MSD Radix sort for a natural language using Unicode characters.

Anurag Parla, College of Engineering, Northeastern University, USA
Hiral Nagda, College of Engineering, Northeastern University, USA
Pranoti Dhabu, College of Engineering, Northeastern University, USA

ABSTRACT

Radix sort is an efficient non-comparative sorting algorithm for floating point numbers and integers that can be generalized for strings. This can be achieved theoretically with linear run-time complexity. That aside In-place radix sort can do this using only linear memory. In this paper we are implementing MSD radix sort algorithm that sorts Unicode characters and out-performs its peers.

1 INTRODUCTION

Sorting is a fundamental problem in computer science, with wide applications. Sorting is one of the most important steps in managing and processing the data, especially when the amount of data is very large like a dictionary –which becomes useful only after sorting it alphabetically, i.e., searching becomes simple. It simplifies the task of processing the data as it gets placed in a particular order and the job of locating the required figure becomes time efficient. Based on the way of generating the output, sorting algorithm can be value-based or index-based. Value-based sorting generates the output directly in the form of values being sorted while index-based sorting generates a list of indices which are rearranged in such a way that the values at those indices form a sorted list.

Based on the logic implemented to sort the values, sorting algorithms are broadly categorized into 2 categories:

- comparison based sorting algorithms
- non-comparison sorting algorithms

First category is named so as the data elements are placed in ordered fashion by comparing them with each other. These include insertion sort, bubble sort, selection sort, merge sort, heap sort, quicksort. Second category is named so as the data elements are placed in a particular order without performing any comparison. Non comparison sorting algorithms include counting sort, radix sort and bucket sort.

The comparison-based sorting algorithms have lower bound $\Omega(n\log n)$ on the time complexity of execution. This lower bound is overtaken by non-comparison sorting algorithms - counting sort, radix sort, bucket sort - at the cost of demanding extra resources for execution or at the cost of assuming on the values to be sorted.

Depending on how the duplicate values in the input series get placed in the output, sorting algorithm may be a stable or unstable sorting algorithm. Stable sorting algorithms preserve the initial ordering of elements with the same key value which is preferred when the application demands to preserve the order.

Out of various algorithms, the one chosen to be implemented depends on various factors. These factors include:

- Distribution of Values
- Range of Values

If a simple algorithm is to be implemented, irrespective of the time and space complexity, then one would choose to implement insertion sort, selection sort, bubble sort. Apart from simplicity, they do perform well when the size of data set is small, but their performance degrades as the size of data set increases.

If distribution of values is taken into consideration, then sorting algorithm would be chosen based on the scenario:

- If values are evenly distributed across a particular range, then bucket sort would be preferred to be implemented as all the values would get distributed across the buckets evenly
- If values are distributed in such a way that there is not much difference between the positions of a value in the input and their respective positions in the sorted array, then insertion sort would generate good performance
- If the values to be sorted do not deviate much, and lie in a small range, or when the data is in the frequency distribution of values though the values may be large, then counting sort is preferred to be implemented
- If the values to be sorted are large figures, and all are expected to consist of same number of digits, then radix sort would perform the best.

2. BACKGROUND

A great variety of general-purpose sorting methods have been proposed. However, many of the best-known methods are not particularly well-suited for sorting of strings. Consider for example the behavior of quicksort (Hoare 1962, Sedgewick 1998). If the pivot is well-chosen, an array of strings to be sorted is recursively partitioned, with the size of each partition roughly half at each stage. The strings in each partition become increasingly similar as the sorting progresses; that is, they tend to share prefixes. As determining the length of this shared prefix is expensive, they must be thoroughly compared at each stage; as a result, the lead characters are inspected many times, which is an unnecessary cost. It is for this reason that methods designed specifically for strings can be substantially more efficient. One is Bentley and

