Knowl Inf Syst (2008) 16:359–391
DOI 10.1007/s10115-007-0115-1

REGULAR PAPER

Efficient mining of maximal frequent itemsets from databases on a cluster of workstations

Soon M. Chung · Congnan Luo

Received: 15 April 2005 / Revised: 31 August 2007 / Accepted: 8 September 2007 /
Published online: 12 December 2007

Abstract In this paper, we propose two parallel algorithms for mining maximal frequent itemsets from databases. A frequent itemset is maximal if none of its supersets is frequent. One parallel algorithm is named *distributed max-miner* (DMM), and it requires very low communication and synchronization overhead in distributed computing systems. DMM has the local mining phase and the global mining phase. During the local mining phase, each node mines the local database to discover the local maximal frequent itemsets, then they form a set of maximal candidate itemsets for the top-down search in the subsequent global mining phase. A new prefix tree data structure is developed to facilitate the storage and counting of the global candidate itemsets of different sizes. This global mining phase using the prefix tree can work with any local mining algorithm. Another parallel algorithm, named *parallel max-miner* (PMM), is a parallel version of the sequential max-miner algorithm (Proc of ACM SIGMOD Int Conf on Management of Data, 1998, pp 85–93). Most of existing mining algorithms discover the frequent k-itemsets on the kth pass over the databases, and then generate the candidate $(k+1)$-itemsets for the next pass. Compared to those level-wise algorithms, PMM looks ahead at each pass and prunes more candidate itemsets by checking the frequencies of their supersets. Both DMM and PMM were implemented on a cluster of workstations, and their performance was evaluated for various cases. They demonstrate very good performance and scalability even when there are large maximal frequent itemsets (i.e., long patterns) in databases.

Keywords Parallel data mining · Maximal frequent itemsets · Association rules · Scalability

S. M. Chung (✉) · C. Luo
Department of Computer Science and Engineering,
Wright State University, Dayton, OH, USA
e-mail: soon.chung@wright.edu

C. Luo
e-mail: luo.3@wright.edu

1 Introduction

Discovery of association rules is an important problem in data mining. With numerous practical applications, such as consumer market-basket data analysis, it has received tremendous amount of attention from the data mining research community. 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]. 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 *frequent itemset*. 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. 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*.

In mining association rules, the most time-consuming job is finding all frequent itemsets from a large database with respect to a given minimum support. Many sequential and parallel algorithms have been proposed to solve this problem. The most common sequential algorithms are Apriori [2] and its variations. Apriori-based algorithms employ a strict bottom-up, breadth-first search and enumerate every frequent itemset [4]. They require multiple passes over the database. At each pass, they count the candidate itemsets to identify frequent itemsets and then use them to generate longer candidates for the next pass. In Apriori, if F_{k-1} denotes the set of frequent $(k-1)$-itemsets discovered in the $(k-1)$th pass over the database, then the set of candidate k-itemsets for the next pass, denoted by C_k, is obtained by the natural join of F_{k-1} and F_{k-1} on the first $k-2$ items. For example, if F_2 includes $\{1, 2\}$ and $\{1, 3\}$, then $\{1, 2, 3\}$ is a candidate 3-itemset. However, a candidate k-itemset is pruned if any of its $(k-1)$-subsets is not frequent. Thus, for $\{1, 2, 3\}$ to remain as a candidate 3-itemset, $\{2, 3\}$ also should be a frequent 2-itemset. This *subset-infrequency based pruning* step prevents many candidate itemsets from being counted in each pass. Many algorithms, such as direct hashing and pruning (DHP) [21], Partition [23], DIC [5], Multipass [16] and inverted hashing and pruning (IHP) [17], are based on Apriori and use various approaches to improve candidate generation and pruning.

All the above Apriori-based algorithms share the basic candidate generation method proposed in Apriori, where they have to generate and count 2^l subsets for a single frequent itemset with length l. This exponential complexity makes Apriori-based algorithms just suitable for mining small frequent itemsets (i.e., short patterns) [4]. To address this problem, algorithms that can efficiently mine the maximal frequent itemsets (MFIs) were proposed. The basic idea is that if we find a large frequent itemset (i.e., a long pattern) early, we can avoid counting all its subsets because all of them are frequent. Another interesting approach is mining only the closed frequent itemsets [22,30]. A frequent itemset is *closed* if it has no proper superset with the same support. Obviously, the relationship between the set of all frequent itemsets (FIs), the set of closed frequent itemsets (CFIs), and the set of maximal frequent itemsets (MFIs) is $\{\text{MFIs}\} \subseteq \{\text{CFIs}\} \subseteq \{\text{FIs}\}$. Compared with mining MFIs, which does not provide the count information about non-maximal frequent itemsets, an advantage of mining CFIs is that the count information about all frequent itemsets is also obtained. However, in many dense databases, the set of CFIs could be still too large and orders of magnitude larger than the set of MFIs [6,12]. In those cases, the only recourse is to mine MFIs. In many real-life data mining applications, the mining of association rules is an interactive process, where quickly obtaining the set of MFIs first and then selecting only the interesting patterns subsumed by this set for further counting may be desirable. More importantly, managing, querying, representing and visualizing a small set of maximal patterns is much easier, time-saving, and space-saving. There are many applications where the set of maximal patterns is adequate, such as Web-log analysis and combinatorial pattern discovery in biological domains.

Representative MFI mining algorithms include max-miner [4], Pincer-Search [18], DepthProject [1], MIFIA [6], GenMax [12], MaxClique and MaxEclat [27,29]. Both MaxEclat and MaxClique generate the set of potential maximal frequent itemsets using the frequent 2-itemsets discovered. They look ahead only during their initialization phase; and two clustering schemes based on equivalence classes and maximal hypergraph cliques are used, respectively, to generate potential maximal frequent itemsets. The equivalence class approach works at a lower computation cost but sacrifices the quality of the candidate itemsets it generates, whereas the hypergraph clique approach can generate a more refined candidate set at a higher computation cost by using a dynamic programming algorithm. When the edge density of the equivalent class graph increases, the hypergraph clique approach may have a high computation overhead [27]. Max-miner, Pincer-Search, DepthProject, MAFIA and GenMax attempt to find out the maximal frequent itemsets as early as possible throughout the whole mining process.

In our research, we compared the performance of max-miner against Apriori, MAFIA, and GenMax on synthetic transaction databases generated as in [2]. The result shows that, in general, the larger the average length of potential maximal frequent itemsets, the better max-miner performs than Apriori. Max-miner also demonstrated much better scalability than MAFIA and GenMax in terms of the number of items and the number of transactions in the database. We will discuss more details about these algorithms later in Sects. 2 and 4.

Another interesting sequential algorithm is the FP-Growth [15] which 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 advantages of FP-Growth come directly from 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 may not fit in main memory, and then the processing time will increase considerably. Our experiments showed that FP-Growth is much slower than Apriori when there are many candidate itemsets as the database size is large and the minimum support level is low.

