Novel Parallel Method for Mining Frequent Patterns on Multi-core Shared Memory Systems

Lan Vu
Dept. of Computer Science and Engineering
University of Colorado Denver
1380 Lawrence St. Denver, CO 80204, USA
Lan.Vu@ucdenver.edu

Gita Alaghband
Dept. of Computer Science and Engineering
University of Colorado Denver
1380 Lawrence St. Denver, CO 80204, USA
Gita.Alaghband@ucdenver.edu

ABSTRACT
Frequent pattern mining is an important problem in data mining with many practical applications. Current parallel methods for mining frequent patterns unstably perform for different database types and under-utilize the benefits of multi-core shared memory machines. We present ShaFEM, a novel parallel frequent pattern mining method, to address these issues. Our method can dynamically adapt to the data characteristics to efficiently perform on both sparse and dense databases. Its parallel mining lock free approach minimizes the synchronization needs and maximizes the data independence to enhance the scalability. Its structure lends itself well for dynamic job scheduling resulting in well-balanced load on new multi-core shared memory architectures. We evaluate ShaFEM on a 12-core multi-socket server and find that our method runs 2.1 - 5.8 times faster than the state-of-the-art parallel method. For some test cases, we have shown that ShaFEM saves 4.9 days and 12.8 hours of execution time over the compared method.

Categories and Subject Descriptors
B.3.2 [**Memory Structures**]: Design Styles – *Shared memory*; C.1.2 [**Processor Architectures**]: Multiple Data Stream Architectures – *Multiple-instruction-stream, multiple-data-stream processors*; D.1.3 [**Programming Techniques**]: Concurrent Programming – *Parallel programming*; E.1 [**Data Structures**]: Trees; H.2.8 [**Database Management**]: Database Applications – *Data mining*;

General Terms
Algorithms, Performance

Keywords
Frequent pattern mining, multi-core, shared memory, association rule mining, parallel algorithm, databases

1. INTRODUCTION
Frequent pattern mining is commonly used to find many types of relationships among variables in large databases such as associations, correlations, causality, sequential patterns, episodes and partial periodicity. It plays a crucial role in association rule mining and is also applied in data indexing, classification and clustering [1, 2]. This frequent pattern mining task is considered one of the most important problems in data mining with numerous practical applications such as consumer market-basket analysis, web mining, similarity search of complex structured data, and network intrusion detection [2, 3]. It is also a component of many commercial database systems like Oracle Database (RDBMS), Microsoft SQL Server and IBM DBS2 Database and statistical software like R, SAS and SPSS Clementine [4, 5, 6].

DISCS-2013 November 18, 2013, Denver, CO, USA

http://dx.doi.org/10.1145/2534645.2534653

1.1 Motivation
Several studies have shown that existing frequent pattern mining methods have typically worked well for certain types of databases. Most methods performed efficiently on either sparse or dense databases but poorly on the other [7, 8, 9, 10, 11, 12, 13, 14]. Furthermore, mining frequent patterns is very time-consuming, especially when the database size is large. Hence, it is essential to design and apply parallel computing techniques to speedup this mining task. Most existing works propose parallel solutions for distributed-memory systems [15, 16, 17, 18, 19, 20]. For example, Google deploys this task in their query recommendation system using MapReduce on a distributed-memory machine with thousands of cores [15]. Some surveys [16, 17] have showed that few studies investigated on parallel frequent pattern mining algorithms for shared memory multi-core computers which under-utilize the benefits of shared memory. None of previous parallel works took into consideration the data characteristics to improve the mining performance on different database types.

1.2 Contributions
In this paper, we present a novel parallel frequent pattern mining method named ShaFEM for the new multi-core shared memory platforms to solve the above issues. The proposed method uses a new data structure named XFP-tree, an extension of frequent pattern tree [7], that is shared among processes (also known as threads) to compact data in memory. Then, each parallel process independently mines frequent patterns using a dynamic combination of two mining strategies: one is suitable for sparse data and the other works well on dense data. The main contributions of our study include:

1) A novel parallel mining method that is able to dynamically switch between its two mining strategies to adapt to the characteristics of the database and runs fast on both sparse and dense databases.

2) A new efficient parallel lock free approach that applies new data structures to enhance the independence of parallel processes, minimize the synchronization cost and improve the cache utilization. Additionally, its dynamic job scheduling for load balancing helps increase the scalability on multi-core shared memory systems.

