Volume 2, No. 3, March 2011
Journal of Global Research in Computer Science
ISSN-2229-371X
RESEARCH PAPER
Available Online at www.jgrcs.info

IMPROVING PERFORMANCE OF RANDOMIZED SIGNATURE SORT USING HASHING AND BITWISE OPERATORS

Tamana Pathak*[1] and Dr. Deepak Garg[2]

*[1]Department of computer Science and Engineering, Thapar University, Patiala, Punjab, India
tamana.pathak@gmail.com[1]
[2]Department of computer Science and Engineering, Thapar University, Patiala, Punjab, India
dgarg@thapar.edu[2]

Abstract: Research done in the area of integer sorting has considerably improved the lower bound and achieved with comparison sorting i.e. $O(n \log n)$ to $O\left(n\sqrt{\log\log n}\right)$ [1] for a deterministic algorithms or to $O(n)$ for a radix sort algorithm in space that depends only on the number of input integers. Andersson et al. [2] presented signature sort in the expected linear time and space which gives very bad performance than traditional quick sort. It is well known that n integers in the range [1, n^c] can be sorted in $O(n)$ time using radix sorting. Integers in any range [1,U] can be sorted in $O\left(n\sqrt{\log\log n}\right)$ time [1]. However, these algorithms use $O(n)$ words of extra memory. We present a simple and stable variant of signature sort for integer sorting, which works in $O(n)$ time and uses only $O(1)$ words of extra memory. In this we are trying to improve the performance of the signature sort by implementing differently and comparing its performance against traditional sorting algorithms and to see the effect of register size on the algorithm.

Keywords: Randomized algorithms; Sorting; Integer Sorting; Linear Complexity.

INTRODUCTION

Integer sorting has always been an important task in connection with the digital computer. The sorting problem is to sort *n* elements, according to a given ordering, has a tight $O(n\log n)$ bound in the comparison model. This lower bound is achievable without randomization. The table-1 depicts various existing comparison based algorithms and their complexities with relative performance.

Table I. Complexity and Performance Comparison

Name	*Complexity*	*Stable*	*Memory*	*Relative Performance*
Insertion Sort	$O(n^2)$	Yes	$O(1)$	Avg.
Shell Sort	$[O(n(\log n)]$	No	$O(n)$	Avg.
Binary tree sort	$O(n\log n)$	Yes	$O(n)$	Good
Selection sort	$O(n^2)$	No	$O(1)$	Avg.
Heap sort	$O(n\log n)$	No	$O(1)$	Good
Bubble sort	$O(n^2)$	Yes	$O(1)$	Avg.
Merge sort	$O(n\log n)$	Yes	Depends	Good
Quick sort	$O(n^2)$	Depends	$O(\log n)$	Good
Randomized Quick sort	$O(n\log n)$	Depends	$O(\log n)$	Good
Signature Sort	$O(n)$	Depends	Linear	Bad
Randomized Signature Sort	$O(n)$	Depends	Linear	Bad

Recent works in the field of deterministic algorithms have optimized this lower bound. Andersson et al. presented with new idea which gives a lower bound of $O(n \log \log n)$ but uses $O(m)$ extra memory. Later on, Yijie Han also improved the lower bound to $O(n \log \log n)$ with linear space. And he then introduced to new lower bound of $O(n\sqrt{\log\log n})$ with linear space.
Using randomization the lower bound for deterministic algorithms can be optimized further as Andersson et al. presented a concept which takes $O(n \log \log n)$ and linear space. Later on, he came up with signature sort with expected linear time and space. But the relative performance of signature sort with traditional deterministic sorting algorithms is very poor.
The Randomized Signature Sorting algorithm works in two phases: One of the phases is Word Formation phase which packs multiple integers into a single machine word to sort quickly by operating on multiple integers with a single instruction. Another Phase of Randomized signature sorting is Comparison Sorting i.e. Sorting between words and Sorting within words, discussed later. To accomplish linear time sorting of n integers requires to pack $\log n \log \log n$ of integers into one machine word.
The existing signature sort divide integers into fields and then pack them into different words to perform packed sorting. This concept consumes lot of running time of computer as integers must be divided into fields and then each field must be packed into different word. After that comparison is performed and based upon that the sorted list

