CACHE-AWARE AND CACHE-OBLIVIOUS ALGORITHMS

Thesis submitted in partial fulfillment of the requirements for the award of degree of

Master of Engineering
in
Computer Science and Engineering

Submitted By
Ritika
(Roll No. 800932017)

Under the supervision of:
Dr. Deepak Garg
Assistant Professor, CSED

COMPUTER SCIENCE AND ENGINEERING DEPARTMENT
THAPAR UNIVERSITY
PATIALA – 147004

June 2011

CERTIFICATE

I hereby certify that the work which is being presented in the thesis entitled, "**Cache-aware and cache-oblivious algorithms**", in partial fulfillment of the requirements for the award of degree of Master of Engineering in Computer Science and Engineering submitted in Computer Science and Engineering Department of Thapar University, Patiala, is an authentic record of my own work carried out under the supervision of Dr. Deepak Garg and refers other researcher's work which are duly listed in the reference section.

The matter presented in the thesis has not been submitted for award of any other degree of this or any other University.

Signature:

(Ritika)

This is to certify that the above statement made by the candidate is correct and true to the best of my knowledge.

(Dr. Deepak Garg)
Computer Science and Engineering Department,
Thapar University,
Patiala

Countersigned by

(Dr. Maninder Singh)
Head
Computer Science and Engineering Department
Thapar University
Patiala

(Dr. S. K. Mohapatra)
Dean (Academic Affairs)
Thapar University
Patiala

ACKNOWLEDGEMENT

It is a great pleasure for me to acknowledge the guidance, assistance and help I have received from Dr. Deepak Garg. I am thankful for his continual support, encouragement, and invaluable suggestions. He not only provided me help whenever needed, but also the resources required to complete this thesis report on time.

I am also thankful to Dr. Maninder Singh, Head, Computer Science and Engineering Department for his kind help and cooperation.

I would also like to thank all the staff members of Computer Science and Engineering Department for providing me all the facilities required for the completion of my thesis work.

I would like to say thanks for support of my classmates. I want to express my appreciation to every person who contributed with either inspirational or actual work to this thesis.

I am highly grateful to my parents and brother for the inspiration and ever encouraging moral support, which enabled me to pursue my studies.

Ritika

ABSTRACT

With the development in technology, the difference between the speeds of processor and memory access impose a penalty if the programs do not take advantage of the memory hierarchy. Also while sorting strings in external memory this latency plays a very vital role as it degrades the performance of the algorithms. 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. Cache-oblivious algorithms help in achieving optimal use of cache without the knowledge of its size.

This thesis consists of discussion of cache-aware and cache-oblivious algorithms for general algorithms like large integer multiplication and for string sorting algorithms. A comparison of various existing algorithms has also been done in this thesis based on various parameters such as the type of algorithm, technique used, basic principle of working and the type of data structure used. The thesis also provide a new algorithm for string sorting which uses the concept of tried linear hashing and uses the cache-oblivious blind trie for strings. The algorithm developed has been tested for random strings and URLs and the result is depicted in the form of graphs.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

1. CD: Compact Disc
2. CPU: Central Processing Unit
3. I/O: Input/Output
4. LCP: Longest Common Prefix
5. MSD: Most Significant Digit
6. MT: Memory Transfers
7. RAM: Random Access Memory
8. URL: Uniform Resource Locator

Chapter-1
INTRODUCTION

1.1. Computer Evolution

Earlier computers only had a flat memory structure, that is, there were only the main memory. This was acceptable because the speed of the processors was very slow and they were the main bottleneck. During those times the program performance and efficiency depended upon the number of instructions it contains and memory references was not considered. The complexity in flat memory system was measured using the RAM-model, consisting one CPU and a unit cost random access memory [2].

With the development in technology, the speed of processors increased following the Moore's Law stating the increase in transistor to be more than 55% every year [20]. In contrast to this the increase in speed of memory has been slow i.e., nearly 7% per year [16]. Thus implying that the speed of memory will be slower than the processor speed and this difference will increase over time.

1.2. Memory Hierarchy

The common way of handling the problem is the introduction of memory hierarchy. In this a fast and expensive memory is placed close to CPU and slow and cheaper memories are placed far from the processor. The memory closer to processor are expensive that is why they are small and thus hold less data i.e., acting as a buffer for slower memories. A typical memory hierarchy is illustrated in Figure 1.1.

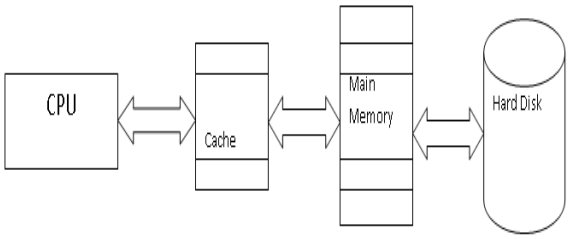


Figure 1.1: A typical memory hierarchy

The hierarchy is constructed considering the locality of reference, mainly temporal locality which is a memory location once referenced by a program is likely to be used again several times within short interval, and spatial locality which is once a program has referenced a memory location it is likely to refer the nearby locations shortly. The fast memory placed near the processor is generally called the cache. When a program reference a data which is not present in cache and has to be accessed from the next level of memory is called a cache miss.

With the development in hardware, the cost of accessing different levels in hierarchy has created implications like algorithms processing large data are dependent on relative memory latency for them to have increased performance, and the performance of processor bound problems is also limited by the memory performance. To make the algorithms perform better the programmers should take advantage of the memory hierarchy.

1.3. String Sorting

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 [23]. The performance of traditional string sorting algorithms degrades when the problem set does not fit into internal memory.

1.3.1. Sorting in Internal Memory

In RAM-model, the memory is thought of as a large array where every element can be accessed at a constant cost of time. The algorithm analysis in this model does not take into account the time required to access the data but focus on the number of instructions. For many practical purposes this is consistent with the real world performance of programs and justifies this model of computation, but only as long as the entire data set fits into memory. Sorting in internal memory is well understood and has been studied for many years. A fundamental result in the RAM model is the (N logN) worst case lower bound by using a comparison based algorithm.

1.3.2. Sorting in External Memory

When data do not fit in main memory (RAM), external (or secondary) memory is used. Magnetic disks are the most commonly used type of external memory. Because access to disk drives is much slower than access to RAM, analysis of external memory algorithms and data structures usually focuses on the number of disk accesses (I/O operations), not the CPU cost. When data is stored on the disk, algorithms that are efficient in main memory may not be efficient when the running time of the algorithm is expressed as the number of I/O operations. External memory algorithms are designed to minimize the number of I/O operations.

1.4. Memory Performance

The advances in hardware development have created a great mismatch between the processors speed and memory speed thus making it important to consider the cost of accessing at each level in the memory hierarchy. Naturally, this lead to an increasing problems accessing the higher levels in the memory hierarchy, thus being dependent on slower memory. This disparity will therefore have two implications: Current algorithms that process large data sets, and are dependent on main memory and disks, will not gain the full speedup of the improved CPUs due to the increase in relative latency in caches. The other implication is that problems which today are CPU intensive will, as the CPUs get faster, become limited by the memory system. All in all, many algorithms are influenced by this development, and memory performance is becoming increasingly important.

1.5. Models

1.5.1. RAM-model

Under RAM model the run time of algorithm is measured by counting the number of steps (instructions) an algorithm contains. It assumes a flat memory system. It is the simplest model representing the working of computer. The main drawback of the RAM model was that it considered only one memory and is not an approximation of the actual

hardware. This raised the need for new models to measure the performance and efficiency of algorithms correctly.

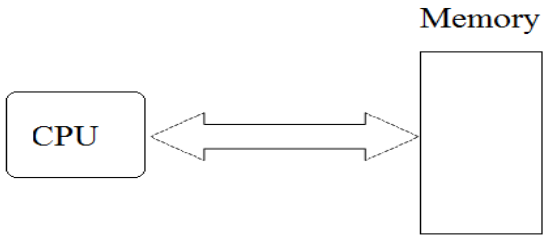


Figure 1.2: The RAM-Model

1.5.2. External-memory Model

To overcome the drawback of RAM model an external memory model is considered. It is a two-level memory hierarchy model with block transfer also known as the I/O model, the disk access model or cache-aware model [1]. The basic idea was to compute the efficiency of algorithm on the basis of number of disk access as calculating number of instructions is not a solution in the case of memory hierarchy. The model contains a cache which is a fast memory placed near processor but its size is limited and a disk which is relatively slow memory placed at a distance from the processor having a limitless size.

In external memory model, the algorithms are specifically tuned according to the memory parameters, for example, in sorting [1] the optimal number of I/Os needed is $\Theta(N/B \log_{M/B} N/B)$, where N is the number of elements to be processed, M is the number of elements that fit in internal memory and B is the number of elements that fit in one block of the internal memory. The limitation of this is that the implementation is highly platform independent as the size of memory varies from system to system and also the complexity of algorithm increases proportional to the increase in memory levels. In this model it is the responsibility of the programmer to control the block transfer. The external memory algorithms dependent on the memory parameters like block size or memory size are called external-memory algorithms or cache-aware algorithms.

1.5.3. Cache-oblivious Model

The basis behind the cache-oblivious model is designing external memory algorithms without knowledge of memory parameters. The cache-oblivious algorithm overcomes the limitations imposed to external memory algorithms i.e., they are not architecture independent and it is very difficult to adapt them to multiple levels of memory. A cache-oblivious algorithm as described by Prokop [14] is the one in which problem variable is independent of the memory parameters like cache size or block size; and is tuned to reduce the number of cache misses. Considering this definition the RAM model algorithms can also be considered as cache-oblivious. These algorithms are evaluated in an ideal-cache model.

The ideal-cache model consists of two-level of memory hierarchy i.e., a small cache and a very large main memory as in Figure 1.3. The cache in ideal-cache model is of M words and cache line of size B. But this simple idea has several surprisingly powerful consequences.

