A Detailed Experimental Analysis of Library Sort Algorithm

Neetu Faujdar
Department of CSE
Jaypee University of Information Technology, Waknaghat
Solan, India
neetu.faujdar@mail.juit.ac.in

Satya Prakash Ghrera
Department of CSE & IT
Jaypee University of Information Technology, Waknaghat
Solan, India
sp.ghrera@juit.ac.in

Abstract— **One of the basic problem in computer science is to arrange the items in lexicographic order. Sorting is one of the major research topic. There are number of sorting algorithms. This paper presents the implementation and detailed analysis of library sort. Library sort is also called gapped insertion sort. It is a sorting algorithm that uses insertion sort with gaps. Time taken by insertion sort is O (n^2) because each insertion takes O (n) time; and library sort has insertion time O (log n) with high probability. Total running time of library sort is O (n log n) time with high probability. Library sort has better run time than insertion sort, but the library sort also has some issues. The first issue is the value of gap which is denoted by 'ε', the range of gap is given, but it is yet to be implemented to check that given range is satisfying the concept of library sort algorithm. The second issue is re-balancing which accounts the cost and time of library sort algorithm. The third issue is that, only a theoretical concept of library sort is given, but the concept is not implemented. So, to overcome these issues of library sort, in this paper, we have implemented the concept of library sort and done the detailed experimental analysis of library sort algorithm, and measure the performance of library sort algorithm on a dataset.**

Keywords— Sorting; Insertion sort; Library sort; Time Complexity; Space Complexity.

I. INTRODUCTION

In computer science, sorting algorithm [2] is an algorithm that sorts the list of items in a certain order; Insertion sort iterates, takes one input element with each repetition, and put it into the sorted output list. Repeat the process until no input elements remains unprocessed. Insertion sort [10] is less efficient on large number of items as it takes O (n^2) time in worst case, and the best case of insertion sorting occurs when data is in sorted manner and it is O (n) in best case. Insertion sort is an adaptive [3] sorting algorithm; it is also a stable sorting algorithm [4].

Michael A. Bender proposed the library sort algorithm or gapped insertion sort [1]. Library sort is a sorting algorithm that comes by an insertion sort but there is a space after each element in the array to accelerate subsequent insertions. Library sort is an adaptive sorting and also a stable sorting algorithm [9]. If we leave more space, the fewer elements we move on insertions. The author achieves the O (log n) insertions with high probability using the evenly distributed gap, and the algorithm runs O (n log n) with high probability. O (n log n) is better than O (n^2). But the library sort also has some issues. The first issue is the value of gap, range of gap is given, but it is yet to be implemented after implementation, we can only decide that given range is satisfying the concept of library sort. The second issue is re-balancing, re-balancing has done after 2^i elements in library sort, but it also accounts cost and time of library sort algorithm. The third issue is that only a theoretical concept of library sort is given by Bender et al but he has not implemented it. So, in this paper to overcome these issues of library sort, we have implemented the concept, done the detailed experimental analysis and we measure the performance on a dataset. The application of leaving gaps for insertions in a data structure is used by Itai, Konheim, and Rodeh [8]. This idea has found recent application in external memory and cache-oblivious algorithms in the packed memory structure of Bender, Demaine and Farach-Colton [1] and later used in [6, 7]. The remainder of this paper is organized as follows. The detail of library sort algorithm is given in section 2 and the time complexity based testing using the dataset is done in section 3. The space complexity based testing on a dataset is done in section 4 [13]. The re-balancing based testing is done in section 5. We analysis the performance of library sort in section 6 and present the conclusion and future work with a few comments in section 7 and 8.

II. LIBRARY SORT ALGORITHM

The algorithm of library sort is as follows, there are three steps of the library sort algorithm.

1. Binary Search with blanks: In Library sort we have to search a number and the best search for an array is found by binary search. The array 'S' is sorted but has the gap. As in computer, gaps of memory will hold some value and this value is fixed to sentential value that is '-1'. Due to this reason we cannot directly use the binary search for sorting. So we have modified the binary search. After finding the mid, if mid comes out to be '-1' then we move linearly left and right until we get a non-zero value. These values are named as m1 and m2. Based on these values we define new low, high and mid for the working. Another difference in the binary search

978-1-4673-6540-6/15/$31.00 ©2015 IEEE

presented below is that it is not only searches the element in the list, but also reports the correct position where we have to insert the number.

