International Journal of Soft Computing and Engineering (IJSCE)
ISSN: 2231-2307, Volume-1, Issue-2, May 2011

Efficient String Sorting Algorithms: Cache-aware and Cache-Oblivious

R. Angrish, D. Garg

***Abstract*— Sorting is a process of rearranging a sequence of objects into some kind of predefined linear order. String data is very common and most occurring data type. Sorting a string involves comparison it character by character which is more time consuming than integer sorting. Also, sorting forms the basis of many applications like data processing, databases, pattern matching and searching etc. So implementing improvements to make it fast and efficient will help in reducing the computational time and thus making our applications run faster. This paper briefs about various fast and efficient string sorting algorithms. The algorithms have been divided into two categories: cache-aware and cache-oblivious. The various algorithms discussed are: CRadix Sort, Burstsort and cache-oblivious string sorting algorithm. The improvement in CRadix Sort is achieved by starting the sorting with the most significant digit and associating a small block of main memory called the key buffer to each key and sorting a portion of each key into its corresponding key buffer. Burstsort is a trie-based string sorting algorithm that distributes strings into small buckets whose contents are then sorted in cache. The cache-oblivious string sorting algorithm is a randomized algorithm for string sorting which uses signature technique (reduces the sequence by creating a set of "signatures" strings having the same trie structure as the original set) to sort strings.**

***Index Terms*—Cache-aware, Cache-oblivious, External string sorting.**

I. Introduction

A string is a collection or finite sequence of characters or alphabets. Sorting a string mainly consisting of characters involve putting in a lexicographical or dictionary order. String data type is predominant in many areas like databases, pattern matching, etc. The traditional sorting algorithms like merge sort, quick sort, insertion sort, etc measure the complexity based on the number of comparisons that are made. These known comparison based algorithms reads the list elements and determines which of two will occur first and which last but in case of string, each character is sorted and the length is a major factor in measure of complexity.

The string sorting takes time approximately proportional to the length of the largest common prefix plus one, since that many characters have to be compared to resolve the comparison. The variable length string sorting is more challenging than the fixed length integer sorting because string sorting involves pointers to access the string, string comparison is done character by character unlike integer sorting in which the entire key is compared at once and also string lengths are variable and swapping them is more difficult.

Now-a-days manipulation of large data sets is a common thing in every area of application like databases, digital libraries, etc. The size of data sets have increased to such an extent that they now do not fit into the internal memory of the computer systems thus they need to be stored in external memory devices or secondary storages like CD, disks, etc., thus increasing the latency time as the external memories are slower than the cache memory. If the problem set is very large the latency time dominates the overall execution time thereby increasing the computation time. The difference in speeds of the memories is increasing rapidly with increasing technology thus leading to increase in latency time i.e., increasing I/O bottleneck making the situation worse [12].

The performance of traditional string sorting algorithms degrades when the problem set does not fit into internal memory. This paper discusses various algorithms that aim at minimizing the number of cache misses so that the I/O bottleneck problem can be reduced thus making it more efficient and fast..

II. Literature Review

There are two basic categories of sorting: Internal sorting and External sorting. If the problem set is small enough to reside entirely into the internal memory, then the sorting done on the set is known as internal sorting. In this case the transfer time i.e., the time required to read and write is insignificant in evaluating the performance. External sorting applies to large problem set which cannot entirely reside into internal memory and they have to be accessed from secondary storage thus adding latency time to computational time and making it the major factor in determining the performance.

Recent development in hardware technologies demand programmers to pay attention to the memory hierarchy as the performance impact of the memory system is increasing dramatically. The introduction of cache memory helped in improving latency but the penalty imposed by cache misses have degraded the overall performance. So it can be implied that a good overall performance cannot be achieved without a good cache performance. As a consequence the design of algorithms should be done in such a way that they take full advantage of the cache memory [9]. This is entirely the duty of the programmer to write the code which will generate less number of cache misses.

Manuscript received March 29, 2011.
Ritika Angrish, Department of Computer Science, Thapar University, Patiala, India, (e-mail: rangrish@gamil.com).
Deepak Garg, Department of Computer Science, Thapar University, Patiala, India, (e-mail: dgarg@thapar.edu).