© JGRCS 2010, All Rights Reserved

is generated. As we can see this signature sort requires lot of extra operation.
In this paper, we present the concept which reduces the extra operation required by the signature sort. Instead of dividing integer into fields we hash each integer into signature which is only $O(\log n)$ bit size long. It reduces the requirement of dividing integer into field as integer has only one signature of reduced bit size. We can pack multiple signatures into one word. In this way, we only use single word for multiple integers, instead of using one word for one field of integer. Thus, the operation required after this in only comparison. The Signature Sorting algorithm uses the signatures which are computed by applying hash function on the integer i.e. input values to reduce the size of the integers being sorted. The range reduction depends on the hashing algorithm not generating collisions; if collisions occur, the output will not actually be sorted, making this a Monte Carlo algorithm. To change the Monte Carlo algorithm's result to a Las Vegas algorithm's result, the output is checked and if the output is not sorted, the algorithm is re-run. Thus, overall the algorithm runs in expected $O(n)$ time. Thus, the hash function is very important and it must not generate collision at all for better results. We are presenting an idea of hash function in this paper which hashes integer into $O(\log n)$ bit size signature.

MACHINE MODEL

The machine model is a normal computer which holds an instruction set corresponding to what is programmed in common standard programming language such as C/C++ or JAVA. A processor determined word-size W, confines how large integers can be processed in constant time. It is assumed here that each input integer fits in a single word and for generic code, the type of a full word integer e.g. long int, should be a macro parameter in C or template parameter in C++ or a primitive type in JAVA. The unit-cost time measure is adopted where each operation takes constant time [1]. Interestingly, the traditional theoretical RAM model of Cook and Reckhow [11] allows infinite words.
According to Yijie Han and Mikkel Thorup [1], the outcome of infinite words with operations like shifts or multiplications, exponentially big parallel processor can be simulated, solving all problems in NP in polynomial time. So they suggested banning such operations from unit-cost theory RAM, making it even more contrived from a practical view-point. But it is possible to achieve such algorithms that can be implemented in the real world considering the real-world limitation of a fixed word-size in mind. This model is named the *word RAM (*Hagerup [2]*)*. The word RAM has a fairly long tradition within integer sorting, being advocated and used by Kirkpatrick and Reisch [3] and Fredman and Willard [4].
In word RAM, addition and subtractions can be performed on integers and this is considered as an advantage over the comparison based model by word RAM. Hence, this word RAM can be used to code multiple comparisons of short integers combined in single words. The idea of multiple comparisons was first introduced by Paul and Simon [5]. As per [1], the word RAM model is different from the comparison based model as well as from the pointer machine in which integers can be used, and segments of integers, as addresses. This idea links to radix sort where an integer is viewed as a vector of characters, and these characters are used as addresses. Another idea of word RAM is to hash the integers into smaller ranges, naming these hash integers as signatures in this paper. Here radix sort goes back at least to 1929 [6] and hashing goes back at least to 1956 [7], both being developed for efficient problem solving in the real world. Fredman and Willard [4] further use the RAM for advanced tabulation of complicated functions over small domains.
As a simple example, considering computer architecture of 64 bits i.e. 64 bits are processed in single instruction cycle. Actual comparison operation is performed on maximum of 32-bit integers; if we are having a range of integers then these integers can be mapped to lower number of bits. Thus we can merge several integers into a single word. Thus 64 bit machine can compare 16 signatures i.e. hash value of input integers with their size of 8 bits in single instruction cycle.
Summing up, we have discussed that the ideas facilitated by the word RAM are well established in the practice of writing fast code. Hence, if we disallow these ideas, we are not discussing the time complexity of running imperative programs on real world computers. Thus this RAM model plays a vital role in reducing the overhead and enhancing the performance of the algorithm.

