Effect of Adaptive Interval Configuration on Parallel Mining Association Rules*

HU Kan[1] CHEUNG D W[2] XIA Shao-wei[1]

[1](Department of Automation Tsinghua University Beijing 100084)

[2](Department of Computer Science The University of Hong Kong Hong Kong)

E-mail: swxia@mail.tsinghua.edu.cn/dcheung@cs.hku.hk

Abstract All proposed parallel algorithms for mining association rules follow the conventional level-wise approach. It imposes a synchronization in every iteration in the computation which degrades greatly their performance if they are used to compute the rules on a shared-memory multi-processor parallel machine. The deficiency comes from the contention on the shared I/O channel when all processors are accessing the channel synchronously in every iteration. An asynchronous algorithm APM has been proposed for mining association rules on shared-memory multi-processor machine. All participating processors in APM generate candidates and count their supports independently without synchronization. Furthermore, it can finish the computation with fewer passes of database scanning than required in the level-wise approach. An optimization technique has been developed to enhance APM so that its performance would be insensitive to the data distribution. Two variants of APM and the synchronous algorithm Count Distribution, which is a parallel version of the popular serial mining algorithm Apriori, have been implemented on an SGI Power Challenge SMP parallel machine. The results show that the asynchronous algorithm APM performs much better, and is more scalable than the synchronous algorithm.

Key words Association rule, data mining, parallel mining, shared-memory multiprocessor, transactional database.

Mining association rules in large databases is an important problem in data mining[1~8]. The problem can be reduced to finding all large itemsets with respect to a given support threshold. Large itemsets are determined by computing the supports of the candidate itemsets. For this purpose, the mining process has to scan a large database multiple times and to search through many candidates for the large itemsets. Because of the amount of I/Os and computation, mining association rules in general is very costly. Therefore, there is a practical need to develop parallel algorithms for this task. Many parallel algorithms for mining association rules have been proposed for parallel machines with distributed shared-nothing memory[9~11]. In this work, we propose to solve

---

* The research is supported by the National Natural Science Foundation of China (国家自然科学基金, No. 79970052) and by RGC (the Hong Kong Research Grants Council)(No. 338/065/0032). **HU Kan** was born in 1970. He is a research associate at Department of Computer Science, Simon Fraser University, Canada. He received a Ph. D. degree on System Engineering from Tsinghua University in 1998. His research interests are data mining, data warehousing and decision support systems. **CHEUNG D W** is an associate professor of Department of Computer Science, The University of Hong Kong. His current research areas include data mining on large data warehouse. **XIA Shao-wei** was born in 1932. She is a professor and doctoral supervisor of Department of Automation, Tsinghua University. Her current research areas include large systems theory and application, neural computing and data mining.

Manuscript received 1998-01-20, accepted 1998-08-05.

this problem on shared-memory multi-processor parallel machines such as the SGI Power Challenge or Sun Enterprise.

In a parallel system with distributed shared-nothing memory, the database is partitioned and distributed across the local disks of the processors. The processors rely on communication to coordinate their tasks. All proposed algorithms for mining association rules in this model focus their design on tackling the following three issues:

(1) reduction of communication cost;

(2) reduction of computation by pruning unnecessary candidate sets;

(3) ensuring all candidates can reside in the memory by partitioning candidate sets across the processors.

One common aspect of all these algorithms is the adoption of a synchronous protocol to coordinate the exchange of support counts of the candidate itemsets at the end of every iteration. Their computation of the large itemsets is organized by iterations such that all large itemsets of the same size are computed together at the same iteration. In the following, we will explain that these synchronous algorithms are not suitable in the shared-memory model.

In SMP (shared-memory multiprocessor parallel) machine, processors communicate with each other through shared variables, the communication cost becomes a minor factor for the performance. On the contrary, the mining performance is dominated by the I/O cost. All general purpose SMP machines use shared storage. The processors in these machines not only share the memory but also share the I/O channels. In the distributed memory model, processors access their own database partitions independently which are stored on their local disks. There is no contention on the I/O and synchronous algorithm is ideal for this case. In the SMP case, the database will still be partitioned. However, these partitions are stored in the same shared storage system and accessed by the corresponding processors via the same I/O channels. A synchronous algorithm applied to this situation will create I/O contention in every iteration, which has a serious impact on the performance. A major goal of this work is to propose an asynchronous algorithm for this task for the shared-memory model to reduce the I/O contention.

A direct extension of the representative sequential association rule mining algorithm apriori to the distributed memory parallel model has been developed. Its implementation on the IBM SP2 is called CD (count distribution)[9]. CD by nature is a synchronous algorithm. A variant of CD adopted to the shared-memory model has been proposed in Ref. [12]. This work concentrated on parallel candidates generation and the hash tree formation. However, since it is a synchronous algorithm, its performance suffers heavily from the I/O contention. From what we know, no asynchronous algorithm has been proposed for mining association rules on an SMP machine.

In this paper, we will propose an asynchronous parallel algorithm APM (asynchronous parallel mining) for mining association rules on SMP machine. APM has the following merits:

(1) candidate sets are generated asynchronously by different processors and stored in a shared data structure in the shared memory;

(2) there is no major contention between the processors in accessing the shared data structure;

(3) candidate sets pruning is done by individual processors independently without synchronization;

(4) supports of candidates are counted asynchronously by all processors independently against their own partitions, which reduces I/O contention significantly;

(5) the number of passes with which the database is scanned is much less than that required in a synchronous algorithm such as CD.