A. *Traditional Methodology*

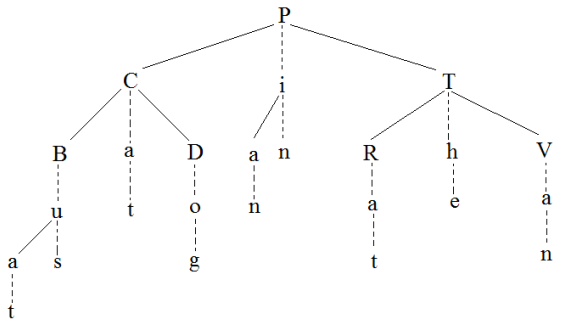


Fig. 1: Ternary search tree depicting the sorting order of Pin, The, Cat, Rat, Dog, Fan, Fun, Pan, Van, Bus, Bat.

Traditionally the computational speed was measured on the basis of comparisons. For string sorting pointers are used and they are then permuted for putting the strings in the required order. Examples include Multikey quicksort [5], radix sort variants [7], [8], etc.

Multikey quicksort is a ternary partitioning algorithm, a variant of quicksort, used for sorting problem having multiple keys i.e., strings. It is explained that the data structure used for this sorting is ternary search tree. Its basic working is same as the quicksort, having the smaller elements on left side and greater on right side. In this algorithm the pivot key can be chosen at random or it can be the first key or the median. After choosing the pivot, a first loop starts at the beginning and compares the two keys if the two are equal it shifts the key to left and halts if the key in comparison is greater. The second loop which works from the end and shifts the key which are equal to pivot and halts when it finds smaller keys. Later the main loop swaps the greater and lesser keys. The multikey quicksort on strings is explained in fig 1 by making the ternary search tree by inserting elements in input order.

A ternary search tree as shown in Fig. 1, for strings stores single character per node and searching a string consists of a character by character binary search for each character. The only drawback in multikey quicksort is the selection of pivot. If we consider median to be the pivot, finding median is very expensive than doing sorting with random pivot.

B. *Cache-Aware and Cache-Oblivious Algorithms*

Earlier, the algorithm efficiency depended on the number of instructions it incur. This model is called the RAM-model where the memory access is said to be done in unit cost regardless of the location of data. With advancing technology, the memory access time depends on the level of hierarchy we deal with. If this factor is not considered the algorithms suffer a major drawback in their performance. To handle this factor another model was introduced called the external memory model or the I/O model [10; 11]. This model takes into account the memory latencies. It considered that the performance of an algorithm depends on the number of disk access needed by the algorithm. The main drawback of this model is that the algorithms developed in this model are platform dependent i.e., they are based on the knowledge of memory parameters; these are called cache-aware algorithms.

The cache-oblivious algorithms [14] help to overcome this drawback. The definition of cache-oblivious algorithm as given by Prokop is “An algorithm is said to be cache-oblivious if it does not depend on the memory parameters like cache line size and cache size”. These algorithms are platform independent: if implemented well in ideal cache model then they can easily be implemented in other memory models as well. The main aim of these algorithms is to minimize the number of cache misses so that there is less memory transfer operations thereby increasing the algorithm performance.

III. Cache-Aware Algorithms

A. *CRadix Sort*

CRadix sort [2] is a cache efficient variant of MSD Radixsort with a little difference that instead of permuting the strings directly using pointers, we use buffer to hold some characters of strings and permute them. This is done to alleviate the drawback caused in MSD Radixsort i.e., the increase in cache misses. Considering MSD Radixsort, it is a cache-oblivious algorithm since it was not developed considering the memory hierarchy. In this the strings are located sequentially and only pointers are swapped instead of swapping the entire strings. So once the pointers are permuted in first sort the strings cannot be accessed sequentially during next sort this causing more cache misses.

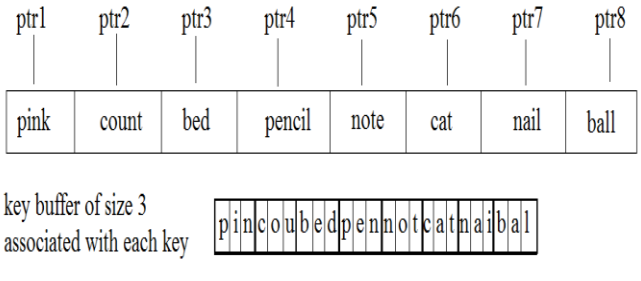
Thus to decrease the number of cache misses a part of main memory is used as buffer which accommodate a part of each string and temporary sorting is done on that. Thus making CRadix sort a cache-aware algorithm. The buffer used to manage the temporary sorting of keys is called key buffer.

