Literature Survey on MSD Radix sort and few other efficient radix sort techniques

Sai Saketh Kosuri - 002199401
Information Systems, Northeastern University, Boston, MA
Email: kosuri.s@northeastern.edu

Eswara Sai Nath Adusumalli - 001565578
Information Systems, Northeastern University, Boston, MA
Email: adusumalli.e@northeastern.edu

Santhosh Maddi - 002127881
Software Engineering Systems, Northeastern University, Boston, MA
Email: maddi.sa@northeastern.edu

Abstract:

Sorting is a fundamental algorithmic task [1]. Many general-purpose sorting algorithms have been created, but efficiency gains can be achieved by designing algorithms for specific kinds of data, such as strings [2]. On string data, where comparison is not a unit-time process, radix sorting techniques exhibit outstanding asymptotic performance. Radix sorting methods have excellent asymptotic performance on string data, for which comparison is not a unit-time operation [3].

In the first part of the paper, we are going to present a brief discussion about classic radix sort and its types (MSD and LSD), and in the second part, we are going to discuss new sorting techniques Adaptive sort, Forward Radix Sort, PARL sort and modified radix sort which is enhancements of classic radix sort and MSD string sorts and compare them with comparison-based sorting algorithms.

Introduction:

Sorting means arranging elements either in ascending or descending order based on the requirement. This order could be lexicographical or numeric. It arranges integers into either ascending or descending order and strings into alphabetical order.

Searching a sorted list or array takes less time as compared to unsorted list. 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. To evaluate the sorting algorithms, we need to consider many factors such as execution time, space complexity, type of input, range of data, stability and a few other factors. Based upon the working and program logic, sorting's are divided into 2 categories.

1. Comparison based sorting
2. Non-Comparison based sorting

Comparison based sorting works by comparing each other elements to find the sorted array/list. Generally, comparison sorting's are well known to a major section of people and think that they are more efficient. A comparison sort algorithm sorts items by comparing values between each other. It can be applied to any sorting cases. And the best complexity is O(n*log(n)) which can be proved mathematically.

A non-comparison sort algorithm uses the internal character of the values to be sorted. It can only be applied to some particular cases and requires particular values. And the best complexity is probably better depending on cases, such as O(n). All sorting problems that can be sorted with non-comparison sort algorithms can be sorted with comparison sort algorithms, but not vice versa.

Dual-pivot quick sort and Tim sort are the most common algorithms which are used by Java to sort primitives and objects respectively.

DUAL PIVOT QUICKSORT:

It is the default sorting algorithm used by JAVA to sort primitives. It is the enhancement of the classic Quick Sort algorithm which uses a single pivot. It is a combination of Insertion sort and Quicksort. Dual pivot quicksort uses the fact that insertion sort is faster when the number of elements in the array is small. Java uses insertion sort under the hood when the number of elements is less than 47 and when the array is larger than 47 it uses dual pivot quick sort. It uses 2 pivots instead of 1.

The larger the array, the more efficient is the dual pivot quicksort in comparison to the classic quick sort. The dual pivot quicksort is also quicker in case of arrays with repeated elements.

Why Quick Sort is not the best approach for sorting the strings:

A great variety of general-purpose sorting methods have been proposed. However, many of the best-known methods are not particularly well-suited to sorting of strings [2].

Consider for example the behaviour of quicksort (Hoare 1962, Sedgewick 1998). An array of strings to be sorted is recursively partitioned, the size of each partition approximately halving at each stage, if the pivot is well-chosen. As the sorting proceeds, the strings in a given partition become increasingly similar; that is, they tend to share prefixes. Determining the length of this shared prefix is costly, so they must be fully compared at each stage; thus, the lead characters are repeatedly examined, a cost that is in principle unnecessary [4].

Radix Sort

Radix sort is a non-comparison-based sorting commonly used to sort Integers. Along with integers it can be useful for strings and floating-point numbers as well [13]. It has been shown in some benchmarks to be faster than other more general-purpose sorting algorithms, sometimes 50% to three times faster. Radix sort is a stable algorithm and is suitable for computers having more memory. Radix sort is classified into 2 types.

1. Least Significant Digit Radix Sort
2. Most Significant Digit Radix Sort

Least Significant Digit Radix Sort

LSD radix sort process Integers from least significant bit and proceeds towards the most significant bit. Since LSD works from the least significant digit, it is considered a stable sort. For the data set with uniform length, Radix Sort works highly efficiently. For data sets with unequal length elements, the number of passes increases because depending on the maximum length of elements in the list, thus increasing Space & Time Complexity [5].

