Literature Survey of papers and publications related to MSD Radix Sort

Thejas Ganjigunte Ramesh

Vikas Venkataraman Shastry

Nithya Viswanathan

College of Engineering,

Northeastern University

Abstract

We present a complete literature survey on relevant papers on MSD Radix sort and its comparison with other sorting algorithms.

Introduction

Sorting is a fundamental time critical computation that has a wide variety of applications. Like any other type of computation, while sorting strings, it is possible to use any comparison-based sorting algorithm. However, there are algorithms that have been proven to be simple and efficient when it comes to sorting strings. The best known amongst these is the MSD Radix Sort algorithm. Earlier, recognition was given to Quick Sort and Heap Sort both of which could reduce the $O(n^2)$ time-complexity of other slower sorting algorithms to O(nlogn) which was a major improvement. These were still comparison-based algorithms that compared any two elements at a given time unlike Radix sort, which is a distribution-based sorting algorithm. A problem arises when sorting strings on GPU is if the input size is greater than the memory of the GPU. Some researchers have proposed a solution to tackle this problem by using Merge Sort which partitions the input. However, this is not feasible if the input data is large because Merge Sort requires a lot of shared memory.

Analysis of "Engineering Radix Sort" by Juha K¨arkk¨ainen and Tommi Rantala [2]

The authors consider the problem of sorting a set of strings $R = \{s1, s2, \ldots, sn\}$ over the alphabet $\Sigma = \{0, 1, \ldots, \sigma - 1\}$ into the lexicographic order.[2]

Name	n	D	Description
URL	10^7	3.1×10^8	URL addresses with the protocol name stripped
Genome	3×10^7	3×10^8	strings of length 9 over the alphabet $\{a, c, g, t\}$ from real genomic data
Unique	3×10^7	2.8×10^8	unique words collected from English documents
Random A	3×10^7	4.6×10^8	strings of single character with the length chosen uniformly at random from $[0, 30)$
Random B	3×10^7	1.2×10^8	strings of length 30 with the characters chosen uniformly at random from $[32, 255)$

Figure 1

Figure 1 [2] shows a description of the data used.

The algorithm used is a simple version of the MSD Radix sort as follows.

MSDRadixSort(*R*, *depth*)
1 **if** $|R| < t$ **then** InsertionSort(*R*, *depth*)
2 **for** $s \in R$ **do** $B[s[depth]] := B[s[depth]] \cup \{s\}$
3 **for** $c \in \Sigma \setminus \{0\}$ **do if** $|B[c]| > 0$ **then** MSDRadixSort($B[c]$, *depth* + 1) [2]

For further optimization, they switch to Insertion sort for smaller buckets.[2] The paper clearly explains the discrepancy between theory and practical implementation and shows with examples how the real behavior is different from the expected behavior owing to the fact that theory often times gives a lot of importance to worst case scenarios. [2]

These difficulties of the practical implementation are addressed by the use of C (Counting) variant and D (Dynamic Buckets) variant.

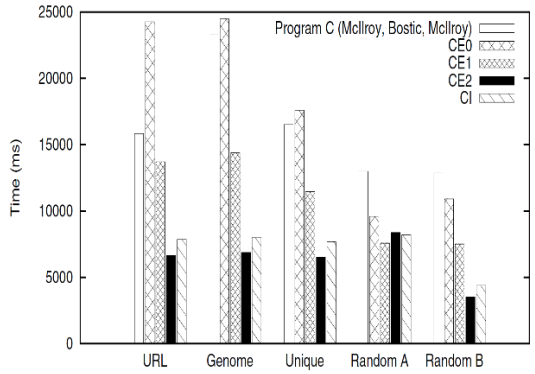

Figure 2

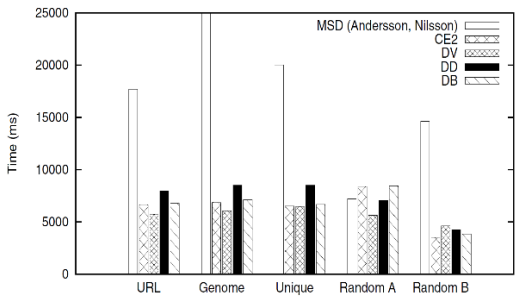

Figure 3

They perform a CI implementation and compare it with the CI implementation of McIlroy, Bostic and McIlroy which is commonly used as a reference implementation in literature. [2] They could successfully prove that their implementation was twice as fast as shown in Figure 2 [2] and Figure 3.[2]

