Vol. 6(21), Jul. 2016, PP. 3032-3042

Contents list available at IJMEC

International Journal of Mechatronics, Electrical and Computer Technology (IJMEC)

Journal Homepage: www.aeuso.org

PISSN: 2411-6173 - EISSN: 2305-0543

On The Implementation of Recursive Data Structures for Cache-Oblivious Algorithms

M. Imani[1], A. Moeini[1] and F. Ghasemi[2]

[1]*Department of Algorithms and Computation, College of Engineering, University of Tehran, Tehran, Iran*

[2]*School of Mathematics, Statistics and Computer Sciences, University of Tehran, Tehran, Iran*

***Corresponding Author's E-mail:** majidimani2012@gmail.com*

Abstract

This paper presents implementation of two important recursive data structures. First data structure is k-funnel that can merge k sorted input lists into a single sorted list. The k-funnel is an integral part of Funnelsort. The second data structure is cache-oblivious search tree that using the van Emde Boas layout for mapping from the nodes in the tree to positions in memory. We focus on the challenges in implementing them. A clever implicit navigation method of Brodal is used in the search navigation process. We also experimentally compare cache-oblivious data structures with traditional RAM model data structures.

Keywords: *data structure; cache-oblivious; k-funnel; Funnelsort; van Emde Boas layout; RAM model*

1. Introduction

The art of analysis of algorithms is dependent on a framework called model of computation. Model of computation is a simplified model of existing hardware resources and their limitations. The most commonly used model is the Random Access Machine (RAM) model [12] that many algorithms are analyzed in this model. In short, RAM model assume the system includes a CPU and a memory of unbounded size, so that each memory location is accessible in constant time. However, unlimited storage for modern computers is like a dream. The memory systems of modern computers often include a memory hierarchy so that memory access time of different levels of memory can be different. L1 cache, L2 cache, L3 cache, RAM memory, and Hard Disk are some common several level of memories that make up the memory hierarchy. An L1 cache access usually is two orders of magnitude faster than a RAM memory access and six orders of magnitude faster than a Hard Disk access [1]. In 1999 Frigo et al. [9] presented a new model of computation, the ideal-cache model or cache- oblivious model. In this model, there are two adjacent levels of the memory hierarchy. Complexity of an algorithm is calculated based on the number of block transfers between these two levels of memory, this type of complexity is known as cache complexity. In the design of algorithms in cache-oblivious model, algorithm designer is not involved in hardware parameters such as block size, memory size, and so on. This property makes the algorithms that are designed in this model are applicable to each two optional levels of memory. Divide and conquer techniques and recursive data structures play an important role in the design of cache-oblivious algorithms. van Emd Boas (vEB) [10] laying out of a B-Tree [11] of Prokop [7] and k-funnel data structure, that is used in funnelsort algorithm due to Brodal and Fegerberg [5] are examples of these commonly used data structures. In this paper our main goal is to describe the implementation challenges of funnelsort algorithm and vEB layout of B-tree, that using the implicit navigation method of Brodal [6]. Each section begins with a brief description of the concepts of the underlying data structure and ends with implementation details. In implementation of funnelsort we

Article History:
Received Date: Jan. 14, 2016
Accepted Date: Jun. 05, 2016
Available Online: Jul. 11, 2016

use from a simple method to generalize k-funnel data structure to any optional value of k, not just for power of two values of it. We also experimentally compare cache-oblivious data structures with traditional RAM model data structures.

We assume throughout the paper that the reader is familiar to the techniques of analysis and design of cache oblivious sorting and searching algorithms.

2. THE K-FUNNEL DATA STRUCTURE

The k-funnel, also called as k-merger, data structure merges k sorted input lists each containing k^{d-1} elements, value of d is equal to 3 for usage of the k-funnel in funnelsort, into a single sorted list using the elegant recursive approach. The k-funnel with respect to the concept of spatial locality of references [2] tries to cache efficiently merge elements in the lists. In addition to input lists, the k-funnel includes a number of middle buffers of size k^2. If k is a power of two, a k-funnel is divided into a top recursive $2^{\lceil \log k^{1/2} \rceil}$-subfunnel and $2^{\lceil \log k^{1/2} \rceil}$ bottom recursive $2^{\lfloor \log k^{1/2} \rfloor}$-subfunnels. Each middle buffer once is considered as output list for one of the bottom subfunnels and once considered as input list for the top subfunnel [4].