Due to the large size of the database to be mined, parallel data mining is very promising, and a few parallel algorithms were proposed for mining association rules. The most well-known one is the count distribution algorithm [3], which is a parallel version of the Apriori algorithm. In count distribution, the database is partitioned and distributed over multiple 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, as in Apriori. 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 nodes is redundant.

The data distribution algorithm [3] is designed to better use the memory space of nodes by partitioning the candidate itemsets. Each node counts just one partition of the candidate itemsets but needs to access all the database partitions. To solve this problem, the intelligent data distribution algorithm [14] is proposed. It uses the ring-based communication between nodes to reduce the network contention caused by the transfer of database partitions between nodes. The PDM algorithm [20] is a parallel version of the direct hashing and pruning (DHP) algorithm, and each node uses a hash table to prune the candidate itemsets. However, like DHP, due to the overhead of hashing, it doesn't show better performance than count distribution.

The FDM [7] and FPM [8] algorithms are enhanced versions of count distribution (CD). Two pruning techniques, named distributed pruning and global pruning, are used to reduce the number of candidates at each pass, and thus reduce both computation and communication overhead of CD. In FDM, two rounds of communication are required in each iteration: one for computing the global supports, and the other for broadcasting the frequent itemsets. Compared with FDM, FPM is more efficient in communication by simply broadcasting local supports to all processors. In addition, how the data skewness affects the power of distributed pruning and global pruning is studied in [8]. However, the communication cost of FPM is still determined by the candidate set size, as in CD. Thus, for small *minsup*, the communication cost could be very high at some passes where the candidate set is large.

All the above parallel algorithms are almost directly based on Apriori by sharing the same candidate generation method. Thus, if some frequent itemsets are long, they also face the same exponential complexity problem as Apriori.

A natural solution to avoid the exponential complexity problem of enumerating frequent itemsets is the parallelization of the MFI mining algorithms. There are only few parallel MFI mining algorithms proposed. Par-MaxEclat and Par-MaxClique [28] are the parallel versions of MaxEclat and MaxClique, respectively. After generating potential maximal frequent itemsets, they are clustered and then distributed over the processors. To reduce the computation and disk I/O, the transaction database is transformed into the vertical format, where each item is associated with a set of transaction identifiers (TIDs) of the transactions containing the item. Based on the clustering of the potential maximal frequent itemsets, related TID-sets are transmitted to the corresponding processors. After this setup phase, each processor can perform the mining independently without any communication and synchronization with others. Like MaxEclat and MaxClique, both Par-MaxEclat and Par-MaxClique look ahead based on the information about frequent 2-itemsets, and it makes them prone to generating many potential maximal frequent itemsets which are not globally frequent.

In this paper, we propose two new parallel algorithms for mining maximal frequent itemsets. They are named *distributed max-miner* (DMM) and *parallel max-miner* (PMM). They are implemented on a Linux cluster system where the database is partitioned over all the nodes involved in the mining.

PMM is a parallel version of the sequential max-miner algorithm. However, it requires multiple passes over the database, like the count distribution algorithm. In each pass, all nodes have the same set of candidate groups to be counted; and after each pass, the support count information is exchanged between nodes, so that every node holds the same global count information and then generates the same candidate groups for the next pass. PMM uses this bottom-up search, but it looks ahead at each pass and prunes more candidate itemsets by checking the frequencies of their supersets.

DMM has two phases: local mining phase and global mining phase. In the local mining phase, each node directly applies a sequential MFI mining algorithm to find all local maximal frequent itemsets in its local database. Then, all nodes exchange and merge local MFIs and their local support counts to construct an initial global candidate set. In the global mining phase, all nodes scan their local databases to count the support of global candidates in each pass. If a global candidate k-itemset is not globally frequent, all its $(k-1)$-subsets are included into the global candidate set for the next global pass. This global mining phase will continue until the global candidate set is empty. We used the sequential max-miner for the local mining phase, but DMM can use any sequential MFI mining algorithm for the local mining phase.

DMM is different from PMM in the way that it allows all nodes independently mine local MFIs first, then uses a top-down search in the global mining phase. As a result, DMM can reduce the synchronization and communication overhead, which is critical in the distributed environment. DMM requires only one synchronization at the end of the local mining phase, and just a few synchronizations during the global mining phase in most cases. Thus, the total number of synchronizations is much smaller than those of PMM and count distribution.

DMM and PMM do not require the transfer of transactions between the processors. As the database partitions allocated to the processors are disjoint from each other, the total amount of data on all the processors is same as the size of the original database. On the other hand, Par-MaxEclat and Par-MaxClique [28] require the transfer of the TID-sets between the processors. Since the potential maximal frequent itemsets distributed to different processors usually have many items in common, the TID-sets for the common items must be transmitted to more than one processor. As a result, the more processors are used for mining, the bigger the communication overhead in the setup phase of Par-MaxEclat and Par-MaxClique. The total amount of the data on all the processors is usually much larger than the size of the original database. Thus, both Par-MaxEclat and Par-MaxClique are not scalable with respect to the number of processors.

As reported in [28], when the number of processors is increased from 1 to 8, the average speedups of Par-MaxEclat and Par-MaxClique are less than 4.0 in mining three different synthetic databases. In our research, we used the same synthetic database generation program and similar parameter values to generate five different databases. Our tests show that, in mining these five databases with our 8-node cluster, PMM and DMM achieved much better average speedups of 6.9 and 7.4, respectively. For the 32-processor cases reported in [28], the speedups of Par-MaxEclat and Par-MaxClique are even lower, only between 2.5 and 5.5. As pointed out by the authors of [28], if the communication cost is not included, the speedup of Par-MaxEclat and Par-MaxClique could be 12 to 16 for the 32-processor cases. This shows that the communication overhead is indeed the performance bottleneck of Par-MaxEclat and Par-MaxClique. In contrast, PMM and DMM are very efficient in terms of communication, and hence more suitable for large-scale parallel/distributed data mining.

DMM uses a prefix tree structure to store and count the global candidates with different lengths during the global mining phase. A similar data structure is also used in some other algorithms, such as DIC [5]. In DIC, the prefix tree grows during the scanning of the database. On the other hand, the prefix tree of DMM is fixed once it is constructed, so it is quite easy to maintain. Furthermore, in DIC, candidate itemsets are generated on-the-fly (during each pass) and the tree contains all candidate itemsets, hence how to control the size of the tree is a problem. In DMM, only potential maximal frequent itemsets are inserted into the prefix tree, so the tree size is much smaller. To maintain the prefix tree as small as possible, three techniques are used in DMM to reduce the size of candidate set: (1) estimation of the support of some global candidates; (2) subset-infrequency based pruning; and (3) superset-frequency based pruning. Our experiments show that (1) and (3) can effectively reduce the size of the initial global candidate set, whereas (2) and (3) can make the candidate set shrink considerably during the global mining phase.

