J Supercomput (2007) 39: 273–299
DOI 10.1007/s11227-006-0008-1

Parallel mining of association rules from text databases

John D. Holt · Soon M. Chung

Published online: 2 March 2007

Abstract In this paper, we propose a new algorithm named Parallel Multipass with Inverted Hashing and Pruning (PMIHP) for mining association rules between words in text databases. The characteristics of text databases are quite different from those of retail transaction databases, and existing mining algorithms cannot handle text databases efficiently because of the large number of itemsets (i.e., sets of words) that need to be counted. The new PMIHP algorithm is a parallel version of our Multipass with Inverted Hashing and Pruning (MIHP) algorithm (Holt, Chung in: Proc of the 14th IEEE int'l conf on tools with artificial intelligence, 2002, pp 49–56), which was shown to be quite efficient than other existing algorithms in the context of mining text databases. The PMIHP algorithm reduces the overhead of communication between miners running on different processors because they are mining local databases asynchronously and prune the global candidates by using the Inverted Hashing and Pruning technique. Compared with the well-known Count Distribution algorithm (Agrawal, Shafer in: (1996) IEEE Trans Knowl Data Eng 8(6):962–969), PMIHP demonstrates superior performance characteristics for mining association rules in large text databases, and when the minimum support level is low, its speedup is superlinear as the number of processors increases. These experiments were performed on a cluster of Linux workstations using a collection of Wall Street Journal articles.

Keywords Parallel association rule mining · Text retrieval · Multipass · Inverted hashing and pruning · Cluster computing · Scalability

This research was supported in part by Ohio Board of Regents, LexisNexis, and AFRL/Wright Brothers Institute (WBI).

J.D. Holt · S.M. Chung (✉)
Department of Computer Science and Engineering, Wright State University, Dayton, Ohio 45435, USA
e-mail: soon.chung@wright.edu

1 Introduction

Mining association rules in transaction databases has been demonstrated to be useful and technically feasible in several application areas [6, 8], particularly in retail sales. Let $\mathcal{I} = \{i_1, i_2, \ldots, i_m\}$ be a set of items. Let $\mathcal{D}$ be a set of transactions, where each transaction T contains a set of items. An association rule is an implication of the form $X \Rightarrow Y$, where $X \subset \mathcal{I}$, $Y \subset \mathcal{I}$, and $X \cap Y = \phi$. The association rule $X \Rightarrow Y$ holds in the database $\mathcal{D}$ with *confidence* c if $c\%$ of transactions in $\mathcal{D}$ that contain X also contain Y. The association rule $X \Rightarrow Y$ has *support* s if $s\%$ of transactions in $\mathcal{D}$ contain $X \cup Y$. Mining association rules is to find all association rules that have support and confidence greater than or equal to the user-specified minimum support (called *minsup*) and minimum confidence (called *minconf*), respectively [2]. For example, beer and disposable diapers are items, and $beer \Rightarrow diapers$ is an association rule mined from the database if the co-occurrence rate of beer and disposable diapers (in the same transaction) is not less than *minsup* and the occurrence rate of diapers in the transactions containing beer is not less than *minconf*.

The first step in the discovery of association rules is to find each set of items (called *itemset*) that have co-occurrence rate above the minimum support. An itemset with at least the minimum support is called a *large itemset* or a *frequent itemset*. In this paper, the term *frequent itemset* will be used. The size of an itemset represents the number of items contained in the itemset, and an itemset containing k items will be called a k-itemset. For example, {beer, diapers} can be a frequent 2-itemset. Finding all frequent itemsets is a very resource consuming task and has received a considerable amount of research effort in recent years. The second step of forming the association rules from the frequent itemsets is straightforward as described in [2]: For every frequent itemset f, find all non-empty subsets of f. For every such subset a, generate a rule of the form $a \Rightarrow (f - a)$ if the ratio of support(f) to support(a) is at least *minconf*.

The association rules mined from point-of-sale (POS) transaction databases can be used to predict the purchase behavior of customers. In the case of text databases, there are several uses of mined association rules between sets of words. They can be used for building a statistical thesaurus. Consider the case that we have an association rule $B \Rightarrow C$, where B and C are words. A search for documents containing C can be expanded by including B. This expansion will allow for finding documents using C that do not contain C as a term. A closely related use is Latent Semantic Indexing, where documents are considered close to each other if they share a sufficient number of associations [16]. Latent Semantic Indexing can be used to retrieve documents that do not have any terms in common with the original text search expression by adding documents to the query result set that are close to the documents in the original query result set. Frequent itemsets mined from a text database may be useful in the task of document ranking.

The word frequency distribution of a text database can be very different from the item frequency distribution of a sales transaction database. Additionally, the number of unique words in a text database is significantly larger than the number of unique items in a transaction database. Finally, the number of unique words in a typical document is much larger than the number of unique items in a transaction. These differences make the existing algorithms, such as Apriori [2], Direct Hashing and

Pruning (DHP) [26] and FP-Growth [19], ineffective in mining association rules in the text databases.

Considering the large number of candidate itemsets to be processed from very large text databases, parallel association rule mining is quite essential. In this paper, we propose a new parallel association rule mining algorithm named Parallel Multipass with Inverted Hashing and Pruning (PMIHP). PMIHP is a parallel version of our sequential Multipass with Inverted Hashing and Pruning (MIHP) algorithm [22], which was shown to be effective for mining association rules in text databases, especially when the minimum support level is low.

We implemented the proposed PMIHP on a cluster of Linux workstations, and analyzed its performance using a collection of Wall Street Journal articles. The new algorithm demonstrates a superlinear speedup as the number of processors increases when the minimum support level is low. PMIHP is also shown to be much faster than the Count Distribution algorithm [3], which is a parallel version of the Apriori algorithm.

The rest of the paper is organized as follows. Section 2 describes the characteristics of text databases, and Sect. 3 reviews existing sequential and parallel association rule mining algorithms. The proposed PMIHP algorithm is described in Sect. 4, and the results of performance analysis and comparison are discussed in Sect. 5. Section 6 contains some conclusions.

2 Text databases

Traditional domains for finding frequent itemsets, and subsequently the association rules, include retail point-of-sale (POS) transaction databases and catalog order databases [6]. The natural item instances are the sales transaction items, but other item instances are possible. For example, individual customer order histories could be used. An item may have a detailed identity, such as a particular brand and size, or may be mapped to a generic identity such as "bread." The number of items in a typical POS transaction is well under 100. The mean number of items and distribution varies considerably depending upon the retail operation.

Mining associations rules between words in controlled vocabularies has been done in [13, 14]. Documents labeled with a controlled vocabulary were mined for association rules. Generalized association rules can also be mined when the controlled vocabulary has a thesaurus by substituting broadening terms. The number of controlled vocabulary terms applied to a particular document is much smaller than the number of unique words in the same document. Using controlled vocabularies can improve precision for many kinds of searching. However, when the terms of interest are new, so that they are not yet in the controlled vocabulary, both precision and recall are adversely affected. Mining association rules to associate free text terms with classification terms has been done in [32]. Documents labeled with classification terms were used as the training set to discover association rules so that new documents could be classified by assigning the appropriate classification terms. The number of candidate itemsets is sharply reduced since every candidate itemset is required to contain at least one classification term, and there are only a small number of classification terms.

Our paper is concerned with finding associations between words in text documents without using a controlled vocabulary or a set of classification terms.

One natural analogue in text data mining to the POS transaction is the document. That is, the words of a document play the same role as the sale items on a POS transaction. Market basket analysis generally ignores the counts for the items purchased. The association of milk and cookies is made without considering the number of quarts of milk or the number of boxes of cookies on any particular transaction. In a similar manner, text mining could be performed without considering the number of times a word appears in the document. Instead of using the whole document as the unit of text mining, we can use other natural text units, such as sentences and paragraphs, and artificial units such as overlapping and non-overlapping text fragments of various sizes.

The word distribution characteristics of text data present some scalability challenges to the algorithms that are typically used in mining transaction databases. A random sample of text documents was drawn from the Text Research Collection (TREC) [23]. The sample consisted of the April 1990 Wall Street Journal (WSJ) articles that were in the TREC. There were 3,568 articles and 47,188 unique words. Most of those words occur in only a few of the documents. Some of the key distribution statistics are:

48.2%	of the words occur in only one document;
13.4%	of the words occur in only two documents;
7.0%	of the words occur in only three documents;
4.5%	of the words occur in only four documents;
3.0%	of the words occur in only five documents.

The average number of unique words (including stop-words) in a document was 207, with a standard deviation of 174.2 words. In this sample, only 6.8% of the words occurred in more than 1% of the documents. At a minimum support level of 0.05%, the average number of frequent words per document was 201. The distribution of 207 words in a document on the average is as follows:

6	words are unique to the document;
4	words occur in only one other document;
3	words occur in only two other documents;
2	words occur in only three other documents;
2	words occur in only four other documents;
40	words occur in less than 1% of the documents;
95	words occur in less than 5% of the documents;
126	words occur in less than 10% of the documents;
157	words occur in less than 20% of the documents;
50	words occur in more than 20% of the documents.

When the stop-words are removed from consideration, the average number of unique words in a document falls to 161, with a standard deviation of 140.9 words.

These distribution characteristics are consistent with the predictions based on the empirically derived rank-frequency law [27], which is also known as Zipf's law. These distribution characteristics are expected to remain the same as the size of the text collection increases.