The first implementation issue is to modify the values of the recursive subfunnels in such a way that for any value of k, recursive procedure to be used properly and with the least amount of the unused memory. Fig 1a shows the number of subfunnels for $k=5$, number of bottom subfunnels is equal to $2^{\lceil \log_2 \sqrt{5} \rceil}=4$ and size of bottom subfunnels is equal to $2^{\lfloor \log_2 \sqrt{5} \rfloor}=2$. So in this case three empty lists and an additional node will be created. Fig.1b shows the number of subfunnels for $k=13$. In this case $2^{\lceil \log_2 \sqrt{13} \rceil}=4$ and $2^{\lfloor \log_2 \sqrt{13} \rfloor}=2$ are the number of bottom subfunnels and the size of bottom subfunnels, respectively. So in this case three lists will be remained so that they do not place in any bottom subfunnel. Therefore, these remaining input lists will not be merged.

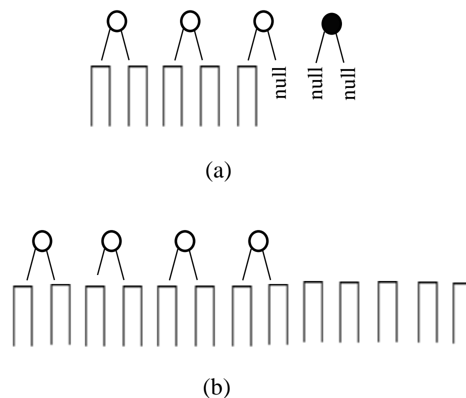

Fig 1: Representation of bottom subfunnels in the k-funnel using numbers $2^{\lceil \log k^{1/2} \rceil}$ and $2^{\lfloor \log k^{1/2} \rfloor}$
(a) Bottom subfunnels for 5-funnel (b) Bottom subfunnels for 13-funnel

The first issue arises from the fact that $2^{\lceil \log k^{1/2} \rceil}$ and $2^{\lfloor \log k^{1/2} \rfloor}$ are not exact numbers. These numbers are the numbers that are close to the exact number of bottom subfunnels and exact size of bottom subfunnels.

We use the following procedure to construct the k-funnel and also to solve the first issue. According to this procedure in line 1 the initial number of bottom subfunnels is computed. In line 2 exact size of bottom subfunnels is calculated using the $getSB$ function. The number of subfunnels will be modified in line 3, this number will be maximum number of bottom subfunnels needed to cover all the input lists. In lines 4 to 7 all bottom subfunnels except the last bottom subfunnel is recursively called. The last bottom subfunnel is called in line 8. Note that the computing of size of last bottom subfunnel is

different from others, to prevent the creation of additional empty lists. Finally, in line 9 size of top subfunnels is obtained and in line 10 top funnel is recursively called.

```
K_Funnel( int k, Point<T>* I, int p, int r, Point<T>* O, BinaryMerger<T>* S)

Inputs:   I          // array of all input lists
Output:  O          // array of merged list

1.   NB = pow( 2.0, ceil( log2( sqrt( k ) ) ) )        // initial value of NB (number of bottom subfunnels)
2.   SB = getSB( k,  NB )                               // initial value of SB (size of bottom subfunnels)
3.   NB = ceil( k / SB )                                // modified value of NB
4.   for( counter = k ; counter > SB ; counter -= SB )
5.       K_Funnel( SB,  … )

6.   i ++                                               // variable i keeps the number of used bottom subfunnels
7.   K_Funnel( k – i*SB, … )                            // recursive call to last bottom subfunnel
8.   ST = i + 1                                         // size of top subfunnels
9.   K_Funnel( ST, … )                                  // recursive call to top subfunnel

getSB( int k, int NB )

Inputs:   k          // number of iput lists
          NB         // number of bottom subfunnels
Output:  SB         // value of SB

1.   SB =  pow( 2.0, floor( log2( sqrt( k ) ) ) )      // computing an initial value of SB
2.   while ( NB * SB < k)                               // refinement of initial value of SB
3.       SB++
```

Fig 2 shows an example of K-funnel for $k = 13$. After modification of values, number of the bottom subfunnels are equal to size of bottom subfunnels ($NB = SB = 4$), but the size of the last bottom subfunnel is 1, and the size of its output buffer is 64.

