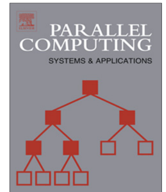
Parallel Computing 40 (2014) 768–785

Contents lists available at ScienceDirect

Parallel Computing

journal homepage: www.elsevier.com/locate/parco



Novel parallel method for association rule mining on multi-core shared memory systems

Lan Vu *, Gita Alaghband

Dept. of Computer Science and Engineering, University of Colorado Denver, Denver, CO 80204, USA

ARTICLE INFO

Article history:
Available online 11 October 2014

Keywords:
Frequent pattern mining
Multi-core
Shared memory
Association rule mining
Parallel algorithm
Databases

ABSTRACT

Association rule mining (ARM) is an important task in data mining with many practical applications. Current methods for association rule mining have shown unstable performance for different database types and under-utilize the benefits of multi-core shared memory machines. In this paper, we address these issues by presenting a novel parallel method for finding frequent patterns, the most computational intensive phase of ARM. Our proposed method, named ShaFEM, combines two mining strategies and applies the most appropriate one to each data subset of the database to efficiently adapt to the data characteristics and run fast on both sparse and dense databases. In addition, our new lock-free design minimizes the synchronization needs and maximizes the data independence to enhance the scalability. The new structure lends itself well to dynamic job scheduling resulting in a well-balanced load on the new multi-core shared memory architectures. We have evaluated ShaFEM on 12-core multi-socket servers and found that our method run up to 5.8 times faster and consumes memory up to 7.1 times less than the state-of-the-art parallel method. For some test cases, ShaFEM can save up to 4.9 days of execution time over the compared method.

© 2014 Elsevier B.V. All rights reserved.

1. Introduction

Association rule mining (ARM) is one of the fundamental tasks in data mining. Since its first application for the analysis of sales or basket data which was introduced by Agrawal et al. [1], ARM has been applied broadly in many fields with an increasing number of applications such as market analysis, biomedical and computational biology research, web mining, decision support, telecommunications alarm diagnosis and prediction, and network intrusion detection [2,4,7,8,13,14,46]. Because of the importance of this mining task, ARM has become an essential mining component of most popular database systems like Oracle Database (RDBMS), Microsoft SQL Server, IBM DBS2 Database and IBM DBS2 and statistical software like R, SAS and SPSS Clementine [24–26,43–45]. The increasing need to analyze big data has led to the development of new ARM method that can leverage the computing power of emerging platforms to support this mining task. Furthermore, widening the applicable areas of ARM requires algorithms that can perform efficiently on different data types.

1.1. Motivation

Several studies have shown that ARM methods typically worked well for certain types of databases. Most methods performed efficiently on either sparse or dense databases but poorly on the other [11,15,17–22,28,30]. Table 1 presents

* Corresponding author.
E-mail addresses: Lan.Vu@ucdenver.edu (L. Vu), Gita.Alaghband@ucdenver.edu (G. Alaghband).

http://dx.doi.org/10.1016/j.parco.2014.08.003
0167-8191/© 2014 Elsevier B.V. All rights reserved.

Table 1
Running time on sparse and dense database.

Databases	Type	*Minsup* (%)	Apriori	Eclat	FP-growth
Chess	Dense	20	1924	77	89
Connect	Dense	30	522	366	403
Retail	Sparse	0.003	18	59	10
Kosarak	Sparse	0.08	4332	385	144

the execution time of three well-known sequential algorithms Apriori [1], Eclat [10] and FP-growth [11] on sparse and dense databases. It shows Eclat performs best on dense data while FP-growth runs fastest on the sparse ones (underline numbers indicate the best execution times among the three algorithms).

Furthermore, the large data size and the amount of computation involved lead to the crucial need of applying parallel computing for this mining task to speed up the large-scale data mining application. Most existing works have proposed parallel solutions for distributed-memory systems [9,34,35,37,38,40]. Some surveys [34,35] show that very few studies were conducted on parallel frequent pattern mining algorithms for the shared memory multi-core platforms. Most of them have based on Apriori that is far less efficient than the other algorithms (shown in Table 1). None of previous parallel work took into consideration the data characteristics to improve the mining performance on different database types.

1.2. Contributions

We present a novel parallel ARM 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 that is shared among processes to compact data in memory. Then, each parallel process independently mines rules and based on the density of mining data being processed dynamically selects and switches between two mining strategies where 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 can dynamically switch between its two mining strategies to adapt to the characteristics of the database and run fast on both sparse and dense databases. This original contribution is based on the recognition for the need to apply different data mining strategies as mining proceeds and the fact that the dataset characteristics change during this processing, and therefore the need for runtime detection of when this should occur.
(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. This is an important contribution as ARM is a challenging problem for high performance computing. It has many dependent subtasks, unpredictable workload and complex data structures and requires many reduction steps.
(3) We demonstrate the efficiency of our approach by conducting intensive experiments to benchmark ShaFEM and other state-of-the arts mining approaches. We present an in-depth analysis of the impact of each technique employed and the contributions made to the final performance of ShaFEM.

1.3. Paper organization

The rest of the paper is organized as follows. Section 2 introduces the problem statement and related works. The parallel frequent pattern mining algorithm, ShaFEM, is presented in Section 3. The first mining stage to construct the XFP-tree is demonstrated in Section 4. Section 5 details the second mining stage and describes the dynamic decision making process to switch between the two mining strategies. We evaluate the scalability and analyze the performance merits of ShaFEM in Section 6. The final section is our conclusion.

2. Background

2.1. The problem statement

Association rule mining (ARM) aims at discovering rules that specify the frequency co-occurrence of groups of itemsets, subsequences, or substructures in a database. For example, an association rule of retail database can be of the form "70% of customers who buy milk and butter also buy bread with confidence 90%". Detection of these interesting rules contributes to the knowledge base used to build intelligence systems such as product recommendation, gene function prediction, network intrusion detection, search engine ranking, etc. Google uses this mining task for their query recommendation system [9].

The association rule mining problem can be stated as follows: Let $I = \{i_1, i_2, \ldots, i_n\}$ be the set of n distinct items in the transactional database D. Each transaction T in D contains a set of items called *itemset*; a *k-itemset* is an itemset with k items. The *count* of an *itemset* x is the number of occurrences of x in D and the *support* of x is the percentage of transactions containing x.

Given a database D, the ARM problem is to find all strong association rules with the form: $X \rightarrow Y | X, Y \subset I$, and $X \cap Y = \emptyset$ whose *support* and *confidence* satisfy a minimum support threshold (*minsup*) and a minimum confidence threshold (*minconf*), two user-specified inputs. The *confidence* of a rule is the percentage of transactions in D that contain X also contain Y. It is specified by the formula: *confidence* $(X \rightarrow Y) = support(X \cup Y)/support(X)$. In other words, the *confidence* of a rule is the conditional probability that a transaction contains Y, given that it contains X. The association rule mining consists of two separate steps: (1) mining all frequent patterns (or frequent itemsets) from the original databases that are highly compute and memory intensive and (2) generating rules from these frequent patterns.

For example, given the database in Table 2 and *minsup* = 20%, the frequent 1-itemsets include *a*, *b*, *c*, *d* and *e* while *f* is infrequent because the *support* of *f* is only 11%. Similarly, *ab*, *ac*, *ad*, *ae*, *bc*, *bd*, *cd*, *ce*, *de* are frequent 2-itemsets and *abc*, *abd*, *ace*, *ade* are the frequent 3-itemset. If *minconf* = 80%, some association rules include $b \rightarrow a$, $c \rightarrow a$, $d \rightarrow a$, $e \rightarrow a$, and $c \rightarrow b$ because their *confidence*s are larger or equal to 80%. In this paper, we use the terms pattern and itemset; database and dataset interchangeably.

While the first stage is computationally quite intensive and requires efficient methods to make mining task feasible, the second stage is a trivial task. Hence, we focus on solving the computing issue of the first stage of finding all frequent patterns in D whose *support* is larger or equal to *minsup*. The following stage of rule generation can be performed easily as in [1].

2.2. Sequential association rule mining

Most current approaches [1,5–11] for finding association rules utilize the property that a *k-itemset* is frequent only if its *sub-itemsets* are frequent to significantly reduce the search space of frequent *itemsets*.

First, the database D is scanned to specify all frequent items (or *1-itemsets*) in D based on the *minsup* value. After this step, only data of frequent items (e.g. the third column in Table 2) are used to determine the frequent *itemsets* as well as to generate the association rules. This considerably reduces the memory usage and computation by avoiding a large amount of infrequent data from loading into memory.

In next steps, the frequent $(k + 1)$-*itemsets* are discovered using frequent *k-itemsets* X of the previous step. To do this, the datasets D_X which are subsets of D and contain frequent items Y co-occurring with X ($X \cap Y = \emptyset$) are retrieved and used to determine the frequency of $(k + 1)$-*itemsets*. Depending on the mining method, D_X can be presented in memory using various data structures such as TID-list [5], Bitmap Vectors [3], FP-tree [6], FP-array [12], diffset [16] or even be obtained by re-scanning the original database D from disks as in the Apriori method [1].

