Frequent Itemset Mining on Large-Scale Shared Memory Machines

Yan Zhang, Fan Zhang, Jason Bakos

Dept. of CSE, University of South Carolina
315 Main St. Columbia SC 29201 USA
{zhangy,zhangf}@email.sc.edu, jbakos@cse.sc.edu

Abstract— **Frequent Itemset Mining (FIM) is a data mining task that is used to find frequently-occurring subsets amongst a database of itemsets. FIM is a non-numerical data intensive computation and is frequently used in machine learning and computational biology applications. The development of increasingly efficient FIM algorithms is an active field, but exposing and exploiting parallelism is not often emphasized in the development of new FIM algorithms. In this paper, we explore parallel implementations of two FIM algorithms, Apriori and Eclat, each using three different representations: vertical transaction id set, vertical bitvector, and diffset. We implemented these algorithms using OpenMP and evaluated their resultant scalability on the 4096-core Intel Nehalem-EX SGI Altix shared-memory machine Teragrid "Blacklight" using 16 processors (one blade) to 256 processors (16 blades) and reported our results. We found that, while scalability generally depends on the input data, Apriori is only scalable when used with diffset. On the other side, Eclat is generally scalable but achieves its best scalability with diffset.**

Keywords-Frquent Itemset Mining; Apriori; Eclat; shared memory; parallel

I. INTRODUCTION

Frequent Itemset Mining (FIM) is a generalized computation for finding frequently-appearing subsets within a database of sets. Many scientific and industrial applications including those in machine learning, computational biology, intrusion detection, web log mining, and e-business benefit from the use of frequent itemset mining.

The objective of FIM is to identify the item subsets that appear together in a transaction database when the number of occurrences exceeds a given threshold. Three popular algorithms for frequent itemset mining are Apriori, Eclat, and FPgrowth [3, 5]. Each algorithm has its own advantages and computational bottlenecks. Apriori is the oldest and simplest of these but generally performs well for most datasets. Eclat is different in that it adopts a depth-first approach to search the candidate space. Compared to Apriori, Eclat has less data dependence when computing candidate itemset support.

Based on serial algorithms, we describe parallel implementation of Apriori and Eclat. Each of these algorithms uses three different representations: *vertical tidset*, *bitvector*, and *diffset*. We find that not only does the performance of these algorithms rely on the choice of data representation, but the scalability of the the parallel implementation is also affected by it. Our implementation uses the OpenMP scheduler to achieve efficient parallelization of these algorithms. For both, we explore the performance and scalability using the massively parallel shared memory machine Teragrid Blacklight, scaling from 16 to 256 processors.

In this paper we perform a comprehensive evaluation of the scalability for two FIM algorithms, each coupled with three different dataset representaitons. Our results show that Eclat is generally more scalable than Apriori, and the diffset data representation is the most scalable among the dataset representations.

II. BACKGROUND

In a widely cited example, supermarkets employ frequent itemset mining to determine which products are often purchased together. A popular anecdote describes how one supermarket used FIM on their transaction database and was surprised to discover that customers who purchased diapers were likely to also purchase beer [7]. It was later discovered that this behavior was caused by husbands picking up beer after being sent out at night to purchase diapers. Discovering the relationship between purchased items can help supermarket management develop effective product promotions and decide which items should be placed together on the shelf. Similarly, online retailers can also use these relationships to recommend related products.

Using a supermarket metaphor, items represent individual items for sale. A *transaction*—or a "basket"--is analogous to a receipt, or a combination of items that were purchased together. A *dataset* is a set of transactions, and an *itemset* is a subset of the items that appear in the transaction database. An itemset of size k is called a *k-itemset*. FIM scans all the transactions and counts the appearance of k-itemsets within the dataset. The support of itemset X, or *support(X)* is the number of the transactions that contain itemset X. An itemset is frequent if its support is greater than a threshold value *min_sup*. The frequent itemset mining is to find all itemsets with support larger than *min_sup* in a given transaction database D. Generally we make assumption that all items in the itemset are sorted according to item number.

A. Candidate Representation

978-0-7695-4516-5/11 $26.00 © 2011 IEEE
DOI 10.1109/CLUSTER.2011.69

The "trie" data structure is most often used to represent candidate itemsets [1-4]. A trie is a rooted, directed tree that is used to generate and store the candidates for each generation. Each path from the root to each leaf represents a candidate. In order to make the code more efficient and amenable to the OpenMP execution model, we represent the trie using a table that stores the nodes associated with each level of the tree.

