Program Structures & Algorithms FALL
In-place MSD Radix Sort for Text Data

Yutong Wang (001530602) & Jingru Xiang (001586653)

Abstract.
In this paper, combining with the benchmark experiments of traditional MSD radix sort, we discuss the traditional MSD radix sort and the in-place MSD radix sort algorithm called "Matesort", which is come up with in reference paper. Compared with traditional linked-list MSD radix sort, the Matesort uses *o(k)* (*k* is the number of bits needed to encode for a value) rather than *o(n)* space, which the traditional MSD radix sort needs. And the binary Matesort shows greater performance than Quicksort. And this paper analyses the performance of optimized Matesort exploited in text data. [1]

1. Introduction

Traditional MSD sort algorithms need *o(n)* space and its time complexity is *o(n) ~ o(nm)*, m is the length of text. However, the Matesort, originally called radix exchange sort and "it is best thought of as a 'mating' of Radix sort and Quicksort", suggested using binary-alphabet with strings. Matesort shows greater performance than Quicksort even in the worst case. To implement Matesort, there are three ways for partitioning – sequential, divide-and-conquer and permutation loop. Though permutation loop partition was concluded as fastest to sort strings, another way, divide-and-conquer, was proved better in performance.

2. MSD Sort, Quicksort and Binary Matesort Algorithm

The binary Matesort is similar to Quicksort, and it has an extra input parameter 'bitloc' to input bit location, note that initial call to Matesort is passed the location of the highest bit:

```
void Matesort(int[] A, int lo, int hi, int bitloc)
{ // initial call: bitloc = highest bit position (starting from 0)
   if ((lo < hi ) && (bitloc >=0))
   { int k = BitPartition(A,lo,hi,bitloc);
    Matesort(A,lo,k,bitloc-1);
    Matesort(A,k+1,hi,bitloc-1); }
}

int BitPartition(int[] A, int lo, int hi, int bitloc) {
   int pivotloc = lo-1; int t;
   int Mask = 1<< bitloc;
   for(int i= lo; i<=hi ; i++)
      // if ( ((A[i]>> bitloc) & 0x1) ==0)
      if ( (A[i] & Mask) <= 0)
      { // swap with element at pivotloc+1 and update pivotloc
         pivotloc++;
         t = A[i]; A[i] = A[pivotloc]; A[pivotloc] = t; }
   return pivotloc;
}
```

Code 1: Binary Matesort algorithm

In the worst case, the binary Matesort algorithm performs *o(kn)* running time, where *k* is the number of bits to encode for a value and *n* is the number of elements to be sorted. The performance is the same as traditional MSD sort but the binary Matesort algorithm needs only *o(k)* space.

The Quicksort is found to be inferior when data is in small size or contains identical ones than binary Matesort. When the repetition factor increases, Matesort has stable performance. Also, for Quicksort, the recursion depth and execution time can reach to *o(n)* when repetition factor rises:

x: (*U*/*x*) Distribution	Recursion depth					Execution time (ms)				
	MS	Quicksort			QSTRO	MS	Quicksort			QSTRO
1	25	64	72	57	17	7843	8156	7953	8078	7890
125	25	204	247	222	13	6859	15437	9531	9421	9953
250	25	351	426	399	13	6812	24093	11984	11484	12812
500	25	619	811	730	12	6781	38656	16921	15750	18750
1000	25	1166	1567	1372	12	6734	69890	27406	24796	30968

Table 1: Comparison of Matesort and Quicksort, Distribution is *U/x*, *x* is repetition factor