The rest of the paper is organized as follows: Sect. 2 reviews more details of the max-miner algorithm, which is a basis of the proposed PMM and DMM algorithms. Section 3 describes the PMM and DMM algorithms, and Sect. 4 presents the results of performance analyses and comparisons of the PMM, DMM, and count distribution algorithms in various cases. Section 5 contains some conclusions and future work.

2 Max-miner algorithm

Unlike Apriori-like algorithms, the max-miner algorithm [4] extracts only the maximal frequent itemsets, from which all frequent itemsets can be obtained. Max-miner always attempts to look ahead in order to identify large frequent itemsets early, so that all subsets of these discovered frequent itemsets can be pruned from the search space. This method is called *superset-frequency based pruning*. Max-miner used the set-enumeration tree to represent the search space. How to construct the set enumeration tree is illustrated in Fig. 1 for four items: 1, 2, 3, and 4. The set-enumeration tree lists all combinations of the four items from level 0 to level 4.

Each node in the tree is called a candidate group, and a candidate group g consists of two components which are actually two itemsets. The first itemset is called the *head* of the group and denoted by $h(g)$. The second itemset is called the *tail* of the group and denoted by $t(g)$. $t(g)$ is an ordered set and contains all the items not in $h(g)$ but can potentially appear in any subnode derived from node g. For example, the node (i.e., candidate group) g_1 enumerating item 1, which is the leftmost node at level 1, has two components: $h(g_1) = \{1\}$ and $t(g_1) = \{2, 3, 4\}$.

The main procedure of max-miner can be explained as follows. From the root of the tree at level 0, we count the support of 1-itemsets. Only the 1-itemsets which are frequent can be enumerated at level 1. In this example, 4 nodes are generated at level 1 if 1, 2, 3, and 4 are all frequent 1-itemsets. For node g_1, which corresponds to item 1 at level 1, we need to count the support of $\{h(g_1) \bigcup t(g_1)\} = \{1, 2, 3, 4\}$. If the support of $\{h(g_1) \bigcup t(g_1)\}$ is greater than or equal to *minsup*, then we do not need to expand the tree from node g_1 anymore due to the superset-frequency based pruning. Similarly, for node g_2, which corresponds to item 2 at level 1, the support of $\{h(g_2) \bigcup t(g_2)\} = \{2, 3, 4\}$ is counted. If $\{h(g_2) \bigcup t(g_2)\}$ is frequent, then the subnodes of node g_2 are not generated at level 2.

At any node g, if $\{h(g) \bigcup t(g)\}$ is not frequent, then for each item i in $t(g)$, we check if $\{h(g) \bigcup i\}$ is frequent. If $\{h(g) \bigcup i\}$ is frequent, a corresponding subnode is generated. The

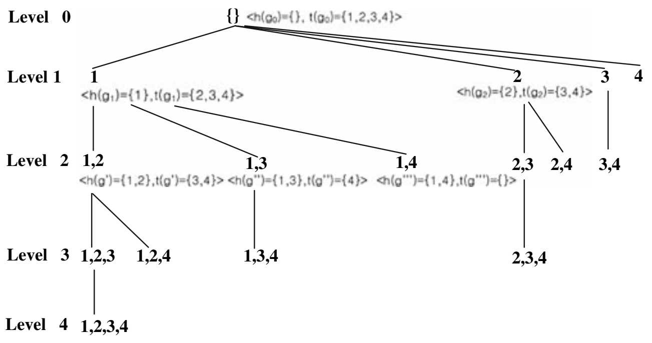

Level 0
{} <h(g0)={}, t(g0)={1,2,3,4}>
Level 1
1
<h(g1)={1},t(g1)={2,3,4}>
2
<h(g2)={2},t(g2)={3,4}>
3
4
Level 2
1,2
<h(g')={1,2},t(g')={3,4}>
1,3
<h(g'')={1,3},t(g'')={4}>
1,4
<h(g''')={1,4},t(g''')={}>
2,3
2,4
3,4
Level 3
1,2,3
1,2,4
1,3,4
2,3,4
Level 4
1,2,3,4

Fig. 1 Search space of max-miner

head of the subnode is $\{h(g) \bigcup i\}$ and its tail contains all the item j, such that $\{h(g) \bigcup j\}$ is frequent and j comes after i in $t(g)$. For example, if $\{1, 2\}$, $\{1, 3\}$ and $\{1, 4\}$ are frequent when $\{1, 2, 3, 4\}$ is not frequent, then we generate three subnodes of node g_1. They are enumerated at level 2 as: $[h(g') = \{1, 2\}, t(g') = \{3, 4\}]$, $[h(g'') = \{1, 3\}, t(g'') = \{4\}]$, and $[h(g''') = \{1, 4\}, t(g''') = \{\}]$. On the other hand, if $\{h(g) \bigcup i\}$ is not frequent, a subnode having $\{h(g) \bigcup i\}$ as the head is not generated because none of its superset would be frequent. This corresponds to the subset-infrequency based pruning of candidates. This process continues level by level until the whole tree is completed.

Here, we can notice that, for a candidate group node g, if an item appears last in the tail of g in ordering, it will appear in most offsprings of node g [4]. For example, item 4 is the last item in the tail of the root node in Fig. 1, and appears in either the head or tail of every node at level 1 and below. On the other hand, item 2 appears in a much smaller number of nodes. Thus, to discover the long patterns early, we better order the subnodes of each node in ascending order of their support. At level 1 of the set-enumeration tree, max-miner orders the frequent 1-itemsets in ascending order of their support. At lower levels, it orders the tail items of each node g in ascending order of the support of $(\{h(g) \bigcup i\})$, for $i \in t(g)$. In the above example, if support($\{1, 4\}$) $\leq$ support($\{1, 2\}$) $\leq$ support($\{1, 3\}$) when we generate the subnodes of node g_1, then the items are reordered in its tail $t(g_1)$ as $\{4, 2, 3\}$; and the corresponding subnodes are generated in that order as: $[h(g') = \{1, 4\}, t(g') = \{2, 3\}]$, $[h(g'') = \{1, 2\}, t(g'') = \{3\}]$, and $[h(g''') = \{1, 3\}, t(g''') = \{\}]$. This heuristic reordering strategy increases the effectiveness of the superset-frequency based pruning.

In [4], they also proposed how to estimate a lower-bound of the support of each itemset using the support of some of its subsets. By discovering the maximal frequent itemsets early, max-miner can reduce the number of passes over the database. As a result, it can reduce the total execution time considerably, compared to other level-wise algorithms.