B. Transaction Representation

The original implementations of Apriori used a traditional horizontal representation for the transactions. In this case, each transaction is represented as a list of items. Storing transactions this way is inefficient and has thus given way to the vertical representation, in which each item is associated with a list of corresponding transactions that contain the item. Vertical representation generally offers one order of magnitude of performance gain since they reduce the volume of I/O operations and avoid repetitive database scanning.

As Figure 1 (a) shows, in the horizontal format items are stored in ordered way for each transaction. Figure 1 (b) shows the equivalent vertical format of the above horizontal format. This is also called the vertical *tidset* (transaction id set). In *tidset*, support counting can be performed by intersection operation. Let $t(X)$ denote the *tidset* of item X and $t(Y)$ denote the *tidset* of item Y. Let P be a prefix. According to definition, $t(PX) = t(P) \cap t(X)$, and $t(PY)=t(P) \cap t(Y)$. And thus, $t(PXY)=t(PX) \cap t(PY)$. $|t(PXY)|$, *i.e.* the size of $t(PXY)$, is the support of *PXY*.

In Figure 1(c), the equivalent vertical *bitvector* format is presented. In this format, the transactions associated with each item are represented as a bitmask. The binary bit is set to 1 if the corresponding transaction is present; otherwise it is set to 0. In this way, the length is fixed for all items. For dense transaction data, the transaction data size of vertical *bitvector* is substantially smaller than the vertical *tidset* format's. This is an advantage in terms of both running time and memory space.

Transaction ID				
1	A	B	C	
2	A	C	E	
3	A	E	F	G
4	B	C	E	G
5	D	F		
6	A	C	D	E

(a) Horizontal Transaction Format

Item	A	B	C	D	E	F	G
	1	1	1	5	2	3	3
	2	4	2	6	3	5	4
	3		4		4		
	6		6		6		

(b) Vertical Tidset Format

Trans. ID \ Item	A	B	C	D	E	F	G
1	1	1	1	0	0	0	0
2	1	0	1	0	1	0	0
3	1	0	0	0	1	1	1
4	0	1	1	0	1	0	1
5	0	0	0	1	0	1	0
6	1	0	1	1	1	0	0

(c) Vertical Binary (Bitvector)Format

Figure 1. Horizontal and Vertical dataset representation

Item(Support)	A(4)	B(2)	C(4)	D(2)	E(4)	F(2)	G(2)
	1	1	1	5	2	3	3
	2	4	2	6	3	5	4
	3		4		4		
	6		6		6		

(a) Vertical Tidset

Item(Support)	A(4)	~~B(2)~~	C(4)	~~D(2)~~	E(4)	~~F(2)~~	~~G(2)~~
	4	2	3	1	1	1	1
	5	3	5	2	5	2	2
		5		3		4	4
		6		4		6	6

(b) Diffset for 1 itemset, with threshold = 3

Item(Support)	AC(3)	AE(3)	CE(3)
	3	1	1

(c) Diffset for 2 itemset, with threshold = 3

Item(Support)	~~ACE(2)~~
	1

(d) Diffset for 2 itemset, with threshold =3

Figure 2. Diffset example

The third vertical data structure is *diffset*. Zaki and Gouda first introduced *diffset* to reduce the memory requirement of the vertical *tidset* representation [8]. It was originally introduced to improve the Eclat algorithm. In this paper, it is used with Apriori for the first time (as far as the authors know).

While *tidset* stores all the tids that contain the item, *diffset* stores the difference in tids between an itemset and its prefix. Suppose P is prefix of any length, X and Y are items. Let $d(X)$ denote the *diffset* of item. Let *PXY* be a new candidate, generated by the union of *PX* and *PY*. A generalized relationship between *diffset* and *tidset* can be summarized like this:

$$d(PXY)=d(PY)-d(PX)$$
$$support(PXY)= support(PX)-|d(PXY)| \quad \text{Equation(1)}$$

