Most Significant Digit Radix Survey

Yanrong Huang

Yuan Lian

Yuxin Liu

College of Engineering, Northeastern University

INFO 6205, Program Structure & Algorithms

Dr. Robin Hillyard

Date: Dec 4th 2021

Abstract

This paper compares the performance of various sorting algorithms through the example demonstration. An implementation method of MSD radix sort is presented. Compared to other implementation methods of radix sort, it is easy to be extended to asynchronous parallel radix sort algorithm.

Keywords: Sorting, Algorithm, Radix Sorting, MSD Sorting.

1. Introduction

Sorting is a fundamental problem in computer science, with broad applications. The moment the algorithm run-time and space complexity were formalized, sorting algorithms were at the forefront for analysis and improvement. Early recognition (Al-Darwish et al. 2004) was given to Quicksort and Heapsort algorithms because they improve the $O(n^2)$ order of running time of other slow sorting algorithms to O(n log n). Heapsort is O(n log n) in the worst case, while Quicksort is O(n log n) in the average case. In practice, Quicksort runs faster than Heapsort most of the time, although it exhibits $O(n^2)$ worst-case order of running time, which happens, for example, when the input data is nearly sorted. In general, sorting algorithms can be divided into two categories: 'com-parison-based' and 'distribution-based'. A comparison-based algorithm, like Heapsort or Quicksort, sorts by comparing two elements at a time. On the other hand, a distribution-based algorithm, like radix sort, works by distributing the elements into different piles based on their values.

2. Background

Radix sort algorithms fall into two classes: MSD (most significant digit) and LSD (least significant digit). Radix sort algorithms process the elements in stages, one digit at a time (He et al. 2011). A digit is a group of consecutive bits with the digit size (number of bits) set at the beginning of the algorithm. MSD radix sort starts with the most significant (leftmost) digit and moves toward the least significant digit. LSD radix sort of does it the other way. LSD distributes the elements into different groups – commonly known as 'buckets' and treated as queues (first-in-first-out data structure) – according to the value of the least significant (rightmost) digit. Then the elements are recollected from the buckets, and the process continues with the next digit.

On the other hand, MSD radix sort first distributes the elements according to their leftmost digit and then calls the algorithm recursively on each group. MSD needs only to scan distinguishing prefixes, while all digits are scanned in LSD. For example, for radix-2 MSD (i.e., digit size = 1 bit), two buckets are used, and the elements are distributed into either bucket depending on the value of the most significant bit. Then

the process continues with the next bit, considering only groups with more than one element until all the bits have been scanned. Clearly, such an algorithm uses O(n) space for the combined two buckets and is able to sort n non-negative integers in the range [0,m] in O(n log m) order of running time.

2.1 In-place MSD radix sort implementations

Binary MSD radix sort, also called binary quicksort, can be implemented in-place by splitting the input array into two bins - the 0s bin and the 1s bin. The 0s bin is grown from the beginning of the array, whereas the 1s bin is grown from the end of the array (Xiang, 2008). The 0s bin boundary is placed before the first array element. The 1s bin boundary is placed after the last array element. The most significant bit of the first array element is examined. If this bit is a 1, then the first element is swapped with the element in front of the 1s bin boundary (the last element of the array), and the 1s bin is grown by one element by decrementing the 1s boundary array index (Huang, 1999). If this bit is a 0, then the first element remains at its current location, and the 0s bin is grown by one element. The next array element examined is the one in front of the 0s bin boundary (i.e., the first element that is not in the 0s bin or the 1s bin). This process continues until the 0s bin, and the 1s bin reach each other. The 0s bin and the 1s bin are then sorted recursively based on the next bit of each array element. Recursive processing continues until the least significant bit has been used for sorting. Handling signed integers requires treating the most significant bit with the opposite sense, followed by unsigned treatment of the rest of the bits.

In-place MSD binary-radix sort can be extended to larger radix and retain in-place capability. Counting sort is used to determine the size of each bin and their starting index. Swapping is used to place the current element into its bin, followed by expanding the bin boundary. As the array elements are scanned, the bins are skipped over, and only elements between bins are processed until the entire array has been processed and all elements end up in their respective bins. The number of bins is the same as the radix used - e.g.,

16 bins for 16-radix. Each pass is based on a single digit (e.g., 4-bits per digit in the case of 16-radix), starting from the most significant digit. Each bin is then processed recursively using the next digit, until all digits have been used for sorting.

