Paper report

Engineering Radix Sort for Strings

Juha Ka ̈rkk ̈ainen and Tommi Rantala

Department of Computer Science, University of Helsinki, Finland

{juha.karkkainen,tommi.rantala}@cs.helsinki.fi

**Abstract**

This paper describes a new implementation of MSD radix sort which is faster than existing MSD radix sort and also describes a new variant which can save space by adding a little runtime.

The problems for existing Radix sort are that they have irregular memory access patterns that are poorly suited for modern computer architectures with fast CPUs. The paper describes an implementation that reduces the number of slow memory accesses through better utilization of the cache memory. The paper also describes an implementation that reduces the cost of slow memory accesses by better utilization of the out-of-date execution capabilities of modern CPUs.

**Discussion**

Besides the number of the strings needed to sort and the total number of different alphabets, the total length of the distinguishing prefixes of the strings, D, is also important. Since the unique prefix of a string is the substring that separates itself from others, D describes the shortest number of characters that needed to be inspected and provides a lower bound for the problem complexity. The big O of this sorting algorithm is O(D + number of alphabets).

The paper use different datasets including URL, Genome, Unique and random strings to test the algorithm. Basic MSD radix sort begins by distributing the strings into different buckets according to the first character. Then sort the rest of the strings recursively. And the time complexity of basic MSD radix sort is O((σ/t + t)D). Here, t is the cutoff for insertion sort, σ is the number of different alphabets. O(tD) is the time complexity of insertion sort for small buckets, O(D) is the time complexity for iterating through all strings and O((σ/t)D) is the time Complexity for iterating through all buckets. The paper choose t = 32 to do the experiments. According to the theory, number of alphabets and buckets iteration will have greater impacts on the sorting. However the result shows string iteration contributes most to the running time. This is because the other two analysis represents the worst case and in real case they are less likely to happen.

There are also other variants int MSD radix sort. The implementation of MSD radix sort is complicated because we couldn’t get the size of the buckets in advance. There are two ways of describing this difficulty. The paper calls them C-variants and D-variants.

C stands for counting, means perform the distribution twice and int the first time the function only counts the buckets sizes without moving strings.

D stands for dynamic buckets, means implementing the buckets using dynamically expanding data structure.

The C-variant of MSD radix sort has two phases. The first counts the buckets size and the second does the distribution. Most of the time is spent in loops iterating through strings and the character access would cause cache and TLB missed both times. The paper then describes an improvement by copying the character into a separate array. Because this implementation accesses the array sequentially, so that phase two generates fewer cache and TLB misses. The bets implementations are more than twice as fast most data sets.

The D-variant of MSD radix sort fixes the suffers from the load blocking phenomenon by loop fission. It takes the dynamic bucket data structure as a template argument : DB. DB is a well-known data structure: block-list. It is a linked list and each node points to a fixed size array. Using DB to implement MSD radix sort will cause the time complexity to O(n/B + σB ). In the paper, B is 1024. Also, DB implementation doesn’t need to store all the string pointers. As strings are distributed, the space of the beginning becomes free and can be used to store blocks. So that the space needed to store blocks is only O(√nσ).

The paper describes two techniques to reduce the number of slow array accesses, algorithmic caching and superalphabet. Algorithm caching copies characters in advance to a place. It represent each string by both the beginning character and first four character stored to next pointer. The characters in the cache are moved with the string pointer and thus can be accesses efficiently. The superalphabet technique is to treat character pairs as single characters. This can half the number of characters and the number of character accesses.

**Result**

By comparing different implementations of sorting algorithms with multikey quicksort and burstsort, it shows that the implementation DB combines a small space requirement with good runtime.