We also took an 8-day collection of WSJ articles, published from October 1, 1991 through October 10, 1991. Note that the Wall Street Journal is published Monday through Friday, but not on the weekends. Given a minimum support level of 0.1%, the distribution of frequent words in 8 partitions, where each partition corresponds to a 1-day collection of articles, is:

5.0%	of frequent words occur in only 1 partition;
31.8%	of frequent words occur in only 2 partitions;
16.2%	of frequent words occur in only 3 partitions;
10.3%	of frequent words occur in only 4 partitions;
8.0%	of frequent words occur in only 5 partitions;
6.7%	of frequent words occur in only 6 partitions;
6.5%	of frequent words occur in only 7 partitions;
15.5%	of frequent words occur in all 8 partitions.

The April 1990 WSJ sample has a similar distribution when the articles are partitioned into 8 approximately equal-sized partitions. A considerable degree of skew is clearly present in the distribution.

The characteristics of this word distribution have profound implications for the efficiency of association rule mining algorithms. The most important implications are: (1) the large number of items and combinations of items that need to be counted; and (2) the large number of items in each document in the database.

It is commonly recognized in the information retrieval community that words that appear uniformly in a text database have little value in differentiating documents, and further those words occur in a substantial number of documents [27]. It is reasonable to expect that frequent itemsets (i.e., sets of words) composed of highly frequent words would also have little value. Therefore, text database miners need to work with itemsets composed of words that are not too frequent, but are frequent enough.

It is not yet clear how low the minimum support should be for finding effective associations. However, experimental results suggest that it should be lower than 0.5% for the April 1990 Wall Street Journal collection. In this collection, there are six groups of four or more very closely related documents. There should be a significant number of frequent itemsets that are in common for the majority of the documents within each group. Using the frequent itemsets discovered with a minimum support level of 0.5%, only one group had a frequent itemset that was common for more than 50% of the documents in that group. Thus, it is clear that the threshold must be lower than 0.5%. A lower minimum support of 0.1% can be selected because, in the April 1990 WSJ collection, that threshold represents around four documents and four documents is the smallest size of any of the six groups.

Two generally recognized theoretical models supporting document ranking are the *vector space model* and the *probabilistic model* [27]. Both models use the occurrence counts of terms within each document as a factor to influence the weight or score of the document. Given two documents, the document with a higher number of occurrences of a search term should be ranked higher than the document with a fewer number of occurrences of the same search term, on condition that the occurrence counts of the other search terms and the document sizes are equal. This weight adjustment corresponds well to the observation that words expressing the concepts central to the themes of a document tend to have higher occurrence counts than other words that are not germane to the themes of the document. It is not yet clear whether it is also useful to adjust the weights of frequent itemsets (of size 2 or more) based on their occurrence counts within each documents; and, if yes, how to adjust is the next question.

3 Existing algorithms for mining association rules

We examine the existing parallel approaches for finding frequent itemsets. We first review the sequential algorithms, and then describe how these algorithms are parallelized. We discuss these algorithms in the context of text data mining, particularly with respect to the low minimum support level required and the skewed distribution of words in text documents.

There are two basic approaches for mining association rules from databases. The first approach is to discover the frequent itemsets iteratively by length. In the first pass, the support of individual items are counted and frequent 1-itemsets are determined. Then the frequent 1-items are used to generate the potentially frequent 2-itemsets, called candidate 2-itemsets. In the second pass, we count the support of the candidate 2-itemsets, so that we can determine the frequent 2-itemsets. Frequent 2-itemsets are used to generate the candidate 3-itemsets, and so on. This process is repeated until there is no new frequent itemset. The classic implementation of this approach is the Apriori algorithm [2] which we discuss below. There have been three basic directions for improving Apriori: reducing the number of candidate itemsets using hashing [20, 21, 26]; reducing the size of the database through transaction trimming and pruning [20–22, 24, 26]; and reducing the number of passes on the database through partitioning [28], sampling [30], and by finding the maximal frequent itemsets [1, 5, 7, 17, 34]. The second approach is to count the itemsets without generating the candidates by transforming the database into a tree data structure [4, 19].

Apriori, Direct Hashing and Pruning (DHP), Sampling, and FP-Growth algorithms are described in more detail below.

3.1 Apriori algorithm

The Apriori algorithm for finding frequent itemsets from a database that consists of transactions is as follows:

```
1)  $F_1 = \{\text{frequent 1-itemsets}\}$;
2)  **Comment:** $k$ represents the pass number.
3)  **Comment:** $F_k$ is the set of frequent $k$-itemsets.
4)  **for** $(k = 2;\ F_{k-1} \neq \phi;\ k++)$ **do begin**
5)      **Comment:** $C_k$ is the set of candidate $k$-itemsets.
6)      **Comment:** $F_{k-1} * F_{k-1}$ is the natural join of
                $F_{k-1}$ and $F_{k-1}$ on the first $k-2$ items.
7)      $C_k = F_{k-1} * F_{k-1}$;
8)      **foreach** $k$-itemset $x \in C_k$ **do**
9)          **if** $\exists y \mid y = (k-1)$-subset of $x$ and $y \notin F_{k-1}$
10)             **then** remove $x$ from $C_k$;
11)     **foreach** transaction $t \in Database$ **do begin**
```

```
12)             foreach $k$-itemset $x$ in $t$ do
13)                 if $x \in C_k$ then $x.count++$;
14)         end
15)         $F_k = \{x \in C_k \mid x.count/|Database| \geq minsup\}$;
16)     end
17)     $Answer = \cup_k F_k$;
```

The formation of the set of candidate itemsets can be done effectively when the items in each itemset are stored in a lexical order, and itemsets are also lexically ordered. As specified in line 7, candidate k-itemsets, for $k \geq 2$, are obtained by performing the natural join operation $F_{k-1} * F_{k-1}$ on the first $k-2$ items of the $(k-1)$-itemsets in F_{k-1} assuming that the items are lexically ordered in each itemset [2]. For example, if F_2 includes {A, B} and {A, C}, then {A, B, C} is a potential candidate 3-itemset. Then the potential candidate k-itemsets are pruned in lines 8–10 by using the property that all the $(k-1)$-subsets of a frequent k-itemset should be frequent $(k-1)$-itemsets. This property is *subset closure* property of the frequent itemset. Thus, for {A, B, C} to be a candidate 3-itemset, {B, C} also should be a frequent 2-itemset. This subset-infrequency based pruning step prevents many potential candidate k-itemsets from being counted in each pass k for finding frequent k-itemsets, and results in a major reduction in memory consumption. To count the occurrences of the candidate itemsets efficiently as the transactions are scanned, they can be stored in a hash tree, where the hash value of each item occupies a level in the tree [2, 26].

Apriori does not perform well with a large number of frequent items, because it may generate a considerable number of candidate itemsets that do not have the minimum support. In the collection of April 1990 Wall Street Journal articles, there are approximately 15,000 words out of total about 47,000 words that occur in more than 0.1% of the documents. With Apriori, approximately 112 million candidate 2-itemsets would be generated.

3.2 Direct Hashing and Pruning (DHP) algorithm

In the Direct Hashing and Pruning (DHP) algorithm, a hashing technique is used to filter out candidate itemsets generated [26]. Each $(k+1)$-itemset in transactions is hashed to a hash value while the occurrences of the candidate k-itemsets are counted by scanning the transactions. If the support count of a hash value is less than the minimum support count, then none of the $(k+1)$-itemsets with that hash value will be included in the set of candidate $(k+1)$-itemsets in the next pass. Pruning candidate itemsets based upon the support counts of their hash values is safe because there may be false positives (i. e., the retained candidate itemsets that are not actually frequent) but there will be no false negatives.

Transaction trimming and transaction pruning methods are also proposed together with DHP [26], so that the size of database to be scanned to count the occurrences of candidate itemsets can be reduced at each pass. Transaction trimming and pruning are based on the subset closure property of frequent itemsets; that is, any subset of a frequent itemset must be a frequent itemset by itself. This property suggests that a transaction may have a candidate $(k+1)$-itemset only if it contains $k+1$ candidate k-itemsets obtained in the previous pass. Thus, as a transaction is scanned to count the occurrences of the candidate k-itemsets, we can determine if this transaction can be pruned from the database in the next pass. On the other hand, if a transaction contains a frequent $(k+1)$-itemset, any item contained in this $(k+1)$-itemset should appear in at least k of the candidate k-itemsets contained in this transaction. Thus, by counting how many times each item in a transaction is involved in the candidate k-itemsets in that transaction, we can decide whether the item can be eliminated from the transaction in the next pass. A transaction is trimmed by rewriting it without the items that will not contribute to forming the frequent itemsets in the next pass.

How much the DHP can reduce the number of candidate itemsets depends on the number of false positives. The false positives are generated when the hash values are identical for a group of candidate itemsets (the hash synonyms) whose individual frequency is less than the minimum support, but whose hash value frequency is not less than the threshold. The number of candidate itemsets that have the same hash value is directly related to the size of the hash table. Unfortunately, this table is in competition for memory space with the hash tree used to store the candidate itemsets and their counts.