In addition to this, they also discuss how to avoid slow accesses by the use of Algorithmic Caching and Superalphabet. Algorithmic caching reduces cache misses by copying the characters in advance to a place they can be accessed efficiently. The superalphabet technique treats a pair of characters as a single character in a larger alphabet. [2]

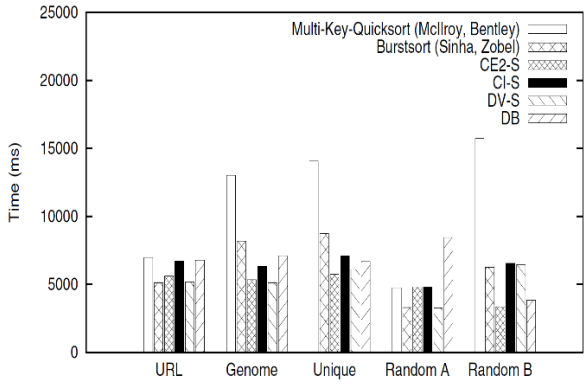

Figure 4

They perform comparison with well-performing Quicksort and Burstsort algorithms as shown in Figure 4. [2]

Implementation	Memory peak				
	URL	Genome	Unique	Random A	Random B
Program C [6]	0 MiB	0 MiB	0 MiB	0 MiB	0 MiB
MSD [2]	117 MiB	352 MiB	352 MiB	352 MiB	352 MiB
CRadix [7]	76 MiB	229 MiB	229 MiB	229 MiB	229 MiB
Adaptive [2]	117 MiB	352 MiB	352 MiB	352 MiB	352 MiB
Multi-Key-Quicksort [3]	0 MiB	0 MiB	0 MiB	0 MiB	0 MiB
Burstsort [11]	102 MiB	249 MiB	376 MiB	133 MiB	194 MiB
CE2	48 MiB	143 MiB	143 MiB	143 MiB	143 MiB
CI	10 MiB	29 MiB	29 MiB	29 MiB	29 MiB
DV	133 MiB	144 MiB	169 MiB	196 MiB	223 MiB
DD	39 MiB	117 MiB	117 MiB	117 MiB	117 MiB
DB	3 MiB	3 MiB	3 MiB	3 MiB	3 MiB
CE2-S	57 MiB	172 MiB	172 MiB	172 MiB	172 MiB
CI-S	19 MiB	57 MiB	57 MiB	57 MiB	57 MiB
DV-S	228 MiB	160 MiB	179 MiB	199 MiB	194 MiB

Figure 5

Figure 5 [2] shows the memory requirements for various implementations. It is important to note that their best implementation DB is a combination of small space requirement with a good runtime.

Analysis of "String sorting on Multi and Many threaded Architecture" by B Neelima, Anjjan S Narayan and Rithesh G Prabhu [1]

The authors focus mainly on the memory performance aspect of MSD Radix sort algorithm while lightly touching upon time-complexity. Graphic Processing Units (GPUs) are powerful parallel processors that can perform large number of tasks unable to be handled by the Central Processing Unit (CPU). In this paper, the authors address the challenges of variable length sorting and compare it to fixed length sorting. [1]

The first algorithm to be considered is the Burst sort. The main principle of working of burst sort is as follows: the input string is inserted into a burst-trie; this creates a sequence of buckets which are sorted but the data inside the buckets remains unsorted. Each string from the input is taken byte-by-byte and then assigned to the bucket. Buckets are sorted using their prefixes. [1]

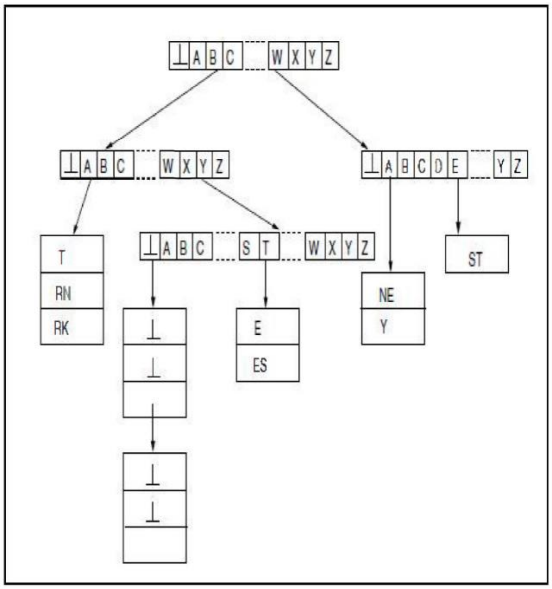

Figure 6

The structure of the burst sort algorithm is shown in Figure 6. [1] The authors further discuss the pros and cons of this algorithm and also mention ways to improve its performance by using a buffer-based technique. They also outline the disadvantages of this method.