The characteristics of these data structures and the behaviors of their mining methods are quite different and will result in different performance for a given database [31,32]. For example, algorithms like Apriori [1], FP-growth [6], H-mine [8], nonordfp [9] and those making use of FP-array data structure [12] exploit horizontal format of data and perform efficiently on sparse databases (e.g. web document data or retail data) while Eclat [5], Mafia [3], AIM2 [10] present data in a vertical format and run faster on the dense ones (e.g. biological sequence data). These mining methods perform unstably on different data types as demonstrated in Table 1. Furthermore, the characteristics of data subsets D_X used to mine $(k + 1)$-*itemsets* can change from very sparse to very dense as mining proceeds. Hence, applying a suitable mining strategy for each D_X is essential to improve the performance of ARM. It leads to the introduction of our parallel mining approach employing two mining strategies based on the characteristics of D_X that could change dynamically during execution.

2.3. Parallel association rule mining

For large-scale transactional databases, applying parallel computing to speed up the mining process is essential. Majority of the existing parallel association rule mining algorithms have been proposed using the distributed memory computing model with message passing [9,34,35,37,38,40]. In the current trend of computer architecture, most computers are equipped with one or many shared-memory multi-core processors. For this type of machines and the memory intensive problems like the association rule mining in this paper, efficient utilization of shared memory MIMD parallelism is essential to improve the overall performance [23] as the large data movement and communication requirements of parallel association rule mining

Table 2
Sample dataset with *minsup* = 20%.

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	a, b, c, e

can be performed seamlessly exploiting the underlying shared memory [33]. The work presented in this paper focuses on exploiting the shared memory platform which could also be a compute node of a larger cluster machine.

FP-growth is usually a selection for large-scale mining applications due to its performance merits [9]. In addition, the divide-and-conquer approach of FP-growth naturally lends itself to parallelism. Several parallel methods inspired by FP-growth have been proposed for shared memory multi-core systems. In the traditional FP-growth-based parallel 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 [35]. 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 [36]. Although this approach reduces the synchronization cost considerably, it adds the overhead of extra partitioning of the workload and is harder to load balance. Moreover, updating of the shared portion constitutes a considerable workload of the FP-tree construction that can reduce the scalability of the algorithm as the number of processes increases.

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 [41]. This approach has shown good scalability on shared-memory multi-core machine. The parallel version of FP-array is another efficient algorithm that uses locks for FP-tree construction. It then converted this data structure into arrays for better cache optimization. This method improves performance significantly compared to the previous parallel methods and has been integrated into the PARSEC Benchmark [39]. In the later stage, frequent patterns are generated by recursive construction of child FP-trees from the parent FP-tree. Because of the divide-and-conquer approach of FP-growth, the mining workload can be partitioned and distributed to parallel processes without data dependence conflicts.

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 parallel solution that works efficiently on shared memory multi-core machine architecture.

3. ShaFEM: a novel parallel association rule mining method

3.1. The overview of ShaFEM

Presentation and manipulation of data in memory are two key elements that decide the performance of the parallel frequent pattern mining algorithm. It is essential for commercial database systems such as Oracle RDBMS, MS. SQL Server and IBM DBS2 and statistical software like R, SAS and SPSS Clementine [24–26,43,44] and other applications to be equipped with a parallel frequent pattern mining method that runs fast and efficient on all types of databases: large, dense, sparse and not have to worry about their size and characteristics as they vary depending on their real applications.

ShaFEM implements a new parallel lock-free approach that uses two different mining strategies (one suited to sparse and one to dense databases) and dynamically adapts its mining behavior at run time for efficient performance on both database types. Our algorithmic design leads to optimized use of shared memory and enhances the data independence among parallel processes for better cache utilization. In an overview, ShaFEM performs its mining task in the following two stages that are presented in detail in Sections 4 and 5:

- **The XFP-tree construction stage**: ShaFEM applies the FP-tree based approach to compact all data in memory to avoid the high I/O cost of multiple database scans. The database is divided into equal parts; each parallel process reads its portion of data to construct its local FP-Tree which is private to the process. 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 to minimize the synchronization cost and enhance the scalability.
- **The frequent pattern generation stage**: all frequent patterns are found using the divide-and-conquer approach. The frequent items in the header table of the XFP-tree are dynamically obtained by the parallel processes as they become available in order to balance the workload. Each parallel process recursively and independently generates all frequent patterns ending with one item being assigned and continues with the next item, Fig. 5. ShaFEM uses two mining strategies for its frequent pattern generation: FP-tree that uses a horizontal data format, and bit vector that uses a vertical data format. A process will dynamically switch between the two strategies in the course of mining for frequent patterns depending on detecting the density of the remaining data to be mined.

3.2. The data structures

FP-tree is a prefix tree storing all sets of ordered frequent items [11]. This tree presents data in a horizontal format and consists of a header table storing the frequent items with their *count*, a root node and a set of prefix sub-trees. Each node of the tree includes an *item name*, a *count* indicating the number of transactions that contain all items in the path from the root node to the current node, and a *link* to its parent node. Each linked list starting from the header table links all nodes of the same frequent item. If two itemsets share a common prefix, the shared part can be merged as long as the *count* properly reflects the frequency of each itemset in the database. Fig. 1 illustrates an FP-tree constructed from the dataset in Table 2 where a pair $\langle x{:}y\rangle$ indicates *item name* and its *count*.

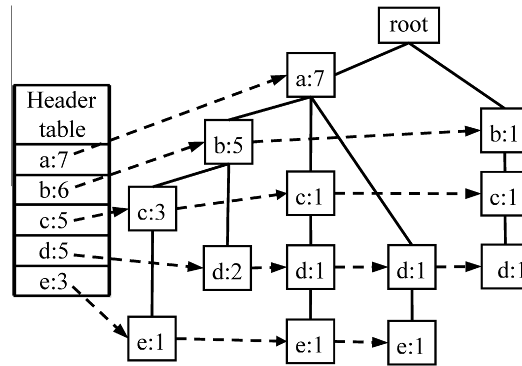


Fig. 1. FP-tree constructed from the database in Table 2.

TID	Frequent Items	Bit Vectors				
		a	b	c	d	e
1	a,b,d	1	1	0	1	0
2	b,c,d	0	1	1	1	0
3	a,c,d,e	1	0	1	1	1
4	a,d,e	1	0	0	1	1
5	a,b,c	1	1	1	0	0
6	a,b,c	1	1	1	0	0
7		0	0	0	0	0
8	a,b,d	1	1	0	1	0
9	a,b,c,e	1	1	1	0	1

Fig. 2. Bit vectors constructed from the dataset in Table 2.

XFP-tree is an extension of FP-tree newly introduced in ShaFEM. This data structure stores all sets of frequent items retrieved from the database and differs from the FP-tree because some degree of node duplication is allowed. It is constructed by combining several FP-trees into a single tree described in detail in Section 4.

An XFP-tree (Fig. 5) is purposely designed so that it is not as compact as a FP-tree (Fig. 1) to achieve higher degree of parallelism and scalability. This data structure is customized for parallel access and does not require using locks during concurrent construction of the XFP-tree.

Bit Vector is a new data structure used in ShaFEM. It includes *item name*, *count* and *vector of binary bits* associating with an item or a pattern. It is used to present the data of a database in memory in the vertical format. The *i*th bit of this vector indicates if the *i*th transaction in the database contains that item or pattern (1: exist, 0: does not exist). For example, the dataset in Table 2 can be presented in five bit vectors as in Fig. 2. The bit vector of the item *f* is removed because this item is infrequent. This structure does not only save memory but also enables low-cost bitwise operations for computations.

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 (Fig. 3a).
  (2) Each process reads its data partition and computes a local *count* list of all items in its portion of the database (Fig. 3b). Data is read in parallel by all processes without any synchronization because the database is equally partitioned and each process can determine its data partition using its process ID. Data are read and processed in blocks to reduce I/O overhead.
  (3) A parallel *summation* is performed to reduce the local *count* lists (private to parallel processes) 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* (Fig. 3c). 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* (Fig. 3c).
- Step 2 – constructing the local FP-trees:
  (1) Each process creates a local header table (private to the process) consisting of the sorted frequent items and their local *counts* using the local *count* lists created in the previous step.

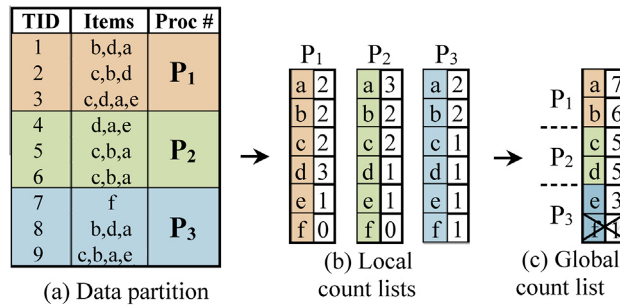


Fig. 3. Parallel construction of the global count list.