ALGORITHM 1. Library Sort Binary Search with blanks

Input: Data to be sorted *n* and number to be searched *k.*
Output: Position to enter the element *d*.

```
while(low < high)
    mid = (low + high)/2
    if (S[mid] = = -1) then
        m1= m2 = mid
        while(S[m1] = = -1 && m1>=low)
            m1= m1-1
        endwhile
        while(S[m2] = = -1 && m2< = high)
            m2 = m2 + 1
        endwhile
        if(m1< 0 && m2 >= high+1) then
            low = high=m1+1
        endif
        if(m1= = 0 && m2>= high+1)
            if(k < S[m1]) then
                low = high = m1
            else
                low = high = m1+1
            endif
        endif
        if(m1 > 0 &&  m2>=high+1)
            if(k >=S[m1]) then
                high = m1+1
            else
                high = m1-1
            endif
        endif
        if(m1 > 0 && m2 < high+1)
            if(k <=S[m1])
                if(k==S[m]) then
                    low = high = m1
                else
                    high=m1-1
                endif
            endif
        endif
        if(k > S[m1] && k < S[m2]) then
            low = m1 + 1
            high = m2 –1
        endif
        if(k >=S[m2])
            if(m2 < high) then
                low = m2 + 1
            else
                low = m2
            endif
        endif
        if(m1= = 0 && m2 <=high)
            if(k <= S[m1]) then
                high = m1
            endif
        endif
        if(k > S[m1] && k < S[m2]) then
            low = m1 + 1
            high = m2 – 1
        endif
        if(k >=S[m2]
            if(m2<=high)then
                low = m2 + 1
            else
                low = m2
            endif
        endif
        else
        if(S[mid] < k) then
            low = mid + 1
        endif
        if(S[mid] > k) then
            high = mid – 1
        endif
        if(S[mid] = = k) then
            return mid
        endif
    endwhile
    if(low==high)
        if(S[low]<k) then
            low++
            high++
        endif
    endif
        return low
end
```

2. Insertion: As we know, library sort is also known by the name 'gapped insertion sort'. If the value to be inserted is in the gap, then it is ok, but if there is an element in that particular position, we have to shift the elements till we find the next gap.

ALGORITHM 2. Library Sort Insertion

Input: Data to be sorted *n* and pass number *i.*
Output: Sorted list but without gaps.

```
Set i1=0 and c1=0
if(i = =1) then
    i1=i - 1
    c1 = 0
endif
S1=pow(2,i)
if(S1 > size) then
    S1 = size
endif
for j= (pow(2,i1-1) - c1) to S1do
    k= search(pow(2,i)+pow(2,i+1),a[j])
    if(S[k] != -1) then
        managetill(k)
    endif
    S[k] = a[j]
```

```
            endfor
        end
```

3. Re-balancing: Re-balancing is done after inserting 2^i elements.

ALGORITHM 3. Library Sort Re-balancing

Input: Sorted data but not uniformly gapped and re-balancing factor *e*.
Output: Sorted list of *n* items.

```
        while(l < n) do
            if(S[j] ! = -1) then
                reba[i] = S[j]
                i++
                j++
                l++
                for k=0 to e do
                    reba[i] = -1
                    i++
                endfor
            else
                j++
            endif
            for k = 0 to I do
                S[k] = reba[k]
            endfor
        endwhile
    end
```

III. Execution Time Testing of Library Sort on a Dataset

We have tested the library sort algorithm on a dataset [T10I4D100K (.gz)] [5, 12] by increasing the value of the gap (ε). The dataset contains the 1010228 items. We have tested on four cases:
(1) Random with repeated data (Random data)
(2) Reverse sorted with repeated Data (Reverse sorted data)
(3) Sorted with repeated data (Sorted data)
(4) Nearly sorted with repeated data (Nearly sorted data)
Table I shows the execution time in microseconds of library sort algorithm using the dataset.