Sedgewick's multikey quicksort (Bentley & Sedgewick 1997, Bentley & Sedgewick 1998), in which the sorting proceeds one character at a time. When a contiguous series of strings with the same first character is created, sorting advances on to the next character. This method is known as multikey quicksort. A family of methods that yields better performance is based on radix sort (Andersson & Nilsson 1993, McIlroy, Bostic & McIlroy 1993, Nilsson 1996, Rahman & Raman n.d.). All these approaches work on the same principle: each string's initial character (or, more broadly, its first symbol) is used to assign it to a bucket. After that, the strings in each bucket are recursively distributed according to the next character (or symbol), and so on, until the buckets are small. After that, the final buckets can be sorted using a simple approach like insertion sort.

Theoretically, radix sorts are interesting because the leading characters in each string that are used to assign the string to a bucket are only examined once. These methodologies approach the least theoretical cost because these distinctive characters must be inspected at least once—if they are not inspected, the value of the string cannot be determined. Note that the cost is still O (n log n) for a set of n distinct strings. While each character must be examined just once, the length of the prefix that must be inspected grows proportionally to log of number of strings. For string sorting, there are various radix sort methods. In 1993, McIlroy, Bostic, and McIlroy (McIlroy et al. 1993) reported several in-place versions that partition the set of strings character by character. This method was observed by Bentley and Sedgewick (Bentley & Sedgewick 1997) to be the fastest general string-sorting method, and in earlier experiments (Sinha & Zobel 2003) found that it was usually the fastest of the existing methods. Sinha & Zobel 2003 called this method MBM radix sort. Stack-based versions of radix sort were developed by Andersson and Nilsson (Andersson & Nilsson 1993, Nilsson 1996), which build and destroy tries branch by branch as the strings are processed.

3. LITERATURE REVIEW

Nilsson [1] re-evaluated the method for managing buckets held at leaf nodes & shows better choice of data structures further improves the efficiency, at a small additional cost in memory. For sets of around 30,000,000 strings, the improved burst sort is nearly twice as fast as the previous best sorting algorithm.

Arne Anderson [2] had presented and evaluated several optimized and implemented techniques for string sorting. Forward radix sort has a good worst-case behavior. Experimental results indicate that radix sorting is considerably faster (often more than twice as fast) than comparison-based sorting. It is possible to implement a radix sort with good worst-case running time without sacrificing average-case performance. The implementations are competitive with the best previously published string sorting programs.

In Radix Sorting & Searching, Stefan Nilsson [1] discusses how distributive partitioning can be used in radix sorting algorithms in Adaptive Radix Sort. The adaptive size of the alphabet is determined as a function of the number of elements remaining, which is a natural extension of MSD radix sort. For binary strings, consider an example, using characters consisting of log k bits to distribute the components of a group of size k into (k) buckets is a natural choice. As a result, the number of buckets that must be browsed will be proportionate to the number of characters that must be reviewed. Because we may read several bits beyond the identifying prefix, the total number of bits read by the method may be more than for basic MSD radix sort. However, because this only happens once per string, it will only result in a linear increase in cost. In practice, the benefit of not having to inspect as many unnecessary empty buckets usually outweighs the additional cost.

Stefan Nilsson [2] describes basic version of forward radix sort as several writers [8, 9] have pointed out that MSD radix sort has a bad worst-case performance due to data fragmentation into many small sub-lists. This is solved by using a forward radix sort. The technique combines the benefits of both the LSD and MSD radix sort algorithms. LSD radix sort's key advantage is that it inspects a complete horizontal strip at a time; its main disadvantage is that it inspects all characters in the input. MSD radix sort simply looks at the strings' differentiating prefixes, so it doesn't make good use of the buckets. Forward radix sort starts with the most significant digit, buckets only once per horizontal strip, and only inspects significant characters.