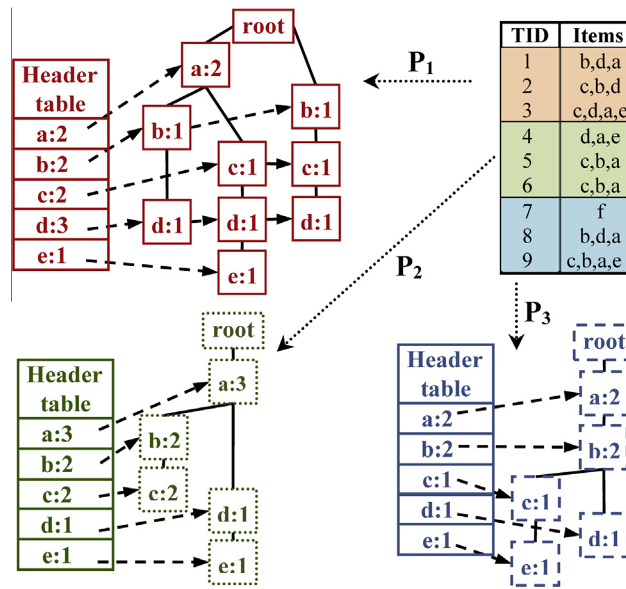


Fig. 4. Local FP-tree construction.

(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. Fig. 4 presents three local FP-trees created concurrently from the dataset in Table 2.

- 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.

(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 Fig. 5. 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. ShaFEM generates all frequent patterns by exploring a very large number of data subsets

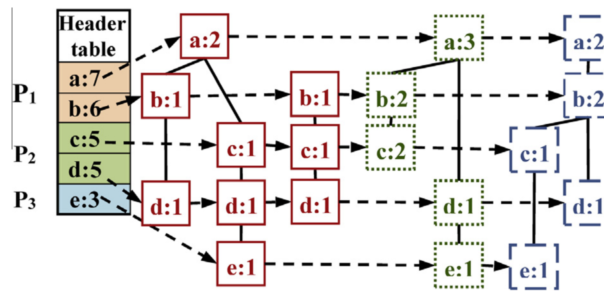


Fig. 5. The global shared XFP-tree.

extracted from the database. Studying many real databases in the well-known FIMI Repository [29], 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. For example, Fig. 6 shows the data subsets with itemsets occurring and ending with c, e and ed which are extracted from Table 2. It can be seen that these data subsets have different characteristics. The data subset of c is considered dense because it has two of the most frequent items a, b while the one of de is less dense because it contains some infrequent items like c, d (items with low frequency value). In ShaFEM, we are able to compute an estimation of the intensity of occurrence of frequent patters at various stages of the execution, K_i, and based on this representative characteristic select the most appropriate mining strategy for a data subset being processed.

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. This approach is distinct to the prior related parallel works [35,36,39,41] which applied a single mining strategy and its performance efficiency will be demonstrated in Section 6.

Fig. 7 presents the overview of the parallel frequent pattern generation process. After the global XFP-tree is constructed, parallel processes independently start searching for all frequent patterns using three tasks *ParallelMinePattern*, *MineFPTree* and *MineBitVector* as described below in more details.

ParallelMinePattern initializes the frequent pattern generation stage and manages the mining workload of parallel processes using dynamic job scheduling. Each parallel process P_i is assigned a frequent item α in the header table of the XFP-tree. It then traverses the XFP-tree in a bottom up direction starting from the nodes in the linked list of the assigned item to retrieve its conditional pattern base C. A conditional pattern base is a "sub-database" consisting of sets of frequent items co-occurring with a suffix pattern, item α in this case (Fig. 9a). The dynamic decision making to switch between the two mining strategies, *MineFPTree* and *MineBitVector*, is based the size of the conditional pattern bases in comparison with a threshold value, K_i, which is estimated and updated at run time using the number of frequent patterns found by the two mining strategies (Section 5.2). If *MineFPTree* is invoked, the parallel process will build the private conditional FP-tree of item α, which is an FP-tree, constructed from the conditional pattern base C instead of the whole database. Otherwise, private bit vectors are generated using the base C and the *MineBitVector* strategy is called. A weight vector w whose elements indicate the frequency of sets in the base C is added as the input of *MineBitVector;* this vector is used to compute the *count* of candidate patterns. Each parallel process P_i maintains its own threshold K_i which reflects the characteristics of the local data being processed. All parallel cores work independently until the mining process is complete. The *ParallelMinePattern* algorithm is presented in Fig. 8.

Fig. 9 illustrates an example. The conditional pattern base of item d in the XFP-tree of Fig. 5 consists of the four sets {a:2,b:2}, {a:1, c:1}, {a:1} and {b:1, c:1} in which {a, b} occurs twice (Fig. 9a). This base is equivalent to the dataset

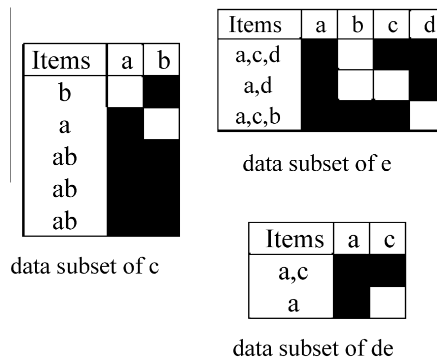


Fig. 6. The data subsets with itemsets occurring and ending with c, e and de (marked in black).

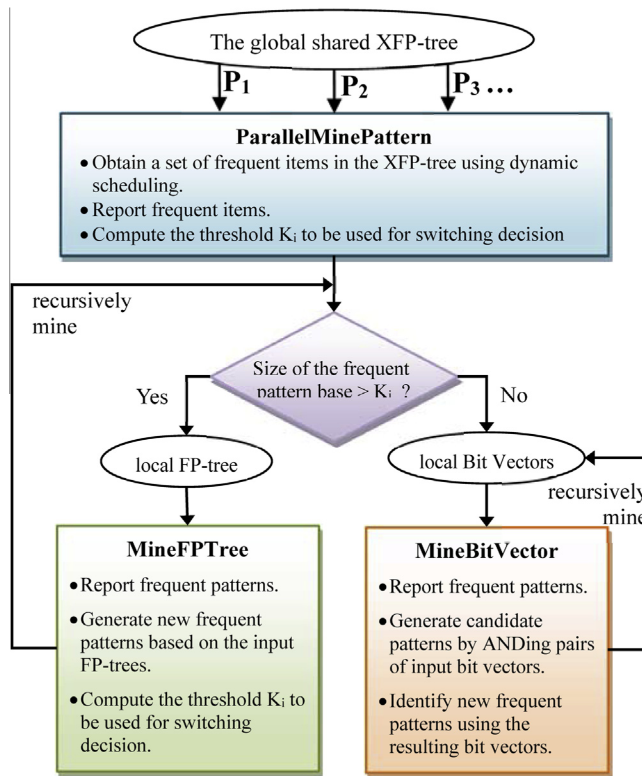


Fig. 7. The frequent pattern generation model of each parallel process.

```
Procedure ParallelMinePattern (XFP-tree XT,minsup)
shared XFP-tree XT, minsup
Ki = 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 Ki
   If Size > Ki 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
}
```

Fig. 8. The ParallelMinePattern algorithm.

represented in Fig. 9b. If *MineFPTree* is selected for mining this base, the conditional FP-tree of item d is constructed as in Fig. 9c. Otherwise, the bit vectors a, b, c and the weight vector w are created instead to generate the frequent patterns using *MineBitVector.*

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. Fig. 10 shows the algorithmic details of *MineFPTree*. This mining approach which uses the horizontal data format does not require generating a large number of candidate patterns and has been shown to perform well on sparse databases [11,18–20]. In addition, *MineFPTree* can switch to the second mining strategy of ShaFEM

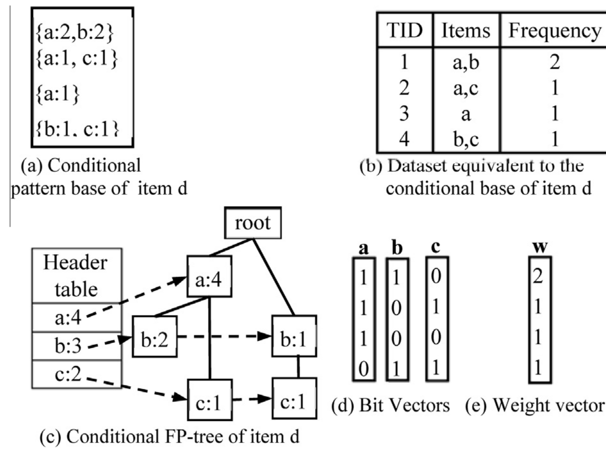


Fig. 9. Illustration of FP-tree and bit vector construction.

when the size of a current conditional pattern base is small enough and suitable to be mined using vertical data layout. In this case, the bit vectors and a weight vector are constructed from that conditional pattern base and mining process switches to *MineBitVector*. The value of K_i is updated at runtime using the method in Section 5.2.