Most Significant Digit Radix Sort

The MSD radix sort begins with the most significant bit and works its way to the right. The technique that we use in the MSD radix sort is Key-indexed counting which is used to sort the strings according to their first character and then recursively sort the sub arrays corresponding to each character excluding the first character. First character is the same for each sub array.

Instead of the two or three partitions in quicksort, MSD string sort divides the array into one subarray for each possible value of the initial character. Small subarrays play an important role in the performance of MSD string sort.

One of the pitfalls of the MSD radix sort is that it can be relatively slow for subarrays containing large numbers of equal keys. MSD also uses two auxiliary arrays to do the partitioning which increases the space complexity of the MSD sort. For comparison sorting methods, performance of sorting depends on the order of the keys but for MSD string sort, the performance is concerned with the values of the keys. In the worst-case scenario, MSD string sort examines all the values in the keys and makes the time complexity O(N) [6].

In theory radix sort is perfectly efficient. It looks at just enough letters in each string to distinguish it from all the rest. There is no way to inspect fewer letters and still be sure that the strings are properly sorted. However, this hypothesis does not account for the fact that it is difficult to keep track of the piles. Because MSD employs recursion, it takes up more space than LSD. When working with a small number of inputs, MSD is much slower than LSD.

Complexity:

Best case time complexity is O(N) and worst-case time complexity is O(N*M) where M = the average length of strings.

REVIEW ON VARIOUS EFFICIENT RADIX SORT TECHNIQUES:

Arne Anderson in his paper "**Implementing Radix Sort**", discussed two techniques of radix sort **adaptive radix sort** which is a modified version of radix sort and **Forward radix sort** that overcomes the fragmentation problem of MSD radix sort[7].

Adaptive Radix Sort:

Author of this paper discussed the implementation of adaptive sort using "distributive partitioning" method by Dubielewicz 1978 which was used primarily to sort real numbers.

Dubielewicz's algorithm:

The algorithm sorts of n real numbers by dividing them into n equal-width intervals. For each interval with more than one element, the method is performed recursively. The numerals are divided into the same number of intervals as there are keys. Dubielewicz gets a sorting algorithm with O(n) expected time for uniform data and O (n log n) worst-case time by combining this approach with a Quicksort partitioning step using the median as the pivot element [8].

The above distributive partitioning method can be applied to radix sort. The ability to change the size of the alphabet is a natural extension of MSD radix sort. The size of the alphabet is determined by the number of elements left. The cost of visiting buckets matches the cost of inspecting bit patters from the strings using this approach. However, in the worst-case situation, after a few steps, the elements will be splitted into different groups, and the algorithm will only read a small constant number of bits at a time. As a result, the worst-case time complexity is (n + B), where B is the total number of distinguishing characters' bits.

Implementation of Adaptive Sort:

Arne Anderson's adaptive sort is implemented using two different alphabet sizes: 8 bits and 16 bits. The implementation is similar to classic MSD radix sort but for alphabets of larger size it applies more sophisticated heuristics to avoid inspecting many empty buckets.

Heuristic:

Bucketing is accomplished by using two successive 8-bit characters, resulting in a 65536-character alphabet. The goal is to keep track of which characters appear in the first and second positions. For example, if n1 and n2 distinct characters were identified in the first and second positions, we would only need to look at n1n2 buckets. This number is often substantially lower than the total 65536 buckets for ASCII-text.

Forward Radix sort:

MSD radix sort has a poor worst-case performance, according to several authors, because the data is fragmented into many sub lists. Forward Radix sort overcomes this problem. Forward radix sort combines the benefits of both LSD and MSD radix sort. The main advantage of LSD radix sort is that it inspects a complete horizontal strip at a time; nevertheless, it also inspects all characters in the input. MSD radix sort looks at the strings' differentiating prefixes but doesn't make efficient use of buckets. Some of the buckets would be empty in the case of MSD radix sort. Forward radix sort begins with the most significant character, buckets each horizontal strip only once, and examines just the significant characters.