The DHP algorithm is not effective for finding the frequent itemsets in the April 1990 Wall Street Journal articles (see Sect. 2 for details on the collection) when the required minimum support level is as low as 0.1%. The reason is that there are 47,000 unique words, so that the maximum number of 2-itemsets to be hashed is about 1.1 billion. This is because the hashing of every 2-itemset within each document is performed before the pruning of the items. With a minimum support count of four occurrences (i.e., minimum support of 0.1%), if every 2-itemset actually occurred in a document, the hash table would have to accommodate more than 250 million entries to avoid counting every 2-itemset, because the expected number of itemsets with the same hash value would be slightly more than four. For this collection of text data, a hash table of 10 million entries resulted in no pruning of candidate 2-itemsets when the minimum support is 0.1%. There was not enough memory available to try a significantly larger hash table.

3.3 Sampling

The idea of the Sampling algorithm [30] is to pick a small sample of the database and find all the itemsets in the sample that are potentially frequent in the whole database. Then, the whole database is scanned to actually count those itemsets. The sample is usually small enough to fit in the memory available.

The itemsets that were found to be frequent in the sample and the itemsets that make up the negative border are counted in the original database. The negative border itemsets are those that are not frequent in the sample but all of their proper subsets are frequent. If a negative border itemset is found to be frequent in the database, then additional counting will be performed for the new negative border itemsets that have the newly discovered frequent itemset as a proper subset.

This approach can be effective when the minimum support level is high enough, so that the expected number of initially missed frequent itemsets is small. However, this is not the case for text databases because they typically require low minimum support levels as explained in Sect. 2.

3.4 FP-Growth algorithm

The FP-Growth algorithm [4, 19] does not generate the candidate itemsets. Instead, it uses a frequent pattern tree (FP-tree) structure to compress the database to avoid repeated scanning of the database.

The FP-Growth algorithm starts with a scan of the entire database to determine the frequent items. Then, the database is read again. Within each transaction, infrequent items are removed and frequent items are ordered in descending order of their occurrence counts. The transactions are then placed into a prefix tree named frequent pattern tree (FP-tree). The occurrence counts of itemsets are updated during insertion. Consider the transactions {A, B, C} and {A, B, D} as the first two transactions in the database. The common prefix {A, B} would have an occurrence count of 2. Each of the two suffix branches would have a count of 1. If the third transaction were {A, B, C, D}, the common prefix {A, B} would have an occurrence count of 3, the {C} branch would have an occurrence count of 2, and the {D} suffix of the {C} branch would have an occurrence count of 1.

There is also a node list for each frequent item, and it links the nodes of the item into a list. This node list is updated during the construction of the FP-tree.

The mining of the FP-tree can be performed by traversing the node lists of frequent itemsets, starting from the least frequent item and proceeding upward to the most frequent item. Let i be the item under consideration. By following the node list of i, we can find every prefix path for node i. Since all the occurrence counts are available on each of the paths, we can discover the frequent itemsets containing i.

The advantages of FP-Growth come from directly reducing the database into a more compact FP-tree, then mining the FP-tree in memory without generating candidate itemsets. However, even with the substantial compression provided by the FP-tree, some FP-tree structures may not fit in main memory, and then the processing time will increase considerably. They proposed a method of efficiently partitioning the database such that each transaction belongs to one and only one partition. The partitioning is done by the common suffix frequent item of the transactions, and the partitions are processed in certain order [19].

There are several aspects of text database that work against the strengths of this approach. First, a typical text document has many words. For example, the average number of frequent words per document (with a minimum support of 0.05%) is 161 in our test database of April 1990 Wall Street Journal articles. There is a total population of about 25,400 frequent words when the stop-word list is used.

Second, only 8 of these words occur in more than 20% of the documents, and about 30 words occur in 10% or more of the documents. A large number of unique prefix paths will sharply limit the degree of compression that can be achieved with the FP-tree.

Finally, the proposed partitioning scheme will result in many very small partitions. For example, there are 6,323 words that occur in just 2 documents (with a minimum support level of 0.05%) out of 3,568 documents. This situation will result in at least 1,784 partitions, because there will not be any common suffix that can exist in more than 2 documents. Considering the large number of items that can be a suffix, it is likely that the number of partitions will be higher. It is important to note though that the number of partitions cannot exceed the number of documents.

We implemented the FP-Growth algorithm and evaluated its performance for both transaction databases and text databases. We found it is not scalable when there are many candidate itemsets as the database size is large and the minimum support level is low.

3.5 Parallel algorithms

Parallel implementations of the above sequential approaches have been explored [3, 4, 9, 12, 18, 25, 29–31, 33]. The sequential algorithms have been parallelized in three basic ways.

The database can be replicated and the candidates can then be partitioned among the processors. This candidate distribution method is a natural way to parallelize the Apriori-like algorithms. After the replication of the database and the distribution of the candidates are done, the mining processes can proceed independently because there is no need for them to coordinate in counting the candidates. However, the replication of large whole text databases can be expensive in terms of data transfer, storage, and processing time at each processor. Thus, it is not a particularly suitable approach for finding frequent itemsets from text databases.

The database can be distributed, and all the processors can work with the same set of candidates. This count distribution approach is quite natural with the Apriori-like and sampling algorithms. However, the mining processes are not independent as they must share the support counts of candidates to determine which of them are frequent. In many cases, large text databases are physically distributed, so partitioning the database into local databases and process them concurrently at different processors is a natural fit. Unfortunately, the very low minimum support levels required for mining frequent itemsets from text databases may result in a very small local minimum support count, so that not many candidates can be pruned locally. In addition, the whole set of candidates needs to fit in the memory of each processor for efficient counting, whereas the number of candidates is usually very large for text databases. The Multipass approach proposed in [20] can be used to control the number of candidates to be processed at the same time.

The transactions in the database can be selectively replicated and some of the candidates can be partitioned so that the workload is balanced among the processors. Each transaction can be allocated only to the processors that are involved with the items within the transaction. As in the pure candidate distribution approach, the mining processes are independent, and no coordination is required to determine which candidates are frequent.

A global hash table of candidates can be used to reduce the interactions between the processors. Park et al. [25] proposed the distribution of the candidate hash table used in the DHP algorithm as a set of clues. Each processor can use the global hash table to determine if a candidate might have a sufficient support count, and also to determine which processors should be polled to obtain the support count of the candidate. This approach is unlikely to be efficient for text databases as the underlying DHP hash table was shown to be ineffective in mining text databases [20].

3.6 Motivation for new parallel mining algorithms

Each of the existing algorithms discussed above was shown to be unsuitable for the task of mining frequent itemsets from text databases. It is mainly because they are not capable of handling the large number of highly disparate candidate itemsets in an effective manner.

The requirement of mining frequent itemsets from a text database with a large number of disparate candidate itemsets motivates the development of new parallel mining algorithms.

4 Parallel Multipass with Inverted Hashing and Pruning (PMIHP) algorithm

The new Parallel Multipass with Inverted Hashing and Pruning (PMIHP) algorithm is a parallel version of the sequential Multipass with Inverted Hashing and Pruning (MIHP) algorithm. The MIHP algorithm is based on the Multipass approach [20] and the Inverted Hashing and Pruning (IHP) [21] that we proposed.

In PMIHP, the Multipass approach reduces the required memory space at each processor by partitioning the frequent items and processing each partition separately. Thus, the number of candidate itemsets to be processed is limited at each instance. The Inverted Hashing and Pruning is used to prune the local and global candidate itemsets at each processing node, and it also allows each processing node to determine the other peer processing nodes to poll in order to collect the local support counts of each global candidate itemset.

The Multipass approach and the Inverted Hashing and Pruning are described in Sect. 4.1 and Sect. 4.2, respectively, and the pseudo-code of the sequential MIHP algorithm is given in Sect. 4.3.

4.1 Multipass approach

The Multipass algorithm partitions the frequent items, and thus partition the candidate itemsets. The partition size is selected to be small enough to fit in the available memory of the processing node. Each partition is then processed separately. Each partition of items contains a fraction of the set of all items in the database, so that the memory space required for counting the occurrences of the sets of items within a partition will be much less than the case of counting the occurrences of the sets of all the items in the database.

The Apriori algorithm described in Sect. 3.1 is modified to be the Multipass-Apriori algorithm as follows:

1. Count the occurrences of each item in the database to find the frequent 1-itemsets.
2. Partition the frequent 1-itemsets into p partitions, $P_1, P_2, \ldots, P_p$.
3. Use Apriori algorithm to find all the frequent itemsets whose member items are in each partition, in the order of $P_p, P_{p-1}, \ldots, P_1$, by scanning the database. When partition P_p is processed, we can find all the frequent itemsets whose member items are in P_p. When the next partition P_{p-1} is processed, we can find all the frequent itemsets whose member items are in P_{p-1} and P_p. This is because, when P_{p-1} is processed, the items in P_{p-1} are extended with the frequent itemsets we found from P_p and then counted. This procedure is continued until P_1 is processed.

Assume, without loss of generality, that the frequent 1-itemsets (or, simply items) are ordered lexically. The frequent items are partitioned into p partitions, $P_1, P_2, \ldots, P_p$, such that for every $i < j$, every item $a \in P_i$ is less than every item $b \in P_j$. Thus, the itemsets under consideration for some partition P_i have a particular range of item prefixes. Notice that if the partitions have the same number of items, the potential number of itemsets that will be formed by extending a lexically lower ordered partition will be larger than the potential number of itemsets from a lexically higher ordered partition.

Since the frequent items are ordered lexically, it is important to process the partitions in sequence from the highest ordered one to the lowest ordered one. This processing order is required to support the subset-infrequency based pruning of candidate itemsets. Figure 1 shows an example of partitions, where items 0, 1, 2, 3, 4, and 5 are frequent 1-itemsets and are partitioned into P_1, P_2, and P_3.