MineBitVector applies the mining strategy that utilizes the vertical data format to generate frequent patterns. The efficiency of this approach on dense data has been shown in [10,15,21,22]. *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. Each subset of data is a frequent pattern base retrieved from the XFP-tree or FP-tree (Fig. 9a and d). The *MineBitVector* algorithm in Fig. 11 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 are collected and used as the input to *MineBitVector* in its recursive loop.

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. We present here the heuristics of how the switching decision is made dynamically.

During the mining process using FP-tree, a very large number of conditional pattern bases are processed to construct new FP-trees from their parent trees. A FP-tree is organized in such a way that the nodes of the most frequent items are closer to the top. The newly generated trees are much smaller than their parents because the less frequent items whose nodes are at bottom of the parent trees are removed. The size of the conditional pattern bases retrieved from these trees also reduces to a level where the bases contain mostly the most frequent items in the database. In these cases, the conditional pattern bases have the characteristic of dense datasets. Therefore, only small conditional pattern bases can be considered for transforming

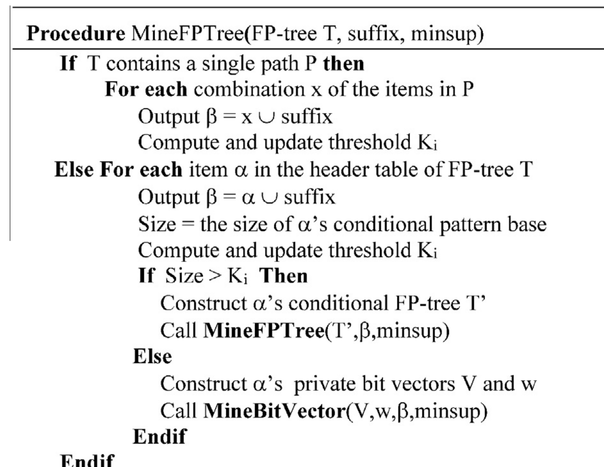

```
Procedure MineFPTree(FP-tree T, suffix, minsup)
  If T contains a single path P then
      For each combination x of the items in P
          Output β = x ∪ suffix
          Compute and update threshold Ki
  Else For each item α in the header table of FP-tree T
          Output β = α ∪ suffix
          Size = the size of α's conditional pattern base
          Compute and update threshold Ki
          If Size > Ki 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
```

Fig. 10. The *MineFPTree* algorithm.

```
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)
```

Fig. 11. The *MineBitVector* algorithm.

```
Procedure UpdateK(Num_NewPatterns , Size)
(* Initialization for the first call to UpdateK for process P_i:
  Create a private array X with N elements, Set all X[j] to zero *)
 newK = 0
 X[0] = X[0] + Num_NewPatterns
 For i = 1 to N – 1
 {   If Size > i*Step
     {   X[i] = X[i] + Num_NewPatterns
       If X[i-1] >= 2* X[i]  Then  newK = (i+1)*Step
     } Else Exit Loop
 }
 i = K_i/Step - 1
 If (i>0 AND X[i-1]  < 2*X[i] ) Then K_i = 0
 If newK  > K Then K = newK
```

Fig. 12. The *UpdateK* algorithm.

into bit vectors and weight vector. The size of a conditional pattern base is specified by the number of sets in that base which is similar to the number of transactions in a dataset. If this size is less than or equal to a threshold K_i, bit vectors and a weight vector are constructed and ShaFEM switches to the mining strategy using bit vectors. Otherwise, *MineFPTree* continues its recursive loop to generate the frequent patterns.

In this context, we use a given K to be the size limit of the bit vectors. The value of K is obtained based on an estimation of the density of database using the *UpdateK* algorithm in Fig. 12. The efficiency of this approach was shown 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. This algorithm requires less computational need in comparison to the one we presented in [46].

In *UpdateK* algorithm, a good value of K is determined by examining different K_i using the algorithm in Fig. 12 where K_i is multiples of 32, i.e., $j * \text{Step}$, $0 \leqslant j \leqslant N$, Step = 32. Instead of computing each $K_i = 0, 1, 2, 3, \ldots N$, we check $K_i = 0, 32, 64, \ldots N$ for two reasons: (1) to reduce the number of computations and (2) to have a good match with most machine's word and cache block sizes because the bit vectors of *MineBitVector* are presented as arrays of 32 bit words. In Fig. 12, $\text{Num}_{\text{NewPatterns}}$ 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.

6. Performance evaluation

In this section, we evaluate the performance of ShaFEM and compare it with prior works.

6.1. Experimental setup

Datasets: Six real datasets with various characteristics and domains were selected for our experiments. They included three sparse, one moderate and two dense databases all obtained from the FIMI repository [29], a well-known repository for frequent pattern mining and association rule mining. The dataset features are reported in Table 3.

Table 3
Experimental datasets.

Dataset	Type	# of Items	Average length	# of Trans.
Chess	Dense	76	37	3196
Connect	Dense	129	43	67,557
Accidents	Moderate	468	33.8	340,183
Retail	Sparse	16,470	10.3	88,126
Kosarak	Sparse	41,271	8.1	990,002
Webdocs	Sparse	52,676,657	177.2	1,623,346

Hardware: We evaluate ShaFEM on two 12-core shared memory dual-socket servers: one with Intel Xeon processors and the other with AMD Opteron processors. Their specifications are listed in Table 4. Because the experimental results on both machines are consistent, for simplification and due to our AMD cluster being dedicated to parallel processing without interference from other jobs, we mostly present the results collected from the machine with AMD processors. Section 6.2 also has a performance comparison of ShaFEM on the two machines to show the performance consistency of our algorithm for different hardware.

Software: ShaFEM has been implemented using our computational method presented in Sections 4 and 5. Furthermore, we have applied some optimization techniques in [27] to improve the input and output processing: (1) the input transactions are preloaded and sorted before they are used to construct the XFP-tree; (2) the most frequent output values are preprocessed and stored in an indexed table; the similar part of two frequent patterns outputted consecutively is processed only once to reduce the computation of output reporting. In addition, we benchmarked ShaFEM, with other methods for mining association rule including FP-array [23], Apriori [1], Eclat [10], FP-growth [11] and FP-growth* [18] whose implementations are available at [29,30,39]. The algorithms were implemented using C/C++. ShaFEM and FP-array were parallelized using OpenMP. We used g++ 4.1.2 for compilation. The reported running time reflects the entire mining process including the data preprocessing time when they are read from the disk as well as the index table generation for output reporting.

6.1.1. Execution time

To demonstrate the efficiency of our proposed method, we study the performance of ShaFEM and compare it with FP-array which is one of the best parallel ARM methods for multi-core shared memory architectures developed by Intel [23]. This method is a component of the PARSEC Benchmark Suite [39]. FP-array inherited the mining features of FP-growth and was shown to run much faster than many mining methods including FP-growth [11], nonordfp [20], AIM2 [21], kDCI [42] and LCM2 [43] on sparse databases. Unlike ShaFEM that constructs a new data structure named XFP-tree, FP-array constructs the FP-tree in parallel and distributes its data to parallel processes using a tiling technique. Then, it converts the global FP-tree to arrays and mines frequent patterns from this data structure. We present in Fig. 13 the running time of ShaFEM and FP-array for the six test datasets. ShaFEM outperforms FP-array for all test cases with different number of cores and different datasets. ShaFEM runs 2.1–5.8 times faster than FP-array for the same number of parallel processes in most test cases. It is important to note that for large datasets such as Kosarak, this speedup of 2.8 for 12 cores translates to a saving of 12.8 hours. Sequentially, ShaFEM runs faster by 117.3 hours or 4.9 days. Although the size of Kosarak is smaller than Webdocs, this dataset was benchmarked with very low *minsup*. Therefore, its execution time was longer the one for Webdocs.

Table 5 shows the result of running ShaFEM and FP-array on two machines with Intel and AMD processors described in Table 4 using 12 cores. The results show that ShaFEM performs better than FP-array for all datasets on both machines.

6.1.2. Speedup

Fig. 14 shows the speedup on 12 cores of ShaFEM and FP-array compared with the sequential running time of FP-array. These results show that ShaFEM has run significantly faster than both sequential and parallel FP-array. Compared to the execution time of FP-array on one core, ShaFEM on 12 cores has performed 13–31.3 times faster while FP-array on 12 cores has been only 5.6–10.0 times faster than its sequential execution time.

Table 4
Test machines.