2.1 Related work

Other MFI mining algorithms using the enumeration tree are DepthProject [1], MAFIA [6] and GenMax [12]. The DepthProject algorithm performs a depth-first search in the enumeration tree of itemsets, and projects the transaction database along the path of search. By projecting the transactions at a node, the mining for the subtree rooted at that node becomes a completely independent itemset generation problem with a substantially reduced transaction database [1]. A look-ahead pruning is used with a dynamic reordering of the children of each node. The MAFIA algorithm integrates three pruning strategies into the depth-first search. The first is the look-ahead pruning used in max-miner. The second is to check if the {head $\bigcup$ tail} of a node is subsumed by a maximal frequent itemset already found. The third is called parent equivalence pruning (PEP), which moves an item from the tail of a node to the head of the node if each transaction containing the head also contains the item. GenMax also performs a depth-first search and employs a technique named *progressive focusing* which reduces the number of maximal frequent itemsets to be checked to determine whether a frequent itemset is subsumed by a maximal frequent itemset or not. GenMax uses the vertical database format, and the *diffset propagation* is used for fast frequency counting.

MAFIA was reported better than DepthProject in [6]. Max-miner was compared with MAFIA and GenMax in [12], and the tests show that max-miner has the best performance for sparse synthetic databases, while MAFIA and GenMax can perform better for small dense databases. There are two reasons why we parallelized max-miner. First, max-miner is much easier to parallelize. Unlike the parallelization of the algorithms using the vertical database format, we don't need to transfer the TID-sets of the transactions between the computing nodes. This transfer can easily cause a heavy communication overhead, which could be a performance bottleneck in the distributed computing environment. Second, but more important, max-miner has much better scalability in terms of the number of items and the number of transactions in the database. Our test results showed that MAFIA cannot mine the database with a large number of items efficiently, and GenMax is not scalable with respect to the database size. This feature makes max-miner more suitable for mining business databases which often have tens of thousands of items and millions of transactions. We will show relevant test results in the performance evaluation section and analyze why max-miner is more scalable. However, we'd like to point out that our proposed DMM algorithm can use any sequential MFI mining algorithm during the local mining phase.

3 Parallel maximal frequent itemsets mining algorithms

We present the PMM and DMM algorithms in this section. They are implemented on a shared-nothing multiprocessor system where each node has local memory and a local disk. All nodes communicate with each other by passing messages through a communication network. The database is evenly divided into N partitions $\{D^0, D^1, D^2, \ldots, D^{N-1}\}$, one for each of the N nodes $\{P^0, P^1, P^2, \ldots, P^{N-1}\}$ involved in the parallel mining, so that each node has the same number of transactions allocated.

For DMM, the set of local maximal frequent itemsets discovered at node P^i is LM^i, for $0 \leq i \leq N-1$. For each pass in the global mining phase of DMM, the candidate set at each node is the set of all global potential maximal frequent itemsets that should be counted during the pass. The global candidate set is denoted by GC_t, where t represents the tth pass in the global mining phase. For PMM, all nodes have exactly the same set of candidate groups at each pass k, which is denoted by C_k. In both algorithms, the final result is the set of all (global) maximal frequent itemsets, and is denoted by GM. For any k-itemset, we will call each of its $(k-1)$-subsets a *largest proper subset*.

3.1 Parallel max-miner (PMM)

PMM requires multiple passes over the database. For each pass k, every node counts the occurrences of the candidate groups in C_k by scanning the local database, independently. At the end of each pass, all nodes exchange the count information so that they can generate the same set of candidate groups C_{k+1} for the next pass. After the first pass, each node determines which items are frequent, and in the second pass, each node counts the occurrences of all the pairs of frequent 1-itemsets using a two-dimensional data array to maintain the counts of the candidate 2-itemsets appearing in the local database. Here, we do not count the occurrences of the large itemset $\{h(g) \bigcup t(g)\}$ of each candidate group g because those large itemsets turn out to be infrequent most of the time at this stage, as reported in [4].

After exchanging and merging of the local counts of the candidate 2-itemsets, every node can identify the same set of frequent 2-itemsets. Then, each node generates the same set of candidate groups for the following third pass. The procedure of generating the candidate groups for the kth pass, $k \geq 3$, is exactly same as that of sequential max-miner algorithm. For example, if $\{1, 2\}$, $\{1, 3\}$, and $\{1, 4\}$ are frequent 2-itemsets, then we generate candidate groups whose heads are $\{1, 2, 3\}$, $\{1, 2, 4\}$, and $\{1, 3, 4\}$, respectively. Like max-miner, at any node in the enumeration tree, if $\{h(g) \bigcup i\}$ is not frequent, where i is an item in $t(g)$, the subnode having $\{h(g) \bigcup i\}$ as its head will not be generated because none of its supersets would be frequent. Thus, the subset-infrequency based pruning is also applied in PMM. In general, each candidate group generated for the kth pass has k items in its head, and a hash tree is used to store and count the candidate groups when $k \geq 3$.

This process of counting candidate groups, exchanging and merging count information, and generating candidate groups for the next pass is repeated until there is no more candidate group for the next pass. PMM algorithm is simple and efficient, and it has low communication overhead because it does not require the transferring of transactions between nodes during the processing. Like count distribution, PMM requires the synchronization between nodes to exchange the count information after each pass. However, the number of passes required in PMM is smaller than that of count distribution because the superset-frequency based pruning is performed in each pass by looking ahead to find maximal frequent itemsets.

3.1.1 Cube-based communication between processors

To perform the communication between processing nodes efficiently, we impose a logical binary n-cube structure on processing nodes. Then, nodes can exchange and merge the local count information through increasingly higher dimensional links between them [9]. 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 different dimensional links. For example, there are 8 nodes in a 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 nodes can exchange and merge their local counts 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.

The MPICH [24] library provides the *allreduce* function which can perform basic predefined operations like sum, product, max, min, etc. on primitive data types, like integer and double, on multiple processing nodes. For the user-defined data types, we need to define the corresponding operators to perform the *allreduce* function. However, we didn't use the *allreduce* function because we have to deal with the itemsets of different length, and not only the addition of the local counts of itemsets, but also complex pruning operations.

3.2 Pseudo-code of PMM

As we assume a homogeneous distributed computing environment where all the processing nodes are the same, we just give the pseudo-code of the PMM algorithm running on a node P^i.

```
/* Pass 1 */
$P^i$ counts the occurrences of items in $D^i$;
$n = \log_2 N$; /* $N$ nodes are used for mining */
**for** ($j = 1$; $j \leq n$; $j++$)
    $P^i$ exchanges and merges the local counts of items
    with a neighbor node through the $j$th-dimensional link;
$P^i$ determines $F_1$; /* $F_1$ is the set of frequent 1-itemsets */
```

```
/* Pass 2 */
$P^i$ generates $C_2$ by pairing the members of $F_1$;
$P^i$ counts the occurrences of the $C_2$ members in $D^i$;
for $(j = 1; j \leq n; j++)$
    $P^i$ exchanges and merges the local counts of $C_2$ members
    with a neighbor node through the $j$th-dimensional link;
$P^i$ determines $F_2$ /* $F_2$ is the set of frequent 2-itemsets */
$P^i$ generates $C_3$ /* all nodes have the same $C_3$ */
/* $C_3$ includes all candidate groups generated based on $F_2$ for pass 3 */


/* Pass $k$, for $k \geq 3$ */
$k = 3$;
/* $C_k$ includes all candidate groups generated based on $F_{k-1}$ for pass $k$,
    where $F_{k-1}$ is the set of frequent $(k-1)$-itemsets */
while $(C_k \neq \phi)$ {
    $P^i$ scans $D^i$ to count the candidate groups in $C_k$;
    /* counting all $\{head \bigcup i\}$, for $i \in tail$, and $\{head \bigcup tail\}$ for each
        candidate group in $C_k$ */
    for $(j = 1; j \leq n; j++)$
        $P^i$ exchanges and merges the local counts of $C_k$ members
        with a neighbor node through the $j$th-dimensional link;
    $P^i$ identifies frequent itemsets;
    $P^i$ inserts frequent itemsets into $GM$ and keeps only maximal
    frequent itemsets in $GM$;
    $P^i$ generates $C_{k+1}$ using $F_k$; /* the superset-frequency based pruning
        is applied */
    $k++$;
}
/* $GM$ is the set of all maximal frequent itemsets */
```