Neither in-place binary-radix sort nor n-bit-radix sort is stable algorithms.

2.2 Stable MSD radix sort implementations

MSD radix sort can be implemented as a stable algorithm but requires the use of a memory buffer of the same size as the input array(Wu et al. 2015). This extra memory allows the input buffer to be scanned from the first array element to the last and move the array elements to the destination bins in the same order. Thus, equal elements will be placed in the memory buffer in the same order they were in the input array. The MSD-based algorithm uses the extra memory buffer as the output on the first level of recursion but swaps the input and output on the next level of recursion to avoid the overhead of copying the output result back to the input buffer. Each of the bins is recursively processed, as is done for the in-place MSD radix sort. After the sort by the last digit has been completed, the output buffer is checked to see if it is the original input array, and if it's not, then a single copy is performed. If the digit size is chosen such that the key size divided by the digit size is an even number, the copy at the end is avoided.

3. Methodology

3.1 Traditional (Recursive) MSD Radix Sort Implementation

MSD radix sort is named after its sorting operation. Numbers are examined, starting from the most significant to the least significant digits (Al-Darwish et al. 2004). A high-level pseudocode and diagram for the recursive MSD radix sort algorithm sorting are presented in Figure1. In this figure, b[i] refers to a bucket corresponding to digit value I and holds a pointer to a linked list of numbers whose digit value of the radix being examined equals i.

The sorting process begins with passing the entire input(main vector) to the Recursive MSD function by reference. This allows the function to make changes directly to the input. Numbers from the input vector are placed into buckets based on the value of the corresponding digits being examined (most to the least significant digit order). Each of these buckets is sent to the Recursive MSD function for further execution. There are two conditions to stop the recursive call: when the size of the bucket is equal to one and if the digit count is smaller than one. The latter indicates that the prior call has worked on the least significant digit, and there are two digits left. Since buckets from recursive calls are sorted, they are merged into the main input.

3.2 Non-recursive MSD Radix Sort Implementation

The fact that MSD radix sort begins its sorting operation by examining numbers from the most significant digit provides the opportunity to allocate each of the numbers in their final nuckets after the first iteration (Aydin et al. 2013). Therefore, each bucket can be considered as an independent vector to be sorted in this range. Non-recursive bucket data structures are implemented as a two-dimensional vector. We also use a middle_bucket data structure to sort individual vectors belonging to the first_bucket for the subsequence digits as described below. Before the sorting process begins, the maximum number of inputs is found, and its digit count is calculated. Digit count determines the number of iterations of MSD radix sort over the input.

The sorting process begins by calculation the most significant digit of each number in the main vector. It then adds each number with the corresponding digit value i (i= 0…9) in i th bucket of the first_bucket. After the first iteration, all numbers are written to the first_bucket vector. Each index in this structure is considered an individual vector which is sorted for the next most significant digit using the middle_bucket vector. The content of the middle_bucket is processed, and the first_bucket is empty. During the next iteration, each index of the second_bucket is processed based on the

third most significant digit, and numbers are sent from second_bucket to the first_bucket using the middle_bucket in reverse order.

During this sorting process, numbers can be either in the first_bucket or the second_bucket. At each iteration, the index length is checked. If length is 1, the index can be added to the main vector until reaching an index that's length is greater than 1. This technique decreases the size of the input for further iterations.

3.3 Hybrid MSD Radix Sort Versions

The hybrid versions use a combination of MSD radix sort and Quicksort algorism (Al-Darwish et al. 2004). MSD radix sort is deployed to create independent vectors, and QuickSort is applied to sort each of the generated vectors efficiently. As described earlier, digit count of the maximum number in the input vector determines the number of iterations needed in the complete MSD radix sort. When the digit count = d, the input will be sorted by MSD radix sort after d iterations.

In the hybrid case, MSD radix sort is applied to input a number of times, referred to as a step to create a number of independencies.

