Our summary of related papers about Radix Sort

Xuhui Zhu,
Weifeng Lai,
Cheng Yan

Abstraction

We look at two papers about Radix Sort and this is our summary about it. The first paper presents a unified treatment of a number of related in-place MSD radix sort algorithms with varying radices, collectively referred to here as ‘Matesort’ algorithms. The second paper is about a new radix sort algorithm that takes advantage of virtual memory and makes use of write-combining yields a per-pass throughput corresponding to at least 89% of the system’s peak memory bandwidth.

1. Formulation and analysis of in-place MSD radix sort algorithms

In this paper contributed by Nasir Darwish, It mentions a space-optimized Radix Sort, which is called Matesort. Matesort algorithm modifies the classical radix-exchange sort. It uses the idea of in-place partitioning and bit processing operation which is an efficient improvement over the traditionally linked list implementation of radix sort that uses O(n) space.

```
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);
    }
}

void Quicksort(int[] A, int lo, int hi) {
    if (lo<hi) {
        int k = Partition(A,lo,hi); Quicksort(A,lo,k-1); Quicksort(A,k+1,hi);
    }
}

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;
}
```

```
int Partition(int[] A, int lo, int hi) {
    int t; int pivot = A[lo]; int pivotloc =lo;
    for(int i= lo+1; i<=hi ; i++)
    if (A[i] <= pivot) { // swap with element at pivotloc+1 and update pivotloc
        pivotloc++;
        t = A[pivotloc]; A[pivotloc] = A[i]; A[i] = t;
    }
    // move pivot to its proper location
    A[lo] = A[pivotloc]; A[pivotloc] = pivot; return pivotloc;
}
```

Listing 1. Binary Matesort algorithm in comparison with Quicksort algorithm.

The binary Matesort algorithm implements the partition operation in Quicksort. In the recursion calling part, it does the same as common Quicksort. In the partitioning, it uses an extra ‘bitloc’ (bit location) input parameter, which starts from the highest bit to the lowest bit, and splits the array into two parts, A[lo ... k] and A[k+1 ... hi] based on the bit location. It also makes use of bit operation to split two parts, which can improve the performance of the comparison partitioning as used in Quicksort as before. All codes are in listing 1 (in c#, same as java).

The binary Matesort algorithm has O(kn) worst-case order of running time, where n is the number of elements to be sorted and k is the number of bits needed to encode an element value. The algorithm is O(k) space. In partitioning, it is shown to be O(n) since it performs constant time per element. And the partition will be done k times because it will do the partition for every bit of the element. The space used is dominated by the stack space associated with recursive calls. Each such call uses O(1) space. In the worst case, there may be k calls pending. Thus the algorithm is O(k) space.

This paper presented and analyzed a number of in-place MSD radix sort algorithms, collectively referred to as Matesort algorithms. These algorithms are evolved from the classical radix exchange sort. Experiments have shown that binary Matesort is of comparable speed to Quicksort for random uniformly distributed integer data. Unlike Quicksort, which becomes slower as data redundancy increases and may degenerate into $O(n^2)$ algorithm, the binary Matesort algorithm is unaffected and remains bounded by O(kn), where k is the element size in bits.

String array Size (n) = 10^6 String length	Execution time (msc)	
	Quick Sort	Binary Matesort
20	3593	3593
30	3718	3718
40	7125	4109
50	7460	4531

Table 1. Execution times for English text data. The times in parentheses are for restricted radix values to 27 characters (ASCII Codes 64–90).

2. Engineering a Multi-core Radix Sort

In this paper contributed by Jan Wassenberg and Peter Sanders, a new Radix Sort algorithm is mentioned. It builds upon “a microarchitecture-aware variant of counting sort. Taking advantage of virtual memory and making use of write-combining yields a per-pass throughput corresponding to at least 89% of the system’s peak memory bandwidth”(Wassenberg,2011).

The key technique of this paper is as presented below.

Algorithm 1: Single-pass counting sort

```
storage := ReserveAddressSpace(N · M);
for i := 0 to M − 1 do next[i] := i · N;
foreach key,value do
    storage[next[key]] := value;
    next[key] := next[key] + 1;
```

In this algorithm, M arrays of N are preallocated for preventing overhead and check whether the current bucket is full. Each value is inserted into a per-key container, e.g. a list of data blocks. The advantage of this algorithm is that it only writes and reads each item once, which makes use of Software Write-Combining.

Nowadays, applications use non-temporal streaming store instructions that write directly to memory to avoid cache pollution since they circumvent the cache. However, this leads to a new problem: single memory accesses involve significant bus overhead. This is where Software Write-Combining comes to help: ”The data to be written is first placed into temporary buffers, which almost certainly reside in the cache because they are frequently accessed. When full, a buffer is copied to the actual destination via consecutive non-temporal writes, which are guaranteed to be combined into a single burst transfer”(Wassenberg,2011).

This algorithm still take space of M*N, but this can be easily handled by 64-bit CPUs with paged virtual memory.. Physical memory is only mapped to pages when they are first accessed, thus reducing the actual memory requirements to O(N +M ·pageSize).

In this paper, the algorithm is used to improve a multi-core Radix Sort.

Algorithm 2: Parallel Radix Sort

```
parallel foreach item do
    d := Digit(item, 3);
    buckets3[d] := buckets3[d] ∪ {item};
Barrier;
foreach i ∈ [0, 2^D) do
    bucketSizes[i] := Σ_PE |buckets3[i]|;
outputIndices := PrefixSum(bucketSizes);
parallel foreach bucket3 ∈ buckets3 do
    foreach item ∈ bucket3 ∀ PE do
        d := Digit(item, 0);
        buckets0[d] := buckets0[d] ∪ {item};
    foreach bucket0 ∈ buckets0 do
        foreach item ∈ bucket0 do
            d := Digit(item, 1);
            buckets1[d] := buckets1[d] ∪ {item};
            d := Digit(item, 2);
            histogram2[d] := histogram2[d] + 1;
    foreach bucket1 ∈ buckets1 do
        foreach item ∈ bucket1 do
            d := Digit(item, 2);
            i := outputIndices[d] + histogram2[d];
            histogram2[d] := histogram2[d] + 1;
            output[i] := item;
```

Following the pseudocode, each Processing Element (PE) first uses counting sort (the algorithm above) to partition its items into local buckets by the MSD (digit = 3). Note that items consist of a key and value, which are adjacent in memory. After all are finished, the output index of the first item of a given MSD is computed via prefix sum.

Algorithm	K=32,V=0	K=32,V=32
VM only	391	238
Intel x2	400	307
GPU+PCIe	501	303
KNF MIC	560	(?)
VM+WC	657	452

This table shows throughputs (million items per second) of Radix Sort for 32-bit keys and optional 32-bit values. The basic algorithm ('VM only') reaches a throughput of 391 M items/s, as shown in the second column of the table. After enabling write-combining ('VM+WC'), performance nearly doubles to 657 M/s, which beats Intel's radix sort by a factor of 1.64 and also outpaces a Fermi GPU when data transfer overhead is included.

As a conclusion, this paper introduces a counting sort algorithm and uses it to improve the performance of Radix Sort. By using write-combining, it fixes the problem of single memory accesses involving significant bus overhead and nearly doubles the performance of Radix Sort.

Reference

1. Al-Darwish, N.: Formulation and analysis of in-place MSD radix sort algorithms. In: Journal of Information Science, pp 467–481 (2005)

2.Jan Wassenberg, Peter Sanders: Engineering a Multi-core Radix Sort. In: Euro-Par 2011 Parallel Processing, pp 160-169 (2011)