By analyzing the table I, we can see that when we increase the gap value between the elements the execution time will decrease. The following figures show this effect. In all the figures X-axis represents the increasing value of the gap and the Y-axis shows the time in microseconds.

TABLE I. Execution Time of Library Sort in Microsec Based on Gap Value

LibrarySort Algorithm	Time in Microseconds			
Value of ε	Random	Nearly Sorted	Reverse Sorted	Sorted
ε =1	981267433	864558882	1450636163	861929937
ε =2	729981576	620115904	1065247938	609647355
ε =3	119727535	358670053	278810310	356489846
ε =4	23003046	117188830	263693774	116590140

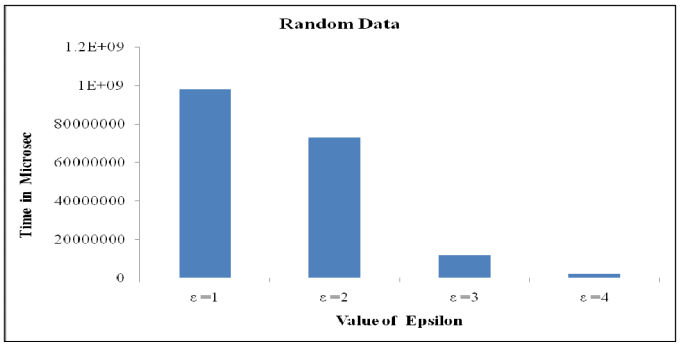

Fig. 6. Graph shows the execution time of random data using value of gaps

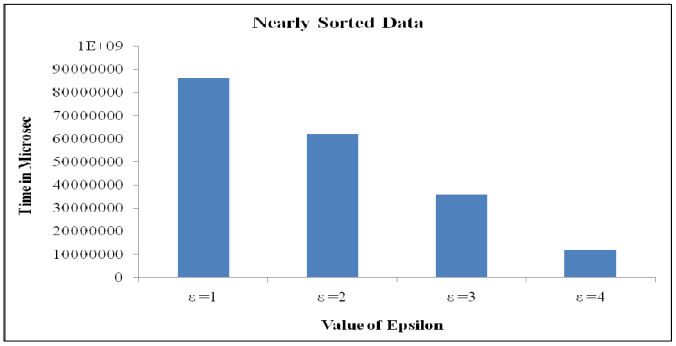

Fig. 7. Graph shows the execution time of nearly sorted data using value of gaps

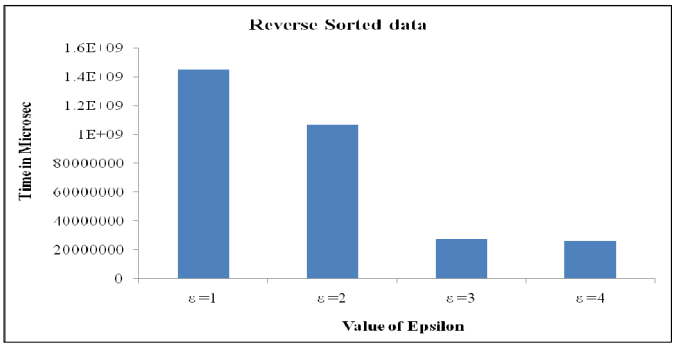

Fig. 8. Graph shows the execution time of reverse sorted data using value of gaps

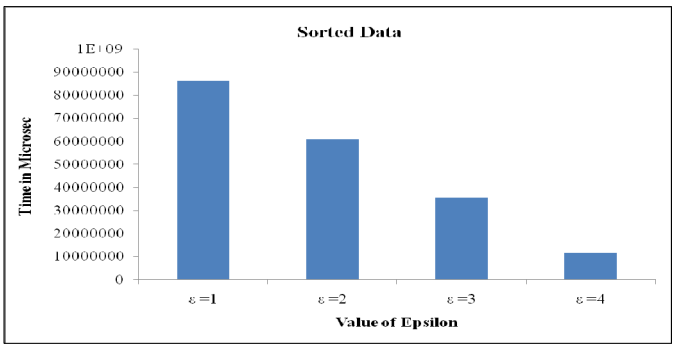

Fig. 9. Graph shows the execution time of sorted data using value of gaps

We have plotted figure 1, 2, 3, 4 by using table I. By examining these figures, we can see that how the execution

time is decreasing when the gap value between items is increasing. In figure 1-4, we are representing the execution time in microseconds in all the four cases of dataset.