3.3 Distributed max-miner (DMM)

DMM includes two phases: local mining phase and global mining phase. In the local mining phase, each processing node P^i, for $0 \leq i \leq N-1$, applies the sequential max-miner algorithm on its local database D^i to find the set of local maximal frequent itemsets LM^i. Then, $LM^0, LM^1, \ldots, LM^{N-1}$ are exchanged and merged between nodes into the initial global candidate set GC_1 for the first global pass. GC_1 includes only maximal itemsets and is available at each node.

Since any itemset which is globally frequent must be frequent in at least one local database, each global frequent itemset should appear in GC_1 or be a subset of some maximal itemset in GC_1. Thus, a top-down search is applied for the itemsets in GC_1 in the global mining phase although these itemsets may have different length. Obviously, the maximum size of GC_1 is $|LM^0| + |LM^1| + \cdots + |LM^{N-1}|$, where $|LM^i|$ represents the size of LM^i. However, since some itemsets may be frequent in multiple local databases, the size of GC_1 is usually smaller than this upper bound. Furthermore, after the local mining phase, we already have local counting information related to these global candidates. It allows us to estimate the global counts of some candidate itemsets, so that we can reduce the size of GC_1. This estimation method is described in Sect. 3.4.2.

With the initial global candidate set GC_1, all nodes start the first pass of the global mining phase. In pass t, each node scans its local database to count the occurrences of the candidates in GC_t, then nodes exchange and merge local counts to find the global frequent itemsets. If the global support of a candidate is above *minsup*, it is included in the set of global frequent itemsets, named *FrequentSet*; otherwise, it is included in the set of infrequent itemsets, named *InfrequentSet*. At the same time, all the largest proper subsets of each infrequent itemset are considered to be included in GC_{t+1} for the next pass.

In DMM, we utilize *InfrequentSet* and *FrequentSet* to perform the subset-infrequency based pruning and superset-frequency based pruning on GC_{t+1}. Before each largest proper subset of an infrequent itemset is included in GC_{t+1} as a new candidate, we check whether it can be pruned or not. If a new candidate appears in *InfrequentSet* or has any subset in it, then it is already identified as infrequent. In this case, we should split it into its largest proper subsets, and repeat the same procedure. This subset-infrequency based pruning technique help us avoid unnecessary computations, especially for large candidates. We can break them down early into subsets that are close to really frequent itemsets.

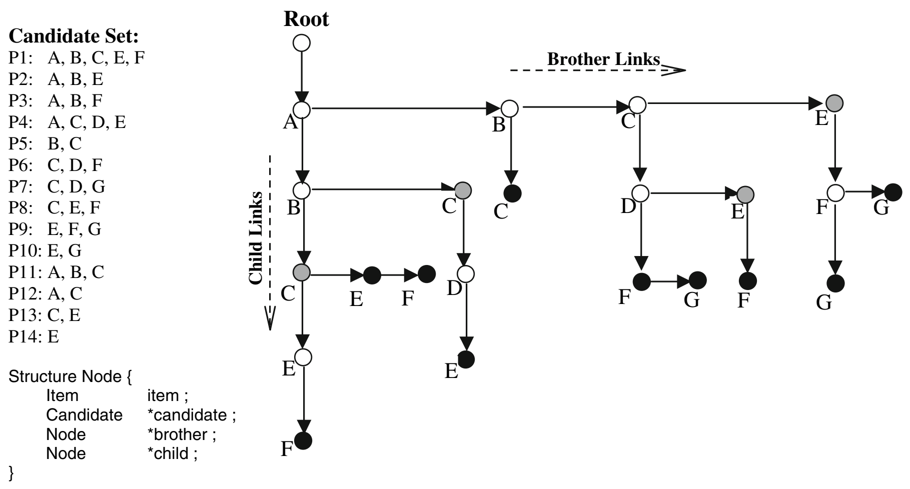
After the subset-infrequency based pruning step, we can check if each remaining candidate appears in *FrequentSet* or has any superset in it. If yes, it will not be included in GC_{t+1} according to the concept of superset-frequency based pruning. In fact, after the first couple of passes, there may be many frequent itemsets already identified. They help us remove a large number of new candidates, especially those split from short infrequent itemsets. In practice, these two pruning techniques considerably reduce the size of candidate sets, pass after pass. How to maintain *FrequentSet* and *InfrequentSet* are described in Sect. 3.4.2. The global mining phase must continue until there is no more candidate for the following pass.

3.4 Features of DMM

3.4.1 Prefix tree for counting global candidates with different length

Structure of the prefix tree: In DMM, we use the prefix tree to count the global candidates with different length. An example prefix tree is shown in Fig. 2 together with some candidate itemsets to be counted.

The root node is at level 0, while the first item of each candidate itemset is placed at level 1, the second item at level 2, and so on. Each node in the prefix tree corresponds to an item and is connected to maximum two other nodes, one through a *child link* and the other through a *brother link*. All the items following a common prefix item in different candidate itemsets are regarded as the children of the prefix item, and they are linked through brother links at the same level in their lexicographical order, but only the first child is directly connected to the prefix item through a child link. The root node corresponds to a null item and has four children in Fig. 2: A, B, C and E, which are the first items appearing in the candidates. Node A is the first child of the root node as it comes first in the lexicographical order among the children of the root node, and other children are linked one by one in ascending order through brother links. Similarly, nodes B and C have a common prefix A in different candidate itemsets, so they are linked together at level 2 as children of node A at level 1, while only node B is directly connected to node A through a child link. If a node corresponds to the last item of an candidate itemset, then it references the candidate itemset through the candidate pointer. In Fig. 2, each of those last nodes is represented as a gray node or a black node, depending on whether it is an internal node or a leaf node in the prefix tree.