When P_3 is being processed, we consider only the candidate itemsets whose members are in $\{2, 3, 4, 5\}$, not including 0 and 1. Thus, the number of candidates will be smaller compared to the case of considering all the frequent 1-itemsets. Similarly, when P_2 is processed, item 0 is not considered, and many of the candidate itemsets can be pruned. Therefore, it requires less memory space during the processing as one partition is processed at a time.

Suppose that $\{2, 3\}$ is the only frequent 2-itemset found as a result of processing the frequent items in partition P_3. When the next partition P_2 is being processed, item 1 in P_2 is extended with each of the items in P_3 first. As a result, we can find some frequent 2-itemsets including item 1. Let's assume that $\{1, 2\}$, $\{1, 3\}$, and $\{1, 5\}$ are found frequent. Then, $\{1, 2\}$ and $\{1, 3\}$ are joined into $\{1, 2, 3\}$, and we need to check if $\{2, 3\}$ is also frequent to determine whether $\{1, 2, 3\}$ is a candidate 3-itemset or not. Since $\{2, 3\}$ was found frequent when P_3 was processed, $\{1, 2, 3\}$ becomes a candidate 3-itemset. This explains why we need to process the last partition of fre-

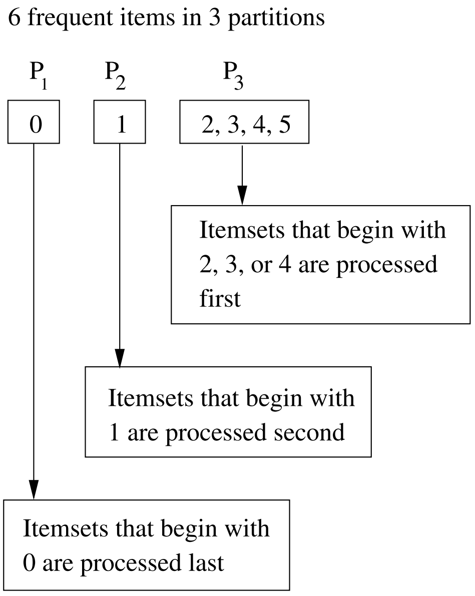

6 frequent items in 3 partitions
P1
P2
P3
0
1
2, 3, 4, 5
Itemsets that begin with 2, 3, or 4 are processed first
Itemsets that begin with 1 are processed second
Itemsets that begin with 0 are processed last

Fig. 1 Partitioning a set of 6 frequent items for Multipass-Apriori

quent items first. On the other hand, $\{1, 2, 5\}$ and $\{1, 3, 5\}$ are not candidate 3-itemsets because $\{2, 5\}$ and $\{3, 5\}$ were not found frequent.

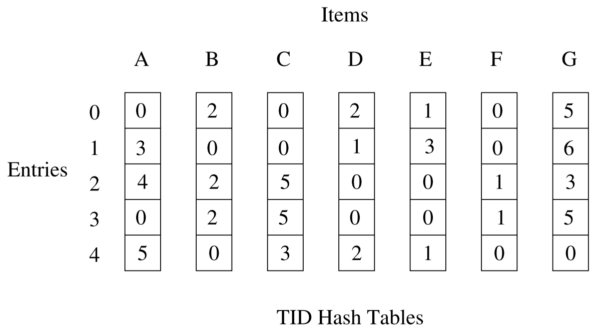
In practice, if the estimated number of candidate itemsets to be generated is small after processing a certain number of partitions, then we can merge the remaining partitions into a single partition so that the number of database scans will be reduced.

4.2 Inverted Hashing and Pruning (IHP)

In the Inverted Hashing and Pruning algorithm, a hash table, named TID Hash Table (THT), is created for each item in the database. When an item occurs in a transaction, the transaction identifier (TID) of this transaction is hashed to an entry of the THT of the item, and the entry stores the number of transactions whose TIDs are hashed to that entry. Thus, the THT of each item can be generated as we count the occurrences of each item during the first pass on the database. After the first pass, we can remove the THTs of the items which are not contained in the set of frequent 1-itemsets, and the THTs of the frequent 1-itemsets can be used to prune some of the candidate 2-itemsets. In general, after each pass $k \geq 1$, we can remove the THT of each item that is not a member of any frequent k-itemset, and the remaining THTs can prune some of the candidate $(k + 1)$-itemsets.

Consider a transaction database with seven items: A, B, C, D, E, F, and G. Figure 2 shows the THTs of these items at the end of the first pass. In our example, each THT has five entries for illustration purpose. Here we can see that item D occurred in five transactions. There were two TIDs hashed to 0, one TID hashed to 1, and two TIDs hashed to 4.

If the minimum support count is 7, we can remove the THTs of the items B, D, E, and F as shown in Fig. 3. Only the items A, C, and G are frequent and are used to determine C_2, the set of candidate 2-itemsets. As in the Apriori algorithm, {A, C}, {A, G}, and {C, G} are generated as candidate 2-itemsets by pairing the frequent 1-itemsets. However, in IHP, we can eliminate {A, G} from consideration by using the THTs of A and G. Item A occurs in 12 transactions, and item G occurs in 19 transactions. However, according to their THTs, they can co-occur in at most 6 transactions. Item G occurs in 5 transactions whose TIDs are hashed to 0, and item A occurs in no transactions that have such TIDs. Thus, none of those 5 transactions that contain G also contains A. Item A occurs in 3 transactions whose TIDs are hashed to 1, and item G occurs in 6 transactions with those TIDs. So, in the set of transactions



Entries	A	B	C	D	E	F	G
0	0	2	0	2	1	0	5
1	3	0	0	1	3	0	6
2	4	2	5	0	0	1	3
3	0	2	5	0	0	1	5
4	5	0	3	2	1	0	0

Items
TID Hash Tables

Fig. 2 TID Hash Tables at the end of the first pass

Fig. 3 TID Hash Tables of frequent 1-itemsets

Items

Entries	A	B	C	D	E	F	G
0	0		0				5
1	3		0				6
2	4		5				3
3	0		5				5
4	5		3				0

TID Hash Tables

whose TIDs are hashed to 1, items A and G can co-occur at most 3 times. The other THT entries corresponding to the TID hash values of 2, 3, and 4 can be examined similarly, and we can determine items A and G can co-occur in at most 6 transactions, which is below the minimum support level.

In general, for a candidate k-itemset, we can estimate its maximum possible support count by adding the minimum TID hash count of the k items at each entry of their THTs. If the maximum possible support count is less than the required minimum support, it is pruned from the candidate set.

4.3 Multipass with Inverted Hashing Pruning (MIHP) algorithm

The MIHP algorithm is a combination of the Multipass-Apriori algorithm and the Inverted Hashing and Pruning described above, and the details of MIHP is presented below:

```
1)  $Database$ = set of transactions;
2)  $Items$ = set of items;
3)  transaction = $\langle TID,\ \{x \mid x \in Items\}\rangle$;
4)  **Comment:** Read the transactions and count the
                 occurrences of each item and create a
                 TID Hash Table (THT) for each item
                 using a hash function $h$.
5)  **foreach** transaction $t \in Database$ **do begin**
6)      **foreach** item $x$ in $t$ **do begin**
7)          $x.count{+}{+}$;
8)          $x.THT[(h(t.TID)]{+}{+}$;
9)      **end**
10) **end**
11) **Comment:** $F_1$ is a set of frequent 1-itemsets.
12) $F_1 = \{x \in Items \mid x.count/|Database| \geq minsup\}$;
13) Partition $F_1$ into $p$ partitions, $P_1, P_2, \ldots, P_p$;
14) **Comment:** Process the partitions in the order of $P_p, P_{p-1}, \ldots, P_1$.
15) **for** $(m = p;\ m > 0;\ m--)$ **do begin**
16)     **Comment:** Find $F_k$, the set of frequent $k$-itemsets,
                 $k \geq 2$, whose members are in partitions $P_m, P_{m+1}, \ldots, P_p$.
17)     **for** $(k = 2;\ F_{k-1} \neq \phi;\ k{+}{+})$ **do begin**
```

```
18)           Comment: Initialize $F_k$ before the partition $P_p$ is processed.
19)           if $m = p$ then $F_k = \phi$;
20)           Comment: $C_k$ is the set of candidate $k$-itemsets whose members
                      are in $P_m, P_{m+1}, \ldots, P_p$.
21)           Comment: $F_{k-1} * F_{k-1}$ is the natural join of
                      $F_{k-1}$ and $F_{k-1}$ on the first $k-2$ items.
22)           $C_k = F_{k-1} * F_{k-1}$;
23)           foreach $k$-itemset $x \in C_k$ do
24)           if $\exists y \mid y = (k-1)$-subset of $x$ and $y \notin F_{k-1}$
25)                then remove $x$ from $C_k$;
26)           Comment: Prune the candidate $k$-itemsets using the THTs.
27)           foreach $k$-itemset $x \in C_k$ do
28)               if $GetMaxPossibleCount(x)/|Database| < minsup$
29)                   then remove $x$ from $C_k$;
30)           Comment: Scan the transactions to count the occurrences
                      of candidate $k$-itemsets.
31)           foreach transaction $t \in Database$ do begin
32)               foreach $k$-itemset $x$ in $t$ do
33)                   if $x \in C_k$ then $x.count++$;
34)           end
35)           $F_k = F_k \cup \{x \in C_k \mid x.count/|Database| \geq minsup\}$;
36)       end
37)   end
38)   $Answer = \cup_k F_k$;
```

Here, *GetMaxPossibleCount*(x) returns the maximum number of transactions that may contain a k-itemset x by using the THTs of the k items in x. Let's denote the k items in x by $x[1], x[2], \ldots, x[k]$. Then *GetMaxPossibleCount*(x) can be defined as follows:

```
GetMaxPossibleCount(itemset $x$)
begin
    $k = size(x)$;
    MaxPossibleCount = 0;
    for ($j = 0$; $j < size(THT)$; $j++$) do
        MaxPossibleCount+ = $min(x[1].THT[j], x[2].THT[j], \ldots,$
                              $x[k].THT[j])$;
    return (MaxPossibleCount);
end
```

size(x) represents the number of items in the itemset x and *size*(THT) represents the number of entries in the THT.