The value of epsilon: when we increase the value of epsilon, the execution time will decrease, but at some point, value of epsilon gets saturated point because we are allocating more gaps, but these gaps are more than are actually required for the operation, so it will only be an extra memory overhead because we need more memory to store the elements. So in this way the space complexity of the algorithm increases linearly, when we increase the value of epsilon. The concept of space complexity will be explained in the next section with the help of graph.

IV. Space complexity Testing of Library sort on a Dataset

Auxiliary space complexity of library sort is O(n), but the space complexity is not only limited to auxiliary space. It is the total space taken by the program which includes the following [11].

(1) Primary memory required to store input data (M_{ip}).
(2) Secondary memory required to store input data (M_{is})
(3) Primary memory required to store output data (M_{op}).
(4) Secondary memory required to store output data (M_{os})
(5) Memory required for holding the code (M_c)
(6) Memory required for working space (temporary memory) variables + stack (M_w)
Table II shows the detail of total space complexity taken by the library sort algorithm on a dataset using gap values and re-balancing factor.

TABLE II. Total Space complexity in bytes of Library sort with Increasing Value of Gap and Re-balancing Factor

Reba-lancing	ε	M_{ip}	M_{is}	M_{op}	M_{os}	M_c	M_w	Total
2	1	4040932	4932283	4	4932283	81,920	16163752	30151174
	2	4040932	4932283	4	4932283	81,920	24245576	38232998
	3	4040932	4932283	4	4932283	81,920	32327400	46314822
	4	4040932	4932283	4	4932283	81,920	40409224	54396646
3	1	4040932	4932283	4	4932283	81,920	16163752	30151174
	2	4040932	4932283	4	4932283	81,920	24245576	38232998
	3	4040932	4932283	4	4932283	81,920	32327400	46314822
	4	4040932	4932283	4	4932283	81,920	40409224	54396646
4	1	4040932	4932283	4	4932283	81,920	16163752	30151174
	2	4040932	4932283	4	4932283	81,920	24245576	38232998
	3	4040932	4932283	4	4932283	81,920	32327400	46314822
	4	4040932	4932283	4	4932283	81,920	40409224	54396646

In table II, we have seen the total space complexity taken by the library sort using the dataset. From table II, we can see that there is no effect of re-balancing factor, but there is an effect of epsilon values. When we increase the gap value, the space taken by the program will also increase. We can see this effect with the help of graph shown in figure 5. In figure 5, the X-axis represents the value of epsilon and the Y-axis represents the memory occupied by the library sort algorithm in bytes. We can see that space complexity of the library sort algorithm increases linearly, when we increase the value of epsilon or gaps between the elements. It increases because we require more memory to store the elements and it is directly proportional to the value of epsilon. Due to this fact, the memory required is directly proportional to the value of epsilon, where epsilon is $(1+\varepsilon)n$.