The candidate set generation technique used in APM is the dynamic candidate generation methods introduced in the DIC (dynamic itemset counting) algorithm[3] for sequential mining. It is a break-away from the conventional level-wise candidate set generation technique used in Apriori. Compared with Apriori, it generates candidates and starts to count their supports at a much earlier time. If the data distribution is homogeneous, DIC can compute the large itemsets in fewer passes than the level-wise Apriori algorithm. One problem of dynamic candidate generation is the possibly larger number of candidates generated. To remedy this, we introduce a pruning technique by which a processor can asynchronously prune candidates by estimating upper bounds of their supports. In our performance studies, we found out that APM is much more efficient than CD because of the reduced I/O contention, fewer passes in database scanning and a much smaller set of candidates.

However, the dynamic candidate generation technique in DIC is very sensitive to the data distribution. It divides a database into intervals and generates candidates from these intervals instead of the entire database. If the distribution of the itemsets in these intervals is very similar to that in the entire database, then the candidates generated from the intervals will contain very few false hits and DIC will be very efficient; otherwise, the gain from using the dynamic technique will be greatly reduced by the overheads from the counting of the large number of unpromising candidates. To overcome this problem, we have developed an optimization technique called AIC (adaptive interval configuration) to preprocess the database partitions for APM. The output of the optimization is a set of intervals for each partition such that the itemset distributions in these intervals are more homogeneous than without the optimization. The algorithm APM with this enhancement is called APM-AIC.

We have implemented APM, APM-AIC and CD on an SGI Power Challenge shared-memory multi-processor machine with 8 processors and carried out an extensive performance study. We found out that APM and APM-AIC have a much better performancethan CD. APM-AIC has a more stable behavior and is in general 3 times faster than CD. Our results also show that the main problem in CD is the synchronous I/O contention. And the contention in APM and APM-AIC is significantly reduced because of the asynchronous access mode. We also performed speedup study on the parallelism of the three algorithms. Both APM and APM-AIC have nice parallel performance.

The rest of this paper is organized as follows. Section 1 overviews the parallel mining of association rules. The technique of dynamic candidate generation is described in Section 2. In Section 3, we discuss the details of the APM algorithm. Adaptive Interval Configuration and an enhanced APM algorithm are presented in Section 4. Section 5 reports the results of an extensive performance study. Section 6 is the conclusion.

1 Parallel Mining of Association Rules

1.1 Association rules

Let $I=\{i_1,i_2,\ldots,i_m\}$ be a set of items and D be a database of transactions, where each transaction T consists of a set of items such that $T\subseteq I$. An association rule is an implication of the form $X\Rightarrow Y$, where $X\subseteq I$, $Y\subseteq I$ and $X\cap Y=\varnothing$[1]. An association rule $X\Rightarrow Y$ has support s in D if the probability of a transaction in D containing both X and Y is s. The association rule $X\Rightarrow Y$ holds in D with confidence c if the probability of a transaction in D which contains X but also contains Y is c. The task of mining association rules is to find all the association rules whose support is larger than a given minimum support threshold and whose confidence is larger than a given minimum confidence threshold. For an itemset X, $X._{sup}$ stands for its support count in database D, which is the number of transactions in D containing X. An itemset $X\subseteq I$ is *large* (or *frequent*) if $X._{sup}\geqslant \text{min}sup\times|D|$, where min*sup* is the given minimum support threshold. For the purpose of presentation, we sometimes use support to stand for support count of an itemset. It has been shown that the problem of mining association rules can be reduced to finding all large itemsets for a given minimum support threshold.

1.2 Synchronous parallel mining algorithms

Apriori is the most representative sequential algorithm for mining association rules[2]. It relies on the apriori_gen function to generate the candidate sets at each iteration. CD (count distribution) is a parallel version of Apriori proposed for the distributed memory parallel systems[9]. In this model, the database D is partitioned into $D_1, D_2, \ldots, D_n$, distributed and stored on the local disks of n processors. At every iteration (k-th iteration), each processor first computes the same candidate set C_k from L_{k-1}, the set of large itemsets found at the ($k-1$)-th iteration, and then scans its own partition to compute the local supports of the candidates in C_k. All the processors then exchange their local supports by performing synchronous broadcasts. Following that, each processor computes the global supports of the candidates and finds out the globally large itemsets of size k. CD repeats these steps until no new candidate is generated. Note that the support count exchanges in CD are synchronized in every iteration because each node must receive the counts from all other processors before it can determine the globally large itemsets.

Several other parallel algorithms which use the data distribution approach instead of the count distribution approach have been proposed. IDD (intelligent data distribution) and HD (hybrid distribution) are the representatives[11]. The main idea in the data distribution approach is to partition candidates across the processors. Each processor is responsible for counting the supports of a subset of candidates assigned to it. Because of the distribution of the counting responsibility, data in different partitions may have to be shipped across the network to other processors. This approach is feasible only if the network bandwidth is large enough for moving the partitions around. It is important to point out that all the above proposed parallel algorithms are synchronous and not suitable for shared-memory parallel systems mainly because of the I/O contention problem.

2 Dynamic Candidate Generation

The asynchronous candidate sets generation technique used in APM is a modification of the dynamic candidate generation technique developed in Ref. [3]. It was developed for sequential mining of association rules in a centralized database and was integrated into the DIC (dynamic itemset counting) algorithm[3]. Most association rule mining algorithms use the level-wise approach established in Apriori to generate candidates by size. Following that approach, no size-($k+1$) candidates containing $k+1$ items can be generated before all size-k large itemsets are found. As a consequence, size-k candidates will only be generated and counted in k-th iteration, and the database must be scanned as many times as the length of the maximum size large itemset. Dynamic candidate generation technique uses an aggressive approach to avoid this level-wise problem by generating candidates with partial information. It divides the database into a number of equal size intervals and assign checkpoints to the boundaries between the intervals (Fig. 1). Dynamic algorithm scans the database starting from the first interval. In the scanning of an interval, all candidate itemsets found large in the interval will be used as generators to generate candidates at the checkpoints for the next interval. In the first interval, the generators will be the size-1 large itemsets in the interval. In the subsequent intervals, size-k generators found in one interval will be used to generate size-($k+1$) candidates for the next interval. Therefore, candidates will be generated by intervals instead of levels, and supports of candidates of different sizes will be counted together as soon as they are generated. Furthermore, the total support of a candidate can be found after the scanning has wrapped around the database with respect to the checkpoint at which the candidate was generated. The scanning of the database can be terminated when supports of all candidates are counted. Compared with Apriori, dynamic algorithm could generate candidates and discover large itemsets much earlier. Hence, it may require fewer

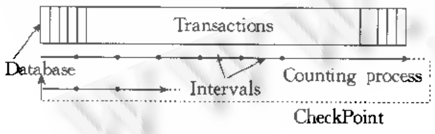

Fig. 1　Dynamic candidate generation

passes of database scanning than Apriori.