2. BACKGROUND AND RELATED WORK

2.1 The Problem Statement

Let $I = \{i_1, i_2, \ldots, i_n\}$ be the set of all distinct items in the transactional database D. The *count* of an *itemset* α (a set of items) is the number of occurrences of α in D and the *support* of α is the percentage of transactions containing α. A *k-itemset* α, which consists of k items from I, is frequent if α*'s support* is at least *minsup*, where *minsup* is a user-specified minimum support threshold. Given a database D and a *minsup*, the problem statement is to find the complete set of frequent itemsets (or frequent patterns) in D. For example, given the dataset in Table 1 and *minsup*=30%, the frequent 1-itemsets include *a, b, c, d* and *e* while *f* is infrequent because its support is only 22%. Similarly, *ab, ac, ad, ae, bc, bd* are frequent 2-itemsets and *abc* is the only frequent 3-itemset.

Table 1. Sample dataset with *minsup* = 30%

Transaction ID (TID)	Items	Sorted Frequent Items
1	b,d,a	a,b,d
2	c,b,d	b,c,d
3	c,d,a,e	a,c,d,e
4	d,a,e	a,d,e
5	c,b,a	a,b,c
6	c,b,a	a,b,c
7	f	
8	b,d,a	a,b,d
9	c,b,a,e,f	a,b,c,e

2.2 Related Works

Many sequential algorithms have been developed for mining frequent patterns on large databases in which Apriori [1], Eclat [21] and FP-growth [7] are most well-known. However, these algorithms typically perform efficiently on either sparse or dense databases but not both [1, 7, 8, 9, 10, 11, 12, 13, 14, 21].

For large-scale transactional databases, applying parallel computing to speed up the mining process is essential. Majority of the existing parallel methods of mining frequent pattern have been proposed for distributed memory systems [15, 16, 17, 18, 19, 20]. For the memory intensive problems like the one being investigated in this paper, efficient utilization of shared memory MIMD parallelism can significantly improve the overall performance [22]. Parallelizing FP-growth is usually selected for large-scale mining applications due to its performance merits [15]. Several parallel FP-growth based methods have been proposed for shared memory multi-core systems. In the traditional approach, parallel processes cooperatively build a shared global FP-tree resulting in extensive use of costly synchronization locks to access each node of the tree [17]. A different approach called Tree Projection partitions the FP-tree into subsections with small portions shared among processes. Only access to the small shared sections would require locks for synchronization [23]. Multiple Local Parallel Trees (MLPT) approach is the first algorithm not requiring locks by constructing local trees separately and mining the frequent patterns from these trees [24]. FP-array which is another efficient parallel algorithm converts the tree data structure into arrays for better cache optimization. This method improves performance significantly compared to the previous parallel methods which we selected to compare with our proposed method. FP-array can be found in the PARSEC Benchmark [25].

Due to inheriting the mining characteristic of FP-growth, the above parallel methods suffered from poor performance on dense databases. In our study, we will focus on solving this issue and propose a new parallel solution that works efficiently on shared memory multi-core architecture.

3. ShaFEM: A NOVEL PARALLEL FREQUENT PATTERN MINING METHOD

ShaFEM implements a new parallel lock-free approach that uses two different mining strategies and dynamically adapts its mining behavior at runtime for efficient performance on both sparse and dense databases. This is the unique approach that have not been found in previous studies. Our algorithmic design leads to optimized use of shared memory and enhances the data independence among parallel processes for better cache utilization. ShaFEM performs in two stages:

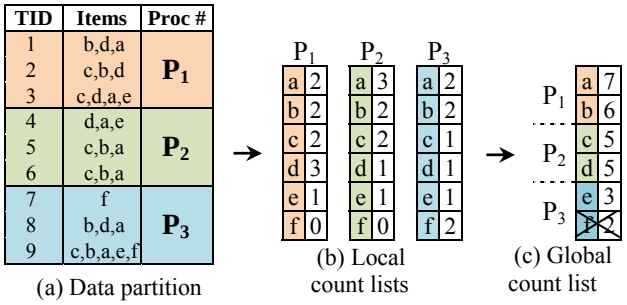
- **The XFP-tree construction stage**: ShaFEM compacts all data in memory into a new data structure that we named XFP-tree, an extension of FP-tree storing all sets of frequent items retrieved from the database. It differs from the FP-tree because some node duplications is allowed. The database is divided into equal parts; each parallel process reads its portion of data to construct its local FP-tree (the structure of these trees described in [7]). The local FP-trees are then merged into a global XFP-tree which is shared among the processes. The trees are implemented and constructed without using locks in nodes to minimize the synchronization cost and enhance the scalability.
- **The frequent pattern generation stage**: The frequent items in the header table of the XFP-tree are distributed to the parallel processes to generate all frequent patterns ending with items being assigned. Its dynamic scheduling approach helps to balance the workload and improves the performance. ShaFEM uses two mining strategies for its frequent pattern generation: one uses FP-tree and the other uses bit vectors. A parallel process will switch between the two strategies for each subset of data being mined depending on the detection of its density characteristics.