The Partition algorithm [28] used the list of TIDs of the transactions containing each item, called a TID-list. Once the TID-list is generated for each item by scanning the database once, we do not need to scan the database again, because the support count of any candidate itemset can be obtained by intersecting the TID-lists of the individual items in the candidate itemset. However, a major problem is that the size of the TID-lists would be too large to maintain when the number of transactions in the database is very large. On the other hand, the number of entries in the THTs can be limited to a certain value, and it has been shown that even a small number of entries can effectively prune many candidate itemsets.

4.3.1 Additional efficiency for MIHP

For further performance improvement, the MIHP algorithm is used together with the transaction trimming and pruning method proposed as a part of the DHP algorithm [26]. The concept of the transaction trimming and pruning is as follows: During the kth pass on the database, if an item is not a member of at least k candidate k-itemsets within a transaction, it can be removed from the transaction for the next pass (transaction trimming), because it cannot be a member of a candidate $(k+1)$-itemset. Moreover, if a transaction does not have at least $k+1$ candidate k-itemsets, it can be removed from the database (transaction pruning), because it cannot have a candidate $(k+1)$-itemset. It is convenient to use a slightly weaker form of the rule for MIHP. The transaction trimming rule as stated is only applied to the items that are in the F_1 partition being processed. The items that are not in the partition being processed are removed if they are not members of any candidate itemset during the current pass. Also, a transaction is pruned during the kth pass if it does not have at least k candidate k-itemsets, each of which contains one or more items from the current F_1 partition being processed.

Transaction trimming and pruning is synergistic with the candidate pruning by IHP. The reduction of candidate k-itemsets for the kth pass, $k \geq 2$, will result in additional items to be trimmed during the kth pass. In addition, the partitioning of the frequent 1-itemsets based on the Multipass is synergistic with the transaction trimming and pruning because more items and transactions can be removed as one partition is processed at a time.

4.4 Parallel MIHP (PMIHP) algorithm

The Parallel MIHP algorithm is based upon the sequential MIHP algorithm. The database is partitioned into almost equal-sized local databases, one for each processing node, and the MIHP algorithm is parallelized by three additional actions: exchanging and merging of the local TID Hash Tables (THTs) of items between processing nodes to produce the global TID Hash Tables replicated in the processing nodes; identifying the global candidate itemsets that are just locally frequent in at least one node, but may have sufficient global support; and determining the global support of the global candidate itemsets and updating the list of global frequent itemsets. The global THT of each item is just a linear cascade of the local THTs of the item, instead of creating a separate global THT whose TID hash count is the sum of the local TID hash counts at each entry.

For an itemset to be globally frequent in the whole database, it must be frequent in at least one local database. If a local support count of an itemset is above the global minimum support count, then it is recorded as a globally frequent itemset. However, if the local count of an itemset is above the local minimum support count but less than the global minimum support count, then it is recorded as a global candidate itemset.

The major steps of the PMIHP algorithm are as follows:

1. Every processing node counts the occurrences of each item in its local database and builds the local THTs.
2. The local support counts and THTs of items are exchanged between processing nodes, so that each processing node can obtain the global support counts of all items. Each processing node then determines the set of globally frequent items and discard the local THTs of globally infrequent items. The received local THTs of the items that are not local to the node are also discarded, and each node creates the global THT of each local item by linearly cascading its local THTs created by the processing nodes.
3. Every processing node partitions the frequent 1-itemsets into the same p partitions, $P_1, P_2, \ldots, P_p$, as in the MIHP algorithm. The number of partitions is determined, such that the candidate itemsets for each partition would be resident in the main memory of processors. It has been shown that the total execution time is not sensitive to the partition size unless it is too large [20].
4. As in MIHP, the partitions are processed one by one, in the order of $P_p, P_{p-1}, \ldots, P_1$, by all the processing nodes. That means, all the processing nodes discover the globally frequent itemsets whose members are in P_p first, and then discover the globally frequent itemsets whose members are in P_{p-1} and P_p, and so on, until partition P_1 is processed.
5. For each partition P_i, $1 \leq i \leq p$, each processing node uses MIHP algorithm to find all the locally frequent itemsets from its local database. Some of these itemsets may have sufficient local support to be globally frequent as well. The locally frequent itemsets that do not have sufficient local support to be globally frequent are global candidate itemsets. The global THTs are used to prune the global candidate itemsets by estimating their maximum global support counts, as in MIHP. When certain number of global candidate itemsets are accumulated at each node, it requests other processing nodes to provide their local support counts of those global candidate itemsets. The other processing nodes to be polled to obtain their local support counts for each global candidate itemset can be easily determined by checking the local THTs contributed by the other processing nodes (in step 1) for the member items of the global candidate itemset. Only the processing nodes that have a positive TID hash count for the global candidate itemset will be polled.
6. The processing nodes exchange their lists of global frequent itemsets after all the partitions are processed.

To perform the communication between processing nodes efficiently, we imposed a logical binary n-cube structure on the processing nodes. Then the processing nodes can exchange and merge the local information through increasingly higher dimensional links between them [11]. In the n-cube, there are 2^n nodes, and each node has n-bit binary address. Also, each node has n neighbor nodes which are directly linked to that node through n different dimensional links. For example, there are 8 nodes in the 3-cube structure, and node $(000)_2$ is directly connected to $(001)_2$, $(010)_2$ and $(100)_2$ through a 1st-dimensional link, a 2nd-dimensional link, and a 3rd-dimensional link, respectively. Thus, in the n-cube, all the nodes can exchange and merge their local information in n steps, through each of the n different dimensional links. When $n = 3$, the three exchange and merge steps are:

step 1: node $(**0)_2$ and node $(**1)_2$ exchange and merge, where $*$ denotes a don't-care bit.
step 2: node $(*0*)_2$ and node $(*1*)_2$ exchange and merge.
step 3: node $(0**)_2$ and node $(1**)_2$ exchange and merge.

It is neither necessary nor desirable to obtain the global support counts of the global candidate itemsets at the end of the processing of each partition P_i, $1 \leq i \leq p$. Instead, when certain number of global candidate itemsets are accumulated at a node, it can start polling the other processing nodes to obtain their local support counts of those global candidate itemsets.

As in MIHP, transaction trimming and pruning can be used together with PMIHP. However, to provide the local support counts of global candidate itemsets, each processing node cannot prune the item that can be a member of some global candidate itemset from the transactions in its local database. Thus, the frequency of polling between processing nodes to obtain the local support counts of the global candidate itemsets must be balanced with the loss in efficiency caused by not trimming and pruning the transactions in the local databases.

5 Performance analysis of parallel MIHP

Some experimental tests have been done to evaluate the performance of PMIHP and to compare it with that of the Count Distribution (CD) algorithm [3]. The Count Distribution algorithm is a parallel version of the Apriori algorithm. In Count Distribution, the database is partitioned and distributed over the processing nodes initially. At each pass k, every node collects local counts of the same set of candidate k-itemsets. Then, these counts are merged between processing nodes, so that each node can identify the frequent k-itemsets and generate the candidate $(k+1)$-itemsets for the next pass. To merge the local support counts of the candidate itemsets, synchronization between nodes is required at every pass, and maintaining the same set of candidate itemsets in all the nodes is redundant.

We implemented PMIHP and Count Distribution on a Linux cluster with 9 nodes connected by a Fast Ethernet switch. Eight of the nodes were used for parallel mining, and one node was used as a management console. Each node has a 800 MHz Pentium processor, 512 Mbytes of memory and a 40 Gbytes disk drive.

Both PMIHP and Count Distribution were implemented in the Java language. The Java virtual machine was the IBM JVM 1.1.8 for the Linux operating system with the just-in-time (JIT) compiler enabled. The interprocess communication was performed using the Java RMI (Remote Method Invocation) protocol.

For all of the tests, the JVM memory for objects was constrained (via the *mx* parameter) to 416 Mbytes. The initial heap size was set to 416 Mbytes (via the *ms* parameter) to control the effect of heap growth overhead on the performance. The partition size used for the PMIHP was 100 frequent items, and the global TID Hash Table (THT) size was 400 entries for each item. As the global THT of each item is a linear cascade of its local THTs, the local THT size for the 1-node case was 400 entries per item, and was 50 entries per item for the 8-node case. It has been shown that the sizes of the partitions and THT are not critical for the overall performance [22].