2.1 Performance issues on dynamic technique

The dynamic candidate generation technique is an aggressive and optimistic approach. It uses partial information to generate candidates from intervals. There are two issues which affect the performance of this technique. The first one is the interval size used in the dynamic candidate generation, i. e. , the distance between two consecutive checkpoints. With respect to the whole database, an interval is similar to a sample from the database. The larger the interval size is, the better it may represent the whole database. Then the dynamic approach may generate fewer unnecessary candidates (false hits) and hence incur less overheads in the counting. However, it would perform more passes in the scanning of the database. On the other hand, a smaller interval size advances the candidate set generation earlier, and hence may require fewer passes over the database. However, it may generate more false hits, and the checking overhead may offset the saving in scanning. So the choice of a proper interval size is a key factor in the performance of dynamic technique.

Data distribution in the database is another factor affecting the performance of dynamic approach. In an ideal scenario, the distributions of itemsets in all the intervals are very similar to each other, and hence very few new candidates are generated from the later intervals in the scanning. Therefore, the scanning could be finished with fewer passes. On the other hand, if the distributions are not very uniform over the intervals, many new candidates may be generated in a much later stage which would prolong the scanning of the database. In the worst case, the number of passes required could be equal to that in Apriori. Therefore, a more homogeneous distribution of itemsets over the intervals is desirable in the application of the dynamic candidate generation. In order to have a more homogeneous distribution, a random access of the transactions could be used. However, as pointed out in Ref. [3], this approach could be expensive and may not work in some cases. To tackle these two performance issues, we introduce some techniques in APM to dynamically create a near optimal interval size and to control the scanning of the transactions in the database to create a logically homogeneous distribution across the intervals. Details will be discussed in Section 4.

3 Asynchronous Parallel Mining

In a shared-memory parallel machine, the model for computing the large itemsets is called common candidates partitioned database. The database D is stored in the shared storage system and divided logically into partitions $D_1, D_2, \ldots, D_n$, where n is the number of processors. Each processor counts the supports of the common candidates against its own partition. Results of the counting are stored in a shared data structure in the memory.

3.1 Asynchronous dynamic candidates generation and counting

APM uses the dynamic candidates generation technique to generate the candidates asynchronously. In order to store candidates of different sizes, a trie instead of a hash tree is used to store the supports. Figure 2 is an example of the trie structure. In it, every node is associated with a candidate itemset. For example, in the first branch of the trie, the nodes store the support for the candidates A, AB, ABC, AC, and AD.

In order to generate candidates, each partition is divided into smaller intervals. The relations between the database, partitions and intervals are illustrated in Fig. 3. An itemset X is *locally large* in a partition D_i if it is large with respect to the number of transactions in D_i. An itemset X is *interval large* in an interval I if it is large with respect to the number of transactions in I, i. e. , its support count in I is not less than $s \times |I|$, where s is the support threshold.

For the simplicity of the discussion, we assume the trie contains all the items (i. e. size-1 itemsets) initially as the start up set of candidates (This start up condition will be modified for performance reason in Section

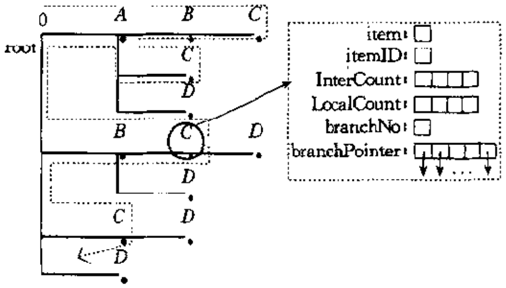

Fig. 2 Trie and its node

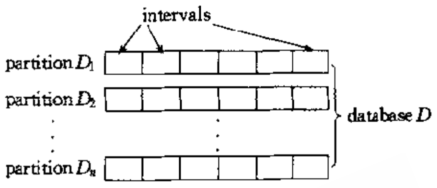

Fig. 3 Database, partitions and intervals

3.4). All the processors scan their own partitions in parallel. For each processor, it starts its scanning in the first interval and counts the supports of the candidates in the trie. The local supports of the candidates with respect to different partitions are stored separately on the trie. In the first checkpoint, the processor determines which itemsets on the trie are *interval large*. These interval large itemsets are then used to generate new candidates for the next interval. They are divided into groups by their sizes and the apriori_gen function[2] is applied to them separately to generate new candidates. The new candidates are stored in the trie once they are generated at the checkpoint. In the subsequent scanning of the other intervals, the same generation and counting processes are repeated by intervals, and candidates are generated from the interval large itemsets found in the previous interval at every checkpoint. Note that candidates generated by one processor is shared by other processors once they are inserted on the trie. All processors perform this scanning and counting cycle in their own partition until all candidates on the trie have been counted by all the processors and no new candidate is generated. Some status information for each processor is stored on the shared trie (Fig. 2) to facilitate the asynchronous candidate generation and counting. Different processors can perform new itemset insertion and support count updates concurrently on the trie. The only constraint is that the local support of a newly inserted itemset by a processor will only start to be counted by another processor on its partition when the latter one starts a new interval scanning cycle. This can be handled by a simple ready flag.