Candidate Set:
P1: A, B, C, E, F
P2: A, B, E
P3: A, B, F
P4: A, C, D, E
P5: B, C
P6: C, D, F
P7: C, D, G
P8: C, E, F
P9: E, F, G
P10: E, G
P11: A, B, C
P12: A, C
P13: C, E
P14: E

```
Structure Node {
      Item          item ;
      Candidate     *candidate ;
      Node          *brother ;
      Node          *child ;
}
```

Fig. 2 Prefix tree structure

Construction of the prefix tree: Initially, the prefix tree has only the root node whose brother and child pointers are null. Then, we insert the candidate itemsets, one by one. Suppose there is a candidate k-itemset $\{I_1, I_2, \ldots, I_k\}$. This candidate itemset can be inserted into the prefix tree in k steps, one for each item. First, we check whether the root node has children. As described above, every node at level 1 should be the first item of a candidate itemset. If the root node doesn't have a child at all, then we need to create a node of I_1 and link it as the first child of root node. If the root node has children, we can reach the first child of the root node through its child link, then check other children through the brother links to make sure whether I_1 is already at level 1. If not, we need to create a node of I_1 at the right position to keep the lexicographical order of the items at level 1. From the node of I_1 at level 1, we repeat the same procedure for the next item I_2 in order to have it at level 2. This procedure is repeated until the last item I_k is located or inserted at level k, and then it will reference the candidate itemset through its candidate pointer.

Counting the global candidates using the prefix tree: For each transaction, we use a recursive method to count all candidates appearing in the transaction. Figure 3 shows how to count the candidates in a transaction containing $\{A, B, D, E, G\}$. We begin with the whole transaction at the root node. At level 1, items A, B and E are matched, and only E references a candidate, so we increase the count of the candidate itemset $\{E\}$ by one. Then, we recursively process the transaction segment $\{B, D, E, G\}$ on node A, $\{D, E, G\}$ on node B, and $\{G\}$ on node E. These three operations let us enter the next level. At level 2, A's first child B appears in $\{B, D, E, G\}$ but does not reference any candidate. So, we continue processing $\{D, E, G\}$ on this B node. Back to the node B at level 1, where we process $\{D, E, G\}$ on, we can find it has no child appearing in $\{D, E, G\}$, so we simply stop here without processing on any of its children. For the node E at level 1, its one child G is matched. Since this G references a candidate, we increase its count by one. On this branch, the transaction segment becomes empty after processing G, so we also stop here. Now, from the node B at level 2, we can enter level 3. Node E, the second child of node B, is matched and the corresponding candidate is counted. When we try to process $\{G\}$ on node E, we cannot find a matching child since

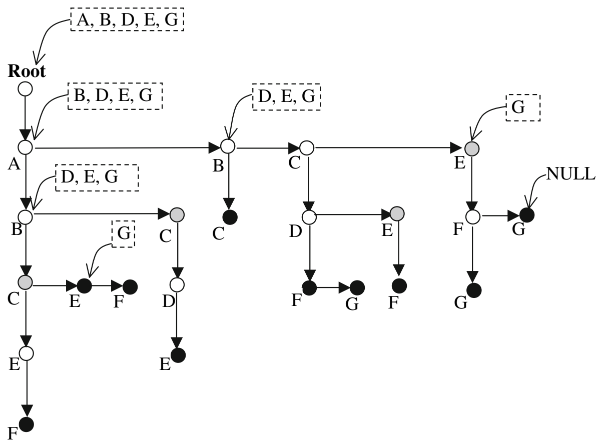


Fig. 3 Counting the global candidates using the prefix tree

E has no child. So, after the transaction is scanned, the counts of $\{A, B, E\}$, $\{E\}$, and $\{E, G\}$ are increased by one.

The task of finding all items in a transaction segment appearing at each level of the prefix tree is like searching common items in two itemsets. For example, at level 1, we need to find common items between $\{A, B, D, E, G\}$ and $\{A, B, C, E\}$. Similarly, from the node B at level 2, we need to find the common items between a transaction segment $\{D, E, G\}$ and B's children $\{C, E, F\}$. As the items in each transaction and the child items of each node in the prefix tree are sorted, we can use the two-way merge-like comparison to speed up the searching. In practice, the level 1 of the prefix tree can be implemented by using an array to speed up the construction and searching; and if an intermediate node has many children, we can create an index for them to speed up the searching.

3.4.2 Reduction of the global candidate set

For DMM, how to reduce the size of the initial global candidate set GC_1 and the subsequent global candidate sets as much as possible is very important for the overall performance. Three techniques were used to solve this problem: global support estimation, subset-infrequency based pruning, and superset-frequency based pruning.

Support estimation of the global candidates: When we merge the local maximal frequent items (MFIs) from all processing nodes, we need to keep just one copy of each local MFI that is frequent in more than one node as a global candidate. If a maximal itemset is frequent at all nodes, obviously it is also a global maximal frequent itemset. We just need to accumulate its local support counts and put it into *FrequentSet*. Such global candidates, however, are very few. Fortunately, even though most itemsets in GC_1 appear as local MFIs in just one or a few nodes, many of them often have their supersets frequent in other nodes. In that case, the support counts of the supersets of a candidate allow us to estimate the minimum support count of the candidate in those nodes. For example, suppose that itemset $\{A, B, G\}$ is a local MFI with the local count of 4,000 in node 1, while $\{A, B, C, E, G\}$ and $\{A, B, G, K\}$ are local MFIs in node 2 with local counts of 3,800 and 4,200, respectively. We can then estimate that the local support count of $\{A, B, G\}$ in node 2 should be at least 4,200. By this way, we can estimate the minimum support count of any itemset in a node if any of its supersets is frequent in that node. Obviously, the estimated minimum support count of a candidate is the largest support count of all its supersets in that node.

Subset-infrequency based pruning and superset-frequency based pruning: During the global mining phase, DMM maintains the following sets: GC_t ($t \geq 1$), *FrequentSet* and *InfrequentSet*. They are changing dynamically with the progress of the mining process. The global mining phase continues until GC_t is empty, for some $t \geq 1$, to ensure that we will not miss any global MFI. Eventually, *FrequentSet* will include all the global MFIs. Since DMM uses *FrequentSet* and *InfrequentSet* to perform the superset-frequency based pruning and the subset-infrequency based pruning, maintaining these two sets without any redundancy is important to make the pruning steps efficient. After each pass in the global mining phase, we determine whether each global candidate is maximally frequent or not. If a global candidate is frequent, we put it into *FrequentSet* only if none of its supersets is already in that set. On the other hand, if a global candidate is infrequent, we put it into *InfrequentSet* only if none of its subsets is already in that set. For example, if $\{A, B, G, H\}$ is infrequent but $\{A, H\}$ is already in *InfrequentSet*, we do not insert it into *InfrequentSet*, because any superset of $\{A, B, G, H\}$ will be also pruned by $\{A, H\}$ when the subset-infrequency based pruning is applied.

