Pincer-search algorithm [10], proposes a new approach for mining maximal frequent itemset which combines both bottom-up and top-down searches to identify frequent itemsets effectively. It classifies the data source into three classes as frequent, infrequent, and unclassified data.

FP-growth is one of the algorithms which is based upon the recursively divide and conquer strategy. FP-growth is proofed to be efficient and is widely used and applied. RElim, however, is proposed to be the algorithm of choice if implemented properly [4].Moreover, RElim was inspired by H-Mine which is similar to FP-growth.

In this paper we compare FP-growth with RElim algorithm in term of time efficiency. We also use multithreading technique to enhance the time efficiency of FP-Growth algorithm. A comparison in terms of execution time were carried out between FP-growth and the enhanced FP-growth.

The rest of this paper is organized as follows: Section 2 discusses state of the art and reviews some related works. Section 3 explains the theoretical background needed. Section 4 describes our proposed multithreaded FP-growth. Section 5 shows experimental setup. Section 6 presents results and discussion. Finally Section 7 concludes the paper.

2. Related Work

Vina et al [1] provided a comparison between H-mine, Fp-Growth and SaM. A framework has developed to allow the flexible comparison of the algorithms. They measured time complexity and came to that the execution time of all the discussed algorithms is nearby but it can also be noticed that the execution time of SaM is comparatively less for higher support threshold.

Christian [4] presented a paper on Recursive Elimination algorithm. He proposed that if a quick and straightforward implementation is desired, it could be the method of choice. Even though its underlying scheme which is based on deleting items, recursive processing, and reassigning transactions is very simple and works without complicated data structures, recursive elimination performs surprisingly well.

Jochen Hipp et al. [11] provided several efficient algorithms that convoy with the popular and computationally expensive task of association rule mining with a comparison of these algorithms concerning efficiency. He proposed that the algorithms show quite similar runtime behavior in their experiments.

Aggarwal and Srikant [12] presented two new versions of Apriori, AprioriT and AprioriTID, for discovering all significant association rules between items in a large database of transactions and compared these algorithms to the previously known algorithms, the AIS and SETM algorithms. They proposed that these algorithms always outperform AIS and SETM.

Borgelt [13] provided efficient implementation of the more sophisticated approaches known under the names of Apriori and Eclat. Both rely on a top down search in the subset lattice of the items. He proposed for free item sets Eclat wins the competition with respect to execution time and it always wins with respect to memory usage. The data set in which it takes lead is for the lowest minimum support value tested, indicating that for lower minimum support values it is the method of choice, while for higher minimum support values its disadvantage is almost negligible. For closed item sets the more efficient filtering gives Apriori a clear edge with respect to execution time. For maximal item sets the picture is less clear. If the number of maximal item sets is high, Apriori wins due to its more efficient filtering, while Eclatwins for a lower number of maximal item sets due to itsmore efficient search.

Győrödi and Holban [14] had performed an experimental comparison between Apriori, DynFP-Growth, FP-growth, the algorithms were implemented in Javaand tested on several data sets. They stated that FP-growth version out performed Apriori in all cases, and Apriori has the most memory consumption. On the other hand, the frequent database scans gave Apriori the maximum number of generated itemsets.

Shankar and Purusothaman [15] presented a comparative study of various methods in existence for frequent itemset mining, association rule mining with utility considerations. THUI (Temporal High Utility Itemsets)-Mine, heap mine (H-mine), and DSM-FI (Data Stream Mining for Frequent Itemsets) algorithms have been evaluated based on their memory usage for mining the frequent itemsets and association rules from large databases.

Vani [16] has conducted a Comparative Analysis of Association Rule Mining Algorithms Based on performance Survey between FP-growth and Eclat, as the fastest algorithms on the survey, and he concluded that their performance varies according to the data set used. In this paper, we compare a java implementation of Recursive Elimination and FP-Growth in term of execution time. We use two different datasets with different numbers of records and attributes, comparing their performance at low and high minimum supports.