3.2 Asynchronous candidate set pruning

It is useful to prune away as many candidates as possible at a checkpoint before a scanning cycle starts. The technique of *global* pruning proposed for distributed memory system is very effective[10]. However, it relies on the support counts of itemsets of one level to estimate the supports of the candidates in the next level to perform the pruning. Its original approach suits only the synchronous mode of operation. We modify it for the asynchronous model.

Suppose D_i, $1\leqslant i\leqslant n$, are the partitions of a database D. For an itemset W, let us use $W._{sup(i)}$ to denote the local support of W in D_i, $1\leqslant i\leqslant n$. Suppose X is a size-k candidate set. If $Y\subset X$, then $Y._{sup(i)}\geqslant X._{sup(i)}$, $1\leqslant i\leqslant n$. Therefore, the local support of X in D_i, $X._{sup(i)}$, is bounded by the value $\min\{Y._{sup(i)}\,|\,Y\subset X$, and $|Y|=k-1\}$. Hence, the value

$$X._{maxsup}=\sum_{i=1}^{n}X._{maxsup},$$

where $X._{maxsup(i)}=\min\{Y._{sup(i)}\,|\,Y\subset X$, and $|Y|=k-1\}$ is an upper bound of the global support of X. If $X._{maxsup}<s\times|D|$, where s is the support threshold, then X can be pruned away. This is referred to as global pruning in Ref. [10].

Note that global pruning requires the local support counts of all the size-$(k-1)$ subsets of X to compute the bound. In the asynchronous case, when a new candidate is generated by a processor, the local supports of some of its subsets may not be available on the trie. We use a more conservative approach to estimate the bound in this case.

Suppose we want to determine whether a candidate X can be pruned at a checkpoint. Let $Y \subset X$. Also suppose a partition D_i, $1 \leqslant i \leqslant n$, has been divided into m intervals and each interval has the length of M transactions. If Y has been counted over t intervals in D_i, and the accumulated support of Y over the t intervals in D_i is $Y._{sup(i,t)}$, then $Y._{sup(i)}$ is bounded by $Y._{sup(i,t)} + M \times (m-t)$. With this bound, we can compute a bound on $X._{sup}$ similar to $X._{maxsup}$. If the bound is less than $s \times |D|$, then X can be pruned away. We call this technique ***upper-bound pruning***. This pruning technique would have a better effect on a candidate X if most size-$(k-1)$ subsets of X have been counted in most of the partitions.

3.3 Removal of small itemsets on the trie

The size of the trie is an important performance factor, which affects the cost of the traversing. A processor can traverse the trie at a checkpoint to identify both the globally large and globally small itemsets. All globally small itemsets found are removed by the processor before it starts a new counting cycle. All the supersets of a small itemset on the same branch can also be removed. For example, if *BC* is globally small, then its superset *BCD* on the same branch (Fig. 2) should also be removed. A simple locking mechanism is needed to ensure that no processor is traversing inside the branch to be removed.

3.4 The APM algorithm

We present the APM algorithm as

```
/* start up condition: all processors scan their partitions to compute local supports of size-1 itemsets;
    then compute $L_1$ and $C_2$ and insert the sets in $L_1$ and $C_2$ in the shared trie */
/* parallel execution: every processor $i$ runs the following fragment on its partition $D_i$ */
(1) while (some processor has not completed the counting on all the itemsets on the trie on its partition)
(2)     { while ( processor $i$ has not completed the counting on all the itemsets on the trie on $D_i$)
(3)         { scan one interval on partition $D_i$ to count supports of itemsets on the trie;
(4)           compute interval large itemsets for the interval scanned;
(5)           generate new candidate itemsets from these interval large itemsets;
(6)           perform upper-bound pruning on these new candidates;
(7)           insert the remaining candidates in the trie;
(8)           remove globally small itemsets found on the trie;
(9)         }
(10)    }
```

The program fragment will be executed by each processor i on it partition D_i, $1 \leqslant i \leqslant n$. We have modified the start up condition of APM. A master processor is assigned to compute L_1 from the private counter arrays which have stored the local support counts of all size-1 items for different partitions. Then C_2 is generated, and the master processor will build a shared trie containing the itemsets in L_1 and C_2. Also the itemsets in C_2 would undergo a global pruning before they are inserted in the trie. With this initialization, the trie would have a smaller size to start with. This will introduce a single round of synchronization. However, this is a tradeoff between memory usage and I/O cost.

4 Adaptive Optimization of APM

The performance of APM will also be influenced by the choice of the interval size and the data distribution across the intervals. We proposed an optimization technique named Adaptive Interval Configuration to preprocess the database partitions to derive a near optimal interval size and interval division. The optimization will be performed independently on each partition by its own processor. Initially, a partition would be divided into a sequence of intervals. These intervals could be aligned with the pages on the storage system for performance sake. The homogeneity of the itemset distributions in these intervals could be increased if we re-arrange the order of the intervals and merge them into larger intervals. In the extreme case, the homogeneity will be guaranteed if all intervals are merged into one big interval. However, we also need to have a small interval size in order to preserve the merit of generating candidates as early as possible.

4.1 Minimum merge

In order to adaptively create a near optimal interval configuration, we need to define a measurement on the homogeneity of a configuration.