The working of CRadix sort is shown in Fig. 2.

Algorithm:

Consider the key buffer to be of size b, c be the number of characters processed and i^{th} filling characters can be computed as $1 + (i-1)b$.

1. Set c=0 and i=1.
2. If buffers are empty or they are completely processed i.e., c=b; set c=0, fill the buffer from $1+(i-1)b^{th}$ character with atmost b characters of the corresponding key and increment i.
3. Increment c.
4. Keys are grouped according to its c^{th} character.
5. Permutation is done in the same order as of the key pointers.
6. Algorithm is recursively applied to each group from step 2 until each group contains single key.



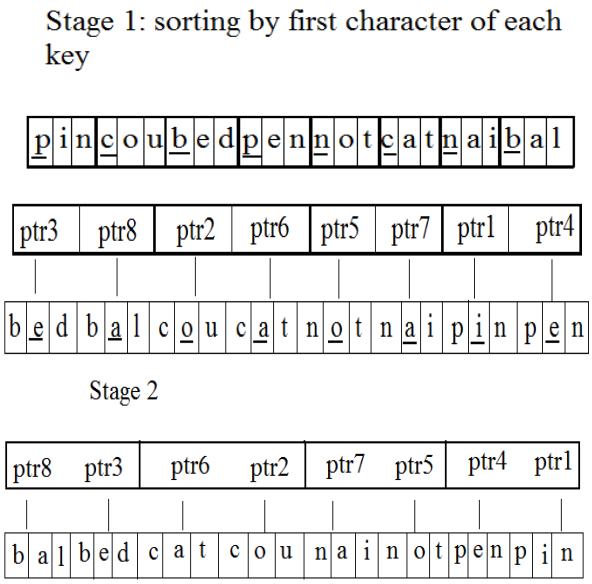


Fig. 2: Sorting of strings: pink, count, bed, pencil, note, cat, nail and bull using key buffer of size 3.

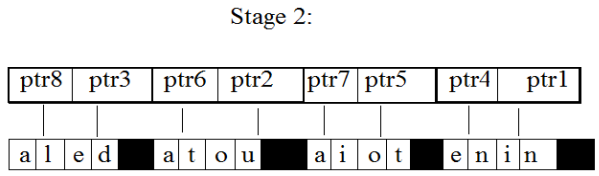


Fig. 3: Implementation using method 2.

The next point of discussion is how to manage the contents of the key buffer. The first method (shown in fig 2) permutes all the untouched characters by finding their offset or using c as the count of the character to be processed. The second method (shown in fig 3) discards the processed character thereby eliminating the task of finding the offset and reduces the buffer size every time. Also while choosing the key buffer size we need to balance the tradeoff between performance loss by cache misses if b=1 and the performance loss suffered due to overhead of permuting large keys if b is very large.

B. Burstsort

Burstsort [1] is a trie based string sorting algorithm in which the contents are divided into small buckets which are later sorted in cache i.e. a combination of burst trie [3] and string sorting algorithms [5; 7]. P-Burstsort is the standard burstsort which proceeds in two stages: making a trie structure of strings and then traversing it in-order and sorting the bucket contents. The output is the pointers to the string in lexicographic order.

A trie is a mutli-way tree structure useful for storing strings over an alphabet. Tries store characters in internal nodes and not keys, records in external nodes and use the characters of the key to guide the search. A burst trie is a trie with accessing nodes as internal nodes and buckets as leaves.

The memory usage of buckets can be reduced by redesigning the buckets or by having attached an array of pointers to sub-buckets i.e., a moving field approach which points to the field where key is to be inserted. To improve the cache efficiency, string suffixes are first copied into a small

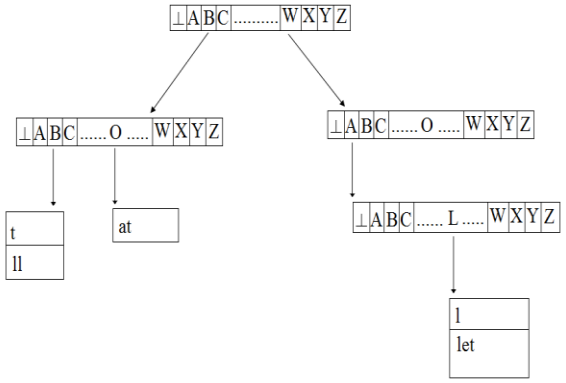


Fig 4: Implementation of Burstsort using trie on strings: bat, ball, wall and wallet.

buffer before a key is stored thereby decreasing the number of cache misses.