SIGNATURES

In order to decrease the size of integers to be operated upon, signatures are created which have the lesser bit size than the original inputs. The signatures are computed for each input with a universal hash function. Such signatures must have the size of $O(\log n)$ where n is the number of input integers. The hash function used to create these signatures, must assure collision free hashing.
The signatures created with this method must follow the property:
If $A_i \le A_j \ \forall \ i,j \in \{0, 1 \ldots n-1\}$

Then $S_i \le S_j \ \forall \ i,j \in \{0, 1 \ldots n-1\}$.

Where A_i's are the input integers, S_i's the corresponding signatures and n is the number of input integers.

HASH FUNCTION

The hash function reduces the size of integers by creating signatures of $O(\log n)$ bit size. The hash function must provide collision free hashing to ensure accurate and better result. The hash function must take $O(n)$ expected time. This will improve the overall performance of algorithm. We are presenting an idea that can be used for hashing of integers into signatures. Division must be avoided in hash function for better performance. The hash function is,

$$h_a(x) = (ax \bmod 2^k) / 2^{k-l}$$

Where k is the number of bits in the input integer,

l is the number of bits in the signature which will be $O(\log n)$.

a is randomly chosen between zero and 2^k.

Since the division in the above function is division by a power of two, it can be implemented as a left shift. This function will be applied to each input integer to produce a

© JGRCS 2010, All Rights Reserved

signature. This will take $O(1)$ time i.e. constant time for each signature. Since there will be *n* signatures, thus the overall expected time for signatures creation will be of $O(n)$. The above said hash function assures $O(log n)$ bit size of signatures and also collision free result.

WORD FORMATION

In Andersson's concept integers are divided into fields and each of these fields is packed into words. In this way signatures are created. This is a bit of overhead as each field of integers is required to be packed and compared. We are modifying this idea using hashing. As discussed above hash function will hash whole integer into a signature with reduced bit size of $O(\log n)$. After that the packing of multiple signatures into one word will be done. This will not only reduce the extra overhead created by dividing integers into fields but also enhance the overall performance. It is an important phase as multiple integers i.e. signatures (hashes) of integers will be packed in a word. We must ensure that this phase runs error free while implementing.

If *w* is word size of the machine, sb is the number of bits in the signature, wl is the word length, m is the number of words, l is the number of signatures in a word and n is the number of input integers then:

To calculate the number of words required will be equal to the product of bits in a signature and number of inputs divided by the total length of the word.

$$m = sb \times n / wl$$

The number of signatures to be packed in a word is equal to the total number of inputs divided by number of words required.

$$l = n/m$$

Consider the example, for 64-bit machine taking 256 input integers of any bit length for simplicity, on which after applying hash function signatures of 8-bit can be obtained corresponding to 256 inputs as signatures are of $O(\log n)$ bit length. Thus, eight different 8-bit integers that are actually signatures can be packed into a word of 64 bit length. Or this can be said that for n inputs there will be $O(\log n \log\log n)$ number of words will be created overall.

In order to pack the signatures in the word, masking is done using shift operators as well as logical operators that make the process faster. Also there will be total n operations in the word formation phase there it will take $O(n)$ time.

COMPARISON SORTING

Comparison sorting is another important phase which is applied on words in order to get the sorted result. Here sorting means exchanging the positions of the signatures from one word to another word and also exchanging positions within word itself. This is the phase where actual comparison will occur, which will result in sorting. As now, there are multiple integers in one word thus, the word itself also be sorted after sorting has occurred in between words. Thus we can divide the whole procedure of sorting into two sub-phases.

The first sub-phase is sorting between words. This can be done by comparing two words and checking corresponding bits of signatures to merge them. The idea is to use bitwise operations to get the result. As bitwise operators are comparatively fast thus use of these operators will speed up the processing the algorithm. We only requires two bitwise operators XOR and AND to perform this task. The task includes applying XOR and AND in such a way that the result will show which signatures need to be swapped. First, we will apply XOR on words and then apply XOR and AND operators on result with the word in which we want smallest of this signatures.