**Formulation and analysis of in-place MSD radix sort algorithms**

**Abstract.**

I have found a centrally processing of several related in- place MSD radix sort algorithms when radices changing, here we collectively call this algorithm ‘Matesort’ algorithm. It is a big improvement in traditional radix sort algorithms which space complexity is O(n) that these algorithms use the think of in-place partitioning. I will introduce the binary Matesort algorithm, this algorithm is a new way to implement the classical radix exchange sort and this algorithm shows the importance in in-place partitioning and bit processing. In this algorithm, the space complexity is O(k), and in the worst case, the time complexity is O(kn), k is the bits number in which element needed to encode and n is the element number in which needed to be sorted. Binary Matesort algorithm can develop in any other algorithms, including the ’continuous Matesort’ and several other radix sorts. We propose the formulas and analysis of three different methods: sequential, divide- and- conquer and permutation loop, to use general radix sort for partitioning. The divide-and- conquer approach is a n efficient way to optimize the sort algorithm, it can even create an encoding method which performs much better than the American Flag Sort algorithm (based in permutation loop).

**Introduction.**

There are two classes in radix sort: MSD and LSD. MSD sort begins with the most left digits and moves to the rightest digit. Firstly, MSD radix sort allocates the most left elements, afterwhile recursively invocates the algorithm in each group. MSD only need to allocate fronted digits, while LSD need to scan and allocates all the digits.

Related work to Matesort

Because portioning in the same place is the unique feature, so Quicksort and Radix sort can be mated as radix exchange sort, and this sort can be called as ‘Matesort’. Next one Section will present how to implement a radix exchange sort; the example is named binary Matesort. Radix exchange sort although is easy to implement, runs quickly and its worst running case time is lower than that of Quicksort. However, it is not primarily introduced in many textbooks and references. But the implementation and proof will show its efficiency.

**Binary Matesort algorithm**

It is similar with Quicksort but still has one extra pass parameter named ‘bitloc’. Induction steps lead to this algorithm.

Induction step: Assume each of the element in the array A[1...n] using k bits(bk–1 bk–2 ...b0) to encode and the array is partitioned from the most left bit (bk–1). In this case, each element with bk–1 = 0 arise before elements with bk–1 = 1. Then we need to sort each group.

Theorem 1: The worst running time in binary Matesort is O(kn), n is the number of elements which need to be sorted and k is the bits number which need to encode the corresponding element value. The algorithm needs O(k) space.

Proof: Each element is encoded by using k bits (bk–1 bk–2 . . . b0) and the partition part running time for n elements is definitely O(n) because O(1) is spent in every element. We can use a binary tree to implement the Matesort, and the binary tree node can be an initial call. In each tree level, the array elements cause division among each Matesort calls. In the BitPartition part, when calls are at level 0, bitloc = k – 1, and so on, when calls are at level r, bitloc = k – r – 1. So, any functions in any tree levels should be O(n). In the lowest level of one tree, the bitloc = 0 so the max value of one tree is k. So, O(kn) is consumed in worst order case. Space consumed is related to the recursive call times and it is consumed in the stack space. Every recursive call consumes O(1) space, so the function BitPartition uses O(1) space. In the worst case, every tree element in every level may be waited for calling. So, in conclusion, this algorithm consumes O(k) space in total.

文本

描述已自动生成Compared with other sorts

文本, 信件

描述已自动生成

Figure 1

Figure 1 shows how to generate the input data.

表格

描述已自动生成Table1 compares the time consumption in each sort. The input data is random integer data. 5FFFFFF is almost 100 million elements.

Table 1

The results show the efficiency and low time consumption in Quicksort and Matesort.

**Conclusion**

In this paper, I have shown how MSD radix sort algorithm can be referred as Matesort algorithms. This algorithm is from a rare algorithm which is named radix exchange sort. The experiment result shows the Matesort speed is as quick as Quicksort in random integer data. However, in fact, from the analysis and the implementation, Matesort can be optimized from O(n2) to O(kn), k is the bits size.

Overall, we can see from the result, Matesort and Quicksort are much faster than any other sort, although the sort is the Microsoft built-in sort.