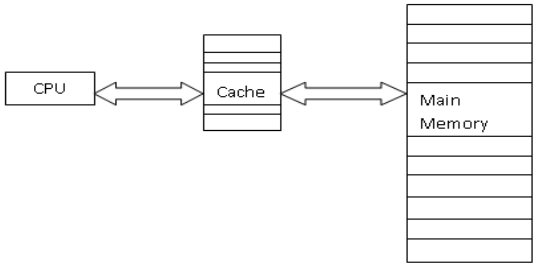


Figure 1.3: The ideal-cache model

One consequence is that, if a cache-oblivious algorithm performs well between two levels of the memory hierarchy (nominally called cache and disk), then it must automatically work well between any two adjacent levels of the memory hierarchy. A further consequence, if the number of memory transfers is optimal up to a constant factor between any two adjacent memory levels, then any weighted combination of these counts (with weights corresponding to the relative speeds of the memory levels) is also within a constant factor of optimal.

Another, more practical consequence is self-tuning. Typical cache-efficient algorithms require tuning to several cache parameters which are not always available from the manufacturer and often difficult to extract automatically. Parameter tuning makes code portability difficult. Perhaps the first and most obvious motivation for cache-oblivious algorithms is the lack of such tuning: a single algorithm should work well on all machines without modification. In contrast to the external-memory model, algorithms in the cache-oblivious model cannot explicitly manage the cache.

Model / Parameters	RAM Model	External-memory Model	Cache-oblivious Model
Complexity	Number of instructions	Number of disk access	Number of memory transfers
Model Type	Simple	Cache-aware	Cache-oblivious

Table I: Comparison of Computational Models

The above comparison table shows that the RAM-model is the older model developed at the time when the architecture was a flat memory system. Being a simplest model it calculated the performance on the basis of number of instructions. On the other side the external-memory model and the cache-oblivious model calculated the performance by counting the number of disk accesses or memory transfers.

1.6. Cache-aware and Cache-oblivious Algorithms

Earlier, the algorithm efficiency depended on the number of instructions it incurs. 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 [1, 26]. This

model takes into account the memory latencies. It considered that the performance of an algorithm depends on the number of disk accesses 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.

Some cache-oblivious algorithms like matrix multiplication algorithm [28], cache-oblivious priority queues [6] when compared to their cache conscious counterpart were found superior.

1.6.1. Introduction to Cache-oblivious Techniques

A number of techniques can be used to develop cache-oblivious algorithms and data structures. The majority of all cache-oblivious results use one or more of these techniques:

Sequential Data Access: The obvious approach is to make an algorithm process data in a sequential way. An example of this is the scanning of an array to find the maximum element.

Divide-and-conquer: The cache-aware approach for utilizing the cache efficiently is to block the problem. That is, dividing it into sub-problems of a size suitable to be in the cache. The cache-oblivious counterpart is to use a divide-and-conquer approach. The idea is, that once a sub-problem fits into a level in the memory hierarchy the further dividing down to a constant size will not cause any further cache misses, and is therefore cheap.

Recursive Layout: For a static data structure-like, for example, a tree-data can be placed in memory in such a way that it improves the usage of the memory hierarchy. One example is to use a recursive layout called the van Emde Boas layout.

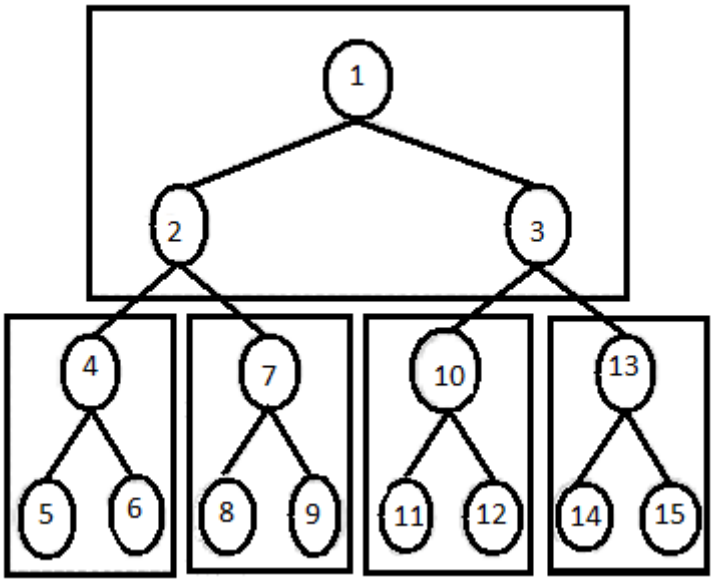


Figure 1.4: The recursive van Emde Boas layout of binary tree

Lazy Evaluation using Buffers: Another approach, for achieving sequential data processing is by using buffers. This technique is described by Arge, and used to develop the cache-aware buffer-tree data structure [5]. The idea can, however, be reused in a cache-oblivious context, if the sizes of the buffers are not fixed to the sizes of the actual memory hierarchy, but instead grow, beginning with a small constant size. This way, the buffers will at some point fit into a cache level, and processing a buffer of this size will not incur more cache misses on that particular level.

The advantage of this technique is that the work is buffered in a lazy fashion. Only when the buffer is full, the content of a buffer is moved to the next larger level, and the elements stored in the buffer are all processed in one operation. This way, the main data processing occurs on the smallest level, and elements are moved up or down in the structure in a lazy or batched fashion.

Chapter-2
LITERATURE REVIEW

2.1. Cache-oblivious Data Structures

In this section we discuss about the various cache-oblivious data structures that have been proposed till date. This includes cache-oblivious trees both static and dynamic and hash table using cache-oblivious hashing.

2.1.1. Cache-oblivious B-Trees

B-Tree is a balanced tree data structure that keeps data stored and allows searches, sequential access, insertions and deletions in logarithmic amortized time. They are optimized tree structures used when part or the entire tree must be maintained in secondary storage. Cache-oblivious B-Trees are the one that perform well for all levels of the memory hierarchy and do not depend on the number of memory levels, the block size and number of blocks at each level.

2.1.1.1. Static B-Trees

The static cache-oblivious search tree having search cost equal to the search cost of standard cache aware B-tree (O ($\log_B N$) I/Os) was first proposed by Prokop [24]. The search tree is related to the van Emde Boas layout as shown in figure 2.1.

The van Emde Boas layout of a complete binary tree T is to have a recursive layout by dividing the tree at the middle level. In this a tree having a single node is said to be a trivial case and it is set in memory as the single node. In other words a tree of height h=logN, have the top tree T_0 as a sub tree consisting of nodes at the topmost $\llcorner h/2 \lrcorner$ levels of T and the bottom $T_1,\ldots, T_k$ trees to be the $\Theta(\sqrt{N})$ sub trees rooted in the nodes on level $\ulcorner h/2 \urcorner$ of T.

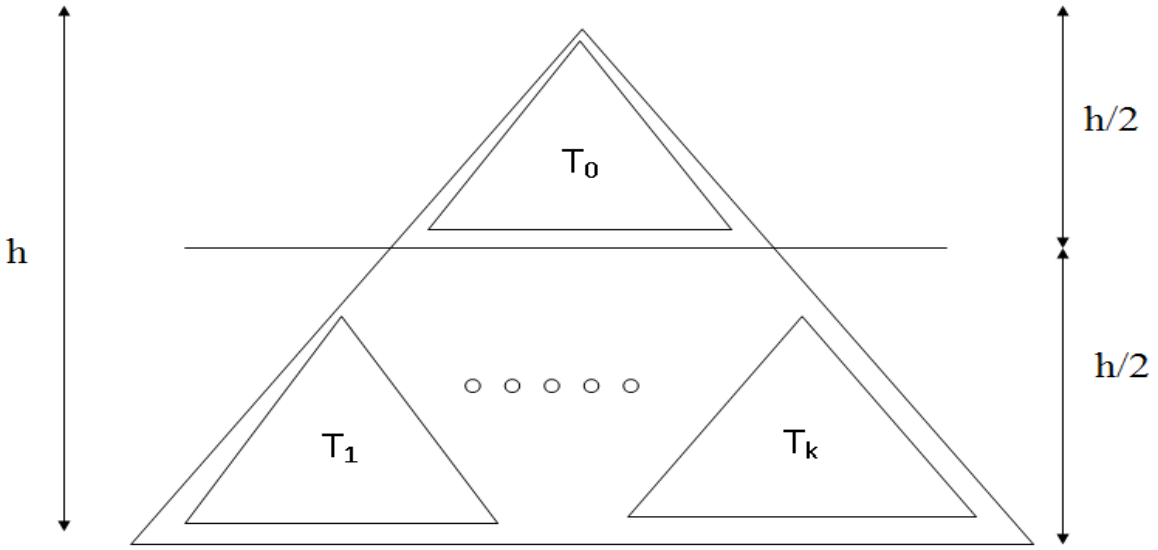


Figure 2.1: The Van Emde Boas Layout

2.1.1.2. Dynamic B-Trees

The basic idea of dynamic B-Tree [8] was to embed a dynamic binary tree of height logN+O(1) onto a static complete binary tree which in turn is embedded into an array using the van Emde Boas recursive layout as shown in figure 2.2. Techniques for maintaining small height in a binary tree used to maintain the dynamic tree takes $O(\log^2 N)$ amortized time per update. The algorithm works by rebalancing the sub tree rooted at the node on whose leaf the violation occurred.