The above discussed operations will give non-zero value on corresponding bits where signatures are needed to be swapped. Thus we will swap those signatures again using XOR operator. Hence, only a constant time is required to perform the above said operation. As we know there will be a maximum of $O(\log n \log\log n)$ words which are to be compared. This will lead us to $O\left((\log n \log\log n)^2\right)$ number of maximum comparisons. We can further reduce these comparisons by using any traditional deterministic algorithm like Quick Sort. By applying the traditional deterministic algorithm the number of comparison will be reduced to $O(\log n \log\log n \log(\log n \log\log n))$.

We know that,

$$\log n \log\log n \leq (\log n)^2$$

Taking $\log$ on both sides,

$$\log(\log n \log n) \leq 2\log n$$

Multiplying both sides with $\log n \log\log n$,

$$\log n \log\log n \log(\log n \log\log n) \leq 2\log n (\log\log n)^2$$

Now as we know,

$$(\log\log n)^2 \leq \log n$$

Thus,

$$\log n \log\log n \log(\log n \log\log n) \leq 2(\log n)^2$$

Now, $(\log n)^2 \leq n$

hence,

$$\log n \log\log n \log(\log n \log\log n) \leq 2n$$

This proves that total number of comparisons will be $O(n)$. As we know that there is constant number of signatures in every word thus we need to perform this procedure only a constant number of times which is independent of number of integers. The overall expected time for this sub-phase is only $O(n)$.

Now after first sub-phase the words will be in sorted order with each other, but there are multiple signatures in every word thus we need to take care of that. Here we need to perform sorting within word itself. This can be done in similar fashion as sorting is done between words. The one more thing we need to take care of here is masking of signatures which are not participating in sorting operation. The task consists of comparing signatures within word. Here also we will use bitwise operators as used above. Hence, there is constant number of signatures in a word. This will take constant number of operations. There is at most $O(\log n \log\log n)$ number of words, that implies we require $O(\log n \log\log n)$ number of operations which is less than $O(n)$.

Let's suppose there are m signatures in a word. So if we start comparing each signature with other, it will take $O(m^2)$ expected time. We can reduce this expected time to

© JGRCS 2010, All Rights Reserved

enhance the performance by simply using divide and conquer technique on word. This is shown in the figure-1:

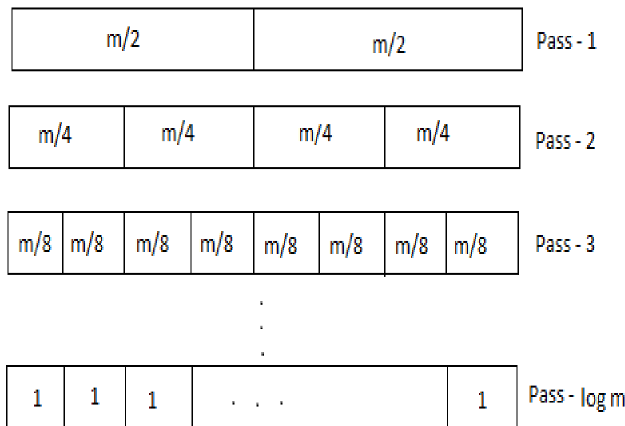

Figure1. Comparison within word

The algorithm for comparison within word is given as follows:

Step 1: Set i = 1.

Step 2: Divide word into 2^i parts.
Step 3: Compare each adjacent parts

i.e 1st with 2nd, 3rd with 4th and so on.
Step 4: Exchange the positions of signatures according to the result of step 3 as exchanged while performing

comparison between words.

Step 5: Stop.

This algorithm first divide a word into 2 halves and compares them bitwise operations. And according to the result of bitwise operations, the positions of signatures are exchanged, and in second pass each half considered as individual word and same task is performed on those. In this way, we are exploiting parallelism of uniprocessor system as done while performing comparison between words. The number of passes will be log m and total number of comparison will be c log m where m is the number of signatures in a word. We have to call the algorithm for all words formed. Thus, finally words will be sorted itself. This shows that there will be at most $O(\log m)$ passes and in each pass only constant number of comparisons is required. Thus total number of comparisons will be $c \log m$.
Considering the space requirement it is clear that the only extra space requirement is $O(1)$ which is used to perform bitwise operations on words.
The above discussion shows that the whole sorting operation can be completed in $O(n)$ expected time and uses $O(1)$ extra memory.