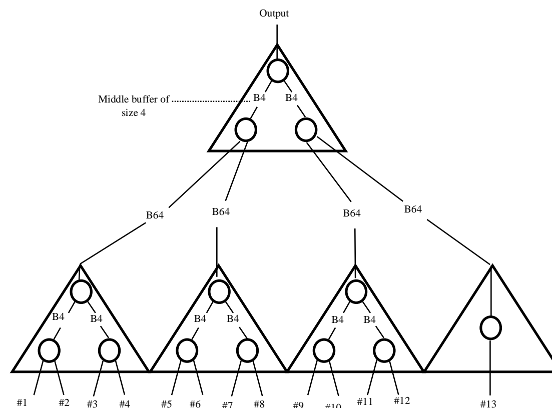

Fig 2: K-funnel for k = 13

The k-funnel data structure starts the filling buffers to merge elements just after creating all the required buffers and memories. The second issue is due to the following questions:

When do we say that a node is full? When do we say that a node is empty?

A node or a binary merger is a 2-funnel. Our binary merges can be have at most two inputs. There are important fields in each node as follows:

O: Output array

$I\ (I_1, I_2)$: Input array

$S\ (S_1, S_2)$: Source nodes (at most two binary mergers at the bottom of the binary merger that needed in recursively fill procedure calls)

$count$: The total number of elements in the inputs of a node and all its sources that are not merged.

$left_index$: Index of left input

$right_index$: Index of right input

k: Number of inputs (one or two)

p: Start index of input array

r: Maximum index of input array

z: Maximum index of output array

$standardSI$: Equal size of input lists of a subfunnel except probably the last input list. In fact, we only define an input array for each subfunnel but we use it as the input lists just by using the p, r and $standardSI$ indexes. So in a binary merger with $k = 2$, start and end indexes for first input list are p and $p + standardSI - 1$, and for second input list are $p + standardSI$ and r, respectively.

2.1. Full and Empty Conditions

Filling a node occurs when the output array is filled as well as in the other case we can say that a node is full when its count value is equal to zero. The count variable is also used in the pruning of the k-funnel data structure so that additional recursive calls to fill function is not performed. Fig 3 shows an example of full condition for K-funnel with N = 128 elements and k = 5 input lists. Shaded lists are lists that completely merged. Nodes with a zero count are full nodes. The additional recursive calls to fill function is not performed in the shaded node. Emptiness of a node depends on the value of k and input buffers, and it occurs under one of the following conditions:

- Index ($left_index$ or $right_index$) is greater than the maximum allowable input index and S is not null.
- Current input element is equal to minus one, where minus one shows the end of elements in the array, and S is not null.