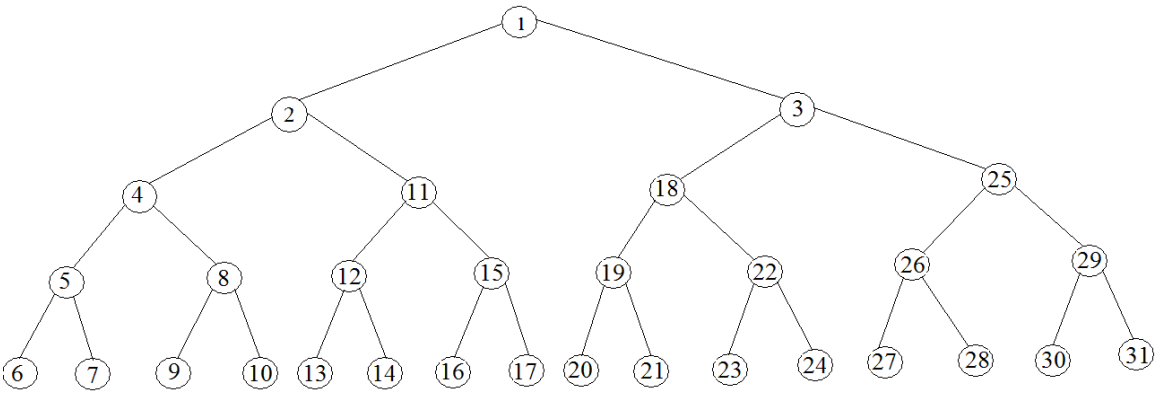


Figure 2.2: (a) van Emde Boas Layout of a complete binary tree of height 5

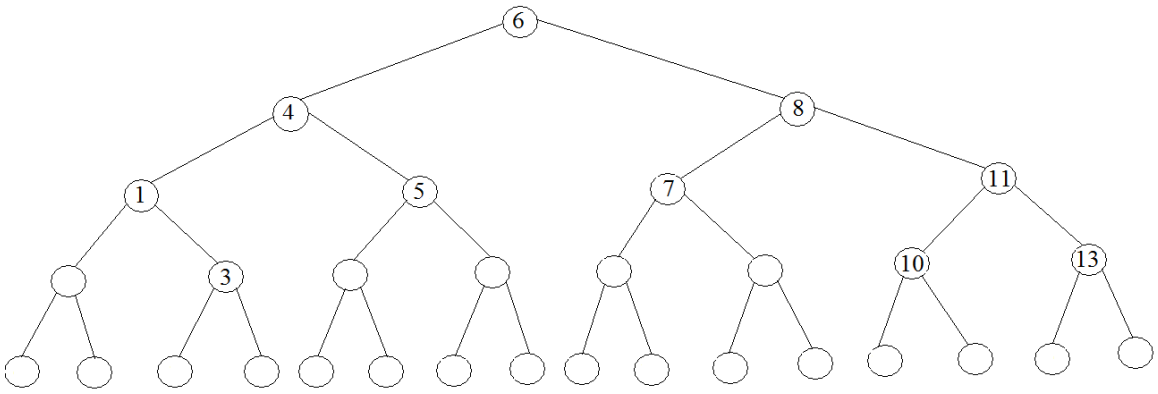
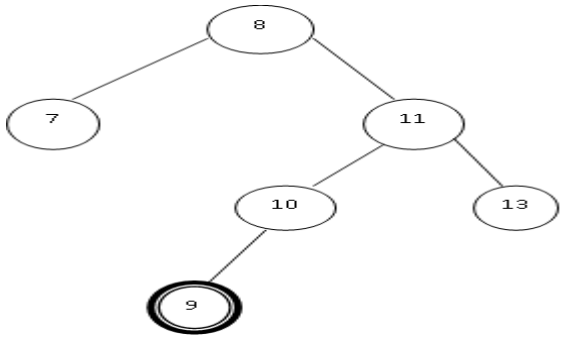
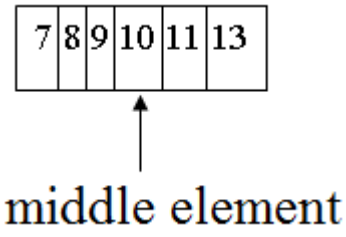


Figure 2.2: (b) Embedding tree of height 4 on complete binary tree of height 5 [10]

The search procedure in the algorithm is same as the search procedure in binary tree but balancing needs to be done when we insert a node and an imbalance is created. This is explained in the example shown in figure 2.3.



(a) Insertion of node 9 creates imbalance



(b) A sorted array of all elements of sub tree is created and a middle element if found

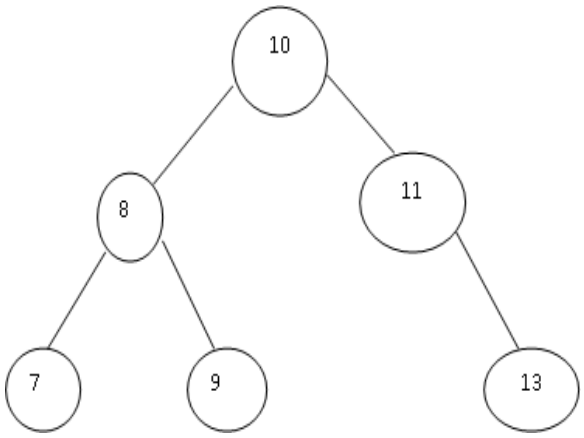


Figure 2.3: (c) The sub tree formed after rebalancing

The major advantages of a dynamic cache-oblivious B-tree are it minimizes the number of memory transfers and no pointers are used which gives better space utilization as van Emde Boas layout is used which is an implicit layout.

2.1.2. Hash Table using Cache-oblivious Hashing

Hash table is one of the simplest and most important index structures in databases. In this a hash function is used to map keys into their positions and various collision resolution strategies are used such as linear probing, chaining etc. The cache-oblivious hashing [22] uses linear probing as a collision resolution strategy ignoring the blocking (not considering the memory blocks) as chaining strategy would perform worse cache-obliviously because the list associated is not laid out consecutively. The search cost of cache-oblivious hashing was found to be $1+O(\alpha/b)$ I/Os where α is the load factor. Average search time of $1+2^{\Omega(b)}$ [18] can also be obtained in cache-oblivious hashing thus matching the cache-aware bound under the following two conditions: (a) b is a power of 2; and (b) every block starts at a memory address divisible by b.

2.2. Cache-oblivious Algorithms

2.2.1. Cache-oblivious Matrix Multiplication Algorithm

Matrix multiplication is the most examined problem and the cache-oblivious algorithm for matrix multiplication outperforms the most efficient cache-aware matrix multiplication algorithm. The cache-oblivious technique used for making matrix multiplication algorithm cache-oblivious is the divide-and-conquer technique.

The algorithm proceeds by assuming two matrices A and B each of size N×N. The layout for the matrices can be any of the following in figure 2.4.

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 7 & 8 & 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 & 17 & 18 \\ 19 & 20 & 21 & 22 & 23 & 24 \\ 25 & 26 & 27 & 28 & 29 & 30 \\ 31 & 32 & 33 & 34 & 35 & 36 \end{pmatrix}$$

(a) Row major layout

$$\begin{pmatrix} 1 & 7 & 13 & 19 & 25 & 31 \\ 2 & 8 & 14 & 20 & 26 & 32 \\ 3 & 9 & 15 & 21 & 27 & 33 \\ 4 & 10 & 16 & 22 & 28 & 34 \\ 5 & 11 & 17 & 23 & 29 & 35 \\ 6 & 12 & 18 & 24 & 30 & 36 \end{pmatrix}$$

(b) Column major layout

$$\begin{pmatrix} 1 & 2 & 3 & 19 & 20 & 21 \\ 4 & 5 & 6 & 22 & 23 & 24 \\ 7 & 8 & 9 & 25 & 26 & 27 \\ 10 & 11 & 12 & 28 & 29 & 30 \\ 13 & 14 & 15 & 31 & 32 & 33 \\ 16 & 17 & 18 & 34 & 35 & 36 \end{pmatrix}$$

(c) Blocked layout

Figure 2.4: Various layout of Matrices

The total memory transfers required in computing one element of the product C of the two matrices are O(N/B+1), where N is the size of the matrices and B is block transfer. The complexity of traditional matrix multiplication algorithm is $O(N^3)$. Thus giving the net memory transfer cost to be $O(N^3/B)$.

Using Strassen's algorithm [26] for cache-oblivious analysis, the three matrices were broken into four sub matrices of size N/2×N/2, rewriting the equation C=AB as:

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

The memory transfer for the algorithm are $MT(N) = 7\ MT(N/2) + O(N^2/B)$ which solves to $MT(N) = O(N^{lg7}) = O(N^{2.81})$. Thus the results proved that cache-oblivious algorithms are faster than traditional algorithms.

2.2.2. Cache-oblivious Large Integer Multiplication Algorithm

The large integer multiplication algorithm using divide-and-conquer [17] uses $\Theta(m^2)$ work and incurs $\Theta(m^2/LZ+m/L)$ cache misses where m and n are the lengths of the two integers, m being the length of the largest integer, Z is the cache size and L is the line length of the cache. The concept behind this is to divide the two integers into two equal length parts and diving the sub problems further till m=1, in which case only two digits are multiplied and in the end combining the entire result. The algorithm is explained in the example.

Example: Multiply 456 with 678

1. Consider two large integers A and B as shown in figure 2.5. Recursively divide the two numbers into equal parts till m=1or the sub problems are small enough to fit into cache.

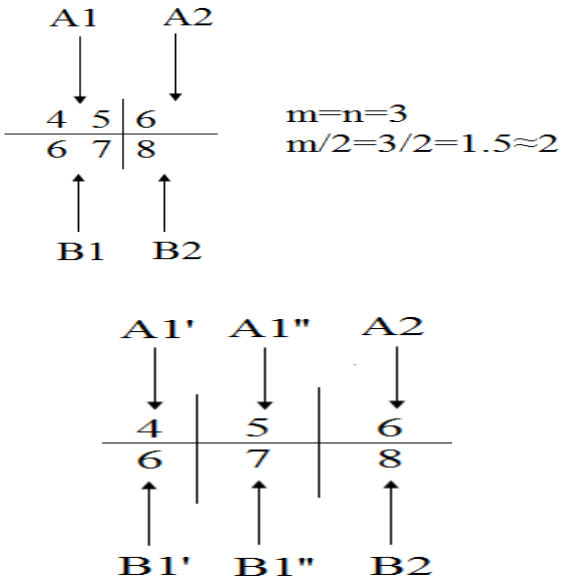


Figure 2.5: Recursive division of two large integers

2. Solve by multiplying A2×B2, A1"×B1", A1'×B1', A1"×B2 and so on, and then combine all the solutions to get the final result.