4. XFP-TREE CONSTRUCTION

In the first stage of ShaFEM, the global XFP-tree, shared among all cores, is built. This process involves three main steps:

- **Step 1 - Finding the frequent items:**

1. The database is evenly divided into horizontal partitions with same data size and is distributed to parallel processes. For example, the dataset in Table 1 is partitioned into 3 parts (Figure 1-a.)
2. Each process reads its data partition and computes a local *count* list of all items in the database (Figure 1-b.)



(a) Data partition (b) Local count lists (c) Global count list

Figure 1. Parallel construction of the global count list

3. A parallel s*ummation* is performed to reduce the local *count* lists into a shared global *count* list. Each process P_i is responsible for a separate set of items in the global *count* list

to compute their *count* (Figure 1-c). Hence, no implementation of locks is required.

4. The frequent items are identified and sorted in the descending order using their *count* and the user-supplied *minsup* (Figure 1-c).

- **Step 2 - Constructing the local FP-trees:**

1. Each process creates a local header table consisting of the sorted frequent items and their local *counts* using the local *count* lists created in the previous step.
2. Each process reads the transactions from its data portion for the second time to get frequent items of each transaction and inserts them into an FP-tree in their frequency descending order. This is the most time consuming step of the first stage and in our design, all processes work independently to build their local FP-trees. Figure 2 presents three local FP-trees created concurrently from the dataset in Table 1.

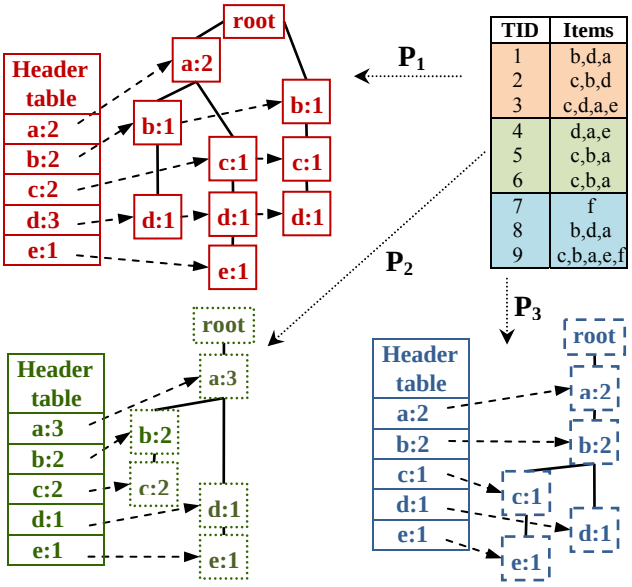


Figure 2. Local FP-tree construction

- **Step 3 - Merging local FP-trees into a global XFP-tree:**

1. The construction of the global XFP-tree is initialized by converting the header table of one local FP-tree into the header table of the global XFP-tree. The frequent items in this table are divided into even subsets and assigned to the parallel processes. For example, *a, b* are assigned for P_1; *c, d* for P_2 and *e* for P_3. Each P_i updates items of this table with the global *count* using the global *count* list of Step 1.

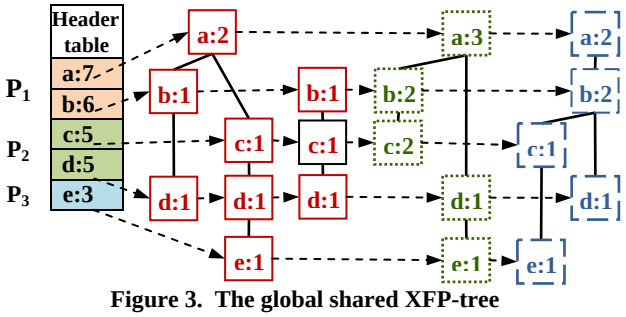


Figure 3. The global shared XFP-tree

2. Each process P_i then joins the local linked lists of their assigned items in the local FP-trees in into the global ones by starting from the existing linked list of the global header table. When all processes complete their work, the XFP-tree is created as in Figure 3. The time to perform this step is negligible because the manipulation of linked lists can be performed in parallel without changing the local FP-trees. Because the next pattern mining stage uses this XFP-tree by traveling in bottom-up direction, the root node of XFP-tree is not needed and not created.