An equivalent *diffset* representation is shown in Figure 2. Figure 2(a) shows the original *tidset* from Figure 1. The support is shown in brackets. For 1-itemset, each *diffset* stores the items which are not included in the *tidset*. In this example, there are 6 items in total. Given threshold 3, there are only three valid 1-itemsets, *A*, *C* and *E*. Others are struck out. For the 2-itemset, we can get $d(AC)= d(C) – d(A)=(3,5)-(4,5)=(3)$. Item 5 is shared by both *diffset*, hence it is deleted. Item 4 is only in *diffset* of A while not found in *diffset* of C, it's not needed in *diffset* of AC. In this case, the *diffset* of *C* subtracts the *diffset* of *A* is the *diffset* of *AC*. The support of *AC* can be obtained by $support(AC) = support(A)-|d(AC)|=4-|(3)|=3$, where $|d(AC)|$ is the size of itemset *AC*. Similarly, the *diffset* and the support of *AE* and *CE* can be achieved in the same way. For 3-itemset, *ACE*, the *diffset* can be summarized as d(ACE) = d(AE) –d(AC), support(ACE)= support(AC)-|d(ACE)|.

III. PARALLELLIZING APRIORI

Algorithm 1: Apriori

Input: D ---Transaction Dataset
min_sup -- Minmum support
Output: C^i --Frequent Candidate for generation i
S^i -- Support for i generation candidate

```
1:   i= 1; //generation number
2:   C^i initialized to all 1-item frequent candidate
3:   Let ||C^i| be the size of set C^i
4:   While ||C^i|| != 0 do
5:   {
6:      Let s^i_j be the number j counter within S^i
7:      Reset all s^i_j ∈ S^i to 0;
8:      S^i = support_counting(C^i, D);
9:      candidate_pruning(C^i, min_sup, S^i );
10:     C^(i+1) = candidate_generation(C^i ) ;
11:     i++;
12:  }
```

Figure 3 Apriori Algorithm

In general, Apriori consists of iteratively performing three stages: candidate generation, support counting, and candidate pruning. Figure 3 gives pseudo code of Apriori. The algorithm takes, as input, the transaction dataset and *min_support*, and provides as output frequent candidates and their corresponding support after each successive candidate size or generation.

Lines 4-12 describe the loop for one single generation. Support counting determines the support of each candidate. Candidates with support less than the threshold are pruned. The next generation of candidates is generated based on the remaining candidates. The algorithm terminates when no more frequent candidates can be found.

For *tidset*, the task is to find the common id within two parents' t*idset*s. Thenew *tidset* is the intersection part of two pareents' tidest.The size of new tidset, or *cardinality*, is the support. The support counting of the vertical *bitvector* and *diffset* is very similar to the *tidset*'s implementation. The difference lies in the way that they calculate the support. For vertical *bitvector*, the algorithm performs a "bitwise and" operation on two parent candidates and the support is computed with a population count on the *bitvector*. For *diffset*, the way to calculate support and *diffset* is listed in Equation 1.

The time consuming part is the support counting. In traditional horizontal data representation, when using parallel threads to perform support counting, each thread is assigned to count the candidates within a transaction. If multiple threads try to increment the support counter for a candidate, race condition is inevitable. In this case, the program needs to use locks, atomic or critical pragma to protect the data. Thanks to the vertical data representation, in vertical support counting, each thread calculates an independent support and does not have data dependency of each other. In this way, the data and task will be automatically distributed among threads and no complex parallel algorithm is needed. Compared to horizontal Apriori, vertical representation not only improves the performance but it also simplifies the parallelization by removing data dependency.

In the support counting of Algorithm 3, we choose to parallelize the outer-most loop in support counting, which do the support counting for each candidate. In this loop, threads are assigned to compute the new transaction representation and the support. Different scheduling policies like dynamic, guided and static are available in OpenMP. Dynamic and guided schedulings are usually chosen to allieviate the load imbalance. In this case, the static scheduling can partition the workload as there enough iterations in Apriori. We don't need to worry the load imbalance problem in this case.

IV. PARALLELLIZING ECLAT

Eclat is the abbreviation for Equivalence CLAss Transformation algorithm. Apriori is a breadth-first search in the sense that all frequent candidates of one generation are calculated and pruned, and then algorithm moves to the next generation.

Eclat uses a depth-first approach: once a candidate is found, it is used as a prefix to recursively search for larger candidates with that prefix. Figure 4 shows the algorithm for Eclat. Line 1 initializes the candidate to 1-itemset. Lines 2 to line 12 is the recursive function to find frequent itemsets. Lines 3 to line 12 is the loop to scan all input candidates. From lines 4 to 6, a pair of parent candidates generates a new candidate *p*, and then the support is calculated using whatever dataset representation is specified. If the candidate is frequent, new candidates will be generated and added to candidate set C^{i+1} and function *Eclat()* is recursively called.

