INFO 6205 Fall 2021 Project: A review of MSD radix sort using adaptive and forward techniques

-Jason Paul Darivemula(002198774) & Ebenezer Ajay Williams (002166250)

***Abstract*—We present a study on the MSD radix sort and explore the working of the Adaptive radix sort and Forward Radix sort in particular. The main goal of this work is to explore the idea of the MSD implementations presented inorder to gain insight to where else it can be employed. We perform a literature survey of an article that compares the Adaptive radix sort and Dorward radix sort with other sorts by sorting ASCII strings.**

***CCS Concepts:* • Theory of computation in Sorting and searching; Data structures and algorithms for data management; • Software and its engineering in Data types and structures; • Computer systems organization under Multicore architectures;**

***Index Terms*—Sorting, Quicksort, Timsort, Radix Sort, Husky Sort, Insertion sort, Complexity, Comparison Sorting**

I. INTRODUCTION

With the dawn of the internet, data began to grow exponentially everyday. The proper ordering of this data was a tradeoff between time and computational power. However with the introduction of efficient sorting algorithms this data can be sorted in a fraction of time. There are two conditions that need to be satisfied for data to be sorted: 1) the resultant data ordering is a permutation of the original data elements. 2) the order must have a monotonic relation(either increasing or decreasing).

Although sorting algorithms are fairly a straightforward topic which have been explored through the ages, the future requires sorting algorithms that are further efficient to sort the futuristic volumes of data. In the past century every few decades brought forth a new method of sorting by either suggesting new approaches or by employing different data structures or strategies. As far as comparison sorting algorithms are concerned, the aspects to be considered include the stability, the partitioning, merging, swapping and above all the comparison operation itself. Based on the data elements being compared and the cost of comparison an algorithm may be chosen. An example between two algorithms that are quadratic in nature O(n^2) would be to choose the insertion sort algorithm if comparison was expensive while swapping was cheap and the selection sort algorithm if comparison was cheap and swapping was expensive. [2] TimSort, an algorithm developed by a software engineer in 2002, demonstrated its efficiency for sorting actual data, and was adopted as the default sorting algorithm in programming languages such as Python and Java. The reason why this algorithm works well is because it works well with computer architecture(cache dealing) and realistic distributions of data(partially sorted data).

In [1] we see that the authors approach sorting with the idea of minimizing the number of array accesses. The authors encode the elements to 64 bit longs thus speeding up the comparison time and swap the actual data elements at the same time as swapping the longs. They call this approach the HuskySort.

Radix sort, a sorting algorithm that uses significant bits comparison can be dated back to the day of tabulating machines. It was not until 1954 when Harold H. Seward, a computer scientist at MIT solved the issue involved with variable allocation of buckets of unknown sizes and developed the first memory efficient radix sort algorithm.

II. LITERATURE SURVEY

In their work titled Implementing Radix Sort[3] , the authors Arne Andersson and Stefan Nilsson explain two new approaches and optimizations to the standard MSD Radix Sort. Their claim that MSD Radix sort is best to sort characters, is based on the idea that it inspects only the distinguished prefixes, the minimum number of characters that has to be inspected to assure that the strings are, in fact, sorted. According to this idea, distinguished prefixes are the shortest pair-wise different prefixes in a set of strings $x_1,x_2,x_3\ldots,x_n$ that constitutes the data being sorted. For example, given the strings “apple”,”apes” and “append”, the distinguished prefixes would be “app”,”ap” and ”appe” respectively.

One drawback of radix sort is the issue that there might be a need to inspect a large number of empty buckets. They mention two traditional approaches to overcome this drawback. This first one being the common approach of switching to an elementary sorting algorithm such as Insertion sort after identifying a cut off point. The other solution is by Bentley and Sedgewick [4], is to replace the bucket based sorting with a comparison sorting technique. This algorithm seems to be more or less the same as quicksort with string comparisons being replaced by character comparisons.