The next algorithm to be analyzed is the Multi-key Quick Sort. The principle behind working of a quicksort algorithm is that a partitioning element (also known as pivot element) is chosen first and the elements are permuted by placing the elements of smaller size when compared to the pivot element in one end and others in the other end and recursively sort the two sub arrays. In extension, the multi-key quick sort is a hybrid algorithm consisting of quicksort and MSD radix sort. It eliminates the comparison of one entire string with another entire string.[1]
The implementation of this is as follows:

sort (s, n, d)
I. if n :s; I return;
2. choose a pivot element value v;
3. partition s based on v on component d to form partitions s <, S =, S > of sizes n<, n=, n>;
4. sort(s <, n<, d);
5. sort(s =,n=,d+l);
6. sort(s >, n>, d); [1]

Later, the authors discuss MSD Radix sort. The algorithm scans the input data and iteratively places the strings in buckets according to their prefixes. The buckets are arranged in lexicographic order. It then uses the next character to partition the bucket into smaller buckets. After splitting the bigger buckets to smaller ones, the algorithm switches to insertion sort or any other sorting technique for the smaller buckets. The LSD radix sort is similar to MSD Radix sort except that it starts with the least significant digit. [1]

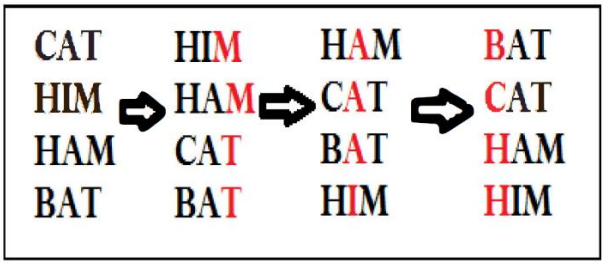

Figure 7

Figure 7 [1] represents the working of the LSD Radix Sort algorithm.

In the next section, the authors compare the performance of the fore-mentioned algorithms. Figure 8 [1] shows the description of the input data used.

Datasets	Size	Distinct Words ($x10^5$)	Word occurrences ($x10^5$)
URLs	304 Mb	12.898	100
Genome	302Mb	2.620	316.230
No Duplicate	382Mb	316.230	316.230
Duplicate	304Mb	70	316

Figure 8

All the programs have been compiled using gcc 4.6.3 and on a Linux operating system on Intel i7 processor model running at 3.4 GHz and 4 GS DDR3 1333 Zion RAM. All the details are measured using Linux clock function.
The performance of the algorithms was observed to be as shown in Figure 9. [1]

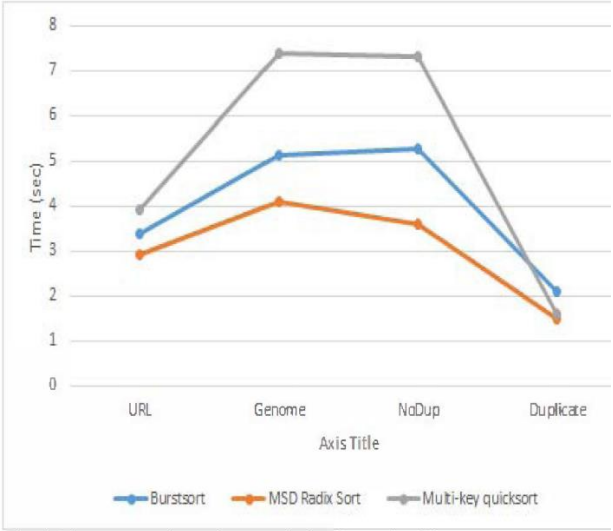

Figure 9

From the comparisons it is evident that MSD radix sort performs the best when the input is a large data set and perform least efficiently when the input contains a lot of duplicates. [1]

In the next section, the authors discuss the performance of these algorithms on GPU.

Figure 10 [1] shows the performance of these algorithms on a graphic processor.