Definition 1. Given two sets of large itemsets L^i and L^j, the distance between them is defined as

$$dist(L^i,L^j)=1-\frac{|L^i\cap L^j|}{|L^i\cup L^j|}.$$

Example 1. Let two large itemsets $L^1=\{A,B,C,AB,BC\}$ and $L^2=\{A,B,D,AB,BD\}$, then the distance between L^1 and L^2 is

$$dist(L^1,L^2)=1-\frac{|L^1\cap L^2|}{|L^1\cup L^2|}=1-\frac{3}{7}=\frac{4}{7}.$$

Definition 2. A division $Dv=\{I_1,I_2,...,I_m\}$ of a database partition D_i is a set of disjoint intervals dividing D_i such that, $D_i=\bigcup_{j=1}^{m}I_j$. The evenness factor $E_f(D_v)$ of Dv is defined by

$$E_f(Dv)=\frac{\sum_{j=1}^{m}dist(L^j,L)}{m},$$

where L^j is the locally large itemsets in I_j, $1\leqslant j\leqslant m$, and L is the large itemsets in D_i.

Evenness factor is a value in $[0,1]$. It equals zero if all the intervals have the same set of large itemsets.

Let $Dv_a=\{I_1,I_2,...,I_p\}$ be a division of a partition D_i. A division $Dv_b=\{I_{s1},I_{s2},...,I_{sp}\}$ of D_i is an equivalent division of Dv_a if it is a re-ordering of the intervals in Dv_a.

Definition 3. Given a division $Dv_a=\{I_1,I_2,...,I_p\}$ of a partition D_i, a division $Dv_m=\{I_{m1},I_{m2},...,I_{ml}\}$ of D_i is a k-merge of Dv_a, if $l=[p/k]$, and there exists an equivalent division $Dv_b=\{I_{s1},I_{s2},...,I_{sp}\}$ of Dv_a such that $I_{mi}=\bigcup_{j=k(i-1)+1}^{ki}I_{sj}(1\leqslant i\leqslant l-1)$, $I_{ml}=\bigcup_{j=k(l-1)+1}^{p}I_{sj}$.

A k-merge of a division is the shuffling and merging of the intervals in a division such that each resulted interval contains k intervals from the original division. Note that the last interval Iml in the k-merge may contain more than k intervals.

Example 2. Let $Dv_a=\{I_1,I_2,...,I_{10}\}$ be a division of a partition D_i, then $Dv_i=\{I_{j1},I_{j2},...,I_{j3}\}$, where $I_{j1}=I_1\cup I_4\cup I_5$, $I_{j2}=I_2\cup I_3\cup I_7$, and $I_{j3}=I_6\cup I_8\cup I_9\cup I_{10}$, is a 3-merge of Dv_a

We want to merge the intervals of a division as less as possible, so that the interval size can be as small as possible. Therefore, we define the minimum merge.

Definition 4. Given a division $Dv_a=\{I_1,I_2,...,I_p\}$ of a partition D_i, and an evenness factor threshold $\delta\in(0,1)$, a k-merge Dv_m of Dv_a is a minimum merge if $E_f(Dv_m)<\delta$ and for all k_j-merges Dv_j of Dv_a, $E_f(Dv_j)<\delta$, $k\leqslant k_j$.

For a given interval division, it is computationally very costly to compute the minimum merge because of the combinatorial effect of all the possible re-orderings. We have developed a technique based on clustering to compute efficiently a sub-optimal solution to this problem.

4.2 AIC (adaptive interval configuration)

We propose a strategy called AIC (adaptive interval configuration) to compute approximate minimum k-merges for each partition to enhance the performance of APM. AIC has two goals: the first one is to derive a more homogeneous itemset distribution over the intervals; the second one is to adaptively derive a proper interval size.

Initially, each database partition is divided into a number of small intervals, e.g., 1024 equal size intervals. In the initialization phase of APM, while the partitions are scanned to compute L_1 and C_2, (see section 3.4), we can compute all support counts for all size-1 itemsets in every interval.

Suppose there are m items, associated with each interval I in a partition. There is a length-m *interval vector* $(v_1, v_2, \ldots, v_m)$ in which v_i, $1 \leqslant i \leqslant m$, equals the support count of item i in I. These vectors can be viewed as points in an m-dimensional space. The Euclidean distance between them is a measurement of the difference of the itemset distributions among the intervals. Note that when we measure the evenness factor of a division, it is defined on large itemsets of all sizes. However, it is too costly to compute large itemsets of all sizes in order to find a good merge for a division. Instead, we use the information on the size-1 large itemsets which is much more efficient to compute. Furthermore, size-1 large itemsets are a good approximation of the large itemsets of larger size, because if an itemset is large, then all size-1 itemsets contained in the itemset must also be large. We use the Euclidean distance between the interval vectors to generate clusters among the intervals (Fig. 4). Any good clustering algorithm can be used for this purpose. In our implementation, we use the k-means clustering algorithm[13].

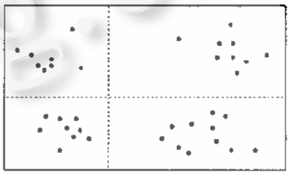
Fig. 4 Clustering of the intervals vectors

4.3 Reordering and merging intervals by clusters

Let Dv be a division of a partition D_i, and $G_1, G_2, \ldots, G_k$ be the k clusters generated from the interval vectors. The clusters are arranged in a decreasing order by their sizes. We generate a cluster-even re-ordering of the intervals of D_v by picking intervals alternatively from the clusters according to the decreasing order of the clusters. The intervals will first be selected randomly from G_1, and then from G_2, continuously until G_k, one from each cluster. After that, the selection will start from the first cluster again until all clusters run out. A k-merge is then performed on the re-ordered division to create a new division with larger interval size.

Because one sample interval is chosen from each cluster to perform the merge, the new intervals are more similar with each other than before the merge. Hence, the homogeneity of the new division would be increased. We modify the definition of the evenness factor defined in Section 4.1 to cover only size-1 itemsets in order to determine whether a merge has enough homogeneity. Therefore, if $D_v = \{I_1, I_2, \ldots, I_m\}$ is a division on a partition D_i, we modify

$$E_f(Dv) = \frac{\sum_{j=1}^{m} dist(L^j, L_1)}{m},$$

where L_1^j is the size-1 interval large itemsets in I_j, $1 \leqslant j \leqslant m$, and L_1 is the set of size-1 large itemsets in D_i.