If a global candidate k-itemset is identified as infrequent, we split it into k $(k-1)$-subsets, which will be considered as new candidates. However, some of them may not be a valid candidate for the next global pass if it appears in *InfrequentSet* or has a subset in it. In that case, we need to split the invalid candidate into its largest proper subsets. For example, if $\{A, B, C, D\}$ is infrequent, we split it into its largest proper subsets {B, C, D}, {A, C, D}, {A, B, D} and {A, B, C}, which are considered as new candidates. If $\{A, D\}$ is already in *InfrequentSet*, we split {A, C, D} and {A, B, D} to their largest proper subsets $\{A, B\}$, $\{A, C\}$, $\{B, D\}$ and $\{C, D\}$. As a result, we get the following new candidates: $\{A, B, C\}$, $\{B, C, D\}$, $\{A, B\}$, $\{A, C\}$, $\{B, D\}$ and $\{C, D\}$. In practice, these two pruning techniques can make the global candidate set shrink drastically for each pass.

3.5 Pseudo-code of DMM

As we assume a homogeneous distributed computing environment where all the processing nodes are the same, we just give the pseudo-code of the DMM algorithm running on a node P^i.

```
/* Local Mining Phase */
$P^i$ applies the sequential max-miner algorithm on $D^i$ and stores local
MFIs in $LM^i$;
$n = \log_2 N$; /* $N$ processing nodes are used for mining */
**for** ($j = 1$; $j \leq n$; $j++$)
    $P^i$ exchanges and merges $LM^i$ with that of a neighbor node through the
    $j$th-dimensional link, and the result is stored in $LM^i$;
$GC_1 = \phi$;
*FrequentSet*$= \phi$;
**foreach** local MFI $x$ in $LM^i$ {
    **if** the estimated global support of $x$ is above the minimum support
        **then** put $x$ into *FrequentSet* unless it contains a superset of $x$;
    **else** put $x$ into $GC_1$;
}
```

```
apply the superset-frequency based pruning on $GC_1$;


/* Global Mining Phase */
*InfrequentSet*= $\phi$;
/* global pass $t$, for $t \geq 1$ */
$t = 1$;
**while** ($GC_t \neq \phi$) {
    $P^i$ scans $D^i$ to count the candidates in $GC_t$;
    **for** ($j = 1$; $j \leq n$; $j++$)
        $P^i$ exchanges and merges the local counts of $GC_t$ members
        with a neighbor node through the $j$th-dimensional link;
    **foreach** candidate $x$ in $GC_t$ {
        **if** the support of $x$ is above the minimum support
            **then** put $x$ into *FrequentSet* unless it contains a superset of $x$;
        **else** put $x$ into *InfrequentSet* unless it contains a subset of $x$;
    }
    **foreach** candidate inserted into *InfrequentSet* in the current pass {
        split the infrequent candidate into new candidates (i.e., its
        largest proper subsets);
        apply the subset-infrequency based pruning on these new candidates;
        those candidates pruned by the subset-infrequency based pruning
        are put back into *InfrequentSet*;
        /* this process will continue until no new candidate either appears
            in *InfrequentSet* or has any subset in it */
    }
    apply the superset-frequency based pruning on the new candidates;
    /* remove those candidates which appear in FrequentSet or have any
        superset in it */
    put the new candidates that passed the two pruning operations into
    $GC_{t+1}$ for the next global pass $t + 1$;
    $t++$;
}
$GM$ = *FrequentSet*;
/* $GM$ is the set of all maximal frequent itemsets */
```

4 Performance evaluation

Our test platform is an 8-node Linux cluster system where nodes are connected by a Fast Ethernet switch. Each node has a 800 MHz Pentium processor, 512 MB memory, and a 40 GB disk drive. The processes are communicating using the Message Passing Interface (MPI) [19].

In our experiments, first we used the synthetic sales transaction databases generated as in [2]. All parameters used for generating databases are described in Table 1. For all databases, $c = 0.5$, $m = 0.5$, $v = 0.1$, $|L| = 2{,}000$ and $NI = 1{,}000$. Table 2 lists all the synthetic databases used in our performance evaluation experiments. The size of each database is about 360 MB. When running the parallel algorithm on a database, we need to partition it into local databases.

Table 1 Synthetic database parameters

$\lvert D\rvert$	Number of transactions in the database
$\lvert T\rvert$	Average size of the transactions
$\lvert I\rvert$	Average size of the maximal potentially frequent itemsets
$\lvert L\rvert$	Number of maximal potentially frequent itemsets
NI	Number of items
c	Correlation level
m	Mean of the corruption level
v	Variance of the corruption level

Table 2 Databases

Name	$\lvert T\rvert$	$\lvert I\rvert$	$\lvert D\rvert$
T10_I02_D7852K	10	2	7852K
T20_I04_D4288K	20	4	4288K
T25_I06_D3504K	25	6	3504K
T30_I08_D2954K	30	8	2954K
T40_I10_D2256K	40	10	2256K

To balance the size of the local databases, each transaction is randomly allocated to a node. We also used two real datasets obtained from the Frequent Itemset Mining Implementations (FIMI) repository [10], and they are described in Sect. 4.7.

In order to compare the performance of PMM, DMM, and count distribution, we also implemented count distribution on the same platform.

4.1 Max-miner versus Apriori

Before we compare the performance of PMM, DMM and count distribution, we'd like to compare the performance of sequential max-miner and Apriori first, because the result shows clearly when PMM and DMM may outperform count distribution.

As shown in Fig. 4, when the average size of maximal potentially frequent itemsets is large, such as 8, max-miner performs much better than Apriori. In this case, there are many long patterns which are frequent, so max-miner's look-ahead technique can demonstrate its advantage. Figure 5 shows that even when the average size of maximal potentially frequent itemsets is moderate, such as 4, if we decrease *minsup*, then more and more patterns become frequent and some long patterns also become frequent. Thus, max-miner can show better performance in this case as well.

Let's look at the behavior of max-miner in each pass. When we compared the performance of max-miner and Apriori in detail, we found that max-miner doesn't reduce the number of passes much in many cases, which was contrary to our initial expectation. Even in the case considered very favorable to max-miner, such as the case that $\lvert I\rvert$ is 8 and *minsup* is 0.25%, max-miner just reduced the number of passes by 3. However, in this case, the overall performance of max-miner is still much better than Apriori. The reason for this improvement is that max-miner effectively reduces the size of candidate set in many passes during mining. Figure 6 shows the comparison of candidate set sizes of max-miner and Apriori for each pass when we ran them on the T30_I08_D2954K database with *minsup* of 0.25%.