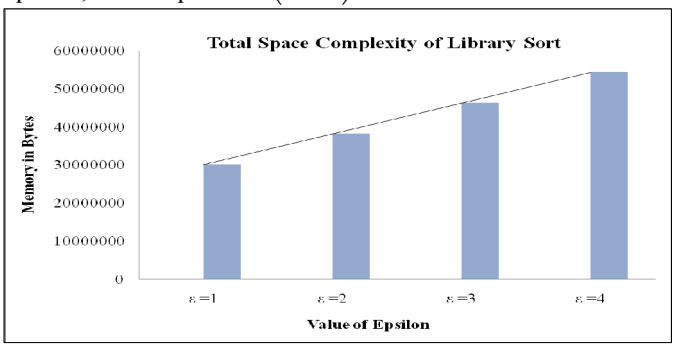

Fig. 10. Graph showing the space complexity of library sort

V. Re-balancing Testing of Library sort on a Dataset

As the rebalancing is done after inserting a^i elements, this increases the size of the array. The size of the array will depend on 'ε'. To do this process, we will require an auxiliary array of the same size so as to make a duplicate copy with gaps. Re-balancing is necessary after inserting a^i elements, but it also accounts the cost and time of library sort algorithm so, what will be the suitable value for 'a' is the question. We have calculated re-balancing till a^i where 'a'= 2, 3, 4 and i = 0, 1, 2, 3, 4….. With the value of gaps 'ε' = 1,2,3,4.

(A) For example, when ε=1, then how re-balancing will be performed if a=2.

$2^i = 2^0, 2^1, 2^2, 2^3, 2^4$ ……
=1, 2, 4, 8, 16 ……

1. Re-balance for 2^0 =1

1	-1

1. Re-balance for 2^1=2

1	2

After re-balancing this array will be-

1	-1	2	-1

2. Re-balance for 2^2=4

1	2	3	4

After re-balancing this array will be

1	-1	2	-1	3	-1	4	-1

So in this manner we can re-balance the array in the power of 2^i.

(B) Example when ε=1, then how re-balancing will be performed if a=3.

$3^i = 3^0, 3^1, 3^2, 3^3, 3^4 \ldots\ldots\ldots\ldots$

$= 1, 3, 9, 27\ldots\ldots\ldots\ldots\ldots$

1. Re-balance for $3^0 = 1$

1	-1

1. Re-balance for $3^1 = 3$

In the above array only one space is empty, so we can insert only one element so the total element will be two, but according to re-balancing factor we require three elements so in this situation we have to shift the data to insert the new elements. So in this way performance of algorithm degrades as we are having the larger number of swapping to generate the spaces which is same as that in the case of traditional insertion sort. So finally we have found that when we increase the re-balancing factor 'a' from 2 to 4 then the execution time of library sort algorithm will also increase. We can see this effect with the help of table III and graphs described in figure 6 to figure 9.

TABLE III. TIME TAKEN BY LIBRARY SORT ALGORITHM IN MICROSECOND DURING RE-BALANCING

Rebalancing	Value of ε	Type of Dataset			
		Random	Nearly	Reverse	Sorted
2	1	981267433	864558882	1450636163	861929937
	2	729981576	620115904	1065247938	609647355
	3	119727535	358670053	278810310	356489846
	4	23003046	117188830	263693774	116590140
3	1	2622591059	2214715182	2832112301	3011802732
	2	2103580421	1964645906	2585747568	2651992181
	3	2043974421	1728175857	2195021514	1962122927
	4	1620914312	1600879365	2130261056	1620374625
4	1	2942693856	2467933298	3239333534	3281368964
	2	2705332601	2510103530	3154811065	2923182920
	3	2676681610	2613423098	3013676930	2378347887
	4	2611656774	2157740458	2993363707	2222906193

From table III, we can see that the execution time of library sort is increasing when the re-balancing factor will increase in all the cases of the dataset. The following graph shows this effect.