4.4 Termination criteria of AIC

After re-ordering and merging have been performed, the evenness factor of the resulted division is computed. If the factor is less than a predefined threshold δ, then AIC terminates and returns the division to APM; otherwise, a new round of clustering, re-ordering and merging will be performed until one of the following three conditions becomes true:

(1) the evenness factor of the resulted division is less than the threshold δ;

(2) the rate of change of the evenness factor is less than a threshold;

(3) the number of resulted intervals is too small and is below a threshold.

The first condition has been discussed. If Ef_i and Ef_j are the evenness factors of two successive rounds in AIC, the rate of change of the evenness factor is defined as

$$\Delta Ef = \frac{|Ef_i - Ef_j|}{Ef_i}.$$

A small rate of change indicates that further re-ordering and merging would not improve much on the factor. Therefore, the preprocessing should terminate and the last merged division is returned to APM. The third condition is to control the number of intervals in the division. If it is below a predefined threshold, the preprocessing should also stop, and let APM to start its task. The main steps of the preprocessing technique AIC is presented as

```
/* Each processor divides its partition into a number of intervals, computes size-1 interval large itemsets
   for each interval and executes the following program in parallel */
(1) Compute interval vectors for all intervals in its partition;
(2) Perform $k$-clustering on the interval vectors;
(3) Perform re-ordering and $k$-merging on the intervals with respect to the clusters to generate a new divi-
    sion $Dv$ with $p$ intervals;
(4) Compute the evenness factor $Ef(Dv)$;
(5) If $Ef(Dv) < \delta$ then return the division $Dv$;
(6) If the rate of change of $Ef(Dv)$ is less than a predefined threshold, then return the division $Dv$;
(7) If $p$ is less than a predefined threshold then return the division $Dv$;
(8) go to step (1).
```

4.5 APM with adaptive interval configuration (APM-AIC)

The enhancement of APM with AIC is called the APM-AIC algorithm whose framework is illustrated in Fig. 5.

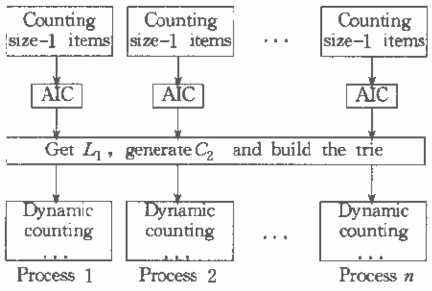

Fig. 5　Framework of APM-AIC

The difference between APM and APM-AIC is mainly the preprocessing to generate adaptively the k-merge divisions for the partitions. In the first scan in APM-AIC, besides computing L_1 and C_2, all support counts of the size-1 itemsets in each interval are stored. AIC takes these interval vectors to generate k-merge divisions for each partition. These merges in fact would give a proper interval size to each partition. Note that the re-ordering and merging of intervals are done logically, no physical movement of the intervals is required. The only change is that the access of the intervals will follow the orders in the merged division. After the optimization, the interval divisions are passed to the main loop of APM described in Fig. 4, and it will access the intervals according to the orders in the divisions.

Our performance studies show that the optimization introduced by the adaptive interval configuration has brought significant performance improvement to APM. This technique will be useful for algorithm which uses local information to speed up the computation of large itemsets. An example of such algorithm is the PARTITION algorithm[7].

5 Performance Studies

We have carried out extensive performance studies on an 8-node SGI Power Challenge shared-memory multiprocessor parallel machine. Each node in the machine is an MIPS R10000 processor, and it has a main memory of 512MB. All processors run IRIX 6.2.

5.1 Synthetic database generation

We followed the methodology proposed in Ref. [2] to develop synthetic database to study the performance of the algorithms. Table 1 is a list of the parameters used in the synthetic databases. The database is generated partition by partition. The result database is achieved by combining all partitions. The parameter n in Table 1 is the number of partitions generated.

Table 1 Synthetic database parameters

$\|D\|$	number of transactions in each partition
$\|T\|$	average size of the transactions
$\|I\|$	average size of the maximal potentially large itemsets
$\|L\|$	number of maximal potentially large itemsets
N	number of items
n	number of partitions

5.2 Relative performance

We have implemented APM, APM-AIC in our studies. In addition, we have also implemented CD on the SGI for comparison purpose. The version of CD in our implementation is a variant of the synchronous version of CD on the shared-memory model. To make CD more efficient, a shared hash tree is created in the memory. Every processor has a private counter array to record support counts of the candidates in each iteration. At the end of each iteration, support counts are copied from these arrays on to the hash tree, and a master processor computes the large itemsets and creates the candidates for the next iteration. This process is repeated until no new candidate is created.

Five databases with different attributes have been generated, and their attribute values are summarized in Table 2. The name of the database is in the form of Dx. Ty. Iz, where x is the number of transactions in each partition, y is the average size of the transactions, z is the average size of the itemsets. The number of partitions is 8, i.e., $n=8$. We also have set $N=1000$, $L=1000$, and correlation level to 0.5[2].

Table 2 Database properties

Name	$\|T\|$	$\|I\|$	Partition Size
D1024K. T5. I2	5	2	28MB
D512K. T8. I2	8	2	20MB
D512K. T10. I2	10	2	24MB
D512K. T10. I4	10	4	24MB
D256K. T20. I4	20	4	22MB

We ran CD, APM and APM-AIC multiple times on the five databases described in Table 2. The minimum support threshold used in the first four databases is 1%, and is 2% in the last database. The average response time of CD, APM and APM-AIC in these executions is shown in Fig. 6. Both APM and APM-AIC are faster than CD for all the five databases, and APM AIC is significantly better than both APM and CD. In all the experiments, it is found that APM-AIC is at least 3 times faster than CD.