   A2×B2= 8×6= 48

   A1'×B2= 5×8= 40 Append 0 in the end for place value= 400

   A1"×B2= 4×8=32 Append two 0s in the end= 3200

   And so on. The resultant sum is equal to the multiplication of the two numbers.

2.3. Traditional Methodology for String Sorting

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 [9], radix sort variants [19; 3], 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 figure 2.6 by making the ternary search tree by inserting elements in input order.

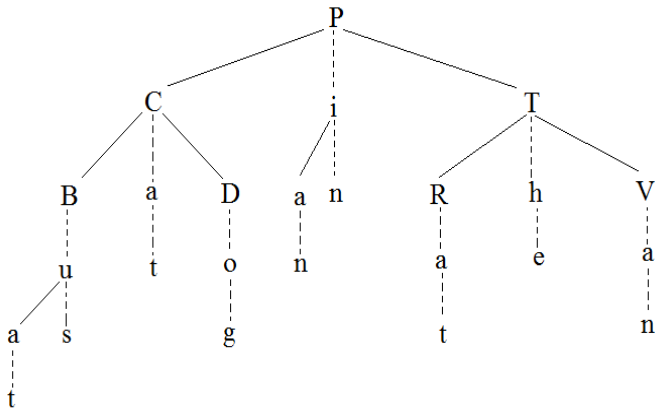


Figure 2.6: Ternary search tree depicting the sorting order of Pin, The, Cat, Rat, Dog, Fan, Fun, Pan, Van, Bus, Bat

A ternary search tree 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.

2.4. Cache-aware String Sorting Algorithms

2.4.1. CRadix Sort

CRadix sort [21] 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 one the basis

of first word the strings cannot be accessed sequentially during next sort thus 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.

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.

The working of CRadix sort is shown in figure 2.7. The next point of discussion is how to manage the contents of the key buffer. The first method (shown in figure 2.7) 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 figure 2.8) 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.

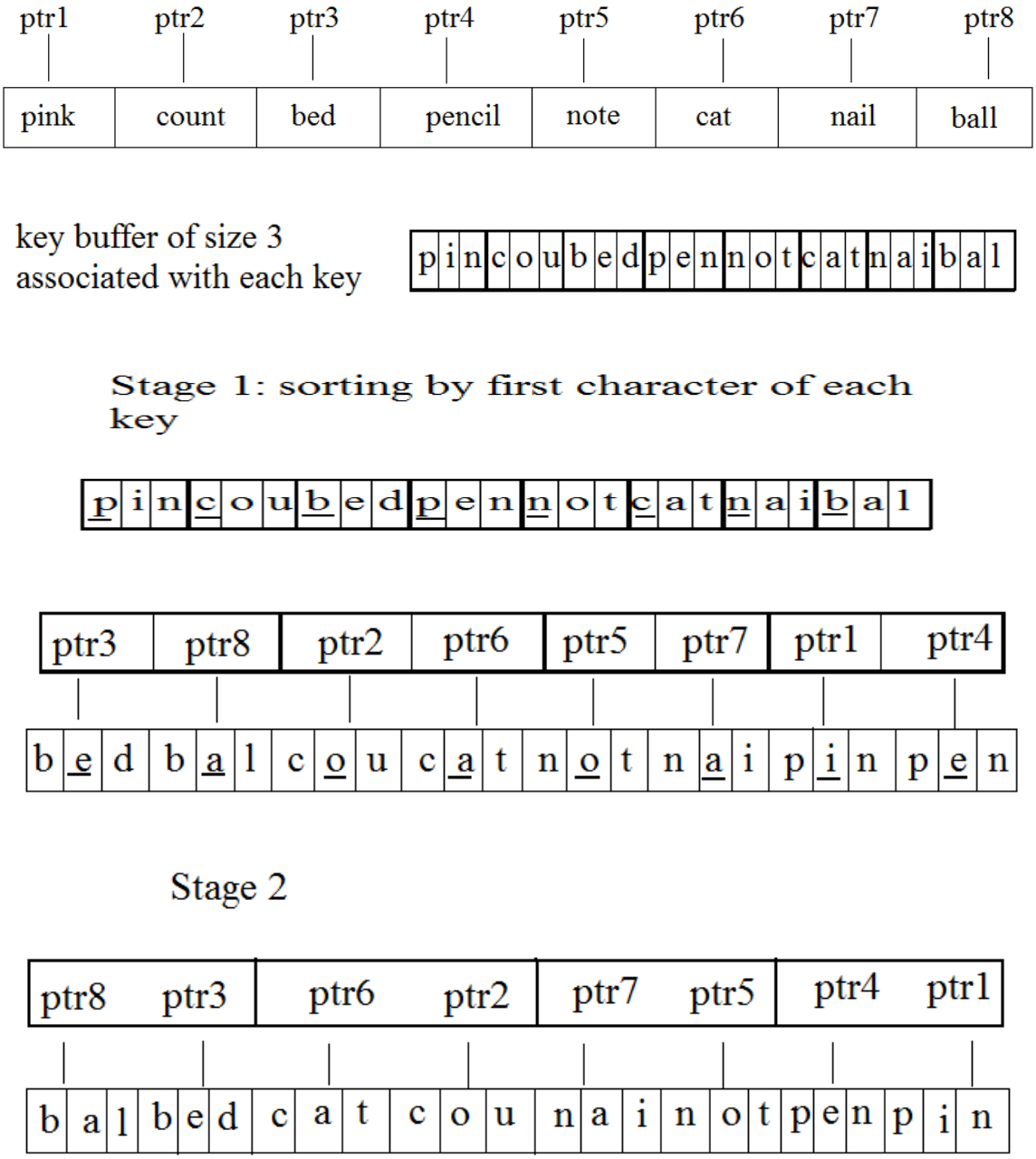


Figure 2.7: Sorting of strings: pink, count, bed, pencil, note, cat, nail and bull using key buffer of size 3

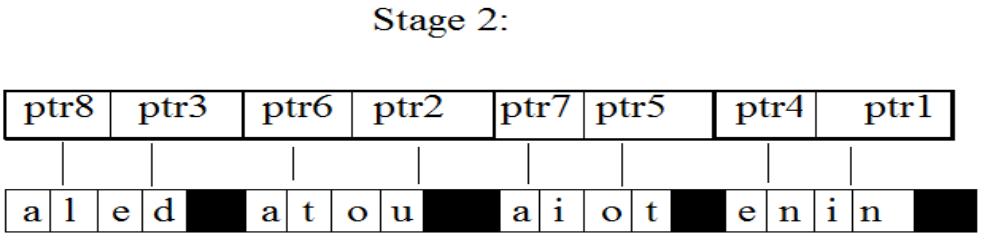


Figure 2.8: Implementation using method 2

2.4.2. Burstsort

Burstsort [25] 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 [15] and string sorting algorithms [9; 19]. 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.

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.

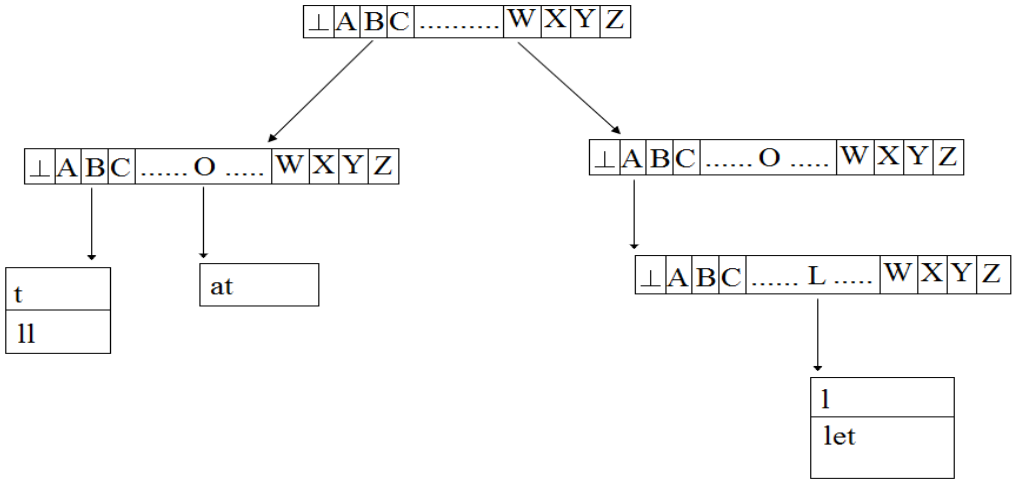


Figure 2.9 Implementation of Burstsort using trie on strings: bat, ball, wall and wallet

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

2.5. Cache-oblivious String Sorting Algorithms

To best state the algorithm we assume the input to be binary strings and the following notation [7].

K = number of strings to sort,

N = total number of words in the K strings,

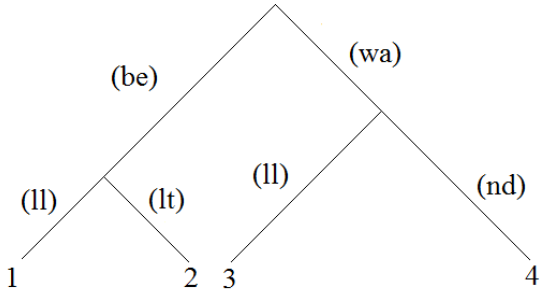
M = number of words fitting in the main memory,

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 [12] 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 lcp 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 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 [6]. This algorithm mainly concern with finding the lcp sequence for strings and then using it for permuting the strings in sorted order.