Arne and Stefan in their work take a different

Manuscript submitted December 05, 2021

Authors : Jason Paul Darivemula(002198774) & Ebenezer Ajay Williams (002166250)

approach and present their study on two new algorithms, the adaptive radix sort and the forward radix sort. Since their motive was to study the working of different algorithms and not determine the fastest algorithm for a particular architecture, they use a LinkedList to store the input array. An optimization they follow is not to swap the strings themselves but instead swap the pointers to the starting characters of the strings instead. This technique in several architectures proves to be less expensive. While implementing radix sort , they avoid the bucket table for every invocation by keeping track of only the occupied buckets in a recursion stack and clearing the bucket table at every invocation of the radix sort. They recursively sort the buckets in a reverse fashion. So, in this implementation for lowercase english characters, at a particular invocation of the MSD radix sort, they would push the buckets corresponding to the characters 'a-z' onto the stack and then unroll the stack starting from 'z' onwards to 'a'.The drawback to this is that the size of the stack is fixed. They overcome this issue by suggesting that if the list on top of the stack and the list about to be pushed are both sorted there would be no use of allocating a new stack record and so they append one list to another. This is done by switching to a comparison based technique when the number of elements to sort was at k(cut-off value) or less than k. Thus the size of the stack would be a maximum of n/k. This optimization minimizes the stack size and also improves run time.

To avoid looking at empty buckets they adopt a technique suggested by McIlroy, Bostic, and McIlroy [5] where the maximum and minimum characters in the strings are kept track of and only the occupied buckets between those characters are looked at. Another optimization they make is by taking advantage of the clustering present in the data. Consecutive elements with a common character and treated as a single unit and moved into a bucket. This helps improve the algorithm for practical situations.

a. *Adaptive Radix Sort*

The authors employ a technique first suggested by Dobosiewicz in [6] where he employs distributive partitioning. The algorithm distributes the input into n intervals by a recursive process as long as n is greater than 1. Hence the number of intervals is the number of keys and by employing the standard quicksort partitioning a sorting algorithm with O(n) expected time for uniform data and O(n log n) worst-case time is achieved. They apply this idea of distributive partitioning to their radix sort. To make the sort adaptive(i.e to leverage the input to improve the algorithm) they suggest modifying the size of the alphabet as a function of the number of elements. Comparing with the N-trees data structure suggested by Ehrlich [7] they see a close relationship between using N-Trees to partition distributively and the adaptive sorting algorithm. From Tamminen [8] they derive that the time taken to sort is utmost 1.8 times for uniformly distributed data and thus from the relationship it can be concluded that adaptive sort works in almost linear time. This is because the maximum depth of the tree would be 4 and this makes it linear.

They implement Adaptive radix sort by using 8 bits and 16 bit alphabet sizes. They use a heuristic consisting of two consecutive 8-bit characters that is used for bucketing, yielding an alphabet of size 65 536 (i.e 2^16). When the size of the list falls below 1500 they use 8 bit character bucketing elsewhile they use 16-bit dual character bucketing. By keeping track of the characters that occur in the first(pos_1) and second position(pos_2), they need to inspect only the pos_1pos_2 buckets. To understand this we consider the alphabet to be lowercase english letters which falls in the ASCII range and has 8-bit characters. The usual implementation of radix sort would bucket the strings into 26(alphabet size) buckets. The adaptive radix sort would bucket the strings into 325(i.e ${}^{26}C_2$) buckets. For ease of understanding consider the buckets to be marked as aa,ab,ac…..zx,zy,zz. They suggest that for larger alphabets it is useful to apply a more sophisticated heuristic to avoid looking at too many empty buckets...

b. *Forward Radix Sort*

It is common knowledge that radix sort has bad worst-case performance due to fragmentation of data into small sublists. The forward radix sort was designed to overcome this issue. The idea behind this is combining the advantages of both the MSD and LSD radix sorts. Forward radix sort combines the LSD advantage that it inspects a complete horizontal strip at a time and the MSD advantage of inspecting only the distinguishing prefixes of the strings.

The forward radix sort approach is to start with the most significant digit and perform bucketing only once for each horizontal strip, by inspecting only significant characters. The invariant is that after the ith pass, the first i characters of the string are sorted. The sorting is performed by grouping strings. The strings are all in one group at start and will be split into smaller groups and after 'i' passes the strings with the same first i characters will be in the same group. The groups are sorted according to the prefixes and each group is associated with a number to identify it. The group number is the rank of the smallest string in it in the final sorted output..

Manuscript submitted December 05, 2021
Authors : Jason Paul Darivemula(002198774) & Ebenezer Ajay Williams (002166250)

The finished and unfinished groups are kept track of - a finished group contains only one string or all strings in the group are equal and not longer than i (pass count).