The performance improvements of APM and APM-AIC are achieved through the following two sources: (1) saving in computation from the reduction on candidate itemsets; (2) saving in I/O cost from the reduction in the number of passes in database scanning.

Figure 7 shows the ratio of the

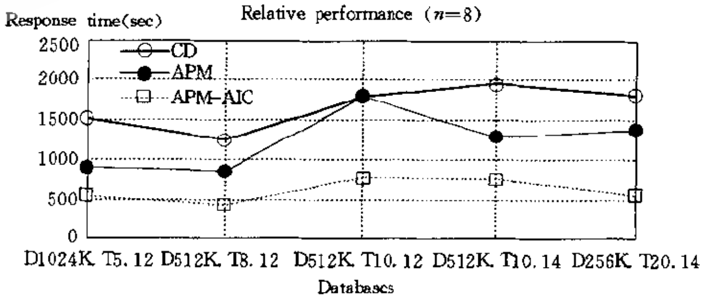

Fig. 6 Relative performance

total candidate itemsets generated by the three algorithms. In most of the cases, the number of candidates in APM-AIC is at least 60% less than that of CD. The stability of the reduction ratio achieved in APM-AIC confirms that the adaptive interval configuration technique is very effective in creating a more homogeneous data distribution among the intervals. Figure 8 provides the evidence that the adaptive interval configuration can reduce the system I/O cost significantly.

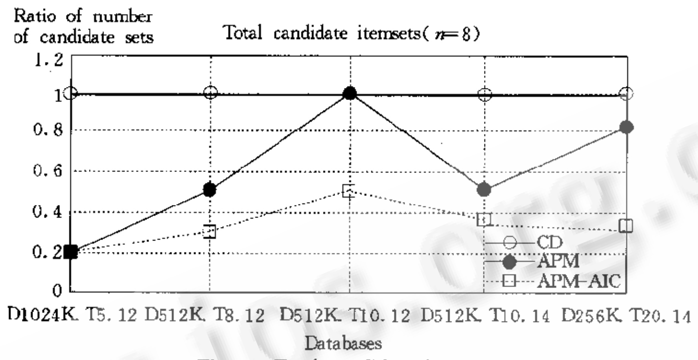

Fig. 7　Total candidate itemsets

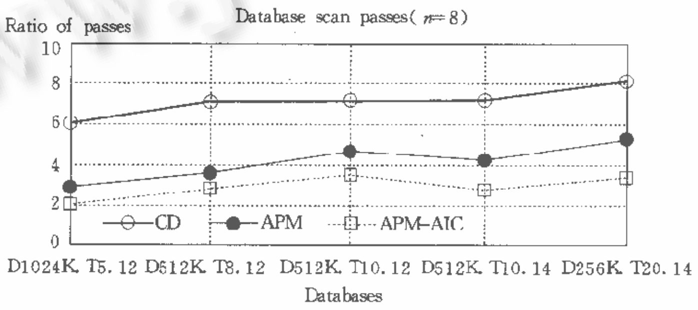

Fig. 8　Saving in database scan

We have also studied the effect of the clustering techniques on the performance of APM-AIC. In k-means clustering, we compare three cases: $k=2$, $k=4$ and a sliding down k. In the sliding down case, we set k to 8 and gradually decrease its value to 4 and then 2 in the later iterations. The sliding down technique starts with a large k which would create more clusters and hence possibly a more homogeneous division than the case of a smaller k. Decreasing the value of k gradually could prevent the interval size to increase too fast in the later iterations. However, the result in Fig. 9 shows that none of these three is better than the others in all cases. This leads us to adopt a small fixed value, i.e. $k=4$, in our experiments. It should be noted that the cost for the optimization is quite small, e.g., only taking 1% of the total response time.

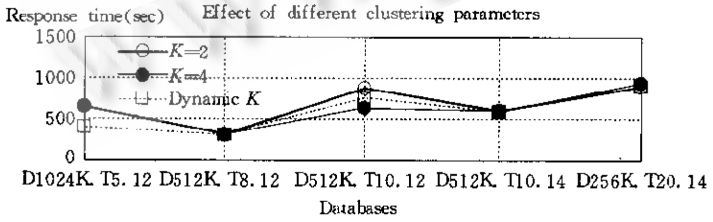

Fig. 9　Effect of different k values in the clustering in APM-AIC

From the experiments presented in this section, it can be concluded that the APM is more efficient than the synchronous CD. APM has less candidates and makes less passes in scanning, and has a much faster overall response time than CD. Furthermore, the enhanced APM-AIC gives APM a more stable behavior and reduces its sensitivity to the data distribution. And it is faster than APM even though it incurs some overhead in the preprocessing of the partitions.

5.3 Parallel performance

In order to study the merit of the parallelism in mining association rules in the shared memory model, we compare the speedup performance of the three parallel algorithms.

In the speedup experiment, we applied different numbers of processors on a fixed size database to study the performance of the three algorithms. The test database is the database D512K. T8.12 which has a total size of 160MB. We ran the three algorithms CD, APM and APM AIC on this database with 1, 2, 4 and 8 processors, respectively. Figure 10 presents the speedup result which is the ratio between the response time of processing