(a) Blind trie for strings: bell, belt, wall, wand

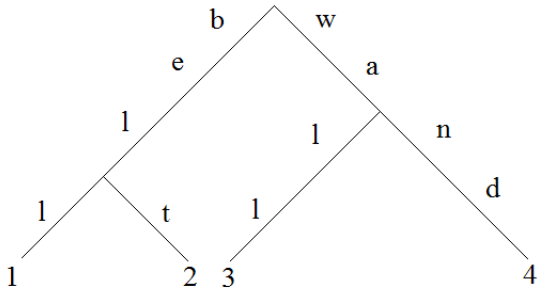


Figure 2.10: (b) Unordered blind trie for strings: bell, belt, wall, wand

Algorithm:

1. Sort the nodes according to the lcp, 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 lcp.

    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 lcp.

Chapter-3
PROBLEM STATEMENT

3.1. Problem Definition

The problem of degradation in the performance of algorithms is caused due to difference in speed of processors and memory with the advance in technology if the programs do not take advantage of the memory hierarchy. The performance of algorithms is measured by taking into account the cost of accessing the data instead of counting the number of instructions when the problem size is very large that it does not fit in the internal memory. The number of cache misses increases if the algorithm does not take advantage of the memory hierarchy thus increasing the accessing cost.

The above disadvantage was overcome by tuning the external-memory algorithm for specific levels of memory hierarchy. In external-memory model it was duty of the programmer to control the movement of data blocks to and from external memory. This was only possible if the memory parameters were known in advance. Thus making it highly platform dependent, as the size of cache and the levels of memory varies for different computers.

In order to overcome this drawback cache-oblivious algorithms are developed in which the algorithm is not dependent on any of the memory parameters and can be tuned to minimize the number of cache misses. The main aim of this thesis is to compare various existing string sorting algorithms cache-aware and cache-oblivious on the basis of parameters like type of algorithm, data structure used, etc. It also aim at developing a cache-oblivious string sorting algorithm as the performance of traditional string sorting algorithm degrades when the problem set does not fit in internal memory.

3.2. Gap Analysis

- The most efficient string sorting algorithms are cache-aware algorithms and cache-oblivious algorithm is only a theoretical concept in this area.
- The discussed cache-aware algorithms need extra memory (for key buffer) thus their performance being dependent on memory parameters which degrades the performance if sufficient amount of memory is not available.
- These algorithms do take advantage of the memory hierarchy and minimizes the number of cache-misses by serializing the access or by using buffer but these same parameters make their implementation dependent on memory parameters and make the platform dependent.
- The new technique used in the cache-oblivious algorithm which is not yet implemented theoretically is using lcp for sorting strings. The implementation of this technique is the main aim of this thesis.

3.3. Importance

The problem of string sorting is chosen as string is the very common and most occurring data type, and 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. The cache-oblivious string sorting algorithm discussed in section 2.5 has not been implemented and tested in practical environment and has been a theoretical concept.

3.4. The Proposed Objectives

The main objectives that are completed in this thesis to achieve the problem described above are:

- To study and analyze existing cache-oblivious Algorithms.
- To find various algorithm characteristics that takes advantage of cache memory.
- To implement cache-oblivious algorithm for a few problems.

- To verify and validate that they are working well with cache memory.

3.5. Methodology Used

- Study all the existing cache-oblivious algorithms that are proved better than their cache conscious counterpart.
- Compare existing string sorting algorithms for external memory on parameters like algorithm type, principle of working, data structure used, etc.
- Find algorithm characteristics that can take advantage of cache memory.
- Implementing a cache-oblivious string sorting algorithm.
- Verify and analyze the behavior of developed algorithm for data sets like random strings, URLs.

Chapter-4
IMPLEMENTATION

4.1. Analysis of Existing Algorithms

The existing cache-aware and cache-oblivious algorithms were analyzed and compared on the basis of data structure used, technique used, type of algorithm and the basic principle of working and the following comparison table was made.

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

Table II: Comparison of algorithms

The above comparison table shows that the main differences between all the algorithms is that they use different types of data structure such as array, trie, etc and the techniques

used by them are using MSD radixsort, using burst trie with string sorting and using lcp technique.

4.2. Design of New Algorithm

The concepts such as tried linear hashing, blind trie used in the algorithm are discussed in detail in this section.

4.2.1. Hashing

Hashing is a procedure of mapping a key to a value, eg., mapping names to an integer value. An important property about hashing is that two keys with distinct values are rarely mapped to same result. The hashing used is the tried linear hashing.

The tried linear hashing is a combination of tried hashing and linear hashing. To implement this buckets of size 4 is used but this size parameters mentioned make it a cache-aware algorithm so ignoring the buckets or taking a bucket of size 1 makes our implementation cache-oblivious. The results for both bucket sizes is calculated and compared.

The tried linear hashing combines the best of linear hashing and tried hashing. Linear hashing allows the data file to expand gracefully and on the other hand, trie hashing uses a trie structure for efficient handling. Tried linear hashing combine these two concepts and it reduced the number of disk access by 5% for a load factor of 0.6 and 62% for a load factor of 0.9 [24]. Tried linear hashing is better than other hashing because it maintains the order of the keys to be hashed which in this case can be the length of the lcp and does not sort the strings instead give an ordered mapping of the strings with the common prefix together.

4.2.2. Blind Trie

A blind trie is a compact representation of a trie in which any node that is an only child is merged with its parents and whose internal nodes store only an integer, which is the length of the strings in the children. This is the cache-oblivious data structure used to

handle strings i.e., it is a variant of compact trie which fits in one disk block. The structure of blind trie is shown in figure 4.1.

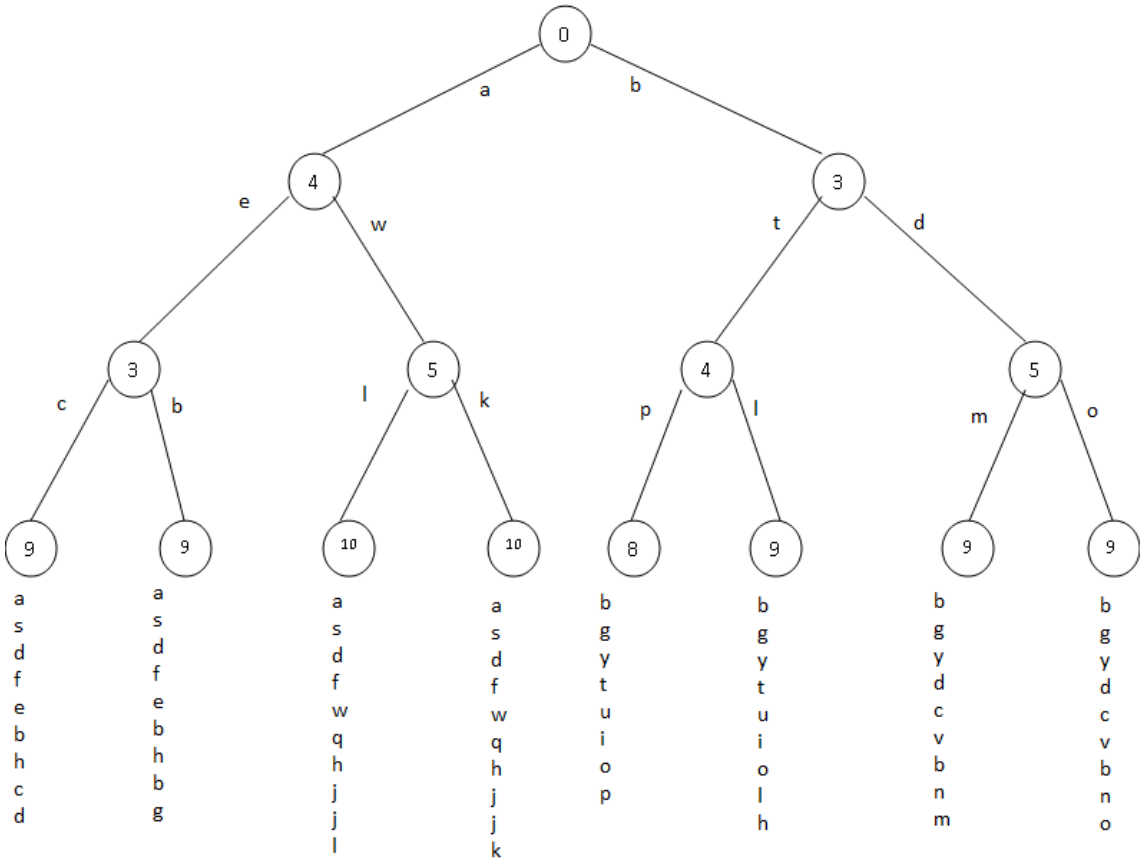


Figure 4.1: Blind trie structure.

4.2.3. List Ranking Algorithm

In list ranking algorithm a linked list of element is ranked with a distance to head/tail. The edges of list are stored contiguously in memory but not necessarily in the order the elements appear on the list. A cache-oblivious list ranking technique was demonstrated using priority queues in [23] and was used to construct other graph algorithms.

4.2.4. Improved Algorithm

1. Hash the strings to blind trie using tried linear hashing considering the order to be the length of the lcp and also mapping the strings with common prefix together.

- Hash value for key is computed using the hash function h(k).
- An array is used to maintain the trie of buckets. The hash value computed above is the bucket number.
- If (bucket number is not in array list)
  - Insert the key in the bucket
  - Bucket creation and trie is handled accordingly.
- Else
  - If (Bucket is not full)
    - Insert the key in the bucket
    - Handle bucket overflow and update trie.
  - Else
    - Linearly find the next empty bucket.