5. FREQUENT PATTERN GENERATION

5.1 Parallel Frequent Pattern Generation Based on Data Characteristics

In this section we introduce a novel parallel approach for frequent pattern generation which can efficiently perform on both sparse and dense databases. Studying many real databases in the well-known FIMI Repository [26], we found that most databases consist of a group of items occurring much more frequently than the others. The more frequent items create subsets of data with the characteristic of dense data while the less frequent items create ones with the characteristic of sparse data. In ShaFEM, the FP-tree based mining strategy named *MineFPTree* is applied for the sparse data portions and the Bit Vector based mining strategy named *MineBitVector* is used for the dense ones. The algorithm switches back and forth between the two mining strategies. This approach is distinct to the prior related parallel works [17, 23, 24, 25] which applied a single mining strategy. Figure 4 presents the overview of the parallel frequent pattern generation process.

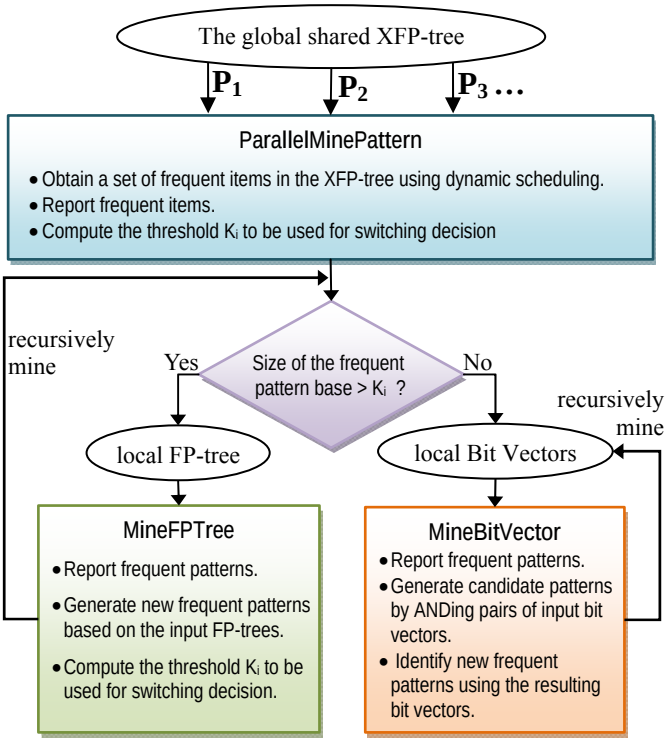


Figure 4. The frequent pattern generation model of each parallel process

ParallelMinePattern initializes the frequent pattern generation stage and manages the mining workload of parallel processes using dynamic job scheduling (Figure 5). Each parallel process P_i is dynamically assigned a set of frequent items in the header table of the XFP-tree and constructs their conditional pattern bases which are "sub-databases" consisting of sets of frequent items co-occurring with a suffix pattern [7]. Based on the size of these

conditional pattern bases, the algorithm switches between the two mining strategies: *MineFPTree* and *MineBitVector*. Each parallel process P_i maintains its own threshold K_i which is used as the switching condition (Section 5.2). All parallel cores work independently until the mining process is completed.

```
Procedure ParallelMinePattern (XFP-tree XT,minsup)
shared XFP-tree XT, minsup
K_i = 0
Parallel Self-Scheduled For j = 1 to number of items in XT
{  α = j^th item in XT
   Output α
   Size = the size of α's conditional pattern base
   Compute and update threshold K_i (* Section 5.2 *)
   If Size > K_i Then
      Construct α's private conditional FP-tree T
      Call MineFPTree(T,β,minsup)
   Else
      Construct α's private bit vectors V and w
      Call MineBitVector(V,w,β,minsup)
   End if }
```

Figure 5. The ParallelMinePattern algorithm

MineFPTree generates frequent patterns by concatenating the suffix pattern of the previous steps with each item α in the header table of the input FP-tree. It then constructs the conditional FP-tree of each item in the input FP-tree and recursively mines new frequent patterns from the new tree. This mining approach does not require generating a large number of candidate patterns and has been shown to perform well on sparse databases [7, 9, 10, 11]. In addition, *MineFPTree* can switch to *MineBitVector,* the second mining strategy that uses the bit vectors and a weight vector. Figure 6 shows the algorithmic details of *MineFPTree.*

```
Procedure MineFPTree(FP-tree T, suffix, minsup)
  If T contains a single path P then
     For each combination x of the items in P
        Output β = x ∪ suffix
        Compute and update threshold K_i (* Section 5.2 *)
  Else For each item α in the header table of FP-tree T
        Output β = α ∪ suffix
        Size = the size of α's conditional pattern base
        Compute and update threshold K_i (* Section 5.2 *)
        If Size > K_i Then
           Construct α's conditional FP-tree T'
           Call MineFPTree(T',β,minsup)
        Else
           Construct α's private bit vectors V and w
           Call MineBitVector(V,w,β,minsup)
        Endif
  Endif
```