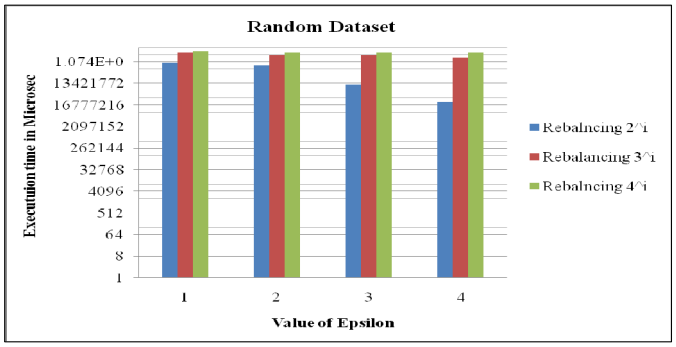

Fig. 11. Graph shows the re-balancing of library sort using random dataset

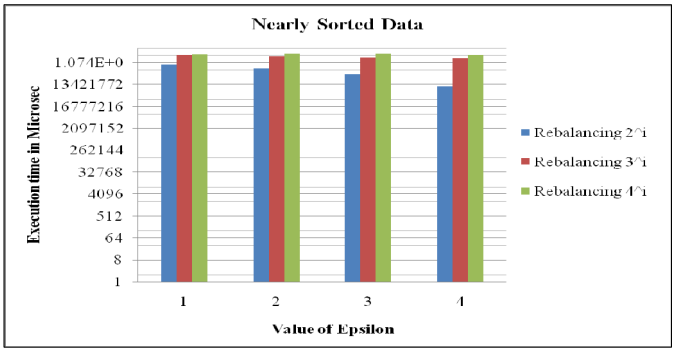

Fig. 12. Graph shows the re-balancing of library sort using nearly sorted dataset

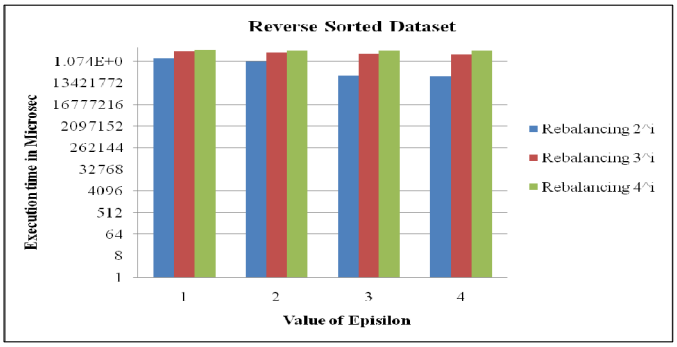

Fig. 13. Graph shows the rebalancing of library sort using reverse sorted dataset

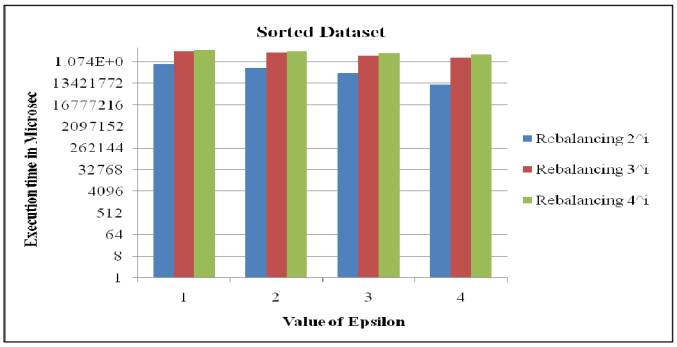

Fig. 14. Graph shows the re-balancing of library sort using sorted dataset

From figure 6 to 9, we can see that the execution time of library sort is increasing when the re-balancing factor is increasing using all the four cases of dataset.

From figure 6 to figure 9, the X-axis represents the value of epsilon and the Y-axis represents the execution time in microseconds when the re-balancing factor value is 2^i, 3^i, 4^i. By analyzing the figures, we can see that the nature of data marginally effected on the re-balancing factor. If the re-balancing factor is 2^i i.e. we have to re-balance the elements in the following manner 2^0, 2^1, ……. 2^n. Then the performance of the algorithm is good because in the array proper space is there to insert the new elements. But the performance of the algorithm is degraded if the re-balancing factor increases from 2^i to 4^i because if we use the re-balancing factor 3^i i.e. we

have to re-balance the elements in the following manner: 3^0, 3^1,…….3^n . Then in the array there is no proper space to insert the new elements in the manner of 3^i and 4^i. So shifting of data was required to insert the new elements and the spaces between many elements have been already consumed so this way performance degrades as we have a larger number of swapping to generate the spaces which is same as that in the case of traditional insertion sort.

VI. Analysis of Performance of Library Sort Algorithm

By execution time analysis, we have found that when we increase the value of epsilon then the execution time will decrease, but at some point the value of epsilon gets a saturated point because we will have the extra spaces, for the data to be inserted in between.

By space complexity analysis, we have found that the space complexity of the library sort algorithm increases linearly that is, when we increase the value of epsilon the memory consumption is also increasing in the same proportion.

