Fast Bit Sort

A New In-Place Sorting Technique

Nando Favaro
February 2009

1. INTRODUCTION

1.1. A New Sorting Algorithm

In Computer Science, the role of sorting data into an order list is a fundamental real world problem which takes on a very important role. There are many various well-known techniques for sorting such as Bubble sort, Heap sort, Insertion sort, Quick sort, Selection sort and Shell sort. Fast Bit Sort is a divide and conquer recursive sort that performs the sort *IN-PLACE.* Fast Bit Sort does not require extra memory allocations for array data to sort subsets nor merging of subsets. Fast Bit Sort provides optimized performance with no knowledge of the unsorted data.

1.2. The Concept

The concept underlying the Fast Bit Sort algorithm is described in the following example:

We are given an unsorted integer array A[0...n-1] containing n integers where each integer is of width B bits. We prepare by having a low pointer L pointing to the lowest element 0 and a high pointer H pointing to the highest element n-1. We also initialize a mask M with the highest bit set as in $M = 2^{(B-1)}$. We sort by looking at the highest bit only and as such, testing and ensuring when the highest bit set, the datum is in the lower part of A and when the highest bit is reset, the datum is in the upper part of A. Then, we proceed through the lower bits for each sub part created by the previous. For signed integers, when the highest bit is set, the data belongs in the lower part but for the lower bits, when set, the datum belongs in the upper part.

2. UNDERSTANDING THE DETAIL

Our first loop iteration begins with analyzing the highest bit using M as a mask for each element starting at L and scanning upwards towards H.

Conceptualizing array A[0..n-1] as a whole but only consider the highest bit only and A[0..n-1] as being virtually split into two partitions. Because the highest bit is the sign bit, any data with the highest bit set belongs in the lower partition of A and the any data with the highest bit reset belong in the higher partition of A. Pointer L is responsible to ensure the datum it points to belongs only in the lower partition (ie. highest bit set) and continues scanning upwards towards H until it finds a datum that does not belong in the lower partition of A. Processing switches to pointer H where it's responsibility is to ensure the datum it points to belongs only in the upper partition (ie. Highest bit reset) and continues scanning downward towards L until it finds a datum that does not belong in the higher partition of A.

At such time, the two elements are swapped. Processing continues and terminates when L and H meet by either L scanning upwards or H scanning downwards. When they meet, L points to the highest element of A with the highest bit set and H points to the lowest element of A with the highest bit reset. Thus, A is sorted only by the highest bit. Conveniently, the original L (before the scanning began) and the ending L represent the limits of A only containing lower sub partition, and the ending H and the original H (before the scanning began) represent the limits of A only containing the upper sub partition.

Recursively, each of those sub partition are separately partitioned the exact same way using M shifted right one each time. It is important to realize only the very first partitioning is for the sign bit and every scan there after must partition the masked bit when reset into the lower partition and when set into the upper partition which is the opposite process for the sign bit.

Even though the deepest recursion is B, recursion to the deepest depth often does not need to proceed fully with when sub partitions are of size 1. A worst-type case scenario would be when consecutive sorted values are alternating odd even at the bit 0 level.

Generally, a pseudo code representation would look like:

```
    Algorithm Fast Bit Sort (integer array A[0...n-1], n)
{       M ← 0x80000000        // global mask, highest bit set for 32 bit integers
        L ← 0
        H ← n-1
        Partition(L,H)
}

    Algorithm Partition( old_L, old_H )
{
        IF old_L == old_H THEN Return     // partition is only one element.
        IF M ==1 THEN Return              // no sorting under bit 0
        Shift M right once
        L ← old_L
        H ← old_H

        In the following block, only scan while L < H
        {
            Scan L upwards while datum A[L] masked bit is reset.
            IF data A[L] is in the wrong partition (ie. bit set)
            THEN
                  {
                      scan H downwards while datum A[H] masked bit set.
                      IF data A[H] is in the wrong partition (ie.bit reset)
                      THEN
                            swap A[L] ↔ A[H]
                            GOTO scanning L upwards again
        }
        Partition (old_L, L)    // sub partition only the reset bit partition
        Partition (H, old_H)    // sub partition only the set bit partition

        Shift M left once
}
```

3. COMPLEXITY

3.1. Time Complexity

At most, Fast Bit Sort scans completely once through A for each bit B. This means the time complexity can be defined as,

$T(n) = O(B \cdot n)$

For integers where B=32 (ie. 32 bit integers) the complexity is,

$T(n) = O(32 \cdot n)$

The maximum time complexity when B=32 occurs only when there

3.2. Space Complexity:

Fast Bit Sort is an in-place sort and thus there is no requirement for any extra temporary memory.