Figure 6. The MineFPTree algorithm

MineBitVector applies the mining strategy that utilizes the vertical data format presented as bit vectors to generate frequent patterns. The efficiency of this approach on dense data has been shown in [12, 13, 21]. *MineBitVector* is different from previous works because it uses a new bit vector structure and does not mine the whole database but only the subsets of data with the dense characteristic. The *MineBitVector* algorithm in Figure 7 generates the frequent patterns by concatenating the suffix pattern with each item in the input data. *MineBitVector* then joins pairs of bit vectors using logical AND operation and computes their *support* using the weight vector to specify new frequent patterns. The bit vectors of these patterns whose structured is described in [27] and used as the input to *MineBitVector* in its recursive loop.

```
Procedure MineBitVector (vectors V, vec. w, suffix, minsup)
  Sort V in support-descending order of their items
  For each vector v_k in V
     Output β = item of v_k ∪ suffix
     For each vector v_j in V with j < k
        u_j = v_k AND v_j
        sup_j =support of u_j computed using w
        If sup_j ≥ minsup Then add u_j into U
     If all u_j in U are identical to v_k
     Then For each combination x of the items in U
        Output β' = x ∪ β
     Else If U is not empty
        Call MineBitVector(U,w,β,minsup)
```

Figure 7. The MineBitVector Algorithm

5.2 Switching Between Two Mining Strategies

Effective determination of how and when to switch between the two mining strategies, is key for ShaFEM to perform efficiently on different database types. While *MineFPTree* and *ParallelMinePattern* proceed, thousands or even millions of child FP-trees are constructed. These trees are much smaller than their parents and their size gradually reduces to a level where the trees contain mostly the most frequent items in the database. In these cases, the conditional pattern bases used to build the trees have the characteristic of dense datasets. Therefore, only small conditional pattern bases whose size are specified by the number of sets are transformed into bit vectors and weight vector. If this size is less than or equal to a threshold K_i, ShaFEM switches to the mining strategy using bit vectors. Otherwise, *MineFPTree* continues its recursive loop to generate the frequent patterns.

We use a given K to be the size limit of the bit vectors. The value of K is indentified based on an estimation of the density of database using the *UpdateK* algorithm in Figure 8. The efficiency of this approach has been proved in our prior study [27] of sequential mining. For parallel mining, each parallel process P_i maintains its own K_i and measures its value based on local data processed by that process. This localization leads to not only more parallelism but also a more accurate estimated value of K_i because the data characteristics of local data may vary for each process P_i.

In *UpdateK* algorithm, $Num_{NewPatterns}$ and Size indicate the number of new frequent patterns and the size of a conditional pattern base consecutively. The number of frequent patterns generated for different values of K is maintained in the array X that will be used to determine the best cut-off point to switch from FP-tree to bit vector. We keep track of the number of frequent patterns for several K values that are multiples of 32, i.e., j*Step, $0 \le j \le N$. The step size of 32 is chosen due to a good match with most machine's word and cache block sizes. The efficiency of this selection was also demonstrated in [27].

```
Procedure UpdateK(Num_NewPatterns , Size)
 (* Initialization for the first call to UpdateK for process P_i:
   Create a private array X of N elements, Set all X[j] to zero *)
 For j = 0 to N – 1
    If Size > j*Step then X[j] = X[j] + Num_NewPatterns
    Else Exit Loop
 K_i = 0
 For j = N-1 to 1
    If X[j-1] ≥ 2*X[j] then K_i = (j+1)*Step and Exit Loop
```