3. Optimized Matesort in Sorting Text Data

For random data, general Matesort spends more time in sorting than binary Matesort, but general Matesort can exploit data redundancy in sorting text data since it uses DigitPartition. Compared with BitPartition, what we mentioned hereinbefore, DigitalPartition represents the text data by array of strings instead of characters, which is faster in data text because 'char[] != A[pivotloc]' is doing character-by-character copy in DigitalPartition. However, BitPartition needs to move points. We just show the codes of BitPartition and DigitalPartition here:

```
int BitPartition(string[] A, int lo, int hi, int bitloc)
{   string t; //next 2 lines: map bitloc to charloc and bitloc within char
    bitloc=charcount*8<\#208>1-bitloc;
    int charloc=bitloc / 8;
    bitloc = 7 - (bitloc % 8);
    int Mask = 1<< bitloc ;
    int pivotloc =lo-1;
    for(int i= lo; i<=hi ; i++)
        // if ( ((A[i][charloc] >> bitloc) & 0x1) ==0 )
            if ( (A[i][charloc] & Mask) <= 0 )
            {   pivotloc++;
                t = A[i];   A[i] = A[pivotloc]; A[pivotloc] = t;
            }
    return pivotloc;
}

int DigitPartition(string[] A, int lo, int hi, int charloc, int pivot)
{   string t;
    charloc=charcount-1-charloc; //order fix: remap charloc
    int pivotloc=lo-1;
    for(int i= lo; i<=hi ; i++)
        if ( A[i][charloc] <= pivot )
        {   pivotloc++;
            t = A[pivotloc];    A[pivotloc] = A[i];          A[i] = t;
         }
    return pivotloc;
}
```

Code 2: BitPartition and DigitalPartition

With DigitPartition, a divide and conquer partitioning, there is an optimized algorithm called GenMatesort_DC_Opt, which shows better performance when sorting text data. The seven-three division rule is applied in GenMatesort_DC_Opt to reduce execution time since in English text, the lower end of the alphabet, such as 'A' – 'D', occurs more times than the higher end, such as 'W' – 'Z'. If the point is closer to letters that occur frequently, it can save 15% time in sorting text data:

```
void GenMateSort_DC_Opt(string[] A,int lo,int hi,int digitloc,int fromdigitval,int todigitval)
{   int mid, k; int i,charloc; int c; int sum;
    if ((lo < hi) && (digitloc >=0) )
    {   sum =0;
        if (fromdigitval==0) // set fromdigitval,todigitval range
        {   charloc = charcount-1-digitloc; //added for order fixing for text
            fromdigitval = A[lo][charloc]; todigitval = fromdigitval;
            for (i=lo+1;i<=hi; i++)
                {   c = A[i][charloc];
                    sum = sum +c;
                    if (c < fromdigitval) fromdigitval = c;
                    else if (c > todigitval) todigitval = c;
                }
        }
    if ( fromdigitval < todigitval)
    {   if (sum > 0) // if range is set then set mid to average value
            mid= sum/(hi-lo);
        else mid = (fromdigitval +todigitval)/2;
    // else mid = (7*fromdigitval +3*todigitval)/10; // seven-three rule

            k= DigitPartition(A,lo,hi,digitloc,mid);
            GenMateSort_DC_Opt(A,lo,k, digitloc, fromdigitval,mid);
            GenMateSort_DC_Opt(A,k+1,hi, digitloc,mid+1, todigitval);
        }
    else GenMateSort_DC_Opt(A,lo,hi, digitloc-1,0, 255);
//In the statement above, fromdigitval parameter (0) is used as a flag to tell
//the start of examination of a digit location (digitloc) for this (lo,hi) group
    }
}
```

Code 3: GenMatesort_DC_Opt: Using seven-three division

Restricted radix values to 27 characters, 'A' – 'Z' and '@' to represent space in ASCII, this form shows the performance of sort mentioned in the paper:

String array Size (n) = 10^6	Execution time (msec)						
String length	.Net Sort	Quick Sort	Binary Matesort	GR MS_Seq	GR MS_MDPLoop	GR MS_DC	GR MS_DC_OPT
20	6578	3593	6375	37859 (6656)	7390 (6343)	4921 (4828)	3468, 3281
30	6859	3718	9015	42859 (7640)	11546 (10140)	5890 (5781)	3640, 3421
40	7125	4109	10703	48973 (8765)	14890 (12406)	6953 (6890)	3828, 3625
50	7406	4531	12546	52593 (9703)	19593 (15968)	7890 (7718)	4046, 3828

Table 2: Performance of sort algorithms for English text data

4. Conclusion

This paper does a survey about an in-place MSD radix sort which has lower space complexity than traditional MSD radix sort. When applied to sort text data, there is a

seven-three partitioning that can optimize time performance for in-place MSD radix sort. In the end, the performance data proved the theory in the paper.

Reference

[1] Al-Darwish, N. (2005). Formulation and analysis of in-place MSD radix sort algorithms. Journal of Information Science, 31, 467 - 481.