Invariant: After ith pass, strings are sorted according to first 'i' characters

In forward radix sort, sorting is implemented by dividing the strings into groups. After each pass groups will be divided into subgroups and sorted according to the prefixes. Each group has a number assigned to it that represents the rank of the smallest string in the group in the sorted set. In ascending order, go through the unfinished groups and insert each string, tagged with the current group number, into respective bucket number. Bucketing is not required if all the inspected characters are equivalent.

Implementation:

Forward radix sort begins with the most significant character, buckets only once for each horizontal strip, and only examines the significant characters.

Separating the strings into groups is used to sort them. Initially, all strings are in the same group, which is labelled group 1. We utilize a linked list to maintain track of the groups, with each entry containing a list of objects with common prefixes.

A pointer is also present in each group which indicates the start of the following unfinished group. We can split a group in constant time using this data structure. The time it takes to traverse the unfinished groups is proportional to the total number of strings in these groups. An array of linked lists, one for each character in the alphabet, is used to implement the buckets. Each bucket has a list of elements at the top of the list with the common tag. Only if the tag of the entry differs from the tag of the list to be placed in the bucket is a new bucket entry created. This, combined with the standard technique of moving to a basic comparison-based sorting algorithm for small groups, results in a significant decrease in space.

Forward radix sort also used same heuristics as adaptive sort to avoid inspecting all empty buckets. Finally, the algorithm examines whether all inspected characters are equal in each pass. If this is the case, there will be no bucketing.

Experimental Results:

In this paper, with 8-bit and 16-bit alphabets, Anderson reported time metrics for MSD radix sort, Adaptive radix sort, and Forward radix sort. All these algorithms have been implemented using linked lists in a consistent manner. He also compared quicksort [Bentley and McIlroy 1993] [9], radix- 10 A. Andersson and S. Nilsson sort [McIlroy et al. 1993] [10], and the recently developed Multikey quicksort [Bentley and Sedgewick 1997] to competitor versions. It is concluded that Adaptive radix sort was the fastest algorithm on average. Forward radix sort was just somewhat slower, and it guarantees good worst-case behaviour. Large alphabets, such as the Unicode 16-bit character set, benefit from forward radix sort [].

Arne Maus, in his paper "**A full parallel radix sorting algorithm for multicore processors**", introduced a parallel left radix sort which can be used on ordinary shared memory multi-core machines [11].

A Parallel Left Radix Sort:

Sorting an integer array, a [] of length n on a shared memory computer with k cores is the challenge addressed in this paper. This work proposed a new complete parallel version PARL of the ARL [2] (Adaptive Left Radix) sequential sorting algorithm, which has been shown to be 3-5 times quicker than the usual Java sorting method Arrays. sort, which is a Quicksort implementation.

Reducing overhead and ensuring thread safety:

A specific design pattern is utilized to generate a thread pool of k threads (an ancient idea), the same number of threads as processor cores, to implement PARL effectively. When an object of the class Para Sort is created, this is done. The sorting technique is also found in the class Para Sort. This object's constructor begins the k threads in an infinite loop controlled by two Cyclic Barriers. The threads then wait for the next call to the PARL-implementing synchronized sorting procedure. Each call to the sorting method in one of these sorting objects will simply allow all threads to complete one more iteration of their loop, sort the array, and then release the calling main thread.

Implementation:

ARL is a radix sorting algorithm that sorts first on the leftmost most significant digit. In ARL, we find the maximum value in the part of an array a[] it is called to sort. This determines how many bits in the keys we must sort, and the number of bits is then divided into one or more digits (assumed to be of the same length, m bits wide, say 6-11 bits) for ease of use. In ARL, the sorting on a digit is done my moving their keys in their permutation cycles. We then select the first untouched key, a[h], and use its value j of the sorting digit to guide border[j] to the location where a[h] should be placed, but before we do so, we select the number there. The last element is then stored, and we've finished one permutation cycle, which can be any length between 1 and n, with all keys moved by this cycle being sorted on this digit. The next untouched key in a [] is then found and moved in a new permutation cycle until all keys have been moved in a permutation cycle. This is sorting, and unlike right radix, it does not require an extra array of length n. Permutation cycles sorting is also as fast as right radix sorting, but it is not as stable — equal-valued elements on the input may be swapped after ARL sorting. In PARL, when all the keys in the array are moved to their proper places, all values in border [] points move one place down in the array where all keys with that value have been stored

Results:

PARL Algorithm vs Parallel Quick Sort vs Quick Sort

Even when no parallelism is used, the parallel Quicksort is somewhat faster than Arrays. Sort. This parallel Quicksort is, on average, twice as quick as Arrays. Sort for bigger values of n. On all computers, sequential ARL is 3-5 times quicker than Arrays. Sort, although sequential ARL is up to 10 times slower than PARL on the server. PARL sorting is 10 to 30 times faster than the conventional Java sort function for sufficiently big values of n, which is a good result.