Algorithm:

1. Insert the key into burst trie and distribute into appropriate buckets according to the most significant bit.
2. If bucket is full, introduce children buckets and insert the keys in it; redistribute the keys according to the next most significant bit.
3. Repeat step 1 and 2 until all the keys are inserted.
4. Traverse buckets in in-order fashion and sort the keys using multikey quicksort.

IV. CACHE-OBLIVIOUS ALGORITHM

To best state the algorithm we assume the input to be binary strings and the following notation [6].

K = number of strings to sort,

N = total number of words in the K strings,

M = number of words fitting in the main memory,

B = number of words per disk block,

where $M<N$ and $1<B\leq M/2$. The input sequence assumed $x_1,\ldots, x_k$ is given in a form such that it can be read in O(N/B) I/Os.

A randomized algorithm [13] for string sorting in external memory inspired by the randomized signature technique that creates a set of "signature" strings having the same trie structure as the original set of strings is discussed here. For K binary strings comprising N words in total, the algorithm finds the sorted order and the longest common prefix sequence of the strings using $O(K/B \log_{M/B}(K/M) \log(N/K) + N/B)$ I/Os. It is a Monte Carlo type randomized, cache-oblivious algorithm which computes the sorting permutation and the lcp (least common prefix) sequence.

The data structure used in this is the unordered blind trie which can be constructed from a blind trie by expanding each single node. The algorithm proceeds by first making the unordered blind trie i.e. the signature reduction for each string and then applying the list ranking algorithm [4]. This algorithm mainly concern with finding the lcp sequence for strings and then using it for permuting the strings in sorted order.

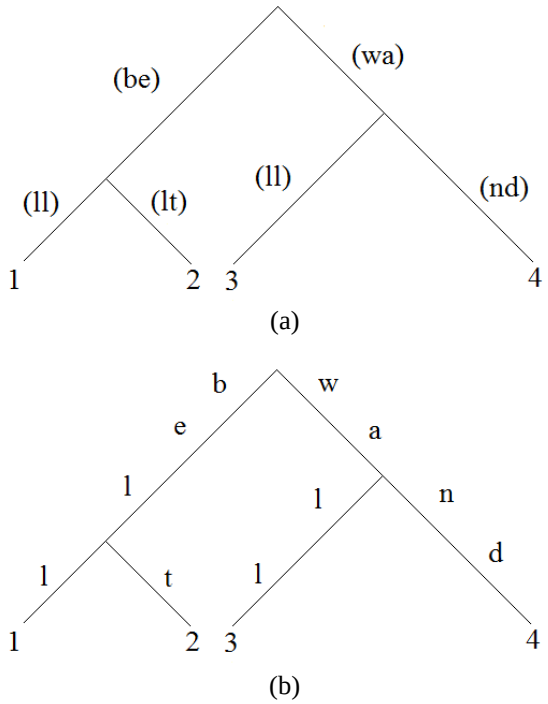


Fig5: (a) Blind trie for strings: bell, belt, wall, wand.
(b) Unordered blind trie for strings: bell, belt, wall, wand.

Algorithm:

1. Sort the nodes according to the longest common prefix, which in this case is called parented and further this parented is sorted according to the branching characters.
2. Construct a directed graph joining the vertices of the unordered blind trie.
   i. For a node containing i children edges are formed and annotated with the longest common prefix.
   ii. Leaf nodes are annotated with the number of strings represented.
3. Order the graph using the list ranking algorithm.
4. Output the sorting permutation and the longest common prefix.

V. Conclusion

CRadix sort may be a cache-efficient variant of MSD Radixsort because of less number of cache misses but requires extra memory for buffer. The large workspace required can be of size of the number of pointers used for representing strings or the extra buffer space reserved for each key whichever is large. The Burstsort variants are already the fastest hardware algorithm known and the memory reduction improvements like reducing the size of buckets which involves dynamic allocation have minimum impact on the sorting time. The randomized sorting algorithm performs well on cache-oblivious model and uses the concept of longest common prefix i.e., signature reduction and uses O (K/B $\log_{M/B}$ (K/M) log (N/K) + N/B) I/Os, where K is number of strings to sort, N is total number of words in the K strings, M is total number of words fitting in memory and B is number of words per block.