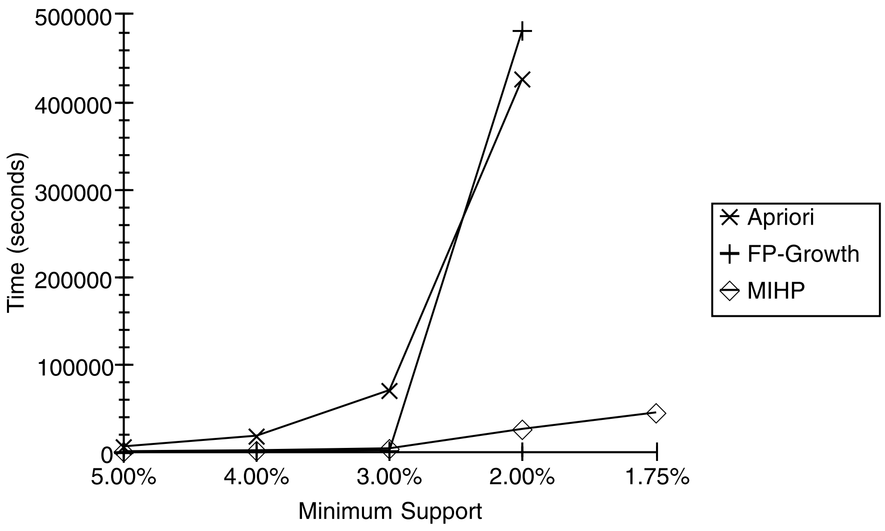


Fig. 4 Total execution time to find all frequent itemsets (in 21,703 documents)

The mining algorithms were executed on 1-node, 2-node, 4-node, and 8-node configurations.

We used a 6-month sample of the Wall Street Journal published from April 2, 1990 through September 28, 1990 for the performance comparison between PMIHP and Count Distribution as well as between MIHP and Apriori. The sample contains 21,703 documents and 116,849 unique words. We varied the minimum support level from 5% to 1.75% to measure its impact on the miners. For PMIHP and Count Distribution, the 6-month sample was sequentially distributed to the processing nodes by assigning the articles of 16 or 17 days to each node.

Figure 4 shows the total execution times of Apriori, FP-Growth and MIHP; and Fig. 5 shows the total execution times of PMIHP and Count Distribution when both algorithms were run on 8 nodes. The Apriori and Count Distribution algorithms were not able to run within the memory constraint of 416 Mbytes when the minimum support level is below 2%. The memory requirement for the candidate itemsets is the limiting factor for both Apriori and Count Distribution. MIHP has much better performance than Apriori because MIHP prunes many candidate itemsets by using the Inverted Hashing and Pruning, and processes a limited number of candidate itemsets at a time based on the Multipass approach. More performance analysis result of MIHP is available in [22].

The FP-Growth algorithm performed well at the higher minimum support levels. However, its performance deteriorated at the 2% minimum support level. The FP-tree becomes too large when the minimum support level is low, and as a result, the total execution time increases sharply. Lower minimum support levels were not attempted.

As shown in Fig. 5, PMIHP performs significantly better than Count Distribution, and the performance gain increases as the minimum support level decreases. It is important to note that the minimum support levels were selected in this test, such that

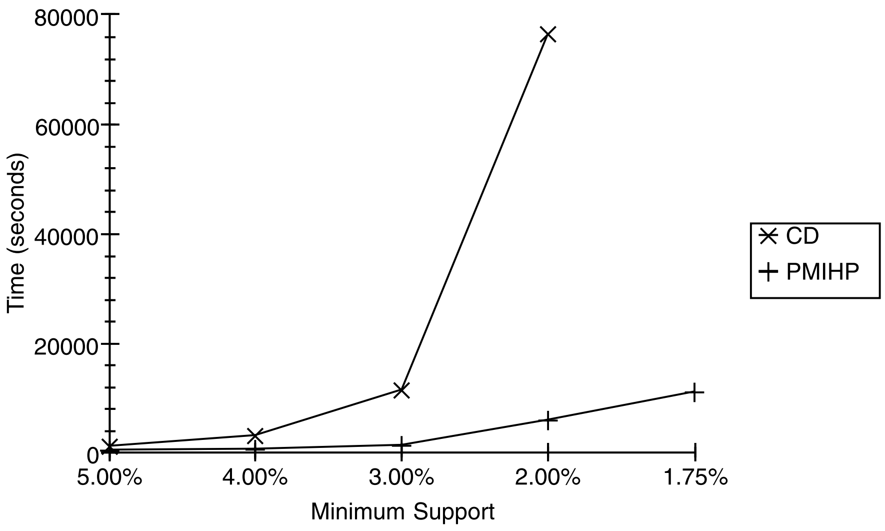


Fig. 5 Total execution time to find all frequent itemsets (in 21,703 documents, on 8 nodes)

the miners would run within the constraint of the main memory and thus eliminates the effect of paging upon the performance.

Comparing Figs. 4 and 5, we can see that the speedup of Count Distribution over Apriori is fairly good. When the minimum support level is 2%, it is about 6, whereas the speedup of PMIHP over MIHP is about 4. However, as we shall see below, the speedup of PMIHP is quite good at a lower minimum support level of 0.15%. Since the amount of computation increases rapidly as the minimum support level decreases, the speedup improvement at low minimum support levels is quite encouraging.

To evaluate the effect of the number of processing nodes on the performance of PMIHP, we used a 8-day sample of the Wall Street Journal, starting from October 1, 1991. There were 1,427 documents and 31,290 unique words. We used a minimum support count of 2 documents (i.e., minimum support of 0.15%) and a stop-word list from Fox [15]. There were 12,828 frequent words. The minimum support count of two documents was selected based upon the result of our experiments showing that low minimum support levels are required to use the frequent itemsets for document retrieval and ranking. We did not stem the words, but we monocased the words.

The database was assigned to the processing nodes sequentially by day. For the 2-node case, one node was assigned the articles of the first 4 days and another node was assigned the articles of the last 4 days. The assignments for the 4-node and 8-node cases were done in a similar manner. These daily collections have a mean of 178, a standard deviation of 22.58, and a median of 174 documents.

Figure 6 shows the total execution time of PMIHP required to find frequent 3-itemsets as the number of processing nodes changes, and Fig. 7 shows the corresponding speedup over the sequential processing. We can see that the speedup increases as the number of processing nodes increases, and the increasing rate is slightly higher than linear. The speedup is 1.65 for the 2-node system, indicating some degree of parallelization overhead mainly due to the interprocess communication to

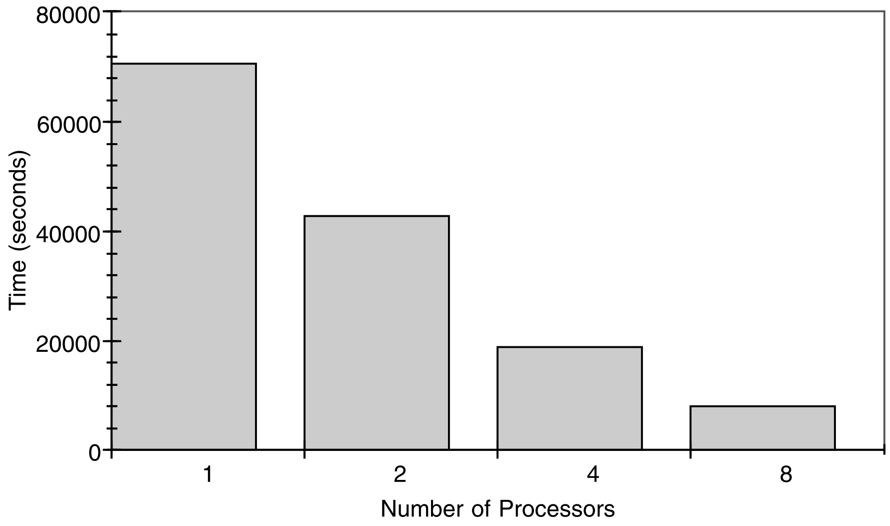


Fig. 6 Total execution time of PMIHP to find frequent 3-itemsets (in 1,427 documents, minsup = 0.15%)

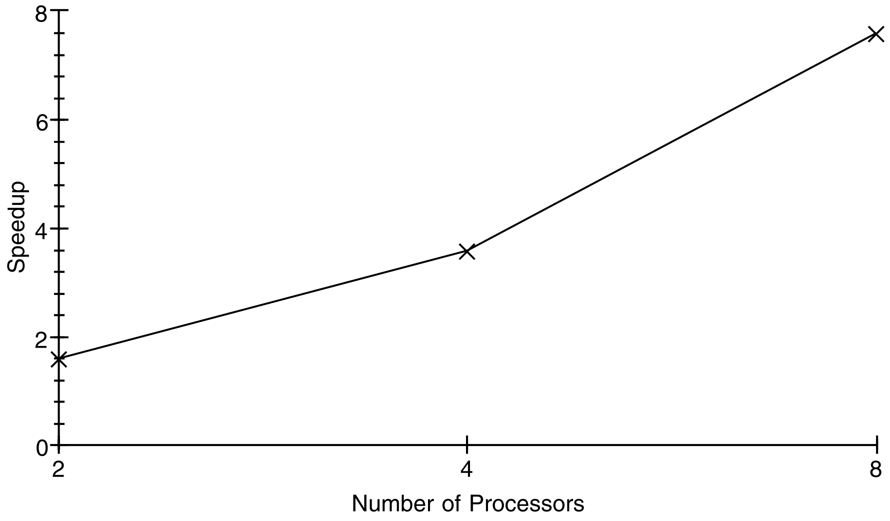


Fig. 7 Speedup of PMIHP (minsup = 0.15%)

exchange the support count information. The speedup is 3.76 for the 4-node system, but the increasing rate of the speedup is 2.27 as the number of nodes is doubled from 2 to 4. As the number of nodes is doubled from 4 to 8, the increasing rate of the speedup is higher than linear again, which indicates that PMIHP is quite scalable.