Stefan Nilsson's Forward radix sort [2], complete version explains that in the basic version of Forward radix sort, we must visit all buckets in each pass, including the empty ones. By including a preprocessing step, this can be avoided. Firstly, make a list P of pairings during the preprocessing. A pair (i; c) denotes that character c will participate in the group splitting in pass i. Using this concept, we may create a three-step extended algorithm. To avoid looking at empty buckets, we first generate P, then sort P, and lastly perform the basic algorithm utilizing the information in P.

Jimenez- Gonzalez [4] introduced a new algorithm called Sequential Counting Split Radix sort (SCS-Radix sort). The three important features of the SCS-Radix are the dynamic detection of data skew, the exploitation of the memory hierarchy and the execution time stability when sorting data sets with different characteristics. They claim the algorithm to be 1:2 to 45 times faster compare to Radix sort or quick sort.

Shibdas Bandyopadhyay and Sartaj Sahni [5] developed a new radix sort algorithm, GRS, for GPUs that reads and writes records from/to global memory only once. The existing SDK radix sort algorithm does this twice. Experiments indicate that GRS is 21% faster than SDK sort while sorting 100M numbers and is faster by between 34% and 55% when sorting 40M records with 1 to 9 32-bit fields.

N. Ramprasad and Pallav Kumar Baruah [6] suggested an optimization for the parallel radix sort algorithm, reducing the time complexity of the algorithm and ensuring balanced load on all processors. [6] Implemented it on the "Cell processor", the first implementation of the Cell Broadband Engine Architecture (CBEA). It is a heterogeneous multi-core processor system. 102400000 elements were sorted in 0.49 seconds at a rate of 207 million/sec.

PARADIS by Rolland He [7] introduces two main improvements over previous radix sort algorithms. First, it introduces a speculative permutation/repair strategy, which can efficiently parallelize the in-place permutation process of radix sort. In addition, the algorithm also implements a distribution-adaptive load balancing technique to balance the work done by different processors during recursive calls. The algorithm is as follows:
After the first pass through the calls are independent and can be parallelized. Therefore, the work and depth can be represented as follows:

$$W(n) = 10W(n/10) + O(n)$$
$$D(n) = D(n/10) + O(n) \text{ which can easily be solved via } W(n) = O(n \log n) \text{ and } D(n) = O(n).$$

A further optimization can be made by parallelizing the bucketizing of elements. Instead of scanning through the elements sequentially, we can instead assign each processor an equal chunk of the data to place in the buckets, since placing each element in a bucket can be done independently. While this does nothing to lower the work, it does improve the depth:

$$D(n) = D(n/10) + O(1) \text{ which can be solved to give } D(n) = O(\log n).$$

4. PROPOSED SYSTEM

MSD Radix sort on natural language using Unicode:

Sorting MSD strings is attractive because you can perform sorting tasks without having to look up every input character. Sorting an MSD string is similar to quicksort in that it splits the array into separate parts, so applying the same method recursively to a subarray completes the sort. The difference is that MSD string sorting uses only the first character of the key type for partitioning, while quicksort uses a comparison that looks at the entire key. The first method described is to create a partition for each character value. Second, always create three partitions for sort keys where the first character is less than, equal to, or larger than the first character of the partition key.

We are using MSD radix sort given by R. Sedgewick and K. Wayne [9], Algorithms by tweaking it using Collator class [10] and pinyin4j library [11] to convert input from Devanagari and Chinese to comparable object thereby, sorting the input accordingly. We benchmarked the tweaked algorithm from 1k to 4M names and recorded the results to derive conclusions.

We also implemented LSD Radix sort, Dual Pivot Quick Sort, Tim Sort and Husky sort [12], made this individual sorting algorithm to sort Unicode characters just like our MSD tweaked with Collator or pinyin4j, benchmarked all algorithms from 1k to 4M partially random data as input names.