Name	Machine 1	Machine 2
Total cores	12	12
Num. of sockets	2	2
Cores/socket	6	6
Processor model	AMD Opteron 2747	Intel Xeon E5-2640
Architecture	Istanbul	Sandy Bridge-EP
Clock rate	2.2 Ghz	2.5 Ghz
LLC/socket	6 MB	15 MB
Memory	24 GB	128 GB
OS	Cent OS 5.8 (Linux)	CentOS 6.4 (Linux)

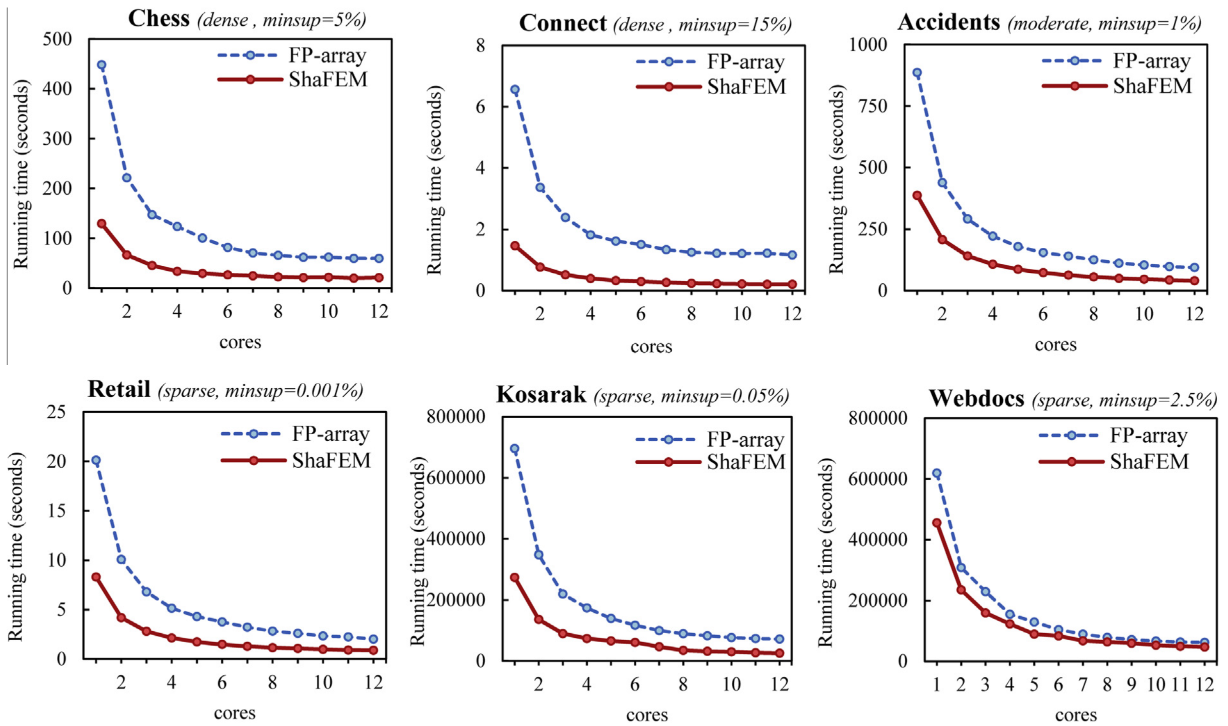


Fig. 13. Running time comparison of ShaFEM and FP-array.

Table 5
Time comparison (seconds) of ShaFEM vs. FP-array on different hardware when 12 cores were used.

Dataset	*Minsup* (%)	Time on machine 1 (AMD)			Time on machine 2 (Intel)		
		ShaFEM	FP-array	Time difference	ShaFEM	FP-array	Time difference
Chess	5	21	60	39	17	35	18
Connect	15	0.2	1.2	1	0.2	0.7	0.5
Accidents	1	41	94	53	32	53	21
Retail	0.001	0.9	2	1.1	0.7	1.5	0.8
Kosarak	0.05	25,941	72,092	46,151	19,719	40,713	20,994
Webdocs	2.5	47,631	62,893	15,262	32,434	36,274	3840

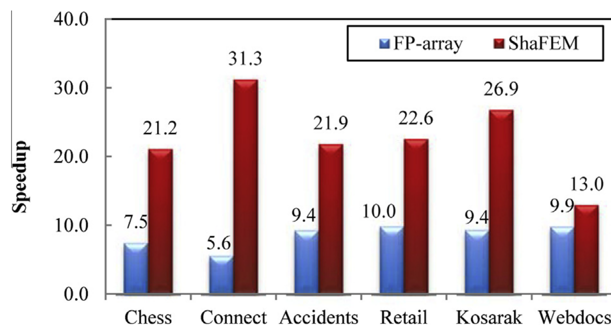


Fig. 14. Speedup of ShaFEM and FP-array on 12 cores relative to FP-array one core.

Fig. 15 shows the speedup of ShaFEM for different number of cores compared to its sequential time on one core. When all 12 cores were used, ShaFEM runs 6.1–10.6 times faster than it did 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

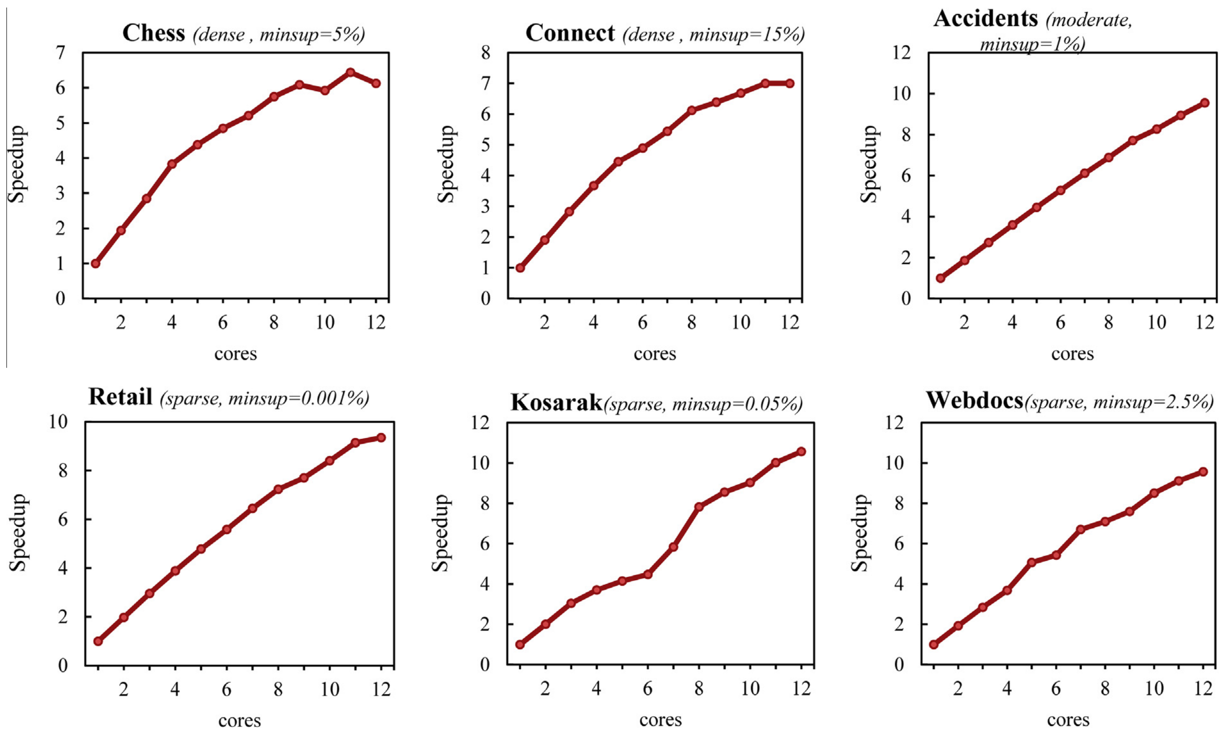


Fig. 15. Speedup of ShaFEM compared to its sequential time on 1 core.

speedup increases slower when 8–12 cores are used. This is due to the nature of the dense (compact) data structure used which make achieving good load balance for higher number of cores difficult. This feature is common for other parallel methods as well. For example, FP-array suffers from similar situation as can be seen from its speedup of 5.6 on 12 cores for dense dataset Connect.

6.1.3. Memory usage

In order to evaluate the memory usage of ShaFEM, we measure the peak memory usage in comparison to FP-array for the six datasets by using the *memusage* command of Linux. Fig. 16 shows their memory usage (megabytes) for different number of cores. As the figure shows, in most test cases ShaFEM consumes much less memory than FP-array. Specifically, the peak memory usage of ShaFEM was 1.5–7.1 times less than FP-array for Chess, Connect, Retails, Kosarak and Webdocs. For Webdocs dataset using 12 cores, ShaFEM used 7 GB of memory less than FP-array. The only case that ShaFEM used more memory than FP-array was for Accidents dataset; their memory usage difference was 23–39%.

Although the memory usage of ShaFEM is 23–39% higher than FP-array in this case, ShaFEM run faster than FP-array. The way in which the allocated memory is accessed and processed has a higher impact in the execution time than the memory usage. The new XFP data structure allows for more lock free parallel access and less memory contention and better cache locality, the bit vector data structure allows for better memory/cache alignment. Our profiling information for Accidents using the Code Analyst tool [47] shows that compared to FP-array, ShaFEM had 61% less data cache access, 59% less miss alignment access, 40% fewer branches and 21% less branch miss predictions.