PARL sort is faster than classic quick sort on a shared memory computer with multiple cores. Sorting times are lower when sorting when sorting arrays with large number of elements [11].

Avinash Shukla, Anil Kishore Saxena, in his paper "**Review of Radix Sort & Proposed Modified Radix Sort for Heterogeneous Data Set in Distributed Computing Environment**"', introduced a Modified Pure Radix Sort for Large Heterogeneous Data Set

Modified Radix Sort

Avinash Shukla in his paper implemented an algorithm that is based on Distributed computing technology which is based on Divide and conquer principle.

In his paper he mentioned the following statements/problems, even though there are effective sorting models for serial execution on a single processor, efficient parallel sorting is still a difficulty. It has been discovered that no single solution is ideal for all data sets of different size, number of fields, and length. As a result, an attempt is made to choose a collection of data sets and optimize the implementation by changing the basic algorithm. In his paper he optimized the above problems by implementing new algorithm.

For example, given heterogeneous list is broken down into two major processes: numeric and string. Both these processes work parallelly. Let's consider x1 and x2 are two processes. Each process has different processor. Process x1 is then divided into sub-lists based on the length of the entries in the list. These lists are sorted based on the logic of even and odd logic at the same time. Passes are alternately transferred on the digits. After sorting these lists, combine them all and sort the main list again. Make a pattern in the case of x2. Get the selected strings using the one-of-a-kind pattern. The same string delivers the same numeric values for all these strings. The proposed algorithm now sorts the given strings using these numeric values.

After implementing this algorithm in two different machines, it is observed that modified radix sort is best for heterogenous data over all other sorting algorithms. Also, he stated that after MRS sort, GPU quick sort is the one of the best options for heterogenous data. This proposed algorithm optimizes both time and space complexities. With MRS sort and GPU

Quick sort, the results showed a 10:20 percent improvement in computational complexity compound [12].

References:

[1] Juha Kärkkäinen, Tommi Rantala, “Engineering Radix Sort for Strings”, SPIRE, 2008.

[2] Ranjan Sinha, Justin Zobel, “Efficient Trie-Based Sorting of Large Sets of Strings” ACSC, Volume 16, 2003.

[3] Peter M. Mcllroy, Keith Bostic, “Engineering Radix Sort”, Vol. 6, No. 1, 1993

[4] Bentley, J., Sedgewick, R., “Sorting strings with three-way radix quicksort”, Dr. Dobbs Journal, 1998.

[5] Sandeep Kaur Gill, Virendra Pal Singh, Pankaj Sharma, Durgesh Kumar: “A Comparative Study of Various Sorting Algorithms”, IJSSR, 2018, ISSN 2460 4010.

[6] https://www.informit.com/articles/article.aspx?p=2180073&seqNum=3

[7] Arne Andersson, Stefan Nillson, “Implementing radixsort” ACM Journal of Experimental Algorithmics Volume 31998 pp 7, 1998.

[8] Dobosiewicz, W., “Sorting by distributive partitioning. Information Processing Letters” 7, 1, 1-6, 1978.

[9] Bentley, J. L. and McIlroy, M. D, “Engineering a sort function. Software Practice and Experience”, 23, 11, 1249-1265, 1993.

[10] Bentley, J. L. and Sedgewick, R., “Fast algorithms for sorting and searching strings. In Proceedings of the Eight Annual ACM-SIAM Symposium on Discrete Algorithms”, pp. 360-369, 1997

[11] Maus, Arne, “A full parallel radix sorting algorithm for multicore processors”, 2011.

[12] Avinash Shukla, Anil Kishore Saxena: “Review of Radix Sort & Proposed Modified Radix Sort for Heterogeneous Data Set in Distributed Computing Environment” ISSN: 2248-9622, IJERA, Vol. 2, Issue 5, 2012.

[13] Adolf Fenyi, Michael Fosu, Bright Appiah: “Comparative Analysis of Comparison and Non-Comparison based Sorting Algorithms” International Journal of Computer Applications (0975 – 8887) Volume 175 – No. 28, October 2020