$S(n) = O(n)$ (data array)
$S(n) = O(3 \cdot B)$ (stack requirement)

Since Fast Bit Sort is recursive, a stack is required but only two values plus the return address is need to be saved for each recursion. For 32 bit integers, this amounts to a maximum recursion depth of 32. Any common Operating System provides this amount of stack without question and is inconsequential.

3.3. Efficiency

Fast Bit Sort is efficient. Even though it must must look at all values in order to sort them, the time needed to sort integers grows in proportion to the number of values. However, storage needs to sort values does not grow. It has a linear growth function in relation to only amount of time needed to sort values.

4. PARALLEL COMPUTING: PARALLELISM

Fast Bit Sort's recursive algorithm lends particularly well when applied to parallel computing. Each sub partitioning can be performed on an available CPU and no CPU will interfere with any other, except for memory access.

Lacking a mechanism to detect when all CPU's are finished, a multiple CPU algorithm would be:

```
    Algorithm Fast Bit Sort (integer array A[0...n-1], n)
{       M ← 0x80000000        // Local mask, highest bit set for 32 bit integers
        L ← 0
        H ← n-1
        Partition(L, H, M)
}


    Algorithm Partition( old_L, old_H, M )
{
        IF old_L == old_H THEN Return
        IF M ==1 THEN Return
        Shift M right once
        L ← old_L
        H ← old_H

        In the following block, only scan while L < H
        {
            Scan L upwards while datum A[L] masked bit is reset.
            IF data A[L] is in the wrong partition (ie. bit set)
            THEN
                  {
                      scan H downwards while datum A[H] masked bit set.
                      IF data A[H] is in the wrong partition (ie.bit reset)
                      THEN
                            swap A[L] ↔ A[H]
                            GOTO scanning L upwards again
        }
        New CPU( Partition (old_L, L, M) )  // sub partition only the reset bit partition
        New CPU( Partition (H, old_H, M) ) // sub partition only the set bit partition
}
```

Ideally, for a 32 bit integer, a maximum of 32 CPU's would be required. Of course, any available idle CPU will reduce the time complexity in a lesser environment. The time complexity could be characterized as,

$T(n) = O(n \cdot \log_2 B)$

For integers where B=32 (ie. 32 bit integers) the complexity is,

$T(n) = O(n \cdot 5)$
$= O(n)$

5. SORTING FLOATING POINT DATA

When dealing with data represented as binary floating point, Fast Bit Sort would need to be modified to sort the data ascending in this following order:

First, the mantissa sign. Sort values negative to positive
Second, for negative mantissas, sort exponents largest to smallest.
Second, for positive mantissa, sort exponents smallest to largest.
Third, for equal exponents, sort the mantissa.

All can be performed in place because each sub partition is independent to any other. Parallelism and multiple CPU's work in this environment also.

6. SORTING STRING DATA

When dealing with string data, Fast Bit Sort would need to be modified in the following fashion assuming strings pointed to by an array of pointers:

First, the string length. Sort the pointers by length of string from highest bit to lowest bit.
Second, for same size strings lengths, sort from highest to lowest bit for each char.

All can be performed in place because each sub partition is independent to any other. Parallelism and multiple CPU's work in this environment also.

7. <u>CONCLUSION:</u>

Fast Bit Sort is a sort technique which is superior than most others. It has a zero extra memory requirement due to being an in-place sort. It has a low code size due to recursion and the simplicity of the algorithm. It parallels easily in multiple CPU environments. It can replace the use of hash tables because sorting is very quick and a binary search can be implemented on the real data instead.

8. <u>CONTACTING THE AUTHOR.</u>

The author can be contacted by email at nando_f@nothingsimple.com