High-performance parallel and distributed computing is becoming increasingly important as data keep growing in size and becoming complicated. Works have been done to parallelize the mining process such as shared memory systems. Rathi et al. [17] proposed a model that implements a parallel FP Growth algorithm that makes use of multiple Graphic processing (GPU) system, the proposed algorithm improves performance of the

algorithm. Wang and Wang [18] designed a parallel algorithm that works on distributed data framework, their algorithm does not need to create the whole FP tree, so it can handle huge data.

Frontier Expansion [19] is a new parallel Frequent Itemset Mining algorithm, its implementation can achieve good performance in heterogeneous platforms with shared memory multiprocessor and multiple Graphic Processing units and speedup 6-30 times sequential Eclat. Accelerating Parallel Frequent Itemset Mining on Graphic Processors with Sorting APFMS [20], is an algorithm that utilizes new generation GPUs to accelerate the mining process on openCL platform, results showed reduction in computation time.

Our work, however, is different in a way that it does not require any special prepared platforms nor hardware equipment such as GPUs. It can work on any computer that supports threading, and all computers do these days. By this we can accelerate the process of mining frequent patterns multiple times regardless of the hardware it runs on.

3. Theoretical Background

3.1. FP-growth

FP-tree algorithm is based upon the recursively divide and conquers strategy; first the set of frequent 1-itemset and their counts is discovered. With start from each frequent pattern, construct the conditional pattern base, then its conditional FP-tree is constructed (which is a prefix tree.). Until the resulting FP-tree is empty, or contains only one single path. (Single path will generate all the combinations of its sub-paths, each of which is a frequent pattern). The items in each transaction are processed in L order (i.e. items in the set were sorted based on their frequencies in the descending order to form a list) [3]. the detail step is as follows:

FP-Growth Method: Construction of FP-tree. Create root of the tree as a “null”. After scanning the database D for finding the 1-itemset then process the each transaction in decreasing order of their frequency. A new branch is created for each transaction with the corresponding support. If same node is encountered in another transaction, just increment the support count by 1 of the common node. Each item points to the occurrence in the tree using the chain of node-link by maintaining the header table.

After the above process mining of the FP-tree will be done by Creating Conditional (sub) pattern bases: Start from node constructs its conditional pattern base. Then, Construct its conditional FP-tree and perform mining on such a tree. Join the suffix patterns with a frequent pattern generated from a conditional FP-tree for achieving FP-growth. The union of all frequent patterns found by above step gives the required frequent itemset. In this way frequent patterns are mined from the database using FP-tree.

Algorithm 1 (FP-tree construction)
Input: A transaction database DB and a minimum support threshold.
Output: Its frequent pattern tree, FP-tree Method: The FP-tree is constructed in the following steps.
1. Scan the transaction database DB once. Collect the set of frequent items F and their supports. Sort F in support descending order as L, the list of frequent items.
2. Create a root of an FP-tree, T, and label it as \null". For each transaction Trans in DB do the following.
Select and sort the frequent items in Trans according to the order of L. Let the sorted frequent item list in Trans be [p | P], where p is the first element and P is the remaining list. Call insert tree([p | P]; T).
The function insert tree([p | P]; T) is performed as follows. If T has a child N such that:
N.item-name = p.item-name, then increment N's count by 1; else create a new node N, and let its count be 1, its parent link be linked to T, and its node-link be linked to the nodes with the same item-name via the node-link structure. If P is nonempty, call insert tree(P;N) recursively.