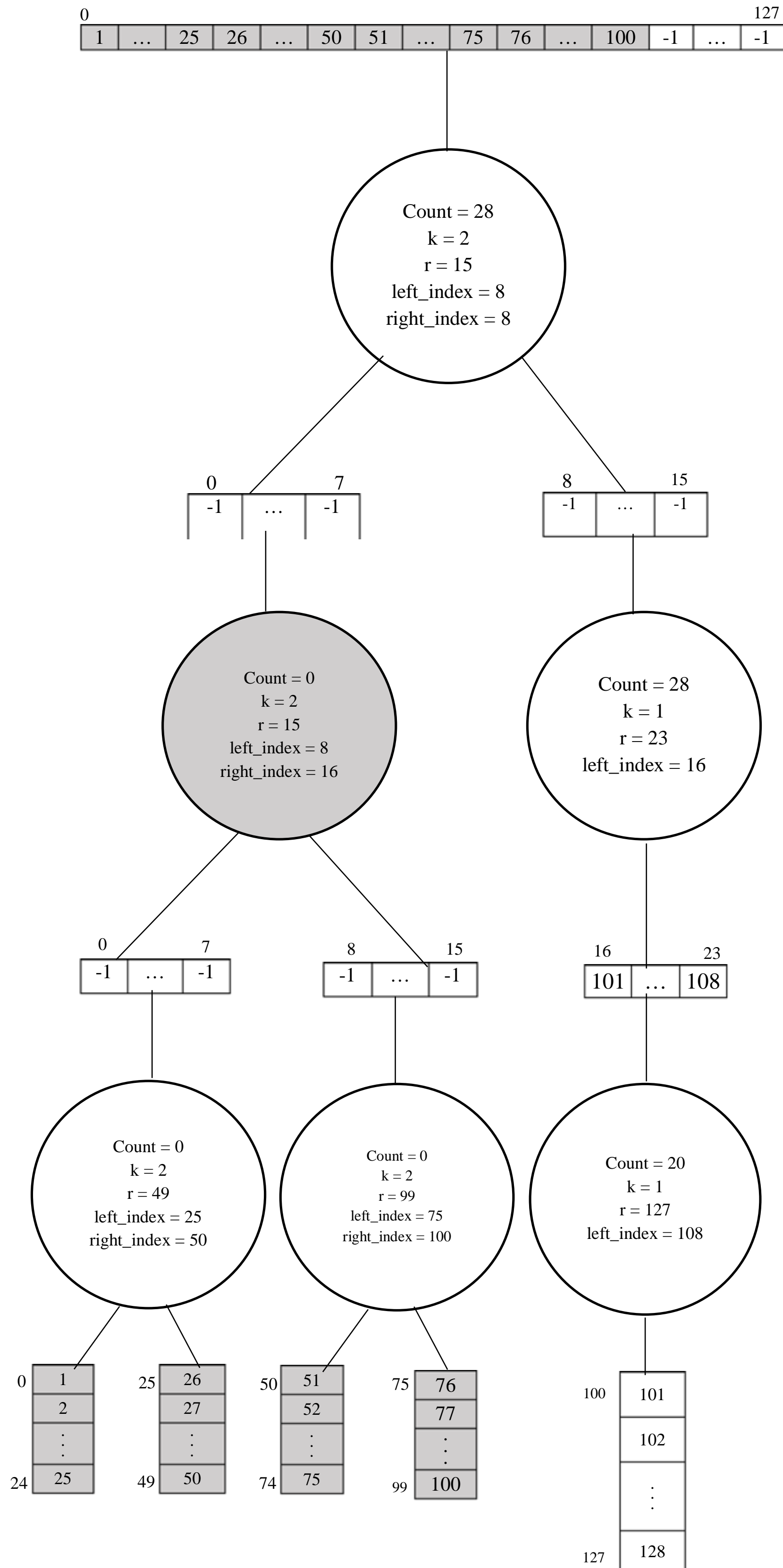

Fig 3: K-funnel for N = 128 elements and k = 5 input lists

3. The van Emde Boas Layout

Consider a binary search tree (BST) of height $\log_2 N$ in which all leaves are on the same level. The purpose is to propose a mapping from the nodes in BST to positions in memory. There are many ways to achieve this such as pre-order, post-order traversal, in-order traversal, and level-order traversal of a BST, but in 2000 Bender et al. [8] presented a new mapping technique. Their proposed layout, called van Emde Boas layout. The idea is all nodes in subtree of size B, that B is block size, rooted at root of original tree first lie in memory. Then all subtrees of size B rooted at its leaves lie in memory and so on.