4. Implementation of MSD radix sorting

The MSD radix sorting process is similar to the sorting process of playing cards in people's lives. The realization process is divided into two types: high priority and status priority. Our report demonstrates the high order priority process. The status priority process is to sort the records according to the highest value when sorting, and then sort them by the highest value on this basis, and so on. From high to low, each pass is based on one bit of the keyword, and all records are sorted on the basis of the previous pass, until the lowest bit, the whole process of radix sorting, is completed (Andersson et al. 1994). The choice between the low priority process and the high priority process depends on the sorting result you expect. In the actual operation process, it is not necessary to strictly follow the order from high to low or from low to high—the basis for grouping each trip can be flexibly selected according to the desired result sequence.

The main design decision is whether to represent the input by a linked list or an array. The array representation has the advantage that the permutation can be performed in place, but we do not get a stable algorithm. Algorithms based on linked lists have the advantage of being simple to implement. Earlier experiments (McIlroy et al. 1993) indicate that the list-based implementations are faster on some architectures and array-based implementations on others. Since our primary goal is to compare different algorithms and optimization schemes and not to write the fastest possible sorting program for a particular architecture (Adersson et al. 1994) we have chosen to implement all of our algorithms using linked lists.

When implementing MSD radixsort we want to avoid using a new bucket table for each invocation of bucketsort. This can be arranged if we use an explicit stack to keep track of the ow of computation (McIlroy et al. 1993). The last bucket is treated first and the other sublists to be sorted are pushed on the stack. The sublists on the stack that are waiting to be sorted are suggested by the dotted line. One possible problem with this implementation is that the stack can be sizable. It is not difficult to construct an example where the stack will contain one entry for each key. We suggest a simple way to overcome this problem. The idea is to take advantage of the fact that short sublists, containing at most k elements, are sorted using a comparison-based algorithm. This sorting is performed before the lists are pushed onto the stack. This means that if both the list on the top of the stack and the next list to be pushed onto the stack contain at most k elements we do not need to allocate a new stack record. Both lists are already sorted and we can simply append the new list to the end of the list on top of the stack. In this way the stack will never contain two consecutive entries that both have less than k elements. Hence, the total size of the stack will be at most 2n/k. In practice, however, the stack will typically be much smaller. Not only does this simple optimization give a considerable reduction of the stack size, it is also likely to improve the running time.

When choosing the size of the alphabet we are faced with a fundamental problem: A large alphabet reduces the total number of passes but increases the total number of buckets that must be inspected. In practice one can sometimes use heuristic methods

to be able to use larger alphabets. For example, if the elements are ASCIIencoded strings from the English language they typically only contain characters in the range 32 to 126. A simple heuristic (McIlroy et al. 1993) to avoid inspecting many empty buckets is to record the minimum and maximum character encountered during the bucketing phase and then only look at the buckets in between these extreme values.

A certain amount of clustering is often present in many kinds of data. One way to take advantage of this fact is to treat consecutive elements with a common character as a single unit when moving them into a bucket. Since we are using a linked-list representation it is possible to move this sublist of identical elements in constant time. This simple optimization improves the performance in many practical situations.

5. Conclusion

Compared with other sorting algorithms, MSD radix sort without comparing keywords, with the advantage of lower computational time complexity. MSD radix sort has ideas clear, simple structure, easy to expand, easy to implement, etc. Features. The future research work is to further test the algorithm of this paper on massive data and make a quantitative comparison with other sorting algorithms.

References

Al-Darwish, N. 2004. Formulation and analysis of inplace MSD Radix sort algorithms. *Journal of Information Science (2005)*, pp.467-481.

Andersson, A. and Nilsson, S. 1994. A new efficient radix sort. *In Proceedings of the 35th Annual IEEE Symposium on Foundations of Computer Science (1994)*, pp.714-721.

Aydin, A. and Alaghband, G. 2013. Sequential & Parallel Hybrid Approach for Non-Recursive Most Significant Digit Radix Sort. *IADIS International Conference Applied Computing (2013)*, pp.51-58.

He, Y., Yan, J. and Bai, Y. 2011. Performance comparison of sorting algorithms and base sorting in the data application in classification. *Fujian Computer* (2011), pp.90-101.

Huang, J. 1999. An implementation method of MSD radix sort. China Academic *Journal Electronic Publishing* (1999).

Wu, Y., Kong, F. and Sun, B. 2015 Applied Data Structure. *Tsinghua Publishing* (2015)

Xiang, L. 2008. Comparison and analysis of various sorting algorithms in data structure. *Taiyuan Urban Career Journal of the Institute of Technology (2008)*, pp.112-124.