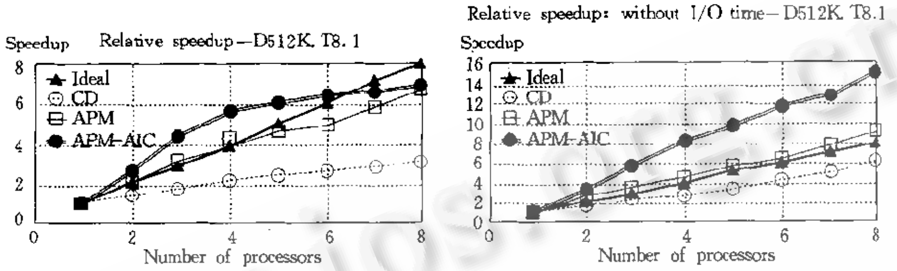

Fig. 10 Speedup

the database by one processor and the response time of processing it by more processors. In the first graph, the I/O time is included in the response time in the calculation of the speedup ratio. In the second graph, the I/O time is excluded from the calculation. In both cases, APM-AIC has a better speedup than APM, and APM is better than CD. The speedup of APM and APM-AIC in the second graph is superlinear through cut the whole range of different numbers of processors, which shows a very high parallelism. The superlinearity comes from the increased pruning effect resulted from the larger number of partitions. On the contrary, in the first graph, even though the speedup of both APM and APM-AIC is better than the linear speedup when the number of processors is between 1 to 4, it starts to degrade when there are more processors ($n \geqslant 4$). Comparing the results in the two graphs, it is clear that the degradation in the speedup is due to the increased contention when more processors share the I/O channel. This confirms our analysis in the beginning that I/O contention is an important negative factor for the performance of a parallel algorithm on a shared-memory system. The results in the first graph also show that CD suffers more from the contention than the other two algorithms because of its synchronous I/O.

6 Conclusion

Synchronous parallel algorithms are not suitable for mining association rules on a shared-memory multiprocessor parallel machine because of the I/O contention problem. We have proposed an adaptive asynchronous algorithm APM to solve this problem. All processors in APM generate candidates and compute their supports independently and asynchronously. The asynchronous access of the I/O channel reduces significantly the I/O cost.

Our second contribution is the proposal of the adaptive interval configuration technique. It defines an access order to the transactions which increases the homogeneity of the distribution of the large itemsets. By using this adaptive technique, the database scanning in APM is reduced to an amount much less than that in the conventional level-wise approach. This further reduces the I/O cost. The performance studies on an SGI Power Challenge machine have confirmed our observation that the asynchronous approach is much more efficient than the synchronous approach. The studies also show that APM has good parallelism.

References

1 Agrawal R, Imielinski T, Swami A. Mining association rules between sets of items in large databases. In: Proceedings of 1993 ACM-SIGMOD International Conference on Management of Data. Washington, D.C., 1993. 207～216

2 Agrawal R, Srikant R. Fast algorithms for mining association rules. In: Proceedings of the 20th VLDB Conference. Santiago, Chile, 1994. 487～499

3 Brin S, Motwani R, Ullman J *et al*. Dynamic itemset counting and implication rules for market basket data. In: Proceedings of 1997 ACM-SIGMOD International Conference on Management of Data. Tucson, Arizona, 1997. 255～264

4 Cheung D W, Han J, Ng V *et al*. Maintenance of discovered association rules in large databases: an incremental updating technique. In: Proceedings of the 12th International Conference on Data Engineering. New Orleans, Louisiana, 1996. 106～114

5 Han J, Fu Y. Discovery of multiple-level association rules from large databases. In: Proceedings of the 21st VLDB Conference. Zurich, Switzerland, 1995. 420～431

6 Park J S, Chen M S, Yu P S. An effective hash-based algorithm for mining association rules. In: Proceedings of 1995 ACM-SIGMOD International Conference on Management of Data. San Jose, CA, 1995. 175～186

7 Savasere A, Omiecinski E, Navathe S. An efficient algorithm for mining association rules in large databases. In: Proceedings of the 21st VLDB Conference. Zurich, Switzerland, 1995. 432～444

8 Toivonen H. Sampling large databases for mining association rules. In: Proceedings of the 22nd VLDB Conference. Bombay, India, 1996. 134～145

9 Agrawal R, Shafer J C. Parallel mining of association rules: design, implementation and experience. Special Issue on Data Mining, IEEE Transactions on Knowledge and Data Engineering, 1996,8(6):962～969

10 Cheung D W, Ng V T, Fu A W *et al*. Efficient mining of association rules in distributed databases. Special Issue on Data Mining, IEEE Transactions on Knowledge and Data Engineering, 1996,8(6):911～922

11 Han E, Karypis G, Kumar V. Scalable parallel data mining for association rules. In: Proceedings of 1997 ACM-SIGMOD International Conference on Management of Data. Tucson, Arizona, 1997. 277～288

12 Zaki M J, Ogihara M, Parthasarathy S *et al*. Parallel data mining for association rules on shared-memory multi-processors. In: Supercomputing'96, Pittsburg, PA, 1996. 17～22

13 MacQueen J B. Some methods for classification and analysis of multivariate observations. In: Proceedings of the 5th Berkeley Symposium on Mathematical Statistics and Probability. 1967. 281～297

自适应区间配置在关联规则并行采掘中的作用

胡侃[1]　张伟荦[2]　夏绍玮[1]

[1](清华大学自动化系　北京　100084)
[2](香港大学计算机科学系　香港)

摘要　现行的采掘关联规则的并行算法基于经典的层次算法.该方法在每一次重复扫描数据库时都需要一次同步,这种同步运算对于共享内存多处理器并行机来说极大地降低了采掘性能,这种低效主要源于对共享的I/O通道的竞争.该文提出了在共享内存多处理机上采掘关联规则的异步算法APM.在APM中,所有参与计算的处理器能独立地产生备选集和计算支持度.而且,APM所需的扫描数据库的次数比层次方法所需的更少.该文还提出了一种增强APM的技术,使得该算法的性能对于数据分布更具有鲁棒性.文中实现了APM的变种算法,还实现了Apriori的并行版本Count Distribution算法.在SGI Power Challenge SMP并行机上,进行了性能分析,结果表明所提出的异步算法APM具有更好的性能和可扩展性.

关键词　关联规则,数据采掘,并行采掘,共享内存多处理器,交易数据库.

中图法分类号　TP311