2. When the trie is complete and ordered, sort the nodes according to the branching character (the first character where the two strings differ).
3. Create a graph for trie constructed above:
   - For internal node p with d children in sorted order, create edges (p^{in},p_1^{in}), (p_1^{out},p_2^{in}) and so on. The horizontal edges are annotated using lcp of the representative strings.
   - For root the edges (p^{in},p_1^{in}) and (p^{out},p_d^{out}) are not created.
   - For leaf node only single edge is created (p^{in},p^{out}).
4. Rank the paths of the graph using the cache-oblivious list ranking technique.
5. Output the sorting permutation.

4.3. Comparison

This section compare the algorithm developed with the cache-oblivious algorithm discussed in [25] on the basis of parameters like type of hashing used and the trie structure used. Consider algorithm developed as algo-1 and the other to be algo-2. The trie structure used in algo-1 is a cache-oblivious compact trie i.e., blind trie whereas algo-2 uses a simple compact trie. The main advantage of blind trie over compact trie is that it has very small space overhead in addition to the input patterns i.e., |patterns|+O(d log n) bits. It also manages unbounded-length strings and performs much powerful search

operations such as the one supported by suffix tree [27]. The structure of both tries is explained in figure 4.2.

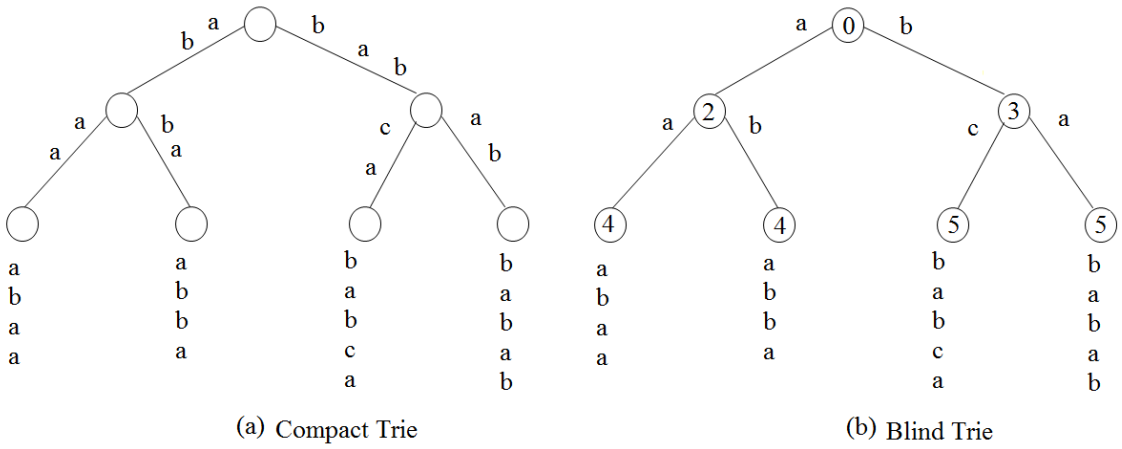


Figure 4.2: Structure of compact trie and blind trie

The next comparison is based on the type of hashing used. Algo-1 uses tried linear hashing whereas simple hashing is used in algo-2. As discussed in section 4.2.1, tried linear hashing is better than any other hashing as it maintains the order of the strings which in this case is the lcp. Thus at the time of mapping the strings are put in order of the lcp and hence saving the time of ordering them when the trie is fully constructed as is the case in algo-2.

4.4. Flowchart

The above discussed algorithm is explained using the flow chart. Figure 4.3 contains the flowchart of tried linear hashing which is the first step of the algorithm. Figure 4.4 shows the flowchart of the entire algorithm. Figure 4.5 shows a flowchart of creating graph from trie.