Fig 4 shows the van Emde Boas layout that recursively splits the tree at the middle level of edges. By cutting the tree from middle level of edges new recursive subtrees $A, B_1, B_2, \ldots, B_m$ with the roughly same size are produced so that the A is a top subtree and others are bottom subtrees. In the layout of the tree first all nodes of A must to lie in memory, then all nodes of B_1 , and so on. If h, height of the tree, is a power of two, m is roughly $\sqrt{N}$ and recursive subtrees all have size roughly $\sqrt{N}$. If h is not a power of two, top subtrees and bottom subtrees of the same size or height are not produced instead height of the bottom subtrees is $2^{\lceil \log_2(h/2) \rceil}$ and height of the top subtree is $h-2^{\lceil \log_2(h/2) \rceil}$. Notice the difference between size of subtrees that produced in van Emde Boas and k-funnel: In vEB layout size of top subtree is smaller than the size of all bottom subtrees, but In k-funnel size of top subtree is larger than the size of all bottom subtrees [4].

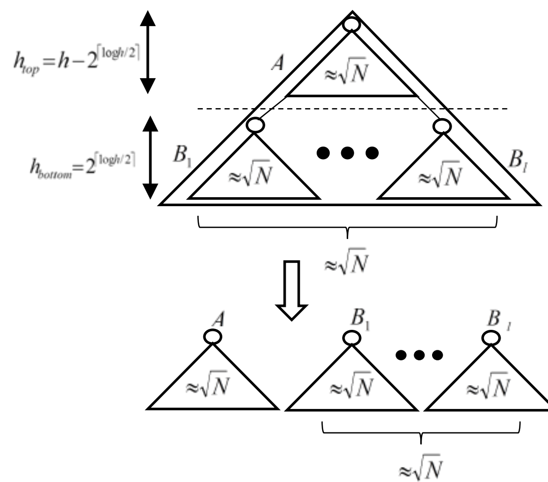

Fig 4: The van Emde Boas layout

3.1. van Emde Boas Layout Construction

Given a sorted sequence of elements Ohashi [3] presented an algorithm for constructing a vEB layout in $O(N \log_2 \log_2 N)$ time and Rønn [4] presented a new $O(N)$ time algorithm.

In the construction of van Emde Boas layout we assume that we have an unsorted array. First, we recursively convert input array into a binary search tree using the linear time median algorithm [13]. It is clear that the cache complexity of this process, binary search tree construction, is $O(N/B \log_2 N/B)$ memory transfers. Nevertheless, there is a better algorithm to construct a binary search tree that its cache complexity is $O(sort(N))$ memory transfers, but here it is not necessary for us. Second, by using the breadth-first search (BFS) traversal of tree from specific nodes, or roots, and until specific heights h_{top} and h_{bottom}, desired van Emd Boas layout subtrees to be obtained. Then we recursively continue

these two steps until the placement of all nodes, or elements, entirely. Fig 5 shows an example of a complete binary search tree of height 5 and its corresponding vEB layout.

3.2. Search Navigation Method

There are two different ways of navigating, obtaining of the position of a node from position of its parent, in a search tree, explicit navigation and implicit navigation. In explicit navigation each parent node contains pointers to its children, and in implicit navigation positions of the child nodes of a parent node are calculated based on the position of the parent node in the tree. The navigation from node to node using pointers is straightforward, just to use the pointer representation of a tree. However, implicit navigation can sometimes be complicated. Although the use of each method depends on several factors, the main advantage of implicit navigation may be that it saves space.

In implementation of binary search using the van Emde Boas layout, we use from the implicit navigation method of Brodal et al. [6]. Since most important part of our vEB layout implementation is implicit navigation, we now review this method.