Data sets	GPU sort removing singleton elements	GPU Radix sort	CPU Bursts-ort	CPU MSD Radix sort
URLs	0.89	0.79	3.368	2.929
Genomes	0.48	0.41	5.108	4.063
No Duplicate	1.18	1.65	5.257	3.584
Duplicates	0.82	1.08	2.128	1.563

Figure 10

The performance improvement achieved is very significant. This clearly gives us the insight that using GPU for string sorting can achieve immense improvement in sorting speed when compared to CPU implementations. [1]

The authors propose a parallel multi-key quicksort using ternary search trees implementation on GPU which can yield efficient results for sorting large datasets. [1]

Analysis of "A survey, analysis and comparison of different sorting algorithms" by Ashok Kumar Karunanidhi. [3]

In this paper, the author compares the performance of different sorting algorithms: Insertion Sort, Selection Sort, Bubble Sort, Quick Sort, Heap Sort, Bucket Sort, Radix Sort, Merge Sort and Counting Sort. This is a mixed pool of algorithms because it consists of both Comparison based and non-comparison based algorithms. Based on their experiments, these are the results the author observed. Figure 11 [3] shows the performance of comparison-based sorting algorithms and Figure 12 [3] shows the performance of non-comparison based sorting algorithms.

Sorting Method	Best Case	Worst Case	Average Case	Stable	Method
Bubble Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	Yes	Exchange
Modified Bubble Sort	$O(n)$	$O(n^2)$	$O(n^2)$	Yes	Exchange
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	Yes	Selection
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$	Yes	Insertion
Merge Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	Yes	Merge
Quick Sort	$O(n \log n)$	$O(n^2)$	$O(n \log n)$	Yes	Partition
Randomized Quick Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	Yes	Partition
Heap Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	No	Selection

Figure 11

Sorting Method	Best Case	Worst Case	Average Case	Stable
Radix Sort	$O(n \cdot \frac{k}{s})$	$O(n \cdot \frac{k}{s})$	$O(n \cdot \frac{k}{s})$	No
Counting Sort	$O(n + 2^k)$	$O(n + 2^k)$	$O(n + 2^k)$	Yes
Bucket Sort	$O(n \cdot k)$	$O(n^2)$	$O(n \cdot k)$	Yes

Figure 12

After running several experiments of various types and sizes, the author makes a summary of the better suited algorithms for various types of inputs in Figure 13. [3]

Problem Definition	Sorting Algorithms
Problems which do not require any extra memory to sort the array of data.	Insertion Sort, Selection Sort,Bubble Sort.
Business applications and Database applications which required a significant amount of extra memory to store data.	Merge Sort.
Problems with the input is extracted from a uniform distribution of a range (0,1).	Bucket Sort.
To sort the record with multiple fields and alphabet with constant size.	Radix Sort
Problems with Small input data sets.	Insertion Sort.
Problems with Large input data sets.	Merge Sort, Quick Sort, Heap Sort.
Problems with the repeated data items in the input array.	Counting Sort.
To sort the record based on address.	Bucket Sort.
Problems with the repeated data items in the input array and the resultant output should maintain the relative order of the data items with equal keys.	Bubble Sort, Merge Sort, Counting Sort, Insertion Sort.
Problems with repeated data items in the input array and the resultant output does not need to be maintained the relative order of the data items with equal keys.	Quick Sort, Heap Sort, Selection Sort.

Figure 13

From his experiments, the author concludes that every sorting algorithm has its own use case and sometimes even complex algorithms use simple sorting algorithms like Selection Sort and Insertion Sort despite their quadratic performance. Sometimes, two algorithms are combined to solve problems using a hybrid method. [3]

Conclusion

While the first paper in this literature survey analyzes the efficiency and time complexity of the MSD Radix Sort algorithm for strings and compares it to other algorithms, the second paper analyzes it's performance on CPU as well as GPU and notes the significant speed-up in performance while also making a similar comparison with other algorithms. The third paper compares various different sorting algorithms and provides suggestions on which algorithm is more suited for different input types. MSD Radix Sort can be concluded to be the superior sorting algorithm both in terms of time complexity and space complexity for sorting large inputs of string data. While, this stands true, it is important to note that there are implementations of MSD Radix sort that cut to simple algorithms like Insertion sort to increase efficiency.

References

[1] B Neelima, Anjjan S Narayan, Rithesh G Prabhu: String sorting on Multi and Many threaded Architecture

[2] Juha K¨arkk¨ainen, Tommi Rantala: Engineering Radix Sort

[3] Ashok Kumar Karunanidhi: A survey, analysis and Comparison of Sorting Algorithms.