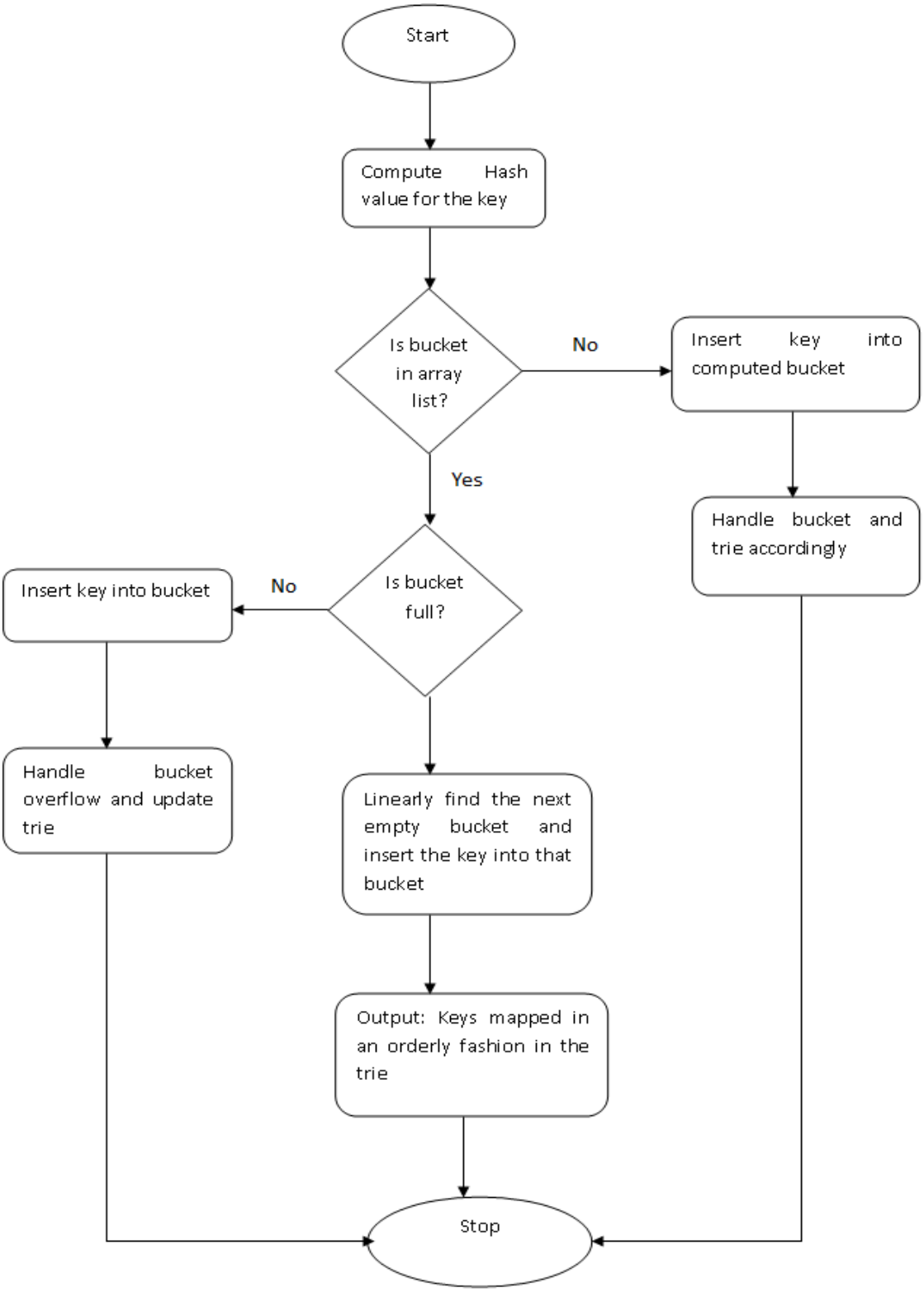


Figure 4.3: Flowchart representing Tried Linear Hashing

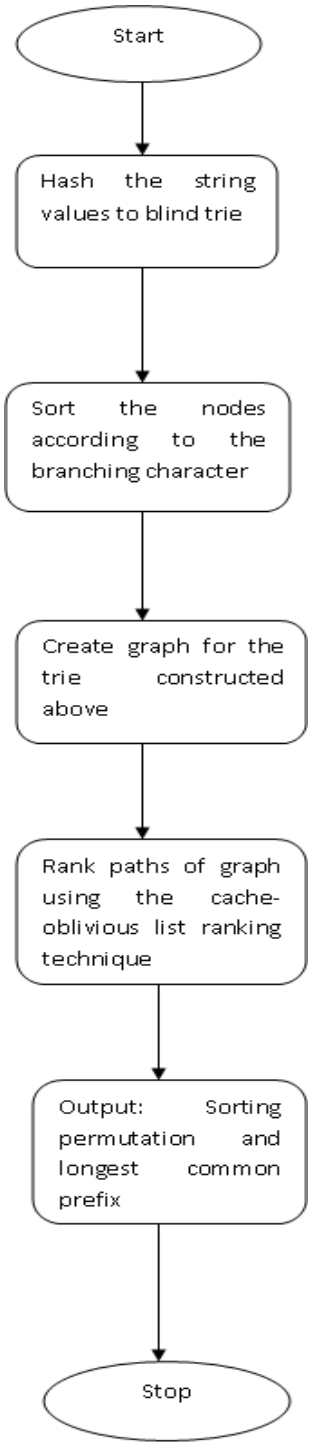


Figure 4.4: Flowchart depicting the entire algorithm

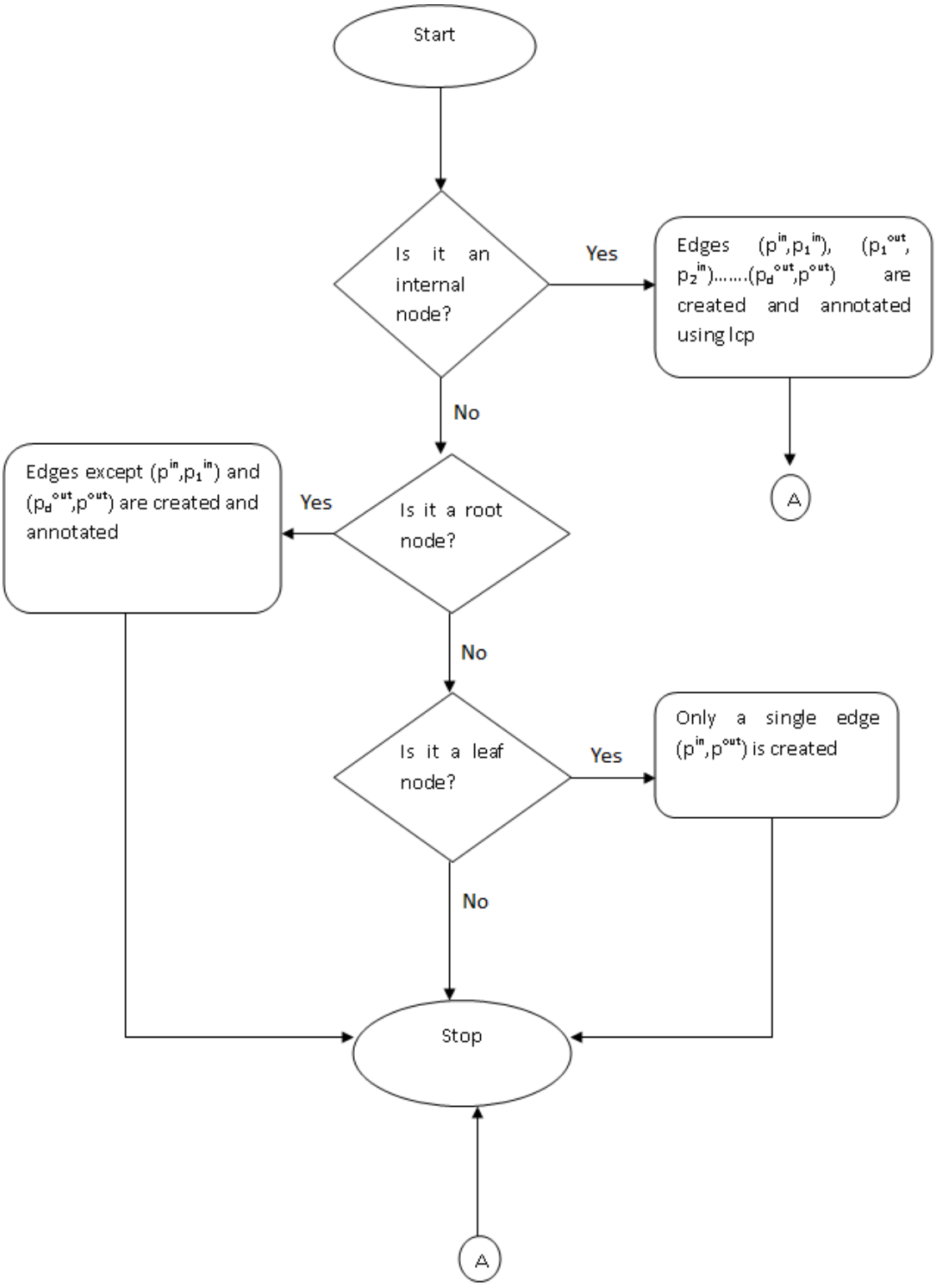


Figure 4.5: Flowchart to create graph from trie

Chapter-5
EXPERIMENTAL RESULTS

5.1. Test Platform and Test Data

In this section, the developed algorithm is tested for two bucket sizes i.e., 1 and 4 for different test data on the following test platform.

Processor: Intel Core2 Duo T6500@2.10 Ghz

Main Memory: 3.00 GB

System Type: 32-bit Operating System

Operating System: Windows Vista

Compiler: Microsoft Visual Studio 2008

The test data sets used in our experiment are:

1. Random strings: A collection of randomly generated character strings of variable size.
2. URL: These are the average webpage addresses with average length of 32 characters.

5.1.1. Test-I (Bucket Size 1)

The test-I is performed for algorithm with bucket size of 1. The bucket size of 1 here indicates the cache-obliviousness of the algorithm. Figure 5.1 shows the test results for random string data and URL data. The output is measured in the form of running time (in milliseconds) computed when the algorithm is run for the two data sets.

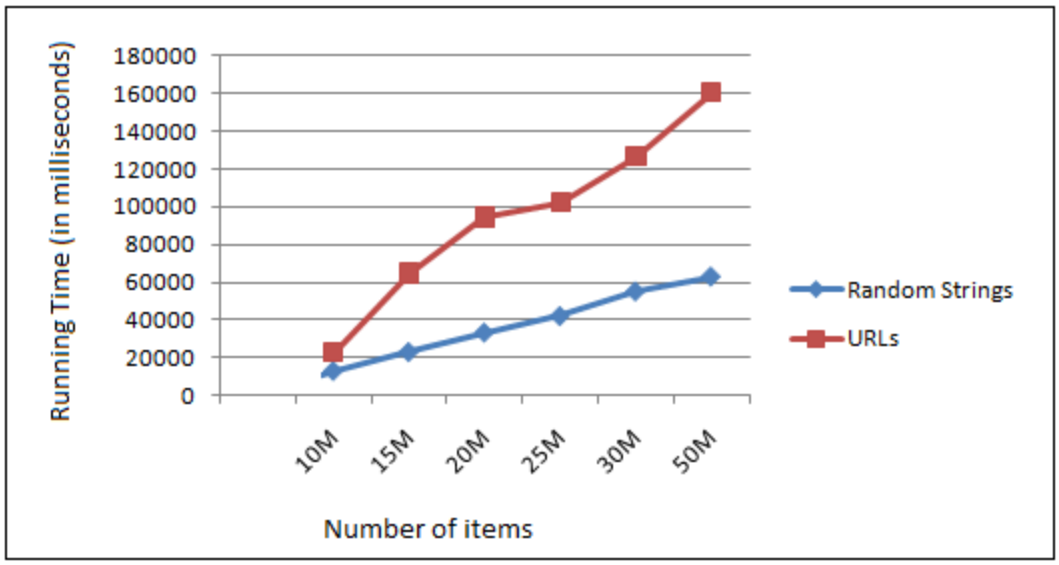


Figure 5.1: Test results for bucket size=1.

Here in figure 5.1, the algorithm takes more time for sorting URLs data set. The URLs contains duplicate strings which are to be handled separately.

5.1.2. Test-II (Bucket Size 4)

In the test we take the cache-conscious counterpart of the developed algorithm. Figure 5.2 shows the test results.

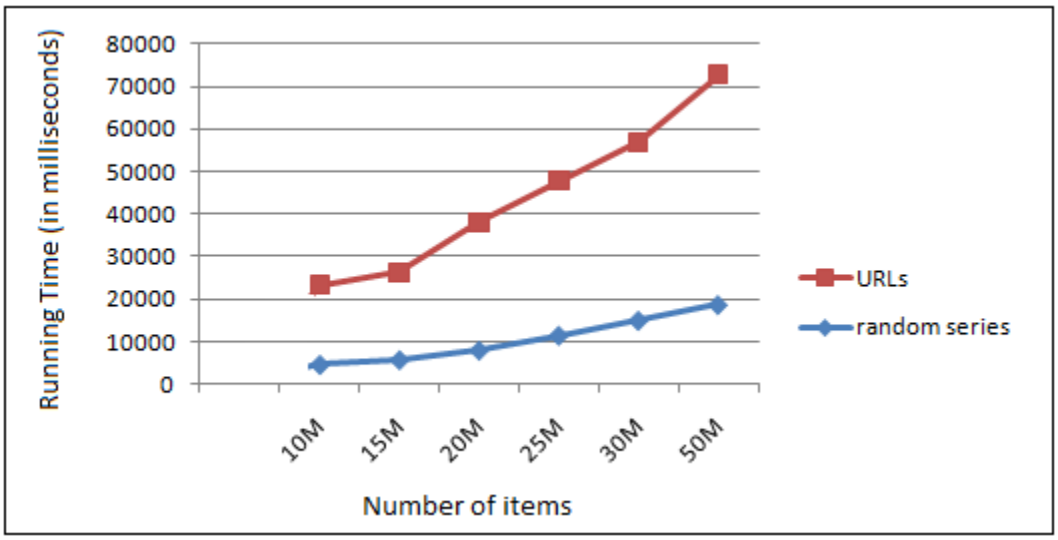
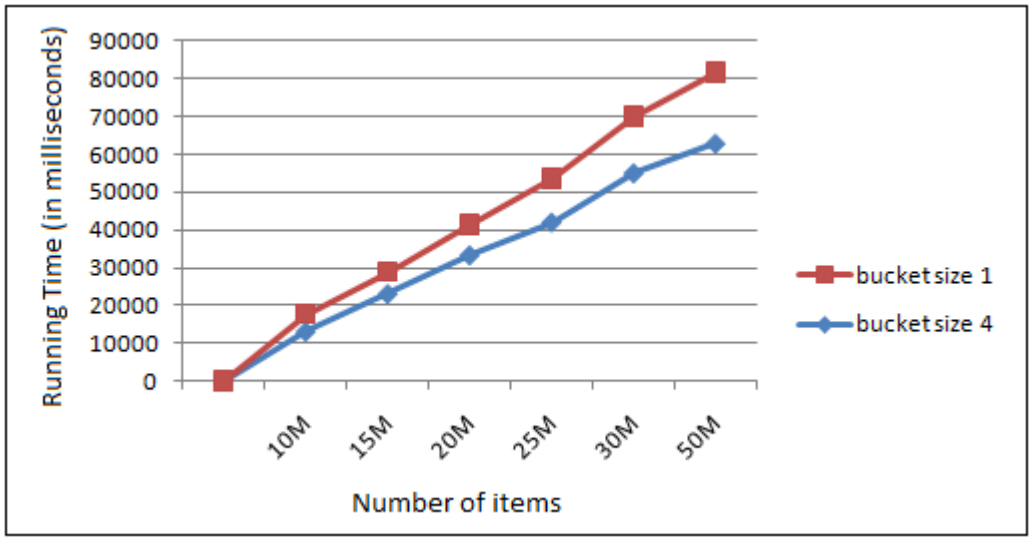