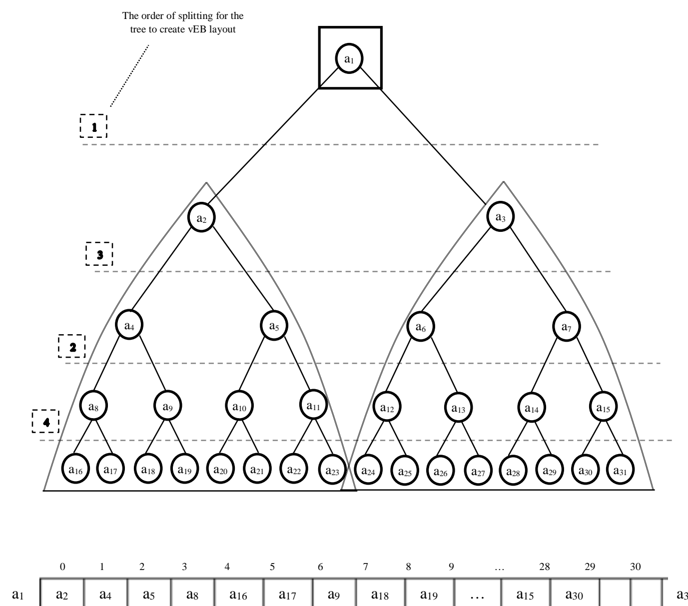

Fig 5: Complete BST of height 5 and its vEB layout

In fact, implicit navigation method of Brodal using the breadth-first layout indexes and vEB layout properties to calculate the vEB layout indexes. Given a node v at position i in a binary tree, the positions of its children are easy to calculate, the position of its left child is given by $2i$ and position of its right child is given by $2i+1$. Consider the search path for element a_{28} in binary search tree in Fig. 7. Its breadth-first layout indexes are 0, 2, 6, and 13, 27 and its vEB layout indexes are 0, 16, 18, 25, and 26. We are able to obtain the position of the root of all bottom subtrees by following equation:

$$position\ of\ r_y = position\ of\ r + S_T + y * S_B \tag{1}$$

In above equation variables are as follow:

r_y: The root of a bottom subtree that is yth subtree at the same level
S_T: The size of corresponding top subtree to r_y

S_B: The size of corresponding subtree to r_y

Now consider a complete binary tree first we have to calculate three arrays of length d, that d is depth of tree:

P_T: the one-dimensional array that stores the size of the top subtree for each depth d
P_B: the one-dimensional array that stores the size of the bottom subtree for each depth d
P_D: the one-dimensional array that stores the depth of the root of top subtree for each depth d

We are able to calculate above three arrays in $O(logN)$, just during the splitting process at time of creating the memory layout. Table 1 shows these arrays in the tree of the Fig 5.

Each vEB layout index of tree is computed by following equation:

$$Pos[d] = Pos[P_D[d]] + P_T[d] + ((i+1) \,\&\, P_T[d])P_B[d] \tag{2}$$

Table 1: Precalculated arrays for vEB search navigation

d	P_B	P_T	P_D
0	0	0	0
1	15	1	0
2	1	1	1
3	3	3	1
4	1	1	3

$(i+1)\&P_T[d]$ will be index of corresponding bottom subtree relative to leftmost bottom subtree at the same level and i is the position of the corresponding node in the breadth-first layout [4]. Let root of the original tree is at position 0 of vEB layout, and it also place in depth 0 of tree.

Following procedure shows the main part of the lines of code of the vEB layout construction:

vEB(Point<T>* vEBLayout, Point<T>* points, int p, int r, int pre, int d, int index, int* PB, int* PT, int* PD)

Inputs: points // points in BFS layout
Output: vEBLayout // same points in vEB layout

```
1.  int N = r - p + 1                                        // total number of points
2.  int h = ( int ) log2( N ) + 1                            // height of BST
3.  int hB = ( int ) pow( 2.0, ceil( log2( h / 2.0 ) ) )     // height of bottom subtrees in vEBLayout
4.  int hT = h – hB                                          // height of top subtree in vEB layout
5.  int sizeOfTop = pow(2.0, hT ) – 1                        // size of top subtree
6.  int sizeOfBottom = pow( 2.0, hB ) – 1                    // size of bottom subtree
7.  PT[pre + hT] = sizeOfTop                                 // computing the P_T[d]
8.  PD[pre + hT] = pre                                       // computing the P_D[d]
9.  Point<T>* top = BFS( points, p, hT, d, index )           // splitting the top subtree
10. vEB( vEBLayout, top, 0, sizeOfTop - 1, pre, d, index, PB, PT, PD )   // recursive call to top subtree
11. for ( int k = 0 ; k <  pow(2, hT) ; k ++)                // recursive calls to bottom subtrees
12.     if ( k == 0 )
13.         PB[pre + hT] = sizeOfBottom                      // computing the P_B[d]
14.     Point<T>* bottom = BFS( points, startIndex + k, hB, d, index )   // splitting k'th bottom subtree
15.     vEB( vEBLayout, bottom, 0, sizeOfBottom - 1, pre + hT, d, index, PB, PT, PD )
```