The PMIHP algorithm has two main data mining activities: counting the support of local candidate itemsets in the corresponding local database; and counting the support of global candidate itemsets in multiple local databases. These two activities are interleaved during the mining as the global support counting is invoked when

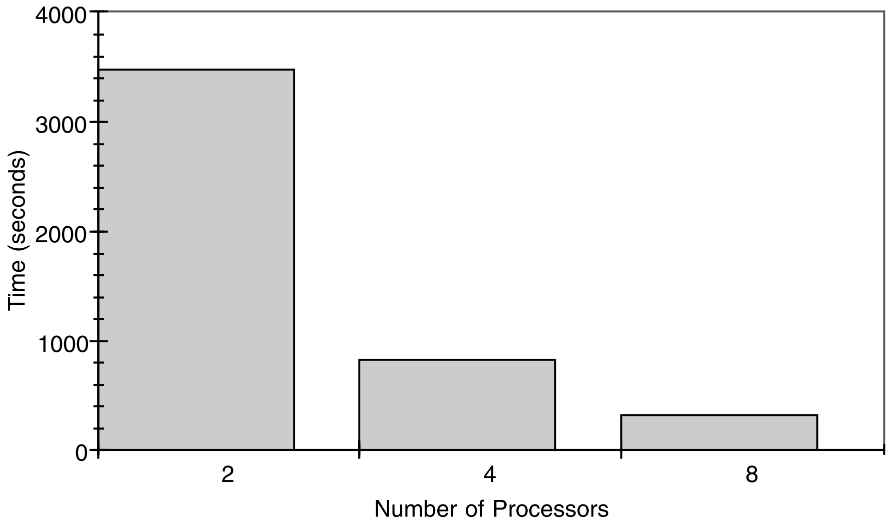


Fig. 8 Global support counting time to find frequent 3-itemsets

the number of identified global candidate itemsets exceeds a certain number at each processing node, which was set to 20,000 in our experiments. To measure this global support counting time, we reconfigured PMIHP to defer the global support counting of the global candidate itemsets at each node and synchronized the nodes before the start of the global support counting phase. Figure 8 shows the global support counting time of the mining process with the longest run time among all the mining processes executed on different processing nodes. Moreover, we used the wall clock time to measure this global counting time, hence it is an upper bound of the actual global support counting time of all the mining processes.

Comparing Figs. 6 and 8, we can see that the 2-node case has a much longer global support counting phase than 4-node and 8-node cases. The portion of the global support counting phase for the 2-node case is about 8% of the total execution time, but it is about 4% for the 4-node case, and about 3% for the 8-node case. Thus, the impact of the global support counting time on the overall speedup is very small, and it is reduced further as the number of processing nodes increases.

Since our processing environment does not provide the statistics for job accounting, exact CPU time measurement was not feasible. So, we measured the average execution of a processing node using the wall clock time. Figure 9 shows the average execution time of a node in the 1-node, 2-node, 4-node, and 8-node configurations. We can see that the 2-node case requires significantly less average execution time per node than the 1-node case; and as the number of processing nodes increases further, the average execution time per node deceases more than linearly. This result is completely consistent with the observed speedup values and also indicates that increased efficiency is behind the performance gain.

Since the identical PMIHP algorithm was executed on all of our system configurations, the differences in execution time must be associated with some workload differences. Figure 10 shows the average number of candidate 2-itemsets processed

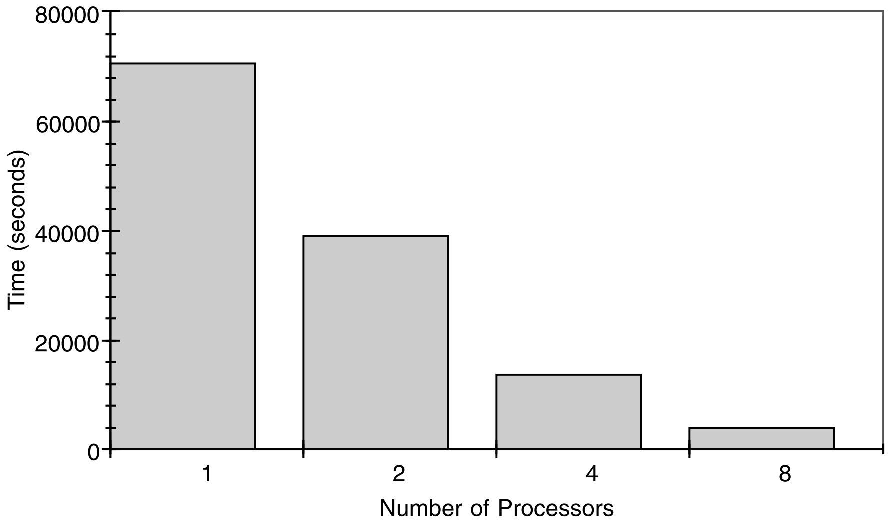


Fig. 9 Average execution time per node to find frequent 3-itemsets

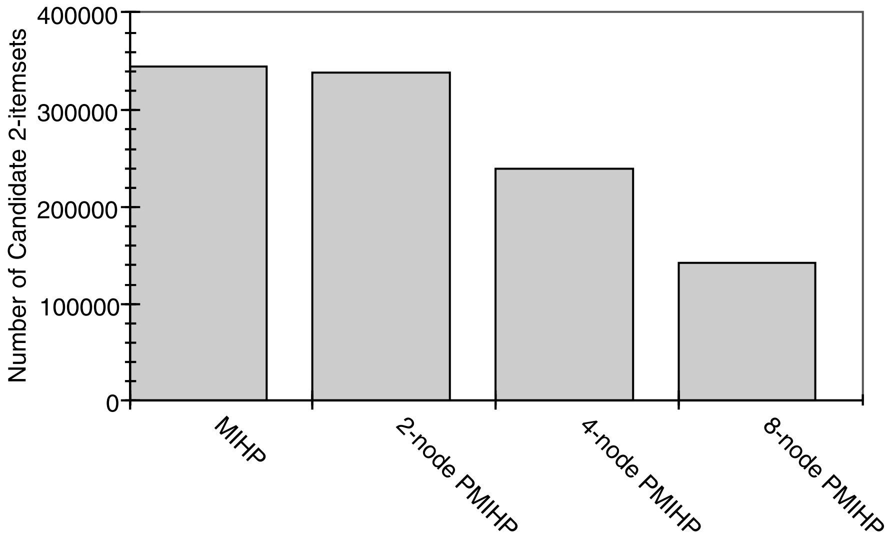


Fig. 10 Average number of candidate 2-itemsets per node

by each node in our four different system configurations. Note that the number of candidate 2-itemsets for the 1-node case is approximately the same as the average number of candidate 2-itemsets for the 2-node case. This result is consistent with the observed total and average execution times for the 1-node and 2-node cases. There is significant reduction in the average number of candidate 2-itemsets processed for the 4-node and 8-node cases over the 1-node and 2-node cases. This result represents the

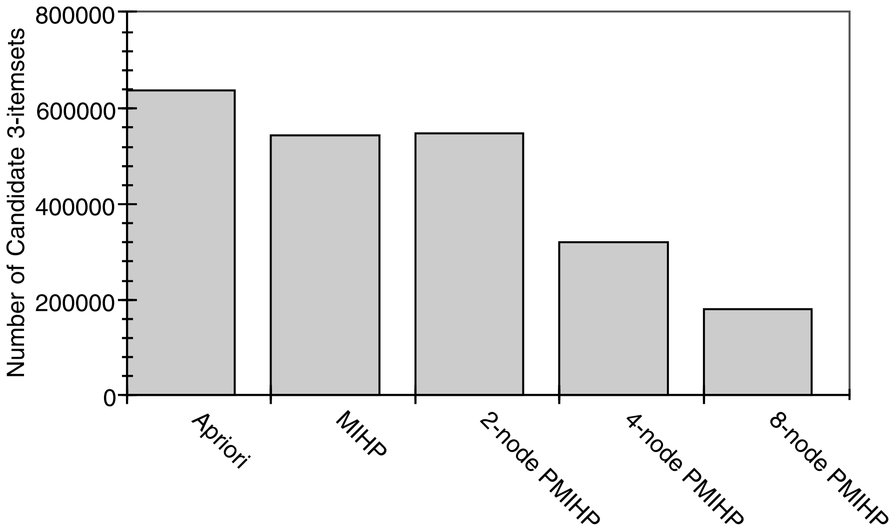


Fig. 11 Average number of candidate 3-itemsets per node

nonuniform distribution of itemsets over the local databases as well as the effective reduction of the candidate itemsets by the Inverted Hashing and Pruning technique.

Figure 11 shows the average number of candidate 3-itemsets processed by each node. We included the number of candidate 3-itemsets processed in Apriori to demonstrate the usefulness of the Inverted Hashing and Pruning. The number of candidate 2-itemsets for Apriori was about 82 million, which is why we did not show that in Fig. 10. We can observe the same pattern of reduction in the candidates 3-itemsets as in the candidate 2-itemsets. This reduction in the average number of candidate itemsets processed by each processing node may be the most clear explanation for the high increasing rate of the speedup observed as the number of processing nodes increases.