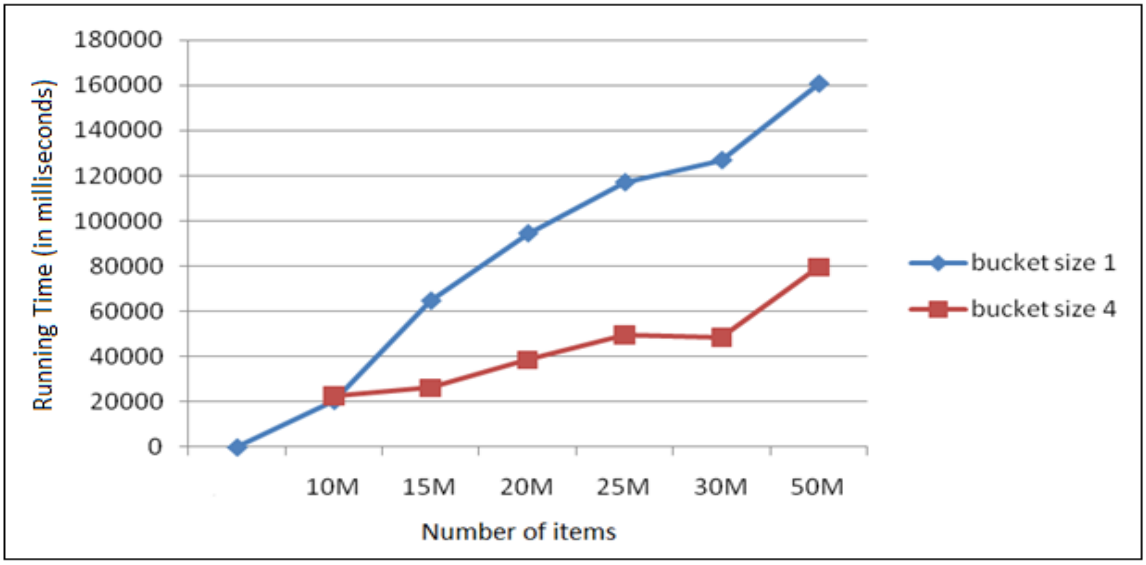
Figure 5.2: Test results for bucket size=4.

5.1.3. Comparison Results

The results in figure 5.3 show the comparison of the two algorithms. The results show that the cache-oblivious algorithm performs approximately similar to the cache-conscious one and with a little bit of improvement it can be made better than it.



(a) Comparison of algorithms for random strings test data.



(b) Comparison of algorithms for URL test data.

Figure 5.3 Comparison of two algorithms

Figure 5.3 shows that the two algorithms developed when compared, the cache-aware was found better in both the cases, but in case of random strings the cache-oblivious performed nearly the same as the cache-aware one. Less difference in performance is seen in case of random strings data (figure 5.3a). The case of URL data set is worse, the performance of both algorithms for initial small database size is almost same but as the data size increases the cache-aware algorithm performed better (figure 5.3b).

Chapter-6

CONCLUSION AND FUTURE SCOPE

The cache-oblivious algorithms are proving to be more beneficial than cache-aware algorithms in some cases and are being improved with time. These if implemented properly can help in improving the efficiency of algorithms. The various string sorting algorithms discussed have their limitations and advantages. 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 but it is still dependent on memory parameters. The randomized sorting algorithm performs well on cache-oblivious model and uses the concept of lcp 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.

The algorithm developed in this thesis is the practical implementation of the randomized sorting algorithm but is different from it as the developed algorithm uses tried linear hashing and blind trie. The experimental results show that the cache-aware algorithm performs better than the cache-oblivious one in both the cases but in case of random strings this difference is less as compared to the case of URL test data. Thus with a little improvement the cache-oblivious algorithm can outperform the cache-aware algorithm.

Future Scope

- The future work in this case can be to improve the implementation of the cache-oblivious case of tried linear hashing.

- The developed algorithm has not yet been compared to other algorithms which can be done in future to know the relative performance of the algorithm.
- For sorting the strings after making trie a general sorting algorithm is used, a cache-oblivious sorting algorithm can be used instead and it can be checked whether they provide better results or not.
- The algorithms are tested only for two data sets, the future work can be to test them for other data sets like webpage words, DNA sequence, etc and it can be checked whether the cache-oblivious algorithm is better than the cache-aware one.

REFERENCES

[1] Aggarwal, A., & Vitter, J. S. The input/output complexity of sorting and related problems. Communications of the ACM 31,9 , 1116-1127. 1988

[2] Aho, A. V., Hopcroft, J. E., & Ullman, J. D. The Design and Analysis of Computer Algorithms. Addison-Wesley Publishing Company . 1974

[3] Andersson, A., & Nilsson, S. Implementing Radix Sort. ACM J. Exp. Algorithmics 3,7 . 1998

[4] Ang, C. H., Tan, S. T., & Tan, T. C. Tried Linear Hashing. ASIAN . 1998

[5] Arge, L. The buffer tree: A new technique for optimal I/O-algorithms. Proceedings of the 4th workshop on Algorithms and Data Structure, Lecture Notes in Computer Science 955, Springer-Verlag , 334-345. 1995

[6] Arge, L., Bender, M. A., Demaine, E. D., Holland-Minkley, B., & Munro, J. I. Cache-oblivious priority queue and graph algorithm applications. In proceedings of the Symposium on Theory of Computing, ACM, New York , 268-276. 2002

[7] Arge, L., Ferragina, P., Grossi, R., & Vitter, J. S. On sorting strings in external memory (extended abstract). In ACM, editor, Proceedings of the 29th Annual ACM Symposiumon Theory of Computing (STOC '97), ACM Press , 540-548. 1997

[8] Bender, M. A., Demaine, E. D., & Farach-Colton, M. Cache-oblivious B-Trees. In Proc. 41st Annual IEEE Symposium on Foundations of Computer Science, IEEE Computer Society Press , 339-409. 2000

[9] Bentley, J., & Sedgewick, R. 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 , 360-369. 1997

[10] Brodal, G. S., Fagerberg, R., & Jacob, R. Cache-oblivious search trees via binary trees of small height. In Proc. 13th Annual ACM-SIAM Symposium on Discrete Algorithms , 39-48. 2002

[11] Christiansen, T. B. Algorithms for string sorting in external emeory. Master's Thesis . Department of Mathematics and Computer Science, University of Southern Denmark.

[12] Fagerberg, R., Pagh, A., & Pagh, R. External string sorting: Faster and cache-oblivious.

[13] Ferragina, P., & Grossi, R. The string B-tree: A new data structure for string search in external memory and its applications. Journal of the ACM, 46(2) , 236-280. 1999

[14] Frigo, M., Leiserson, C. E., Prokop, H., & Ramachandran, S. Cache-Oblivious Algorithms (extended abstract). Proceedings of the 40th Annual Symposium on Fondations of Computer Science, IEEE Computer Society Press , 285-297. 1999

[15] Heinz, S., Zobel, J., & Williams, H. E. Burst tries: A fast, efficient data structure for string keys. ACM Trans. Inform. Syst, Vol. 20, No. 2 , 193-223. 2002

[16] Hennessy, J. L. Computer Architecture: A Quantitative Approach. 3th Edition, Morgan kaufmann Publishers Inc. 2003

[17] Kamal, N., & JaiHui, Z. Cache-oblivious Algorithms, A Parallel Project Report. Singapore MIT Alliance .

[18] Knuth, D. E. Sorting and Searching. volume 3 of The Art of Computer Programming. Addison-Wesley, Reading, MA . 1973

[19] Mcilroy, P. M., Bostic, K., & Mcilroy, M. D. Engineering Radix Sort. Comput. Syst, Vol. 6, No. 1 , 5-27. 1993

[20] Moore, G. Cramming more components into integerated circuits. Electronics magazine 38 , 114-117. 1965

[21] Ng, W. H., & Kakehi, K. Cache efficient Radix sort for string sorting. IEICE TRANS. FUNDAMENTALS, Vol. E90-A, No. 2 . 2007

[22] Pagh, R., Wei, Z., Yi, K., & Zhang, Q. Cache-oblivious hashing. In Proceedings of the 20th ACM SIGMOD-SIGACT-SIGART symposium on Principles of database system of data . 2010

[23] Patt, Y. N. The I/O subsystem- a candidate for improvement. Guest Editor's Intorduction in IEEE Computer, 27 (3) , 15-16. 1994

[24] Prokop, H. Cache-oblivious Algorithms. Master's Thesis, Massachusetts Institute of Technology. 1999

[25] Sinha, R., & Wirth, A. Engineering Burstsort: Towards Fast in-place string sorting. ACM Journal of Experimental Algorithmics, Vol. 15, Article No. 2.5 . 2010

[26] Strassen, V. Gaussian elimination is not optimal. Numer Math , 13, 354-356. 1969

[27] Vitter, J. S., & Shrive, E. A. (1994). Algorithms for parallel memory I: two level memories. Algorithmica 13 , 110-146.

[28] Yotov, K., Roeder, T., Pingali, K., Gunnels, J., & Gustavson, F. An experiment comparison of cache-oblivious and cache concious programs. In proceedings of the Symposium on parallel Algorithms and Architectures, ACM, New York , 93-104. 2007

LIST OF PUBLICATIONS

1. Angrish, R., & Garg, D. Efficient string sorting algorithms: cache-aware and cache-oblivious. International Journal of Soft Computing and Engineering, Vol.1, Issue-2. 12-16. 2011 (Published)
2. Angrish, R., & Garg, D. Algorithms and data structures: efficient and cache-oblivious. International Journal of Advances in Soft Computing Technology, Vol. 1, Issue-2. 2011 (Accepted)