As we did for Apriori, we evaluated the tidset, bitvector, and diffset representations. Each thread stores and calculates support independently, eliminating the need to protect candidate data. Unlike Apriori, once the base transaction data is read, each thread will generate its own transaction representation data and do the support counting. These generated data can be reused and does not need to share with others. The data independence among Eclat loop iterations

Algorithm 2: Eclat

Input: C^1 --Frequent Candidate for generation 1
min_sup -- Minimum support
Output: Frequent Candidate for all generations

```
1:   C^i initialize to C^1
2:   Eclat(C^i) :
3:     for all c^i_j ∈ C^i do
4:       for all c^i_k ∈ C^i , with k>j do
5:         if c^i_k and c^i_j shares prefix of length (i-1)
6:           p = c^i_k ∪ c^i_j ;//generate new candidate p
7:         support_counting(p);
8:         if support(p)>=min_sup
9:         C^(i+1) =C^(i+1) ∪p; // C^(i+1) initially empty
10:        if C^(i+1)not empty do Eclat(C^(i+1))
11:      End forall
12:    End forall
```

Figure 4. Eclat Algorithm

helps parallel implementation minimize the memory exchange and thus improve the scalability.

We parallelized Eclat using shared memory OpenMP. The OpenMP scheduler divides the workload and distributes the data. The most time consuming part of the program is the recursive Eclat() function, which is the outer loop of the program. The threads are invoked at line 3 of algorithm 2 and joined at line 12. Parallelizing this outer loop reduces the thread invocation cost but also poses a limit on the possible number of threads.

For OpenMP scheduler, we hope each thread can start with its own input data. So we choose the chunksize to as small as possible. The scheduler is set to dynamic so that the load imbalance can be minimized, which is different from parallel Apriori's static setting.

V. Evaluation

We tested our implementation on an SGI UV 1000cc machine Blacklight, a Non-Uniform Memory Access shared-memory system comprising 256 blades. It's an SMP platform ideal for applications that require a large shared memory for computational tasks. Each blade holds 2 Intel Xeon X7560 (Nehalem) eight-core processors, for a total of 4096 cores across the whole machine. Each core has a clock rate of 2.27 GHz, supports two hardware threads using hyper thread and can perform 9 Gflops. We did not use hyper thead as it does not improve our program performance. The sixteen cores on each blade share 128 Gbytes of local memory. In this way, each core has 8 Gbytes of local memory- totalling 32 TB of memory for the system. Two Single System Images with 16 TB of shared memory each are connected by a NumaLink5 Shared Memory Interconnect. The program is compiled on Intel's C/C++ compiler 11.1, which supports OpenMP 3.0.

Experimental results of four different databases--chess, mushroom, pumsb, and pumsb_star--are compared. The size of these datasets varies, from 334KB to 15.9MB. All the datasets can be found at the Frequent Itemset Mining Implementations Repository [6] and are popular datasets in frequent itemset mining and their experimental results have been extensively reported.

TABLE II. RUNNING TIME FOR APRIORI WITH DIFFSET

Testset	1 thread	16 threads	32 threads	64 threads	128 threads	256 threads
chess@0.2	1372.6	101.6	58.2	45.8	28.2	30.8
mushroom@0.005	1083.1	87.4	49.6	39.4	25.9	20.7
pumsb@0.5	1564.7	92.9	65.4	42.1	31.5	30.0
pumsb_star@0.23	1362.9	104.5	78.3	47.5	34.0	28.0

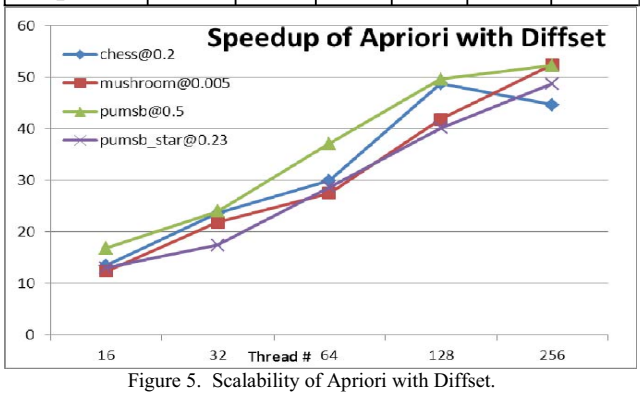

Figure 5. Scalability of Apriori with Diffset.