The memory usage of both ShaFEM and FP-array increased as more parallel cores were employed, but the memory increasing level of ShaFEM was smaller than the FP-array. For the memory intensive problems like association rule mining, efficient utilization of memory can have significant impact on the execution time. Our implementation of ShaFEM minimizes the usage of memory, uses the bit vector structure to save memory and arranges data elements to increase data locality for cache optimization.

6.2. Sequential performance evaluation

6.2.1. Execution time

ShaFEM outperforms other well-known sequential algorithms for mining association for sparse and dense databases as well. This is demonstrated by benchmarking ShaFEM with other sequential algorithms including Apriori [1], Eclat [10], FP-growth [11], FP-growth* [18], FP-array [23]. Apriori, which is the most well known and most widely used method for association rule mining, explores the horizontal data format for its candidate generate-and-test mining strategy. Eclat uses

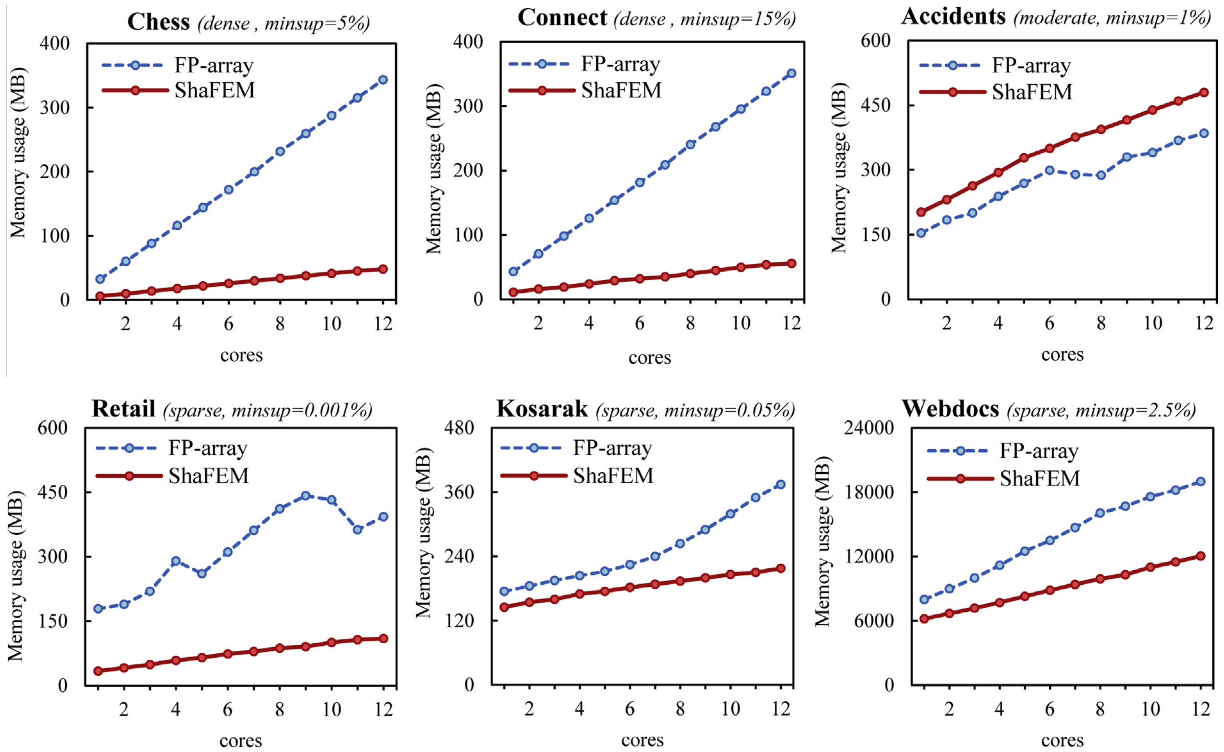


Fig. 16. Peak memory usage of ShaFEM and FP-array.

TID-list, a vertical data structure. FP-growth, FP-growth* and FP-array apply the mining without candidate generation strategy with FP-tree and arrays, horizontal data structures. ShaFEM is distinguished from other algorithms because it deploys two dynamic mining strategies compared to the single strategy of the other methods. In this experiment, all algorithms run on a single core; the test cases use a range of minimum support input values (*minsup*).

Results show that ShaFEM out performs the compared methods for all test cases. Also we can observe the unstable performance of the other methods for different database characteristics. Fig. 17 presents some results on Chess (dense) and Kosarak (sparse) datasets. ShaFEM is the best performing method in all cases. Apriori runs slowest on both datasets. FP-growth, FP-growth* and FP-array performed better than Eclat on the sparse dataset but worst on the dense ones.

6.2.2. Speedup

In order to observe how well ShaFEM performed compared to the other sequential algorithms, we computed the speedup of ShaFEM compared to Apriori, Eclat, FP-growth, FP-growth, FP-array by dividing their sequential execution times d by the sequential time of ShaFEM. Fig. 18 presents the speedup values for Chess (dense) and Kosarak (sparse) datasets. Even

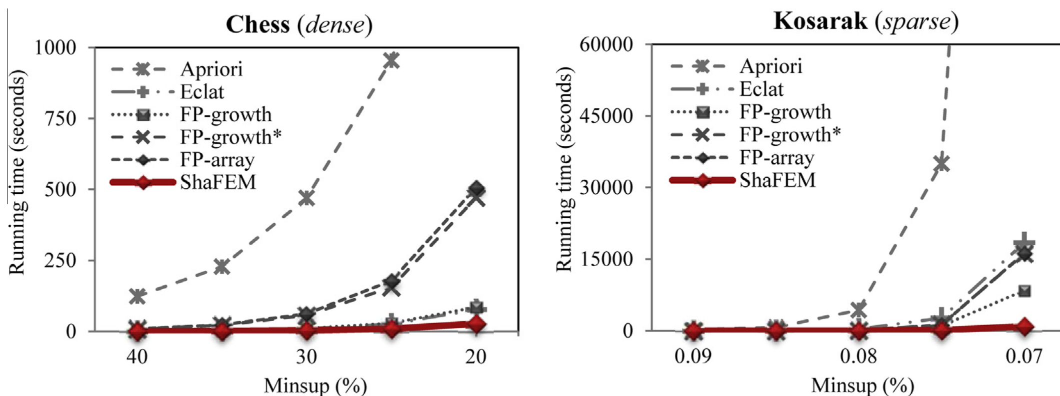


Fig. 17. Comparison of sequential execution time of different algorithms on sparse and dense datasets.

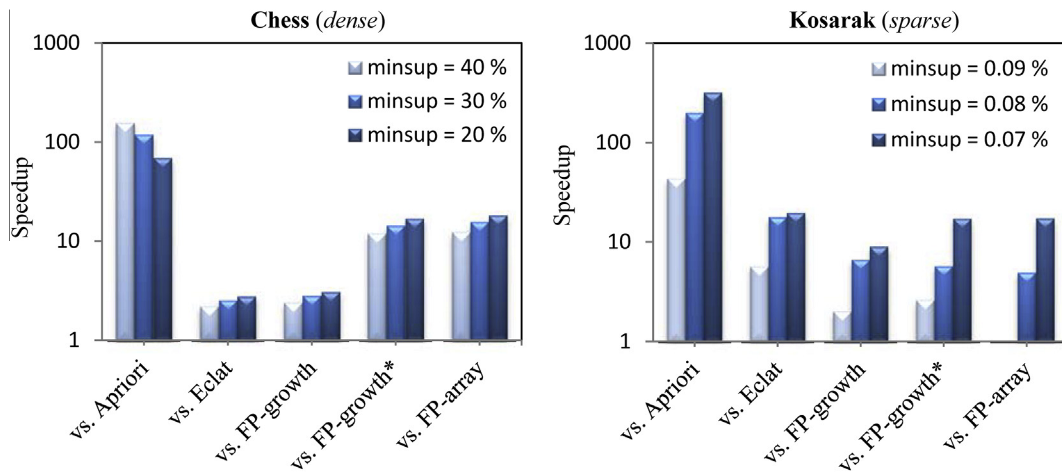


Fig. 18. Speedup of ShaFEM on one core compared to sequential algorithms.

sequentially, ShaFEM runs much faster than the compared methods on both dense and sparse dataset for various *minsup* inputs. For example, in our test cases, ShaFEM runs 43.8–323.4 times faster than Apriori; 2.2–19.9 times faster than Eclat; 2.0–9.0 times faster than FP growth, 2.6–17.4 times faster than FP-growth* and 1–18 times faster than FP-array for the range of *minsup* values indicated in Fig. 18.

6.3. Analyzing performance merits of ShaFEM

The above results have shown the efficiency of ShaFEM for different data types on both sequential and parallel systems. In this section, we analyze the key elements that play important roles in performance enhancement of ShaFEM including (1) the application of two mining strategies in ShaFEM helps the algorithm to adapt better to data characteristics; (2) the dynamic scheduling method of ShaFEM helps to obtain better load balance; (3) adoption of a new lock free approach in the construction of XFP-tree.