Following procedure depicts the $vEBSearch$ program that it use from Equation 2 for parent to child navigation in the vEB Layout:

```
vEBSearch( Point<T>* vEBLayout, int p, int r, Point<T> point, int index, int* PB, int* PT, int* PD )

Inputs:   vEBLayout        // points in vEB layout
           point             // search point
Output:  index             // index of search point

1.  int h = ( int ) log2(r - p + 1) + 1                          // height of BST
2.  while ( D < h && bstIndex <= r )
3.      pos[D] = pos[PD[D]]+PT[D]+((bstIndex + 1 )&PT[D])*PB[D]   // computing pos array using the Equation 2
4.      if (vEBLayout[pos[D]].nums[index] == point.nums[index])
5.   return pos[D]
6.     else if (point.nums[index] < vEBLayout[pos[D]].nums[index])
7.         bstIndex = 2 * bstIndex + 1
8.     else
9.         bstIndex = 2 * bstIndex + 2
10.    D++
```

4. Experimental Results

In the final section of this paper we practically compare a RAM model sorting algorithm, mergesort, with cache-oblivious model sorting algorithm, funnelsort. We also compare a RAM model searching algorithm, binary search, with cache-oblivious searching algorithm, binary search using the vEB layout. The actual funnelsort algorithm is an $N^{1/3}$-way mergesort with $N^{1/3}$-funnel data structure. Following procedure shows the main section of our funnelsort program.

```
cacheObliviousSort( Point<T>* A, int p, int r, Point<T>* O, int q, int z, int index )

Inputs:   A                 // initial points
Output:  O                // sorted points

1.   if (N <= 8)
2.     insertionSort(A, p, r, index)
3.   else
4.     for ( int i = 0; i < N13 - 1; I ++ )
5.        cacheObliviousSort(A, p + i*sizeOfList, p + ((i + 1) * sizeOfList - 1), O, p + i *  sizeOfList, p + ((i + 1) * sizeOfList - 1), index )

6.     cacheObliviousSort(A, p + (N13 - 1) * sizeOfList, r, O, p + (N13 - 1) * sizeOfList, r, index)

7.     BinaryMerger<float> rooot = k_funnel(N13, A, p, r, O, q, z, NULL, sizeOfList, index)

8.     fill(rooot, index)
```

All implementations of this paper is done in C++ programming language. We tested the dataset on a system with the specifications in the following table:

Table 2: System specifications

Resource	*Information*
Processor	Intel® Pentium® P6200 2.13 GHz
L1 cache	128 KB
L2 cache	512 KB
L3 cache	3.0 MB
RAM	3GB DDR3 Memory
Hard	500 GB

Our dataset consist of random numbers that were generated in each repetition of the experiment. For each N, which N is the size of data, each program was executed ten times. Results are visible in the table 3 and table 4, Fig 8 and Fig 9. According to the charts, as long as data grows, cache-oblivious programs have better performance than the RAM model programs. At the same time complexity of design of cache-oblivious algorithms is more than the design of RAM model algorithms.