TABLE I. SUMMARY OF TEST DATASETS

Dataset	Number of Item	Average Length	Number of Transaction	Size
chess	75	36	3,195	334K
mushroom	119	22	8,124	557K
pumsb	2113	73	49,046	15.9M
pumsb_star	2088	50	49,046	10.8M

Table 1 summarizes the datasets we used in our experiments. As you can see, four datasets show various characteristics. The mushroom dataset includes information about various species of mushroom. Chess, pumsb and pumsb_star are from UCI dataset and PUMSB by Robero Bayrdo. The pumsb_star dataset contains census data for population and housing and is a variation of the pumsb dataset but is restricted in that it does not contain any item with a support of 80% or more. In addition, some other dataset from the FIM Repository, like T40I10D100K and accidents have also been tested. Because the number of items is less than the number of processors, they did not show scalability beyond 64 threads and we did not report them here.

A. Apriori

For each representation, we tested the program with all datasets. However, the *tidset* and *bitvector* implementation did not show scalability beyond 16 cores. Due to limited space, we do not report them here. However, in Table II, for Apriori with Diffset, we achieve much better scalability. The table above the plot summarizes the execution time for dataset. The support level is represented as relative number compared to total transaction number. For example, chess@0.2 means dataset chess is tested with support 20%. The execution time is in seconds. Figure 5 shows the speedup relative to one thread. We achieve a speedup of 52X for 256 threads for the mushroom dataset.

The experimental results show the *tidset* and *bitvector* implementation of Apriori are not scalable beyond 16 threads, or one blade. The poor scalability of these two dataset representations is due to two factors. First, *tidset* and *bitvector* have a larger memory footprint than *diffset*. Second, Apriori requires a substantial amount of memory exchange for transactional data. Compared to Eclat, Apriori must store all candidates for each generation, thus the transaction database is much larger. Because the number of candidates can be substantial, the size of tidset and bitvecotor is generally one order of magnitude larger than the diffset's. When generating candidates, each thread must read two parent candidates in order to generate the new candidate. NUMA transactions cause communication overhead when the parent candidates are stored on a remote node. However, Apriori shows scalability from 16 to 256 threads for *diffset*. Experimental result proves that *diffset* not only improves the performance of Apriori, but is also more scalable. The reason is that *diffset*

TABLE III. RUNNING TIME FOR ECLAT WITH TIDSET

Testset	1 thread	16 threads	32 threads	64 threads	128 threads	256 threads
chess@0.2	953.2	60.2	30.6	15.9	8.6	5.7
mushroom@0.005	649.6	41.2	22.1	12.6	8.2	7.3
pumsb@0.52	11335.7	704.4	358.1	186.2	98.5	66.0
pumsb_star@0.22	8993.2	559.4	291.0	149.0	80.4	55.4

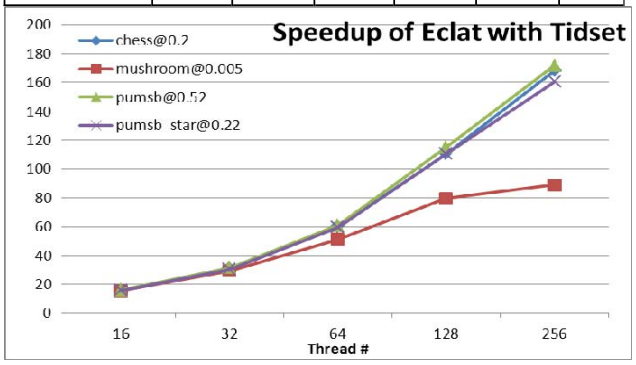

Figure 6. Scalability of Eclat with Tidset.

can reduce the memory exchange by reducing the transaction size dramatically.

B. *Eclat*

Tables III-V show the execution times in seconds of the parallelized Eclat algorithm with Tidset, Bitvector, and Diffset respectively. Figures 6-8 shows the speedup relative to one thread. The horizontal axis is thread count and the vertical axis is speedup. Although speedup varies among dataset; all the datasets are scale with the number of threads. The best results are shown for the Tidset implementation with the pumsb dataset (171 X for 256 threads).