We also ran a test with a larger database: 8 weeks of the Wall Street Journal, published from January 2, 1991 through February 22, 1991 (February 23rd was a Wall Street Journal holiday). There were 6,170 documents and 64,191 unique words, of which 31,948 were frequent words at the minimum support level of 0.03%, i.e., 2 out of 6,170 documents. The 1-node system required 845,702 seconds to find 1,554,442 frequent 2-itemsets, whereas the 8-node system required 33,183 seconds. This performance represents a superliner speedup of 25.5 of the 8-node system over the 1-node system. Thus, we can conclude that the performance of PMIHP is quite scalable when the database is large and the minimum support level is low, which is the case of high workload.

The 1-node case generated 16,174,357 candidate 2-itemsets, whereas the 8-node case generated 2,459,629 candidate 2-itemsets per node on the average. The total number of candidate 2-itemsets counted by the 8 nodes were 19,677,031, which means that only 21.7% of the candidate 2-itemsets were counted at more than one processing node. This implies that the distribution of words across the 8-week sample of the Wall Street Journal is quite skewed.

In the Count Distribution algorithm, all the nodes count the same set of candidate itemsets in each pass over the database regardless of the distribution of items over the local databases. On the other hand, in our PMIHP algorithm, not all candidate itemsets are counted at more than one node when the distribution of items over the local databases is not uniform. Obviously, the more skewed the data distribution, the better the performance of PMIHP. Cheung et al. [10] proposed several approaches to partition the database to achieve a high degree of skewness. Text documents, arranged in a chronological order, do appear to have a high degree of skewness, and benefit the PMIHP algorithm.

Related to the distribution of candidate itemsets over the local databases, the minimum support level is an important factor. For the 6-month sample of Wall Street Journal (from April 2, 1990 through September, 28 1990) used in the first set of tests, we used a relatively high minimum support level of 1.75%. In that case, there was very little skew in the distribution of frequent items over the local databases. The 1-node system generated 1,177,462 candidate 2-itemsets, and the 8-node system generated 1,174,402 candidate 2-itemsets per node on the average. As a result, the speedup of PMIHP was not as high as the case of using a low minimum support level.

6 Conclusions

The proposed Parallel Multipass with Inverted Hashing and Pruning (PMIHP) algorithm is a parallel version of our Multipass with Inverted Hashing and Pruning (MIHP) algorithm, and it is effective for mining frequent itemsets in large text databases. The Multipass approach reduces the required memory space at each processor by partitioning the frequent items and processing each partition separately. Thus, the number of candidate itemsets to be processed is limited at each instance. The Inverted Hashing and Pruning is used to prune the local and global candidate itemsets at each processing node; and it also allows each processing node to determine the other peer processing nodes to poll in order to collect the local support counts of each global candidate itemset.

PMIHP distributes the workload to multiple processing nodes to reduce the total mining time without incurring much parallelization overhead. The average number of candidate itemsets to be counted at each processing node is much smaller than the case of sequential mining, while the time for the synchronization between processing nodes to exchange the count information for the global candidate itemsets is very small compared to the total execution time.

PMIHP is able to exploit the natural skewed distribution of words in text databases, and demonstrates a superlinear speedup as the number of processing nodes increases. It has a much better performance than well-known parallel Count Distribution algorithm [3], because the average number of candidate itemsets to be counted at each processing node is much smaller, especially when the minimum support level is low. Overall, the performance of PMIHP is quite scalable even when the size of the text database is large and the minimum support level is low, which is the case of high workload.

Currently we are trying to apply the mined word associations to enhance the precision of text retrieval by grouping the documents based on their common frequent words. We also investigate the extension of the PMIHP algorithm for the mining of meaningful sequential patterns.

References

1. Agarwal RC, Aggarwal CC, Prasad VVV (2000) Depth first generation of long patterns. In: Proc of the 6th ACM SIGKDD int'l conf on knowledge discovery and data mining, 2000, pp 108–118
2. Agrawal R, Srikant R (1994) Fast algorithms for mining association rules. In: Proc of the 20th VLDB conf, 1994, pp 487–499
3. Agrawal R, Shafer JC (1996) Parallel mining of association rules. IEEE Trans Knowl Data Eng 8(6):962–969
4. Agarwal R, Aggarwal C, Prasad V (2001) A tree projection algorithm for generation of frequent item sets. J Parallel Distrib Comput 61(3):350–371
5. Bayardo RJ (1998) Efficient mining long patterns from databases. In: Proc of ACM SIGMOD int'l conf on management of data, 1998, pp 85–93
6. Brin S, Motwani R, Ullman J, Tsur S (1997) Dynamic itemset counting and implication rules for market basket data. In: Proc of ACM SIGMOD int'l conf on management of data, 1997, pp 255–264
7. Burdick D, Calimlim M, Gehrke J (2001) MAFIA: a maximal frequent itemset algorithm for transaction databases. In: Proc of int'l conf on data engineering, 2001, pp 443–452
8. Chen MS, Han J, Yu PS (1996) Data mining: an overview from a database perspective. IEEE Trans Knowl Data Eng 6(8):866–883
9. Cheung DW, Ng VT, Fu AW, Fu Y (1996) Efficient mining of association rules in distributed databases. IEEE Trans Knowl Data Eng 8(6):911–922
10. Cheung DW, Lee SD, Xiao Y (2002) Effect of data skewness and workload balance in parallel data mining. IEEE Trans Knowl Data Eng 14(3):498–514
11. Chung SM, Yang J (1996) A parallel distributive join algorithm for cube-connected multiprocessors. IEEE Trans Parallel Distrib Syst 7(2):127–137
12. Chung SM, Luo C (2004) Distributed mining of maximal frequent itemsets from databases on a cluster of workstations. In: Proc of the 4th IEEE/ACM int'l symp on cluster computing and the grid—CCGrid 2004, 2004
13. Feldman R, Hirsh H (1998) Finding associations in collections of text. In: Michalski R, Bratko I, Kubat M (eds), Machine learning and data mining: methods and applications. Wiley, pp 223–240
14. Feldman R, Dagen I, Hirsh H (1998) Mining text using keyword distributions. J Intell Inf Syst 10(3):281–300
15. Fox C (1992) Lexical analysis and stoplists. In: Frakes W, Baeza-Yates R (eds), Information retrieval: data structures and algorithms. Prentice Hall, pp 102–130
16. Gordon M, Dumais S (1998) Using latent semantic indexing for literature based discovery. J Am Soc Info Sci 49(8):674–685
17. Gouda K, Zaki MJ (2001) Efficiently mining maximal frequent itemsets. In: Proc of the 1st IEEE int'l conf on data mining, 2001, pp 163–170
18. Han EH, Karypis G, Kumar V (2000) Scalable parallel data mining for association rules. IEEE Trans Knowl Data Eng 12(3):337–352
19. Han J, Pei J, Yin Y (2000) Mining frequent patterns without candidate generation. In: Proc of ACM SIGMOD int'l conf on management of data, 2000, pp 1–12
20. Holt JD, Chung SM (2001) Multipass algorithms for mining association rules in text databases. Knowl Inf Syst 3(2):168–183
21. Holt JD, Chung SM (2002) Mining association rules using inverted hashing and pruning. Inf Proces Lett 83(4):211–220
22. Holt JD, Chung SM (2002) Mining association rules in text databases using multipass with inverted hashing and pruning. In: Proc of the 14th IEEE int'l conf on tools with artificial intelligence, 2002, pp 49–56
23. National Institute of Standards and Technology (NIST) (1997) Text Research Collection
24. Orlando S, Palmerini P, Perego R (2001) Enhancing the apriori algorithm for frequent set counting. In: Proc of int'l conf on data warehousing and knowledge discovery, 2001, pp 71–82
25. Park JS, Chen MS, Yu PS (1995) Efficient parallel data mining for association rules. In: Proc of ACM int'l conf on information and knowledge management, 1995, pp 31–36

26. Park JS, Chen MS, Yu PS (1997) Using a hash-based method with transaction trimming for mining association rules. IEEE Trans Knowl Data Eng 9(5):813–825
27. Salton G (1988) Automatic text processing: the transformation, analysis, and retrieval of information by computer. Addison-Wesley
28. Savasere A, Omiecinski E, Navathe S (1995) An efficient algorithm for mining association rules in large databases. In: Proc of the 21st VLDB conf, 1995, pp 432–444
29. Shintani T, Kitsuregawa M (1996) Hash based parallel algorithms for mining association rules. In: Proc of the 4th int'l conf on parallel and distributed information systems, 1996, pp 19–30
30. Toivonen H (1996) Sampling large databases for association rules. In: Proc of the 22nd VLDB conf, 1996, pp 134–145
31. Zaiane OR, El-Hajj M, Lu P (2001) Fast parallel association rule mining without candidacy generation. In: Proc of IEEE conf on data mining, 2001, pp 665–668
32. Zaiane OR, Antoine ML (2002) Classifying text documents by associating terms with text categories. In: Proc of the 13th australian database conf, 2002
33. Zaki MJ, Parthasarathy S, Ogihara M, Li W (1997) Parallel algorithms for fast discovery of association rules. Data Min Knowl Discov 1(4):343–373
34. Zaki MJ (2000) Scalable algorithms for association mining. IEEE Trans Knowl Data Eng 12(3):372–390