int hi, int d)
{
    if (hi <= lo + CUTOFF) {
        InsertionSort.sort(a, lo, hi, d);
        return;
    }
    if(hi <= lo)
        return;
    int[] count = new int[R+2];
    for (int i = lo; i <= hi; i++){
        int index = charAt(a[i], d);

        if(index >= 2304 && index <= 2423 ){
            count[(index - 2304) + 2]++;
        }

    }

    for (int r = 0; r < R+1; r++)
        count[r+1] += count[r];

    for (int i = lo; i <= hi; i++) {
        int index = charAt(a[i], d);
        if(index >= 2304 && index <= 2423 ) {
            aux[count[(index - 2304) + 1]++] = a[i];
        }
    }
    for (int i = lo; i <= hi; i++)
        a[i] = aux[i - lo];

    for (int r = 0; r < R; r++)
        sort(a, lo + count[r], lo + count[r+1] - 1, d+1);
}
```

3 Related Works

3.1 Approach 1

There are some great optimizations to this MSD Radix algorithm one of which is provided by Arne Andersson and Stefan Nilsson in their paper "A New Efficient Radix Sort [1]"[1]. Their proposal to improve is based on maintaining an invariant after ith pass i.e., the. strings are sorted according to the first 'i' characters.

Initially, all strings are contained in the same group.This group is then split into smaller groups and after i passes all the strings with the same prefix after ith pass i.e., same first 'i' characters will belong in the same group. The group is kept in a sorted order according to the prefixes seen so far. Distinguishing is done between finished and unfinished groups.

The ith step of the algorithm is performed in the following way,

1. Traverse the unfinished groups in sorted order and insert each string x, tagged by its current group number, into bucket number xi

2.Traverse the buckets in sorted order and put the strings back into their respective groups in the order as they occur within the buckets

3.Traverse the groups separately. If the kth string in group g differs from its predecessor in the ith character, split the group at this string. The new group is numbered g + k - 1.

Observe that when moving strings, we do not move the strings themselves but point to them. In this way, each string movement is guaranteed to take constant time.

This forward radix sort time complexity is O (S + n + m*smax). In practice, this is an improvement over that Chen and Reif [2] [3].

3.2 Approach 2

Another improvement for the MSD Radix Sort algorithm is given in PARADIS: An Efficient Parallel Algorithm for In-place Radix Sort [3]. This paper purposes improvement in MSD by parallelizing MSD Radix Sort. Efficient parallelization of in-place radix sort is very challenging for two reasons. First, the initial phase of permuting elements into buckets suffers read-write dependency inherent in its in-place nature. Secondly, load balancing of the recursive application of the algorithm to the resulting buckets is difficult when the buckets are of very different sizes, which happens for skewed distributions of the input data.

In PARADIS, its approach solves both problems. A) "speculative permutation" solves the first problem by assigning multiple non-continuous array stripes to each processor.The resulting shared-nothing scheme achieves full parallelization. Since speculative permutation is not complete, it is followed by a "repair" phase, which can again be done in parallel without any data sharing among the processors. B) "distribution-adaptive load balancing" solves the second problem. We dynamically allocate processors in the context of radix sort, to minimize the overall completion time.

Experimental results show that PARADIS offers excellent performance/scalability on a wide range of input data sets. Two strategies for parallel sort have been proposed for multi-core CPU: top-down and bottom-up. In top-down techniques, the input is first partitioned based on the key (e.g., radix-partition), and then each partition is independently sorted. In bottom-up techniques, the input is partitioned for load balancing, and all individually sorted partitions are merged into the final array. Unlike comparison-based sorting (e.g., quicksort, mergesort), radix sort is a distribution-based algorithm that relies on a positional representation of each key (e.g., keys can be digits or characters).

Working steps for PARADIS is as follows,

Step 1 (lines 4-7) The unsorted input array is scanned Step 2 (lines 8-11) The input array is partitioned into |B| buckets by computing ghi and gti. Step 3 (lines 12-20) Each element is checked on line 15 and permuted on line 16 if it is not in the right bucket. Step 4 (lines 21-25) Once element permutation is completed, each bucket becomes a sub-problem, which can be solved independently and recursively.

Conclusions

MSD Radix Sort is deceptively simple, masking a rather sophisticated computation. MSD string sort is so simple as to be used improperly, it can consume outrageous amounts of time and space. In typical applications, the strings will be in order after examining only a few characters in the key. The method quickly divides the array to be sorted into small use key-indexed counting on the first character recursively sort subarrays.

Reducing run time is different from reducing operation counts. Sorting large data sets tends to be memory bound, so the goal of optimization is not to reduce the number of operations, but to try and do make efficient use of memory — for example, increasing the number of buckets works against the running time required; it does no good to complete a radix sort. So, by manipulating the number of radixes we can improve the time required to complete MSD Radix Sort.

Along with that, switching to Insertion sort algorithm to further sort the smaller subarrays is also a crucial task, and makes a huge difference in the running time of the MSD Radix sort algorithm. Thus, choosing a right value for cut off to switch to Insertion sort algorithm is very important.

References

[1] Arne Andersson and Steffan Nilsson. A new efficient radix sort. *Proc. 35th Annual IEEE Symposium on Foundations of Computer Science*, (1):714–721.

[2] R. C. Agarwal. A super scalar sort algorithm for risc processors. *Proc. SIGMOD*, (3):240–260.

[3] Rajesh Bordawekar Ulrich Finkler Vincent Kulandaisamy Minsik Cho, Daniel Brand and Ruchir Puri. Paradis: An efficient parallel algorithm for in-place radix sort. *Proceedings of the VLDB Endowment Volume 8 Issue*, (2):1518–1529.