Algorithm2 (FP-growth: Mining frequent patterns with FP-tree and by pattern fragment growth)
Input: FP-tree constructed based on Algorithm 1, using DB and a minimum support threshold.
Output: The complete set of frequent patterns.
Method: Call FP-growth (FP-tree; null), which is implemented as follows.
Procedure FP-growth (Tree; α)
{
(1) IF Tree contains a single path P
(2) THEN FOR EACH combination (denoted as β) of the nodes in the path P DO
(3) generate pattern βUα with support =minimum support of nodes in β;
(4) ELSE FOR EACH ai in header of Tree DO {
(5) generate pattern β = aiUα with support =ai.support;
(6) Construct β's conditional pattern base and then β's conditional FP-tree Treeβ;
(7)IF Treeβ≠∅
(8)THEN Call FP-growth (Treeβ; β) }
}

3.2. Recursion elimination

In a pre-processing step delete all items from the transactions that are not frequent individually, i.e., do

not appear in a user-specified minimum number of transactions. This pre-processing is demonstrated in Figure 1, which shows an example transaction database on the left. The frequencies of the items in this database, sorted in an ascending order, are shown in the middle. If we are given a user specified minimal support of 3 transactions, items f and g can be discarded. After doing so and sorting the items in each transaction in an ascending order by their frequencies we obtain the reduced database shown on the right of Figure 1.

Then select all transactions that contain the least frequent item (least frequent among those that are frequent), delete this item from them, and recurse to process the obtained reduced database, remembering that the item sets found in the recursion share the item as a prefix.

On return, remove the processed item also from the database of all transactions and start over, i.e., process the second frequent item etc.

a d f	g 1	a d
c d e	f 2	e c d
b d	e 3	b d
a b c d	a 4	a c b d
b c	c 5	c b
a b d	b 7	a b d
b d e	d 8	e b d
b c e g		e c b
c d f		c d
a b d		a b d

Figure 1. Transaction database (left), item frequencies (middle), and reduced transaction database with items in transactions sorted in an ascending order by their frequency (right) [21]

This process is illustrated for the root level of the recursion, which shows the transaction list representation of the initial database at the very top, see Figure 2.

In the first step all item sets containing the item e are found by processing the leftmost list. The elements of this list are reassigned to the lists to the right (grey list elements) and copies are inserted into a second list array (shown on the right). This second list array is then processed recursively, before proceeding to the next list, i.e., the one for item a.

In these processing steps the prefix tree (or the H-struct), which is enhanced by links between the branches, is exploited to quickly find the transactions containing a given item and also to remove this item from the transactions after it has been processed.

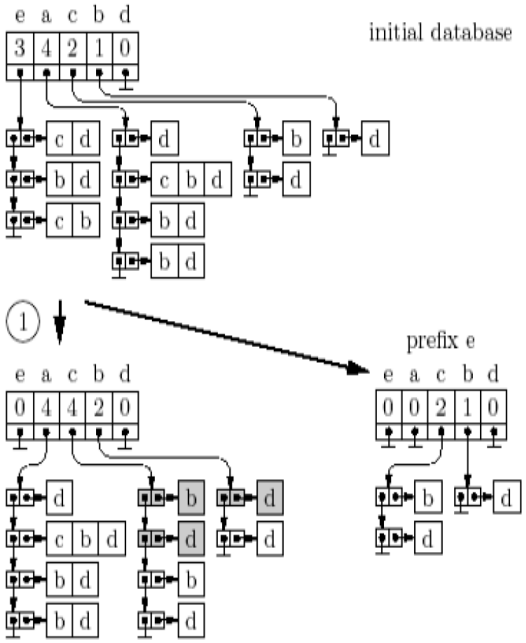

Figure 2. Procedure of the recursive elimination with the modification of the transaction lists (left) as well as the construction of the transaction lists for the recursion (right) [21].

4. Proposed Multi-threaded FP-growth

Every path in the FP-tree keeps track of an itemset along with its support [4]. And it's known that each starting node generate it's related itemsets; for example consider the FP-growth tree in Figure3., taking item 4 and 3 as an example it exists in three branches, below is a list of its related items along it's tree branch path.