6.3.1. The impact of combining two mining strategies

To study the benefits of applying the two mining strategies in ShaFEM, we measured the mining time of ShaFEM in three separated cases: (1) using *MineFPTree* only, (2) using *MineBitVector* only and (3) using both *MineFPTree* and *MineBitVector* (i.e., our approach). From the experimental results (Fig. 19), we observe how and where the combination of the two mining strategies help to significantly improve the overall mining performance on sparse and dense databases compared to the cases that a single mining strategy has been used. It is important to note that note that a database itself may be considered sparse (or dense), but also that given a database its data subsets may have dense and sparse characteristics. For example, by applying both *MineFPTree* and *MineBitVector,* ShaFEM runs 2.1–8.9 times faster for Chess dataset and 4.7–6.4 times faster for Kosarak when compared with the cases of using single mining strategy, Fig. 19. This is explained by the ability of ShaFEM to select the suitable strategy (i.e. either *MineFPTree* or *MineBitVector*) for each subset of data being mined based on their characteristics.

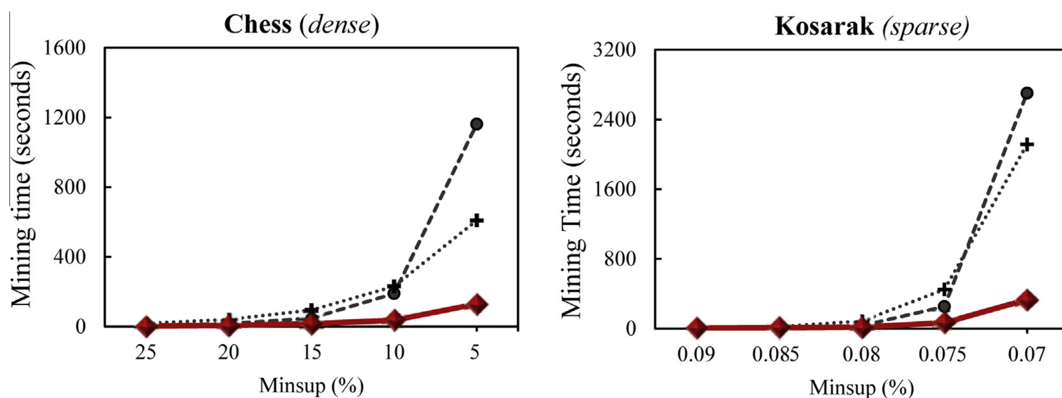


Fig. 19. Impact of combining two mining strategies of ShaFEM compared to using single strategy.

In Case 1, *MineFPTree* is applied to mine all data subsets although it is only suitable for the sparse ones. Performance loss occurs when *MineFPTree* mines the dense data subsets. In Case 2, all data subsets are mined using *MineBitVector* which is more suitable for dense data subsets. As a result, the performance loss occurs when it mines the sparse data portions. ShaFEM applies both mining strategies; it selects *MineFPTree* for the sparse and *MineBitVector* for the dense portions to improve the overall performance. Additionally, data characteristics of mining data also vary as *minsup* changes. For both Chess and Kosarak, Case 1 runs faster than Case 2 for the larger *minsup* values but slower for smaller *minsup*. This is because when *minsup* value reduces, more frequent patterns are generated and the number of small dense data subsets to be processed will be larger than the number of sparse data subsets making *MineBitVector* a more suitable option. In such a case, using *MineFPTree* only (Case1) not only result in a large performance loss but also it will perform worse than using *MineBitVector* only (Case 2). ShaFEM can detect the change of data characteristics to balance the use of its two mining strategies and hence run faster and stably for various *minsup* values.

For in-depth understanding of self-adaptive ability of ShaFEM to data characteristics, we measure the amount of time that ShaFEM spends on each of its two mining strategies separately when both strategies have been applied. Fig. 20 presents their percentage of time distribution for the six test datasets. The results show that both mining strategies of ShaFEM contributed to generate the frequent patterns. However, their percentage of contributions varied depending on the data characteristics of each dataset. The mining strategy using Bit Vector was utilized mostly for the dense datasets. However, the time percentage of this strategy reduces when the data were sparse. This workload distribution has been done automatically because our approach dynamical switches between the two mining strategies. The mining strategy using Bit Vector is more suitable for dense data because the low cost bitwise operations are used to generate the large number of frequent patterns which are usually found from this type of data. In addition, the bit vector data structure, which is more cache friendly and saves memory usage, can boost the mining performance. For the sparse portions of the datasets, the number of frequent patterns is less. Therefore, the mining strategy using FP-tree is a better choice because its mining approach does not require generating the very large number of infrequent candidate patterns.

6.3.2. The impact of dynamic scheduling

In two mining stages of ShaFEM, the second stage of generating frequent patterns usually accounts for most of the execution time. In this stage, load balancing is a critical issue. We use the divide and conquer approach for this stage to enhance parallelism by enabling parallel processes to work independently and select small work portions one at a time (Section 5.1). Because the workload of each portion varies, dynamic scheduling has been used for load balance. In order to study the

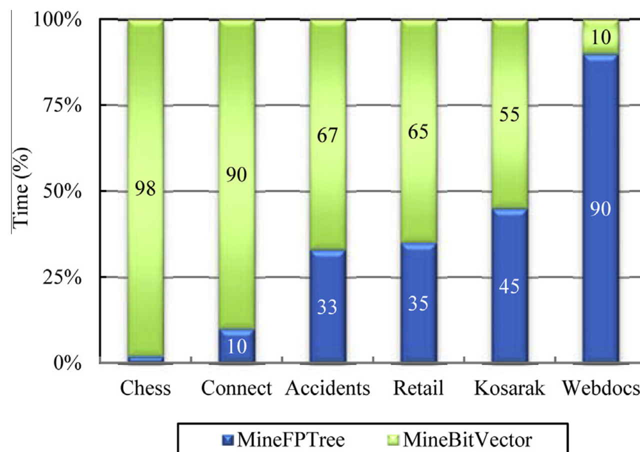


Fig. 20. Time distribution for two mining strategies of ShaFEM.

Table 6
Performance of ShaFEM using dynamic scheduling vs. static scheduling on 12 cores.

Databases	*Minsup* (%)	Static scheduling (1) (sec.)	Dynamic scheduling (2) (sec.)	Time difference (3) = (1) − (2) (sec.)	Performance improvement (4) = (3) ∗ 100/(2) (%)
Chess	5	45	21	24	114
Connect	15	0.51	0.2	0.31	155
Accidents	1	161	41	120	293
Retail	0.001	5.3	0.9	4.4	489
Kosarak	0.055	15,802	6902	8900	129
Webdocs	3	48,183	7862	39,086	497

Table 7
Performance of ShaFEM using lock vs. lock-free on 12 cores.

Databases	*Minsup* (%)	Build FP-tree with lock (1) (sec.)	Build XFP-tree without lock (2) (sec.)	Time difference (3) = (1) − (2) (sec.)	Performance Improvement (4) = (3) ∗ 100/(2) (%)
Chess	5	21.38	21.2	0.18	0.8
Connect	15	0.2	0.2	0	0.0
Accidents	1	42.69	41	1.69	4.1
Retail	0.001	1.14	0.9	0.24	22.2
Kosarak	0.055	8033	6902	1131	16.4
Webdocs	3	8789	7862	927	11.8

efficiency of this scheduling method, we measured ShaFEM in two cases: (1) using static scheduling and (2) using dynamic scheduling and report the results in 6. Applying dynamic scheduling can increase the mining performance 114–489% in our test cases (see Table 6).

6.3.3. The impact of constructing XFP-tree

ShaFEM applies a lock-free approach to construct XFP-tree, a new data structure derived from FP-tree to reduce the synchronization need and enhance the parallelism. In order to evaluate the performance benefits of this approach, we implement a variant of ShaFEM in which parallel processes construct a single shared FP-tree (instead of XFP-tree) by using a lock for each node of the tree. When a node is updated, the lock of that node will be activated until the updating is complete. Table 7 presents the execution time of ShaFEM in two cases: (1) build FP-tree with lock; (2) build XFP-tree without lock (i.e. our proposed solution). The results show that the application of XFP-tree helps increase the mining performance up 22.2%.

7. Conclusion

We have presented ShaFEM, a novel parallel method for association rule mining on multi-core share memory machine, and its efficiency on different database types via a number of experimental results on a 12-core machine. This dynamic parallel method that combines two mining strategies runs faster and consumes less memory than the state-of-the-art methods. It performs stably on both sparse and dense databases. This method can be used to implement the association rule mining component of databases management systems and statistical software like Oracle RDBMS, MS. SQL Server, IBM DBS2, R, SAS, SPSS Clementine, etc. as well as various applications to help the mining task self-adapt to the data characteristics and utilize the benefits of shared memory in multi-core computers. We will integrate ShaFEM into a mining framework that will exploit a combination of distributed memory and shared memory computational models to enable this mining task on very large computer cluster systems.