Figure 8. The UpdateK algorithm

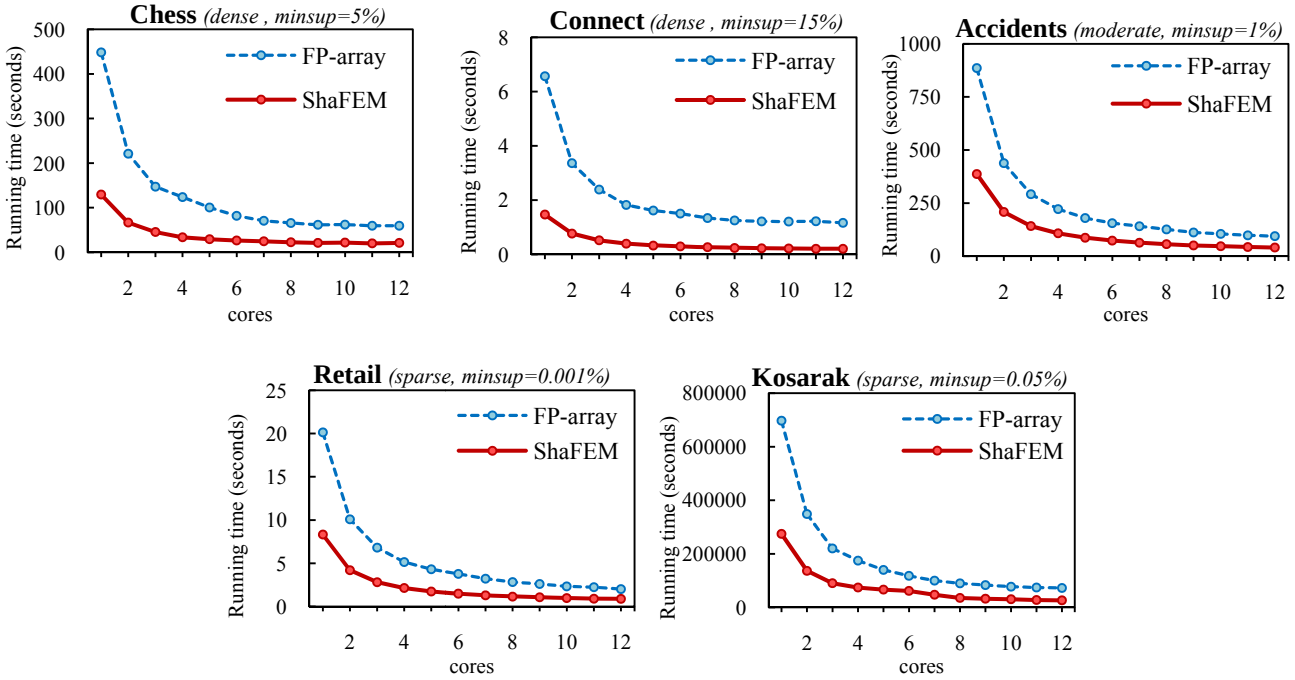


Figure 9. Running time comparison of ShaFEM and FP-array

6. PERFORMANCE EVALUATION

6.1 Experimental Setup

Datasets: Five real datasets including two sparse, one moderate and two dense databases were selected for our experiments (Table 2). They were obtained from the FIMI Repository [26].

Table 2. Experimental Datasets

Dataset	Type	# of Items	Average Length	# of Trans.
Chess	Dense	76	37	3196
Connect	Dense	129	43	67557
Accidents	Moderate	468	33.8	340183
Retail	Sparse	16470	10.3	88126
Kosarak	Sparse	41271	8.1	990002

Hardware: We evaluate ShaFEM on a 12-core dual-socket server with shared memory. It is equipped with two six-core AMD Opteron 2747 processors, 24 GB memory and 160 GB hard drive. This machine is running the CentOS 5.8 operating system.

Software: ShaFEM has been implemented with OpenMP using our computational method presented in Section 4 and Section 5. We study the performance of ShaFEM and compare it with FP-array, one of the best parallel FP-growth like mining methods [22] and is available in the PARSEC Benchmark Suite [25].

6.2 Performance Evaluation

We present the running time of ShaFEM and FP-array for five test datasets in Figure 9. ShaFEM outperforms FP-array for all test cases with different number of cores and different datasets. We find that ShaFEM runs 2.1 - 5.8 times faster than FP-array for the same number of parallel processes in every case for all datasets including both sparse and dense ones. It is important to note that for large datasets such as Kosarak, this speedup of 2.8 compared FP-array for 12 cores translates to a savings of 12.8 hours while our method sequentially runs faster by 117.3 hours or 4.9 days in this test case. Figure 10 shows the speedup on 12 cores of ShaFEM and FP-array compared with the sequential running time of FP-array. It demonstrates that ShaFEM speeds up between 21 to 31.3 times over the FP-array. These results are obtained because ShaFEM balances its workload between its two mining strategies meanwhile FP-array use a single mining strategy.