By execution time analysis of re-balancing, we have found that when we increase the re-balancing factor 'a' from 2 to 4 then the execution time of library sort algorithm will also increase as it moves towards the traditional insertion sort.

So to find out the best result of library sort algorithm, the value of epsilon should be optimized and re-balancing factor should be minimized or ideally equal to 2.

VII. Conclusion

In this paper, we have tested the library sort algorithm on a dataset. There are four cases of the dataset and every case of the dataset contains the 1010228 items. We have applied the library sort algorithm in the four cases of the dataset and compare the performance in each case. And also we have found out the total space complexity taken by library sort algorithm. We also found out how the value of epsilon and re-balancing factor is affecting the execution time of library sort algorithm, and we should keep the value of epsilon and re-balancing factor (a) as optimal as possible and minimum or ideally equal to 2. Library sort algorithm is implemented in C-language. The program of library sort is designed at Borland C++ 5.02 compiler and executed on the Intel® core™ i5 processor-3230 M CPU @ 2.60 GHz machine, and the programs running at 2.2 GHz clock speed.

VIII. Future Work

In this paper, we have used the uniform gap distribution after each element. For further improvement of library sort algorithm, we can use the non-uniform gap distribution after each element. This can be done parallel using GPU distribution.

Acknowledgment

We have done our code of library sort algorithm in C language and the datasets is used which is available at frequent item-set mining implementation repository**.** This work is performed in the frame of a research concerted action. All experimental results are done in the research lab of Jaypee University of Information Technology, Waknaghat Solan, India.

References

[1] Bender, Michael A., Martin Farach-Colton, and Miguel A. Mosteiro. "Insertion sort is O (n log n)". *Theory of Computing Systems,* 39.3 (2006), pp. 391-397.

[2] Cormen, Thomas H., et al. "Introduction to algorithms". *Cambridge: MIT press,* Vol. 2, 2001.

[3] Estivill-Castro, Vladmir, and Derick Wood. "A survey of adaptive sorting algorithms". *ACM Computing Surveys (CSUR)*, 24.4 (1992), pp. 441-476.

[4] Pardo, Luis Trabb. "Stable sorting and merging with optimal space and time bounds". *SIAM Journal on Computing,* 6.2 (1977), pp. 351-372.

[5] Frequent Itemset Mining Implementations Repository,http://fimi.cs.helsinki.fi accessed on 10/11/2014

[6] Bender, Michael A., et al. "A locality-preserving cache-oblivious dynamic dictionary". *Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms*, 2002, pp. 1-22.

[7] Brodal, Gerth Stølting, Rolf Fagerberg, and Riko Jacob. "Cache oblivious search trees via binary trees of small height". *Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms*, 2002, pp. 1-20.

[8] Itai, Alon, Alan G. Konheim, and Michael Rodeh. "A sparse table implementation of priority queues". *Springer Berlin Heidelberg*, 1981, pp. 417-431.

[9] Thomas, Nathan, et al. "A framework for adaptive algorithm selection in STAPL". *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming ACM*, 2005, pp. 277-288.

[10] Janko, Wolfgang. "A list insertion sort for keys with arbitrary key distribution". *ACM Transactions on Mathematical Software (TOMS,* 2.2 (1976), pp. 143-153.

[11] Faujdar Neetu, and Satya Prakash Ghrera. "Analysis and Testing of Sorting Algorithms on a Standard Dataset".*CommunicationSystems and Network Technologies (CSNT), Fifth International Conference on. IEEE*, 2015, pp. 962-967 .

[12] Zubair Khan, Neetu Faujdar, et al. "Modified BitApriori Algorithm: An Intelligent Approach for Mining Frequent Item-Set". *Proc. Of Int. Conf. on Advance in Signal Processing and Communication,* 2013, pp. 813-819.

[13] Faujdar Neetu, and Satya Prakash Ghrera. "Performance Evaluation of Merge and Quick Sort using GPU Computing with CUDA". *International Journal of Applied Engineering Research,* 10.18(2015), pp. 39315-39319.