1. Item 4
   - Branch1: {3,2,1}, supp 5
   - Branch2: {3, 1} , supp 1
   - Branch3: {1} , supp 3
2. Item 3
   - Branch1: {2,1}, supp 5
   - Branch2: {1}, supp 2

So the candidate item sets of 4 are as follows:

- {3,2,1} Sup 5, {3,1} Sup 6, {1} Sup 9

and the candidate item sets of 3 are as follows:

- {2,1} Supp 5, {1} Supp 7

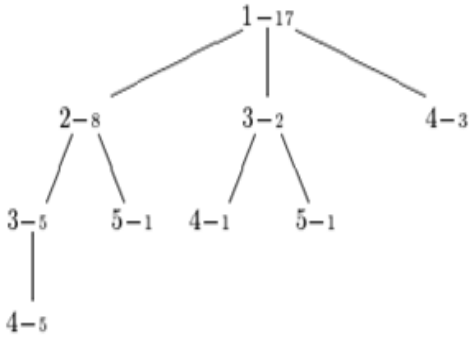

Figure 3. FP-tree

The generated itemsets are concatenated with its support using (:) are

- *Itemsets of 4 are as follows:*

(4):9, (4,1):9, (4,2):5, (4,3):6, (4,1,2):5, (4,1,3):6, (4,2,3):5, (4,1,2,3):5.

- *Itemsets of 3 are as follows*:

(3):7,(3,1):7,(3,2):5, (3,2,1):5.

As we see from the previous example each item generates its own frequent itemsets, for example we can generate the itemset for item 4 and item 3 concurrently and then aggregate the results generated by both items in one file. So our contribution is to create a thread to gather the frequent itemsets of an item and after that combine the result of the finished thread in one single file.

The abstract code of our proposed threaded FP-growth can be found in the appendix.

5. Experiment

A. *System Information***:**

1) *Operating System*: windows server 2008, 64 bit
2) *Memory*: 8192 MB Ram
3) *Processor*: Intel® xeon® CPU ES-26200, 2.00GHz (4 CPUs).

B. *Implementation*:

All the algorithms to be tested have been implemented in Java using eclipse 2015, with jdk8.0

C. *Datasets*:

Two integer datasets were used in the comparison operation between Relim, FP-growth, and the proposed multithreaded FP-growth algorithms, we used integer data sets to be simple to deal with. These datasets contain different number of transactions and attributes.

1) *Mushroom Dataset* [22]**:** it contains 8124 instances, this data set includes descriptions of hypothetical samples corresponding to 22 species of gilled mushrooms in the Agaricus and Lepiota Family.
2) *Connect Dataset* [23]: it contains 67557 instance, 42 attributes. This database contains all legal 8-ply positions in the game of connect-4 in which neither player has won yet, and in which the next move is not forced.

D. *Optimization Issues:*

Finding all frequent itemsets in a database is difficult since it involves searching all possible itemsets (item combinations). The set of possible itemsets is the power set over *I*and has size 2^n-1 (excluding the empty set which is not a valid itemset).

In this paper we assume that by using the multithreading technique we may enhance the execution time of finding the frequent item set in test datasets.

Multithreading [24] is a programming technique for implementing applicationconcurrency and, therefore, also a way to exploit the parallelism of shared memory multi-processors. A traditional "single threaded" process could be seen as a single flow of control (thread) associated one to one with a program counter, a stack to keep track of local variables, an address space and a set of resources. Multithreading programming allows one program to execute multiple tasks concurrently, by dividing it into multiple threads, so we will divide the operation on finding the frequent item sets to multiple threads, 4 threads precisely, one for each CPU, to benefit from the concurrency execution to minimize the time.

E. *Experiment:*

1. The first experiment has been performed to compare performance of FP-growth with RElim using two different datasets.

2. The second experiment is has been performed to see how much enhancement does multithreading add to FP-growth compared to single threaded version. Multithreaded FP-growth has been discussed in section V.

6. Results and Discussion

1. Results of the first experiment show the superiority of FP-growth over RElim on mushroom dataset especially when the minimum support (minsupp) is low as shown in Figure 4. FP-growth outperformed RElim in all cases but the difference is dramatically reduced by the increase of the minimum support. We believe that depends on the nature of the dataset transactions, noticing that the number of frequent itemsets has reduced from hundreds of thousands to a few

thousands with a slight increase in the minsupp. This could be justified mentioning that RElim may perform better if a programming language that supports pointer handling is considered for implementation.