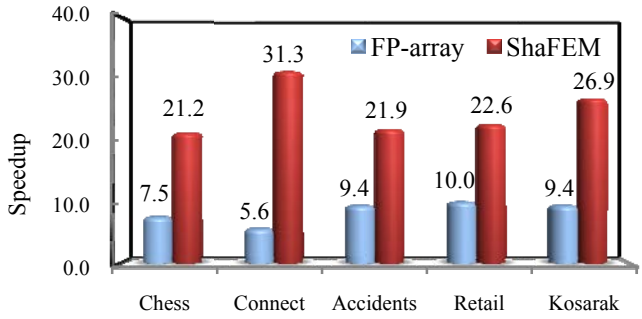


Figure 10. Speedup of ShaFEM and FP-array on 12 cores relative to FP-array one core.

Figure 11 shows the speedup of ShaFEM compared to its own sequential execution time. It demonstrates that the ShaFEM is scalable and performance improvement will continue with additional cores and larger datasets. When all 12 cores are used, ShaFEM runs 6.1 - 10.6 times faster than it does on a single core. ShaFEM scales better for sparse datasets and its scalability is nearly linear for the Accidents, Retail and Kosarak. For the dense datasets Chess and Connect, the speedup increases slower when 8 - 12 cores are used. It causes by the nature structure of dense data which made difficulty for load balancing and it is common for other parallel methods. For example, FP-array suffers from similar situation when its speedup on 12 cores for dense data is 7.5 (Chess) and 5.6 (Connect).

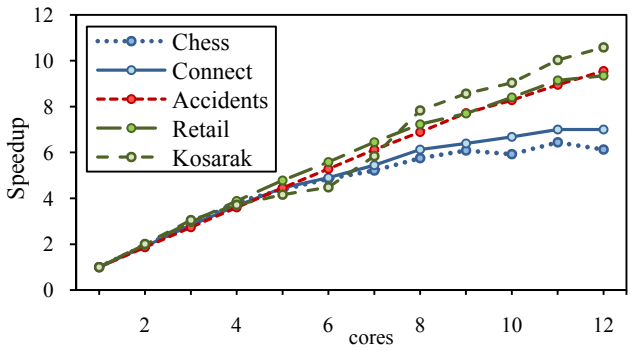


Figure 11. Speedup of ShaFEM on different datasets

7. CONCLUSION

We have presented ShaFEM, a novel method for mining frequent patterns on multi-core share memory machine, and its efficiency on different database types via a number of experimental results on the 12 core machine. By applying the novel dynamic parallel method using the two mining strategies and no locking requirement, ShaFEM has been shown not only running much faster than the state-of-the-art methods but also performing stably both sparse and dense databases. This method can be applied to implement the frequent pattern mining component of databases management systems and statistical software like Oracle Database (RDBMS), Microsoft SQL Server, IBM DBS2 Database, R, SAS, SPSS Clementine, etc. as well as various applications to help this mining task self-adapt to the data characteristic. We will integrate ShaFEM into our mining framework that combines both distributed and shared memory model in near future.

8. REFERENCES

[1] Agrawal , R., Srikant, R. 1994. Fast Algorithms for Mining Association Rules. In *Proceedings of the 20th Int. Conf. on Very Large Databases* (1994). 487-499.

[2] Han, J., Cheng, H., Xin, D., Yan, X. 2007. Frequent Pattern Mining: Current Status and Future Directions. In *Journal of Data Mining and Knowledge Discovery* (Aug. 2007).

[3] Burdick, D., Calimlim, M., Flannick, J., Gehrke, J., Yiu, T. 2005. MAFIA: A Maximal Frequent Itemset Algorithm. In *IEEE Transaction on Knowledge and Data Engineering* (Nov. 2005). vol. 17, no. 11, 1490–1504.

[4] Li, W., Mozes, A. 2004. Computing Frequent Itemsets Inside Oracle 10g. In *Proc. of the 30th Int. Conf. on Very Large Databases* (2004). 1253–1256.

[5] Utley, C. 2005. Introduction to SQL Server 2005 Data Mining. *Microsoft SQL Server 9.0 technical articles* (Jun. 2005). Available at: http://technet.microsoft.com/en-us/library/ms345131.aspx.

[6] Yoshizawa, T., Pramudiono, I., Kitsuregawa, M.. 2000. SQL Based Association Rule Mining Using Commercial RDBMS (IBM db2 UBD EEE). In *Proc. of the 2nd Int. Conf. on Data Warehousing and Knowledge Discovery* (2000). 301–306.

[7] Han, J., Pei,J., Yin Y. 2000. Mining Frequent Patterns without Candidate Generation. In *Proc. of the 2000 ACM SIGMOD Int. Conf. on Management of Data* (Jun. 2000), vol. 29, issue 2, 1-12.

[8] Borgelt C. 2005. An Implementation of the FP-growth Algorithm. In *Proc. of the 1st Workshop on OSDM: Frequent Pattern Mining Implementations* (Aug. 2005). 1-5.