TRACKING INDEX

Traditionally indexing is not required in sorting as the array to be sorted stores pointers to the data structure, and the pointers are changed in order to put the structures in the right order. However, signature sort in word formation and sorting phases works on signature which is lesser in bit size that are packed many to a word, thus there isn't enough space to carry along a pointer to the rest of the structure.
The above mentioned matter can be resolved by not only outputting the sorted data but also outputting their corresponding indexes listed in the original list too. The benefit of index is to track the original position can be looked up afterwards. This association of indexes with the integer can be computed by manipulating a packed list of indexes in the exact way the list of integer to be sorted. In order to rearrange the word of indexes in exact way as the word of integers at every stage, the merge operation is annotated.
At the end, each word of sorted integers matches up with the value of original position in the word of indexes. After applying merge operation on integers, indexes are mirrored on integers. This gives a list consisting of pairs i.e. one list containing the integers in the sorted order and another one of original indexes of each integer in the original list.
This method of maintaining indexes will work fine upto $\log n \leq wl$. This will fail afterwards because at this point, the indexes no longer fit in a field.
Another way of using the benefits of indexes apart of mirroring the indexes with the sorting, the indexes can be created afterward by constructing a table of size $2wl < 2\log n < n$. This will not require more space than before. The table stores the start index of each integer in the sorted list, from which the index of each integer in the sorted list can be computed. The table will help us in sweeping the original list. The table can be computed in $O(n)$ time and the sweep takes $O(n)$ time, for a total of $O(n)$ post-processing time to compute the indexes.
As a side note, if $\log n > wl$, packed sort is unnecessary as the number can be sorted in $O(n)$ time using radix sort.

SUMMING UP

We finally come up with variant of signature sort which is not only stable but better in performance than existing signature sort algorithm with $O(n)$ expected time which uses only $O(1)$ extra space. The variant is also randomized as the existing algorithm is also. The actual running time of this variant is comparatively very low than existing signature sort as we are using bitwise operation. The use of bitwise operator makes the algorithm fast. The comparative performance of this variant with randomized quick sort is

© JGRCS 2010, All Rights Reserved

also good. We have enhanced the relative performance of signature sort.

REFERENCES

[1] Yijie Han and Mikkel Thorup. Integer sorting in $O(n\sqrt{\log \log n})$ expected time and linear space. In *IEEE Symp. on Foundations of Computer Science*, volume 43, 2002.

[2] Andersson, Hagerup, Nilsson, and Raman. Sorting in linear time? In *STOC: ACM Symposium on Theory of Computing (STOC)*, 1995.

[3] D. Kirkpatrick and S. Reisch. Upper bounds for sorting integers on random access machines. 1984.

[4] M. L. Fredman and D. E. Willard. Surpassing the information theoretic bound with fusion trees.1993. Announced at STOC'90.

[5] W. Paul and J. Simon. Decision trees and random access machines. In Proc. Symp. ¨uber Logik and Algoritmik, 1980.

[6] L. J. Comrie. The hollerith and powers tabulating machines.Trans. Office Machinary Users' Assoc., Ltd, 1929-30.

[7] A. I. Dumey. Indexing for rapid random access memory systems. Computers and Automation, 1956.

[8] Mikkel Thorup. Randomized sorting in $O(n \log \log n)$ time and linear space using addition, shift, and bit-wise boolean operations.

[9] Y. Han: Deterministic Sorting in $O(n \log \log n)$ Time and Linear Space, J. Algorithms 50(1): 2004.

[10] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein: Introduction to Algorithms, Second Edition, The MIT Press and McGraw-Hill Book Company2001.

[11] S. Cook and R. Reckhow. Time-bounded random access machines. J. Comp. Syst. Sc., 10(2):1973.

[12] M. Dietzfelbinger, T. Hagerup, J. Katajainen, M. Penttonen, A reliable randomized algorithm for the closest- pair problem, J. Algorithms 25 (1997).

© JGRCS 2010, All Rights Reserved