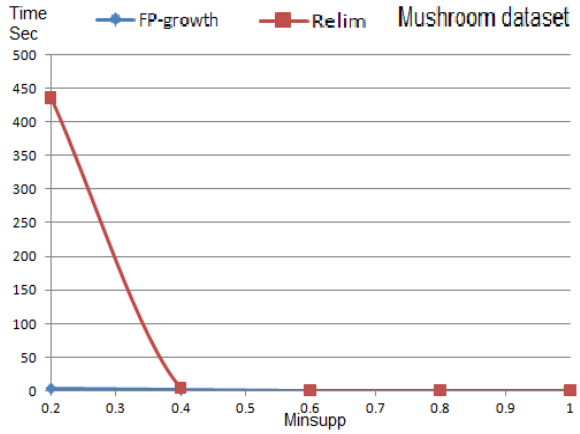

Figure 4. FP-Growth & RElim on Mushroom dataset

Also the superiority of FP-growth is clear in the larger dataset especially when the min support is low, see Figure 5.

These results are consistent with the suggestion of Cristian [21] that if a quick and straightforward implementation is applied it may improve the performance of the algorithm. We believe that the huge difference of execution time between the two algorithms refer to the mechanism of java code optimization, and especially the garbage collection mechanism.

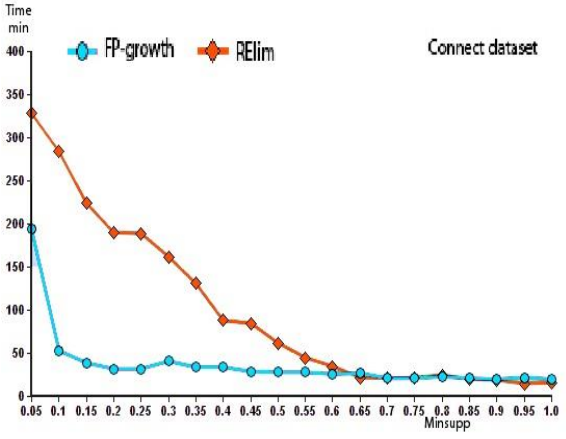

Figure 5. FP-Growth & RElim on connect dataset

2. The second experiment proofs that the multithreaded FP-growth is much faster than FP-growth itself. Multithreaded FP-growth consumed around quarter the time consumed by FP-growth on both data sets, see Figure 6.and Figure 7. We believe that the result is proportional to the number of threads, as we used 4 threads, one for each CPU. These results came consistent to other findings in [17], [19], [20]. Despite using GPUs as special hardware, they all parallelized the work as we did, but we used multithreading instead. And all the results showed improvements in the time execution of the algorithms.