Input Size	MSD Radix	Tim Sort	LSD Radix	Dual Pivot Quick Sort	Husky Sort
1000	9.6	0.21	3.46	25	0.55
2000	13.8	0.36	2.53	54.4	1.32
4000	28.9	0.36	2.68	121.8	2.31
8000	47.8	0.92	3.48	590	5.23
16000	96.1	1.58	5.11	1456.2	6.25
32000	196	3.3	10.2	1662.6	8.81
64000	493.2	9.86	18.22	1427.1	17.12
128000	1139.9	27.15	38.83	2919.4	40.62
256000	2639.8	67.5	90	6387.2	87.12
512000	5266.1	169.74	314.2	16549.2	238.11
1024000	21085.1	380.49	645.98	47029.8	480.24
2048000	21306.8	832.76	1288.6	155186	1078.07
4096000	21581.3	1739.78	2572.97	531048	2144.79

Table 4.1 Benchmark statistics

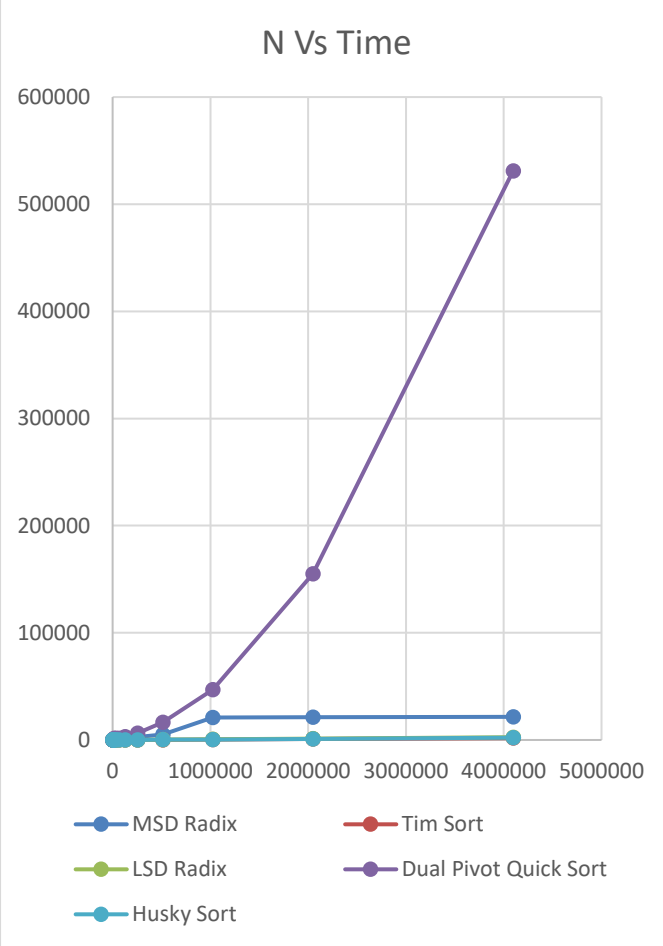

Table 4.2 N vs Time graph

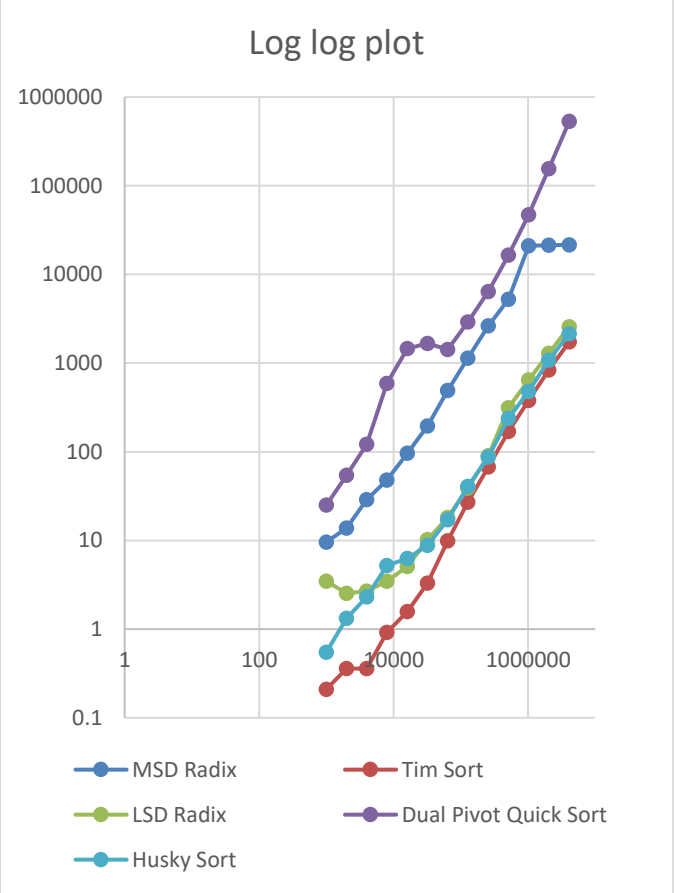

Table 4.3 Log-log plot

CONCLUSION

Theoretically, we had come to conclusion that MSD Radix sort is the fastest string sorting algorithm, but reality is Husky Sort [12] performs better than MSD Radix sort almost like LSD Radix sort and Tim sort.

To get a better picture of the scheme of things we made a log-log graph and it showed that Husky Sort is near to Tim sort and LSD Radix sort and all 3 are way better than MSD Radix sort. The worst performing string sorting algorithm is Dual Pivot Quick Sort as it nearly shows linearithmic growth. For small size input Tim sort performs better and then Husky sort, then LSD Radix sort then MSD radix sort and lastly Dual Pivot Quick Sort.

REFERENCES

1. Nilsson, S. 1996," Radix sorting & searching", Ph.D. thesis, Department of Computer Science, Lund University, Lund, Sweden