VI. CONCLUSION

TABLE VI. RUNNING TIME FOR ECLAT WITH BITVECTOR

Testset	1 thread	16 threads	32 threads	64 threads	128 threads	256 threads
chess@0.2	5101.1	971.9	608.5	395.4	155.7	75.6
mushroom@0.005	1246.9	191.3	200.4	146.2	34.5	23.9
pumsb@0.52	2045.3	162.7	95.3	57.6	43.6	38.5
pumsb_star@0.22	9706.3	622.8	314.8	170.2	96.9	60.1

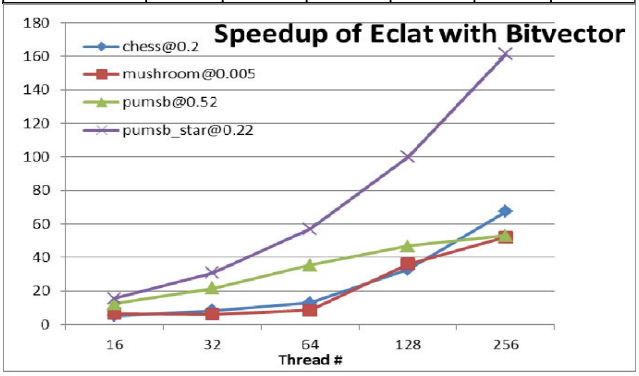

Figure 7. Scalability of Eclat with Bitvector.

TABLE V. RUNNING TIME FOR ECLAT WITH DIFFSET

Testset	1 thread	16 threads	32 threads	64 threads	128 threads	256 threads
chess@0.15	522.2	33.3	17.0	8.9	5.1	4.0
mushroom@0.001	801.7	50.0	26.7	14.3	8.9	7.3
pumsb@0.45	5480.0	350.3	176.2	93.3	54.8	40.9
pumsb_star@0.20	7197.5	453.4	235.4	122.0	67.8	49.2

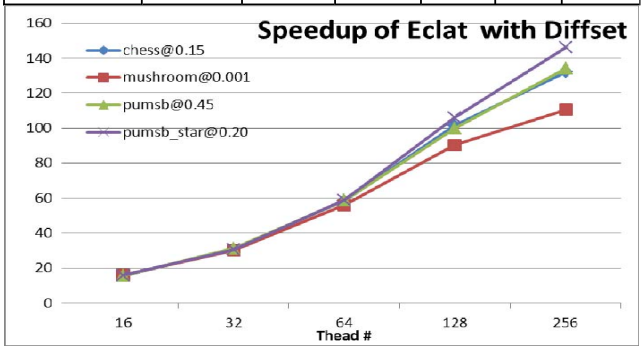

Figure 8. Scalability of Eclat with Diffset.

In summary, we present two parallel FIM algorithms, Apriori and Eclat on shared memory platform, using three different vertical data representations. Experimental results show Eclat is scalable for all three vertical representations, but achieves the best performance with *diffset*. This is because Eclat algorithm has better data independence and requires much less communication overhead than Apriori. Parallel Apriori is only scalable when used with the Diffset transaction representation. The reason is that memory sharing and communication of *diffset* implementation is less compared to those of *tidset* and *bitvector*.

ACKNOWLEDGMENT

This research was supported in part by the National Science Foundation through TeraGrid resources provided by Pittsburgh Supercomputing Centre under grant number [TG-TRA110001].

REFERENCES

[1] F. Bodon, "A Fast Apriori Implementation" Proceedings of the IEEE ICDM Workshop on Frequent Itemset Mining Implementations, (2003).

[2] F. Bodon, "Surprising Results of Trie-based FIM Algorithm" Proceedings of the IEEE ICDM Workshop on Frequent Itemset Mining Implementations (2004).

[3] F. Bodon, A Survey on Frequent Itemset Mining, Technical Report, Budapest University of Technology and Economic, 2006.

[4] F. Bodon, and L. R´onyai, "Trie: an alternative data structure for data mining algorithms," Mathematical and Computer Modelling Volume 38, Issues 7-9, October 2003, pp.739-751.

[5] B. Goethals, *Survey on frequent pattern mining*, Technical report, Helsinki Institute for Information Technology, 2003.

[6] Frequent Itemset Mining Implementations Repository http://fimi.cs.helsinki.fi/src.

[7] Ian H. Witten and Eibe Frank, *Data mining: practical machine learning tools and techniques*.2nd edition, Morgan Kaufmann (2005) pp27

[8] J. Ruoming, Y. Ge, G. Agrawal, "Shared memory parallelization of data mining algorithms: techniques, programming interface, and performance" IEEE Transactions on Knowledge and Data Engineering, Vol 17, Issue 1, 2005.