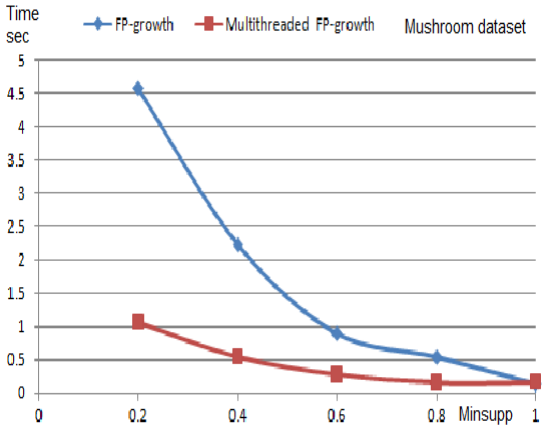

Figure 6. FP-growth & Multithread FP-growth on mushroom dataset

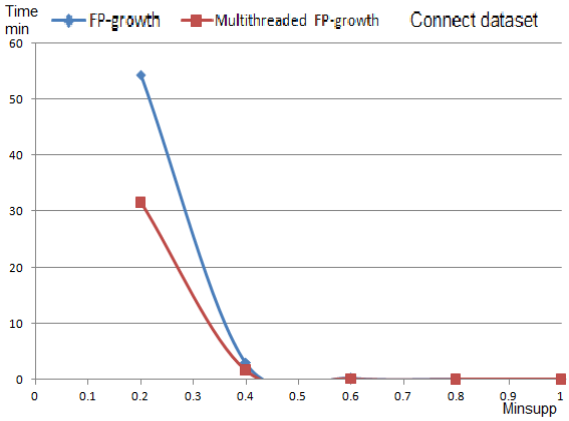

Figure 7. FP-growth & Multithread FP-growth on connect dataset

7. Conclusion and future work

This paper presented a performance comparison between two frequent pattern extraction algorithms RElim and FP-Growth which implemented in Java. These two algorithms are used in finding frequent itemsets in the transaction database. We found out by experiment that FP-growth outperformed RElim in term of execution time. In this context, multithreading technique is used to enhance the time efficiency of FP-Growth algorithm in the part of generating frequent itemsets. The results showed that multithreaded FP-growth is more efficient compared to single thread FP-growth, this result comes clear at low minimum support thresholds

In our experiment, it was difficult to accurately measure the space required by the algorithms because Java garbage collection mechanism make results about space unexplainable. So, in future work

we will try the two algorithms on another platform and compare the space complexity.

8. Appendix

1. The abstract code of our proposed FP-growth threading approach

```
/*************************
-transactionCount is the total number of transactions
-mapSupport is hashmap of support of each item
-relativeMinsupp is the relative minimum support
-freqItemSet is ConcurrentHashMap
which is a centralized container of the generated frequent itemsets
-item is the item which we need to generate its related itemsets.
-threads is a container of the running threads
*************************/
int j=0; //counter of started threads
intThreadNum=4; // number of threads
FPGrowthTree tree = Constructs_FP_growth_tree();
for(inti=0;i<itemListSize;i++){
if(j<ThreadNum){
item = tree.headerList.get(i);
CandidateGenerator c = new CandidateGenerator(tree,transactionCount, mapSupport, relativeMinsupp, freqItemSet,item);
threads.add(c);
c.start();
j++;
}
if(j==ThreadNum || i==itemListSize){
boolean exit=false;
while(!exit){
    // sleep just 10 millisecond
    //to check the finished threads
Thread.sleep(10);
    Iterator Itr = threads.iterator();
while (Itr.hasNext()) {
       CandidateGenerator c = Itr.next();
       if(c.finished){
         // a thread is finished, remove it from
         // the threads list
       Itr.remove();
         //change exit flag to exit while loop
         // to start a new thread.
       exit=true;
        // decrease the number of running threads
        // to start a new one
       j--;
        } //if condition for check finished thread
      } // loop over the threads list
    } // loop to monitor the finished thread
} // if condition to start threads monitoring
} // main item loop
```

9. Acknowledgment

I would like to express my deep sense of gratitude and sincere thanks to my friend and colleague Ramzi A. Matar. His constructive and insightful comments, suggestions and help have improved the quality of this work.

10. References

[1] Soni, V., Shah, N., Prajapati, S., Damor, N., Chaudhari, N., Patel, U., Patel, A., Chaudhari V., &Prajapati, A. (2013, March) A Study of Various Projected Data based Pattern Mining Algorithms. Research Journal of Material Sciences. Vol. 1(2), (PP. 1-5).

[2] Goethals, B. (2003). Survey on frequent pattern mining. Univ. of Helsinki.

[3] Han, J., Pei, J., & Yin, Y. (2000, May). Mining frequent patterns without candidate generation. In ACM SIGMOD Record (Vol. 29, No. 2, pp. 1-12). ACM.

[4] Borgelt, C. (2005, August). An Implementation of the FP-growth Algorithm. In Proceedings of the 1st international workshop on open source data mining: frequent pattern mining implementations (pp. 1-5). ACM.

[5] Agarwal, R. C., Aggarwal, C. C., & Prasad, V. V. V. (2000, August). Depth first generation of long patterns. In Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining (pp. 108-118). ACM.