[9] Grahne, G., Zhu, J. 2003. Efficiently Using Prefix-trees in Mining Frequent Itemsets. In *Proc. of the 2003 Workshop on Frequent Pattern Mining Implementations* (2003). 123–132.

[10] Pei, J., Han, J., Lu, H., Nishio, S., Tang, S., Yang, D. 2001. Hmine : Hyper-structure Mining of Frequent Patterns in Large Databases. In *Proc. of the IEEE Int. Conf. on Data Mining* (Nov. 2001). 441–448.

[11] Racz, B. 2004. Nonordfp: An FP-growth Variation Without Rebuilding the FP-tree. In *Proc. of the IEEE Workshop on Frequent Itemset Mining Implementations (FIMI)* (Nov. 2004).

[12] Shporer, S. 2004. AIM2: Improved Implementation of AIM. In *Proc. of the IEEE Workshop on FIMI* (Nov. 2004).

[13] Schmidt-Thieme, L. 2004. Algorithmic Features of Eclat. In *Proc. of the IEEE Workshop on FIMI* (Nov. 2004).

[14] Agrawal, R., Imielinski T., Swami, A. 1993. Mining Association Rules between Sets of Items in Large Databases. In *Proc of the 1993 ACM SIGMOD Int. Conf on Management of Data* (Jun. 1993). vol. 22, issue 2, 207-216.

[15] Li, H.,Wang, Y., Zhang, D., Zhang, M., Chang, E. 2008. PFP: Parallel FP-Growth for Query Recommendation. In *Proc. of the 2008 ACM Conf. on Recommender systems* (2008). 107-114.

[16] Zaki, M. J. 1999. Parallel and Distributed Association Mining: A Survey. In *IEEE Concurrency Journal* (Oct-Dec 1999). vol. 7, issue 4, 14- 45.

[17] Garg, R., Mishra, P. K. 2009. Some Observations of Sequential, Parallel and Distributed Association Rule Mining Algorithms. In the *IEEE Proc of the 2009 Int. Conf. on Computer and Automation Engineering* (March 2009).

[18] Moonesinghe, H. D. K., Chung, M.J., Tan P.N. Fast Parallel Mining of Frequent Itemsets, *Michigan State University*.

[19] Tanbeer, S.K., Ahmed, C.F., Jeong, B.S. 2009. Parallel and Distributed Frequent Pattern Mining in Large Databases. In *Proc. of the 11th IEEE Int. Conf. on High Performance Computing and Communications* (2009). 407 -414.

[20] Li, J., Liu, Y., Liao, W., Choudhary, A. 2006. Parallel Data Mining Algorithms for Association Rules and Clustering. In *CRC Press* (2006), 3-5.

[21] Zaki, M., Parthasarathy, S., Ogihara, M., Li, W. 1997. New algorithms for fast discovery of association rules. In *Proc. of Knowledge Discovery and Data Mining* (1997). 283-286.

[22] Liu, L., Li, E., Zhang, Y., Tang, Z. 2007. Optimization of Frequent Itemset Mining on Multiple-core Processor. In *Proc. of the 33rd Int. Conf. on Very Large Databases* (2007). 1275-1285.

[23] Chen, D., Lai, C., Hu W., Chen, W., Zhang, Y., Zheng, W. 2006. Tree partition based parallel frequent pattern mining on shared memory systems. In *Proc. of the 20th Int. Conf. on Parallel and distributed processing* (2006), 313-320.

[24] Zaiane, O. R., El-Hajj, M.,Lu, P. 2001. Fast parallel association rule mining without candidacy generation. In *Proc. of the IEEE 2001 Int. Conf. on Data Mining* (ICDM, 2001), 665-668.

[25] Bienia, C. 2008. The PARSEC Benchmark Suite: Characterization and Architectural Implications (PARSEC - freqmine). *Princeton University Technical Report TR-811-08* (Jan. 2008). Available at http://parsec.cs.princeton.edu.

[26] Frequent Itemset Mining Implementations Repository, Available at http://fimi.ua.ac.be

[27] Vu, L., Alaghband, G. 2012. Mining Frequent Patterns Based on Data Characteristics. In *Proc. of the 2012 Int. Conf. on Information and Knowledge Engineering* (Jul. 2012). 369-375.

[28] Uno, T.,Kiyomi, M.,Arimura, H. 2004. LCM ver. 2: Efficient Mining Algorithms for Frequent/Closed/Maximal Itemsets. In *Proc. of ICDM Workshop on FIMI*, (2004).