References

[1] R. Agrawal, R. Srikant, Fast algorithms for mining association rules, in: Proceedings of the 20th International Conference on Very Large Databases, 1994, pp. 487–499.
[2] S. Brin, R. Motwani, C. Silverstein, Beyond market basket: generalizing association rules to correlations, in: Proceedings of ACM SIGMOD International Conference on Management of Data (Jun. 1997), vol. 26, no. 2, 1997, pp. 265–276.
[3] C. Silverstein, S. Brin, R. Motwani, J. Ullman, Scalable techniques for mining causal structures, J. Data Min. Knowl. Discovery 4 (2-3) (2000) 163–192.
[4] R. Agrawal, R. Srikant, Mining sequential patterns, Proc. Data Eng. 1995 (1995) 3–14.
[5] H. Mannila, H. Toivonen, A.I. Verkamo, Discovery of frequent episodes in event sequences, J. Data Min. Knowl. Discovery 1 (3) (1997) 259–289.
[6] J. Han, G. Dong, Y. Yin, Efficient mining of partial periodic patterns in time series database, in: Proceedings of IEEE Data Engineering (Mar. 1999), 1999, pp. 106–115.
[7] J. Han, H. Cheng, D. Xin, X. Yan, Frequent pattern mining: current status and future directions, J. Data Min. Knowl. Discovery 15 (1) (2007) 55–86.
[8] D. Burdick, M. Calimlim, J. Flannick, J. Gehrke, T. Yiu, MAFIA: a maximal frequent itemset algorithm, IEEE Trans. Knowl. Data Eng. 17 (11) (2005) 1490–1504.
[9] H. Li, Y. Wang, D. Zhang, M. Zhang, E. Chang, PFP: parallel FP-growth for query recommendation, in: Proceedings of the 2008 ACM Conference on Recommender systems, 2008, pp. 107–114.
[10] M. Zaki, S. Parthasarathy, M. Ogihara, W. Li, New algorithms for fast discovery of association rules, Proc. Knowl. Discovery Data Min. 1997 (1997) 283–286.
[11] J. Han, J. Pei, Y. Yin, Mining frequent patterns without candidate generation, in: Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data (Jun. 2000), vol. 29, no. 2, 2000, pp. 1–12.
[12] J.S. Park, M.S. Chen, P. Yu, An effective hash-based algorithm for mining association rules, in: Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data (May 1995), vol. 24, no. 2, 1995, pp. 175–186.
[13] H. Toivonen, Sampling large databases for association rules, in: Proceedings of the 1996 International Conference on Very Large Data Bases, 1996, pp. 134–145.
[14] S. Brin, R. Motwani, J.D. Ullman, S. Tsur, Dynamic itemset counting and implication rules for market basket analysis, in: Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data, vol. 26, no. 2, 1997, pp. 255–264.
[15] A. Fiat, S. Shporer, AIM: another itemset miner, in: Proceedings of the 2003 Workshop on Frequent Itemset Mining Implementations, 2003.
[16] M.J. Zaki, K. Gouda, Fast vertical mining using diffsets, in: Proceedings of the 2003 ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, 2003, pp. 326–335.
[17] C. Borgelt, An implementation of the FP-growth algorithm, in: Proceedings of the 1st Workshop on OSDM: Frequent Pattern Mining Implementations, Aug. 2005, pp. 1–5.

[18] G. Grahne, J. Zhu, Efficiently using prefix-trees in mining frequent itemsets, in: Proceedings of the 2003 Workshop on Frequent Pattern Mining Implementations, 2003, pp. 123–132.
[19] J. Pei, J. Han, H. Lu, S. Nishio, S. Tang, D. Yang, Hmine: hyper-structure mining of frequent patterns in large databases, in: Proceedings of the IEEE International Conference on Data Mining (Nov. 2001), 2001, pp. 441–448.
[20] B. Racz, Nonordfp: An FP-growth variation without Rebuilding the FP-tree, in: Proceedings of the IEEE ICDM Workshop on Frequent Itemset Mining Implementations (Nov. 2004), 2004.
[21] S. Shporer, AIM2: improved implementation of AIM, in: Proceedings of the IEEE Workshop on Frequent Itemset Mining Implementations (Nov. 2004), 2004.
[22] L. Schmidt-Thieme, Algorithmic features of eclat, in: Proceedings of the IEEE Workshop on Frequent Itemset Mining Implementations (Nov. 2004), 2004.
[23] L. Liu, E. Li, Y. Zhang, Z. Tang, Optimization of frequent itemset mining on multiple-core processor, in: Proceedings of the 33rd International Conference on Very Large Databases, 2007, pp. 1275–1285.
[24] W. Li, A. Mozes, Computing frequent itemsets inside oracle 10g, in: Proceedings of the 30th International Conference on Very Large Databases, 2004, pp. 1253–1256.
[25] C. Utley, 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>.
[26] T. Yoshizawa, I. Pramudiono, M. Kitsuregawa, SQL based association rule mining using commercial RDBMS (IBM db2 UBD EEE), in: Proceedings of the 2nd International Conference on Data Warehousing and Knowledge Discovery, 2000, pp. 301–306.
[27] L. Vu, G. Alaghband, A fast algorithm combining FP-tree and TID-list for frequent pattern mining, in: Proceedings of the 2011 International Conference on Information and Knowledge Engineering (Jul. 2011), 2011, pp. 472–477.
[28] R. Agrawal, T. Imielinski, A. Swami, Mining association rules between sets of items in large databases, in: Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data (Jun. 1993), vol. 22, no. 2, 1993, 207–216.
[29] Frequent Itemset Mining Implementations Repository, Workshop on frequent itemset mining implementation, 2003–2004. Available at: <http://fimi.ua.ac.be>.
[30] C. Borgelt, Frequent pattern mining implementations. Available at: <http://www.borgelt.net>.
[31] L. Vu, G. Alaghband, Mining frequent patterns based on data characteristics, in: Proceedings of the 2012 International Conference on Information and Knowledge Engineering (Jul. 2012), 2012, pp. 369–375.
[32] L. Vu, G. Alaghband, High performance frequent pattern mining on multi-core cluster, in: Proceedings of the 2012 IEEE International Conference on Collaboration Technologies and Systems (May 2012), 2012, pp. 630–633.
[33] R. Rabenseifner, G. Hager, G. Jost, Hybrid MPI/OpenMP parallel programming on clusters of multi-core SMP nodes, in: Proceeding of the 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing (Feb. 2009), 2009, pp. 427–436.
[34] M.J. Zaki, Parallel and Distributed Association Mining: A Survey, IEEE Concurr. J. 7 (4) (Oct.–Dec. 1999) 14–45.
[35] R. Garg, P.K. Mishra, Some observations of sequential, parallel and distributed association rule mining algorithms, In: IEEE Proceeding of the 2009 International Conference on Computer and Automation Engineering (March 2009), 2009, pp. 336–342.
[36] D. Chen, C. Lai, W. Hu, W. Chen, Y. Zhang, W. Zheng, Tree partition based parallel frequent pattern mining on shared memory systems, in: Proceeding of the 20th International Conference on Parallel and Distributed Processing, 2006, pp. 313–320.
[37] H.D.K. Moonesinghe, M.J. Chung, P.N. Tan, Fast Parallel Mining of Frequent Itemsets, Michigan State University.
[38] S.K. Tanbeer, C.F. Ahmed, B.S. Jeong, Parallel and distributed frequent pattern mining in large databases, in: Proceeding of the 11th IEEE International Conference on High Performance Computing and Communications, 2009, pp. 407–414.
[39] C. Bienia, 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>.
[40] J. Li, Y. Liu, W. Liao, A. Choudhary, Parallel Data Mining Algorithms for Association Rules and Clustering, CRC Press, 2006. pp. 3–5.
[41] O.R. Zaiane, M. El-Hajj, P. Lu, Fast parallel association rule mining without candidacy generation, in: Proceedings of the IEEE 2001 International Conference on Data Mining (ICDM, 2001), 2001, pp. 665–668.
[42] S. Orlando, C. Lucchese, P. Palmerini, R. Perego, F. Silvestri, KDCI: a multi-strategy algorithm for mining frequent sets, in: Proceedings of ICDM Workshop on Frequent Itemset Mining Implementations, 2003.
[43] T. Uno, M. Kiyomi, H. Arimura, LCM ver. 2: efficient mining algorithms for frequent/closed/maximal itemsets, in: Proceedings of ICDM Workshop on Frequent Itemset Mining Implementations, 2004.
[44] M. Hahsler, B. Grün, K. Hornik, C. Buchta, arules – A computational environment for mining association rules and frequent item sets, J. Stat. Softw. 14 (15) (Oct. 2005).
[45] L.E. Hen, S.P. Lee, Performance analysis of data mining tools cumulating with a proposed data mining middleware, J. Comput. Sci. 4 (2008) 826–833.
[46] L. Vu, G. Alaghband, Novel parallel method for mining frequent patterns on multi-core shared memory systems, in: Proceedings of the 2nd International Workshop on Data-Intensive Scalable Computing Systems, Nov. 2013, pp. 49–54.
[47] Code Analyst, AMD. Available at: <http://developer.amd.com/tools-and-sdks/archive/amd-codeanalyst-performance-analyzer/>.