[6] Pei, J., Han, J., Lu, H., Nishio, S., Tang, S., & Yang, D. (2001). H-mine: Hyper-structure mining of frequent patterns in large databases. In Data Mining, 2001. ICDM 2001, Proceedings IEEE International Conference on (pp. 441-448). IEEE.

[7] Liu, G., Lu, H., Yu, J. X., Wang, W., & Xiao, X. (2003, November). AFOPT: An Efficient Implementation of Pattern Growth Approach. In FIMI.

[8] Grahne, G., & Zhu, J. (2003, November). Efficiently Using Prefix-trees in Mining Frequent Itemsets. In FIMI (Vol. 90).

[9] Gouda, K., &Zaki, M. J. (2005). Genmax: An efficient algorithm for mining maximal frequent itemsets. Data Mining and Knowledge Discovery, 11(3), 223-242.

[10] Lin, D. I., &Kedem, Z. M. (1998). Pincer-search: A new algorithm for discovering the maximum frequent set. In Advances in Database Technology—EDBT'98 (pp. 103-119). Springer Berlin Heidelberg.

[11] Hipp, J., Güntzer, U., &Nakhaeizadeh, G. (2000). Algorithms for association rule mining—a general survey and comparison. ACM sigkdd explorations newsletter, 2(1), 58-64.

[12] Aggarwal, S., & Kaur, R. (2013). Comparative Study of Various Improved Versions of Apriori Algorithm. International Journal of Engineering Trends and Technology (IJETT)-Volume4Issue4-April.

[13] Borgelt, C. (2003, November). Efficient implementations of apriori and eclat. In FIMI'03: Proceedings of the IEEE ICDM workshop on frequent itemset mining implementations.

[14] Győrödi, C., Győrödi, R., &Holban, S. (2004). A comparative study of association rules mining algorithms. In SACI 2004, 1st Romanian-Hungarian Joint Symposium on Applied Computational Intelligence (pp. 213-222).

[15] Shankar, S., &Purusothaman, T. (2009). Utility sentient frequent itemset mining and association rule mining: a literature survey and comparative study. International Journal of Soft Computing Applications, 4, 81-95.

[16] Vani, k., (2015). Comparative Analysis of Association Rule Mining Algorithms Based on Performance Survey.International Journal of Computer Science and Information Technologies6 (4).

[17] Rathi, S., &Dhote, C. A. (2015). Parallel Implementation of FP Growth Algorithm on XML Data Using Multiple GPU. In Information Systems Design and Intelligent Applications (pp. 581-589). Springer India.

[18] Wang, Z., Wang, C.: A parallel association-rule mining algorithm. In: WISM'12 Proceedings of the 2012 International Conference on Web Information Systems and Mining, pp. 125–129. Springer, Berlin (2012)

[19] Zhang, F., Zhang, Y., &Bakos, J. D. (2013). Accelerating frequent itemset mining on graphics processing units. The Journal of Supercomputing, 66(1), 94-117.

[20] Huang, Y. S., Yu, K. M., Zhou, L. W., Hsu, C. H., & Liu, S. H. (2013). Accelerating Parallel Frequent Itemset Mining on Graphics Processors with Sorting. In Network and Parallel Computing (pp. 245-256). Springer Berlin Heidelberg.

[21] Borgelt, C. (2005, August). Keeping things simple: Finding frequent item sets by recursive elimination. In Proceedings of the 1st international workshop on open source data mining: frequent pattern mining implementations (pp. 66-70). ACM.

[22] Mushroom dataset: Frequent Itemset Mining Implementations Repository. Site: http://fimi.ua.ac.be. Found at: http://fimi.ua.ac.be/data/mushroom.dat (Accessed 17 December 2015).

[23] Connect dataset: Frequent Itemset Mining Implementations Repository. Site: http://fimi.ua.ac.be. Found at: http://fimi.ua.ac.be/data/connect.dat. (Accessed 17 December 2015).

[24] Negri, A., Scannicchio, D. A., Touchard, F., &Vercesi, V. (2001). Multi thread programming.