Table I: Comparison of algorithms

Algorithm / Parameters	CRadix Sort Algorithm	Burstsort Algorithm	Cache-oblivious randomized algorithm
Algorithm Type	Cache aware	Cache aware	Cache-oblivious
Technique	Modifies MSD Radixsort by making it cache efficient.	Combines burst trie with string sorting algorithms.	Combines signature technique with list ranking algorithm.
Basic Principle	Uniquely associating a memory block called key buffer to each key and then the contents of key buffer are permuted	Distributes strings into buckets whose contents are then sorted in cache	Unordered blind trie is constructed and the permuted using list ranking algorithm
Data structure	Array	Trie or ordered tree structure	Unordered blind trie

VI. References

[1] R. SINHA and A. WIRTH, "Engineering Burstsort: Towards fast in-place string sorting", ACM Journal of Experimental Algorithmics, Vol. 15, Article No. 2.5, 2010.

[2] W.H. Ng and K. Kakehi, "Cache efficient radix sort for string sorting", IEICE TRANS. FUNDAMENTALS, Vol. E90–A, No. 2, 2007.

[3] S. Heinz, J. Zobel, and H.E. Williams, "Burst tries: A fast, efficient data structure for string keys", ACM Trans. Inform. Syst, Vol. ,20, No. 2, 2002, pages192–223.

[4] L. Arge, M. A. Bender, E. D. Demaine, B. Holland-Minkley, and J. I. Munro, "Cache-oblivious priority queue and graph algorithm applications", In ACM, editor, Proceedings of the 34th Annual ACM Symposium on Theory of Computing (STOC '02), ACM Press, 2002, pages 268–276.

[5] J. Bentley and R. Sedgewick, "Fast algorithms for sorting and searching strings", In Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms. Society for Industrial and Applied Mathematics, Philadelphia, 1997, pages360–369.

[6] L. Arge, P. Ferragina, R. Grossi, and J. S. Vitter, "On sorting strings in external memory (extended abstract)", In ACM, editor, Proceedings of the 29th Annual ACM Symposium on Theory of Computing (STOC '97) , ACM Press, 1997, pages 540–548.

[7] P.M. Mcilroy, K. Bostic, and M.D. Mcilroy, "Engineering radix sort", Comput. Syst, Vol. 6, No. 1, 1993, pages5–27.

[8] A. Andersson and S. Nilsson, "Implementing radixsort", ACM J. Exp. Algorithmics 3, 7, 1998.
[9] A. LaMarca and R.E. Ladner, "The influence of caches on the performance of sorting," J. Algorithms, vol.31, 1999, pages66–104.
[10] A. Aggarwal and J.S. Vitter, "The input/output complexity of sorting and related problems", Communications of the ACM, 31(9), 1988, pages1116-1127.
[11] J. S. Vitter and E.A.M. Shrive, "Algorithms for parallel memory I: Two-level memories", Algoritmica 13, 1994, pages110–147.
[12] Yale N. Patt, "The I/O subsystem—a candidate for improvement", Guest Editor's Introduction in IEEE Computer, 27(3) , 1994, pages15-16.
[13] R. Fagerberg, A. Pagh and R. Pagh, "External string sorting: Faster and cache-oblivious".
[14] M. Frigo, C. E. Leiserson, H. Prokop, AND S. Ramachandran, "Cache-oblivious algorithms (extended abstract)", Proceedings of the 40th Annual Symposium on Foundations of Computer Science, IEEE Computer Society Press, 1999, pages 285–297.

Ritika Angrish is a master's student of Department of Computer Science, Thapar University, Patiala and obtained her B.E. Degree from Panjab University, Chandigarh in 2008. Research interest: algorithms and data structures.

Deepak Garg is an assistant professor of the Department of Computer Science, Thapar University, Patiala. Chair, ACM North India SIGACT Chapter; Senior member of IEEE, USA; Secretary of IEEE Computer Society, Delhi Section; Member of ExeCom IEEE Delhi Section; and member of various other national and international societies. Research interest: data structure, Algorithms and Data Mining.