Table 3: Sorting execution time

	Execution time (seconds) average time in 10 repetitions	
N	MergeSort	Funnelsort
$\mathbf{10^2}$	0.058	0.0967
$\mathbf{10^3}$	0.0686	0.0652
$\mathbf{10^4}$	0.0638	0.1181
$\mathbf{10^5}$	0.4149	0.4077
$\mathbf{10^6}$	4	3.3
$\mathbf{2 \times 10^6}$	35.83	7.83
$\mathbf{2.5 \times 10^6}$	59.66	10.33
$\mathbf{3 \times 10^6}$	78.33	13.33
$\mathbf{4 \times 10^6}$	234.66	18.33

Table 4: Searching execution time

	Execution time (seconds) average time in 10 repetitions	
N	Binary Search	vEBSearch
1023	0.00035625	0.00001175
4095	0.00035625	0.00001175
16383	0.000397	0.00004
65535	0.000472	0.000036
262143	0.0004668	0.000042
1048575	0.000595	0.000056

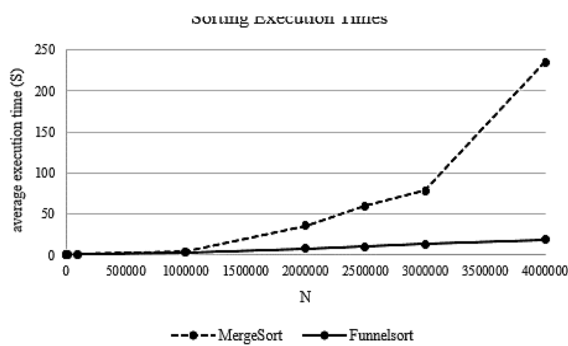

Fig 6: Practical Comparisons between funnelsort and mergesort

Fig 7: Practical Comparisons between binary search and vEBSearch

Conclusions

In this paper we implemented sorting and searching algorithms in RAM and cache-oblivious model of computation. We examined some important cases of basic cache-oblivious algorithms that may be of concern to anyone who works in this area. In the case of funnelsort, we using from generalized version of k-funnel that can be works for all values of k. Finally, we compared funnelsort with mergesort, and binary search with binary search using the van Emde Boas layout.

References

[1] COMPAQ, Documentation library, “http://ftp.digital.com/pub/Digital/info/semiconductor/literature/dsc-library.html”, 1999.
[2] D. A. Patterson, and J. L. Hennessy. "Computer organization and design." Morgan Kaufmann, 2007, pp. 460-470.
[3] D. Ohashi, "Cache oblivious data structures." Master’s thesis, University of Waterloo, 2000.
[4] F. Rønn, “Cache-oblivious searching and sorting.” Diss. Diplomarbeit, Department of Computer Science (University of Copenhagen), 2003.
[5] G. S. Brodal, R. Fagerberg, “Cache-Oblivious Distribution Sweeping.” Proceedings of the 29th International Colloquium on Automata, Languages, and Programming, Lecture Note in Computer Science 2380, Springer-Verlag, 2002, pp. 426-438.

[6] G. S. Brodal, R. Fagerberg, and R. Jacob, “Cache-Oblivious Search Trees via Binary Trees of Small Height.” Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms, ACM Press, 2002, pp. 39-48.
[7] H. Prokop, "Cache-oblivious algorithms." Master’s thesis, Massachusetts Institute of Technology, Cambridge, 1999.
[8] M. Bender, E. D. Demaine, and M. Farach-Colton, "Cache-oblivious B-trees." Foundations of Computer Science, 2000. Proceedings. 41st Annual Symposium on IEEE, 2000, pp. 4-12.
[9] M. Frigo, C. E. Leiserson, H. Prokop, and S. amachandran, “Cache-oblivious algorithms.” Proceedings of the 40th Annual Symposium on Foundations of Computer Science, IEEE Computer Society Press, 1999, pp. 285-297.
[10] P. Van Emde Boas, "Preserving order in a forest in less than logarithmic time and linear space." Information processing letters 6, no. 3, 1977, pp. 80-82.
[11] R. Bayer, and E. McCreight, “Organization and maintenance of large ordered indexes.” Acta Information, 1972, pp. 173-189.
[12] S. A. Cook, R. A. Reckhow, “Time-Bounded Random Access Machines.” Proceedings of the 4th Annual ACM Symposium on Theory of Computing, ACM Press, 1972, pp. 73-80.
[13] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, "Introduction to algorithms." The Knuth-Morris-Pratt Algorithm, 2001, pp. 166-170.