The ith pass can be performed using the following steps:

- Traverse the unfinished groups and insert each string x, tagged by its current group identity number, into bucket number x(i), where x(i) is the ith character of x.
- Traverse the buckets and revert strings back to their respective groups using the group identity number in the order as they occur within the buckets. No bucketing is done if the characters are equal.
- Last step, traverse the groups individually. If the nth string in group g(i.e group identity) differs from its predecessor in the ith character, split the group at this string and it's group identity would be as g + n - 1.

The cut off number of strings to switch to Insertion sort is kept at 30 for the experiments conducted.

The time complexity of the forward radix sort is $O(S + n + m * S_{max})$, where S is the sum of the number of characters of distinguishing prefixes, S_{max} is the length of the longest distinguishing prefixes, and m is the size of the alphabet. S and n terms originate from the fact that the algorithm inspects each distinguishing character and string at least once. The $m* S_{max}$ term is from the fact that the algorithm makes S_{max} passes through m buckets.

The worst case occurs when the algorithm does not perform any bucketing in a pass when all inspected characters are equal. The worst-case time complexity is given by $O(S + n + m^2)$. If the number of strings remaining in unfinished groups are larger than m, then the cost of bucketing is at maximum, the cost of reading the characters. If less than m strings remain, there can be at most m passes in which any splitting of groups occurs ,thus the total number of buckets visited is $O(m^2)$ in this case.

c. *Experimental Results*

The authors compare time measurements for MSD radix sort, Adaptive radix sort, and Forward radix sort with 8-bit and 16-bit alphabets and compare them with standard core library sorts. The measure taken into consideration is the total number of seconds of CPU time consumed by the kernel on behalf of the program. They have concluded that a carefully coded radix sorting algorithm outperforms comparison-based algorithms by a large margin.

They conclude the advantage of Forward radix sort is that it reduces the amount of bucketing. In real world implementations the cost associated with each bucket operation will be larger than the cost encountered in the experiments conducted. By the use of only one tag per group, and not one tag per element the authors reduce the total number of tags to less than 10% of the number of elements and thus decrease the run time by roughly 10%. By not splitting consecutive groups that are already sorted they achieve a drastic reduction in space complexity. They find that the total number of groups created is roughly 6% of the total elements to be sorted and thus reduce the run time by almost 4%.

III. Conclusion

The authors conclude that Forward radix sort the free-choice method was best for worst case but Adaptive radix sort 8-bit alignment was slightly better, but the differences were small. The Adaptive radix sort was the fastest algorithm for the average case. Forward radix sort was only slightly slower and had a guaranteed worst-case behavior which would work well for large unicode 16 bit character alphabets.

IV. Conflict of Interest

The authors declare no conflict of interest

V. Author Contributions

All authors discussed the contents of the manuscript and contributed to its preparation.

References

[1] R C HILLYARD, Yunlu Liaozheng, and Sai Vineeth K R. 2020. Huskysort. ACM Trans. Graph. 37, 4, Article 111 (August 2020), 9 pages. https://doi.org/ 10.1145/1122445.1122456

[2] T Peters. 2002. Timsort - Python. Retrieved November 3, 2020 from https://svn.python. org/projects/python/trunk/Objects/listsort.txt

[3] A. Andersson and S. Nilsson. Implementing Radix sort. In Proceedings of the 35th Annual IEEE Symposium on Foundations of Computer Science, pages 714{721, 1994. https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.54. 4536&rep=rep1&type=pdf

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

[5] P. M. McIlroy, K. Bostic, and M. D. McIlroy. Engineering radix sort. Comp. Systems, 6(1):5{27, 1993.

Manuscript submitted December 05, 2021

Authors : Jason Paul Darivemula(002198774) & Ebenezer Ajay Williams (002166250)

[6] W. Dobosiewicz. Sorting by distributive partitioning. Information Processing Letters, 7(1):1{6, 1978.

[7] G. Ehrlich. Searching and sorting real numbers. Journal of Algorithms, 2(1):1{12, 1981.

[8] M. Tamminen. Analysis of N-trees. Information Processing Letters, 16(3):131{137, 1983.

Manuscript submitted December 05, 2021

Authors : Jason Paul Darivemula(002198774) & Ebenezer Ajay Williams (002166250)