2. Arne Anderson, Stefan Nilsson, "Implementing radix sort", Journal of Experimental Algorithmic (JEA), 3, p.7-es, 1998

3. Mcllroy, P.M., Bostic, K., Mcllroy, M.D.: Engineering radix sort. Computing Systems 6(1), 5–27 (1993)

4. Danial, Jimenez- Gonzalez, J. J. Navarro, Josep [2002]", the Effect of Local Sort on Parallel Sorting Algorithms", 10th IEEE Euromicro Workshop on Parallel Distributed & Network Based Processing (EuromicroPDP' (02) 2002

5. Shibdas Bandyopadhyay and Sartaj Sahni. 4-Feb 2011," GPU Radix Sort for Multifield Records", IEEE Explore High Performance Computing (Hipc), 2010 International Conference on19-22 Dec. 2010 ,1-10, Dona Paula, 11824284, Dept. of CSE, University of Florida, Gainesville, FL 32611.

6. N. Ramprasad and Pallav Kumar Baruah.2007." Radix Sort on

the Cell Broadband Engine", at International Conference High Performance Computing (HiPC) – Posters, 2007.

7. PARADIS: A parallel in-place radix sort algorithm Rolland He https://dl.acm.org/doi/abs/10.14778/2824032.2824050

8. T. H. Cormen, C. E. Leiserson, and R. L. Rivest. Introduction to Algorithms. McGraw-Hill, 1990.

9. R. Sedgewick; K. Wayne. Algorithms. Addison-Wesley, 2011 ISBN-13:978-0-321-57351-3

10. https://docs.oracle.com/javase/7/docs/api/java/text/Collator.html

11. https://jar-download.com/artifacts/com.belerweb/pinyin4j/2.5.1/source-code/net/sourceforge/pinyin4j/PinyinHelper.java

12. R C HILLYARD, Yunlu Liaozheng, and Sai Vineeth K R Husky Sort, ACM Trans. Graph., Vol. 37, No. 4, Article 111. Publication date: August 2020