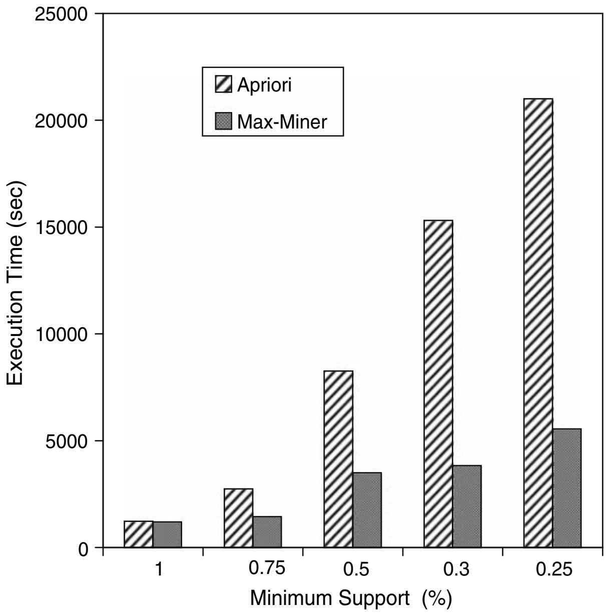


Fig. 4 Max-miner versus Apriori on T30_I08_D2954K

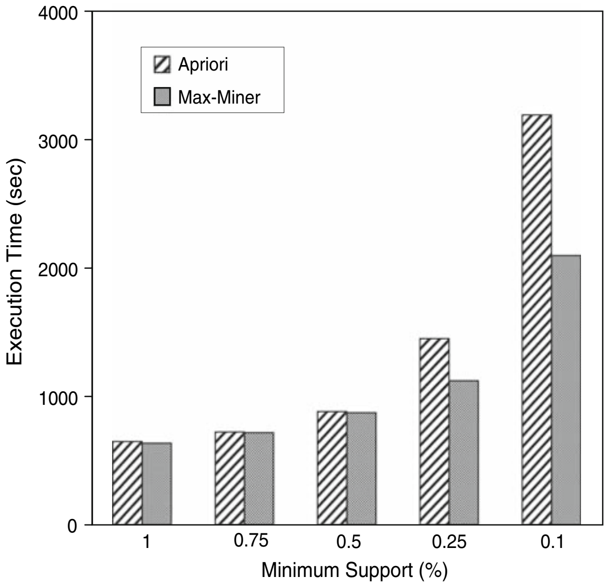
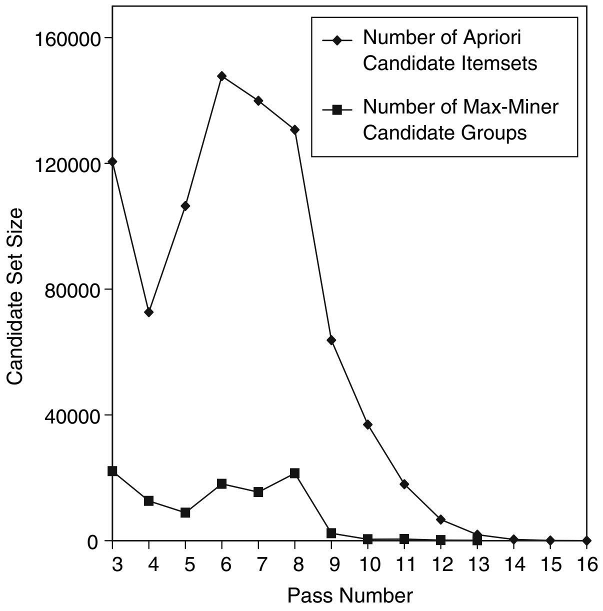


Fig. 5 Max-miner versus Apriori on T20_I04_D4288K

The comparison begins at the third pass because both algorithms use the same technique to count candidate 1-itemsets and candidate 2-itemsets. Although each candidate group in max-miner has a tail, which may be long, max-miner constructs a hash tree just based on the heads of the candidate groups. So, the number of nodes in its hash tree is determined by the number of candidate groups, not by the total number of items in their tails. In Apriori, a larger number of candidate itemsets make the hash tree much bigger in terms of the number of levels and the size of leaf nodes. Furthermore, when we count a candidate group

Fig. 6 Max-miner versus. Apriori: size of the candidate set



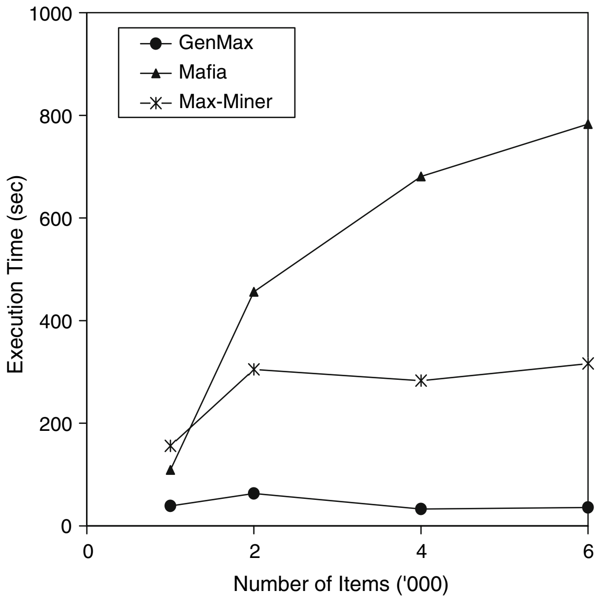
in max-miner, after locating its head in the hash tree, we can easily check multiple items (appearing in the same transaction) against its tail items, because those tail items are listed together with the head in the same leaf node of the hash tree. On the other hand, in Apriori, the candidate itemsets containing some of those tail items are placed in one or more leaf nodes of the hash tree. So, to check them against the candidates, we need to traverse those leaf nodes and compare them with the candidates in each leaf node, one by one. These factors give a substantial performance gain to max-miner when a large number of transactions are processed.

4.2 Max-miner versus MAFIA and GenMax

We also compared max-miner with MAFIA and GenMax by using the source codes provided by their authors. They were run on one node of our cluster system. Our max-miner implementation performed comparably with GenMax but much faster than MAFIA on T10_I2_D100K_NI1K. When $|T|$ and $|I|$ parameter values are increased, MAFIA and GenMax gradually showed their superiority: For T40_I10_D100K_NI1K database, both MAFIA and GenMax outperformed our max-miner implementation, as reported in [12]. Then, we performed some additional comparisons.

First, we investigated how max-miner, MAFIA and GenMax perform when the number of items in the database is increased. As shown in Fig. 7, as the number of items (NI) of the T40_I10_D100K database increases, the execution time of MAFIA increases rapidly. Max-miner and GenMax outperformed MAFIA clearly for large NI values. When NI is large, there are many frequent items for the same minimum support level. MAFIA performs the intersections of the TID-sets of the frequent items, and this cost makes it less competitive. GenMax deals with this problem by transforming the database back to the horizontal format on-the-fly. In the horizontal format of the transaction database, each transaction is a row in the database and consists of a unique transaction id (TID) and a set of items involved in the

Fig. 7 Effect of the number of items (T40_I10_D100K, minsup = 0.5%)



transaction. However, for very large databases, such an on-the-fly transformation may have very high computation overhead and require a large amount of memory.

Second, we investigated how these algorithms perform when we increase the number of transactions in the T40_I10_NI6K database. Compared with MAFIA, which uses bitmaps to store the intermediate data during mining, GenMax requires less memory by using diffsets. However, in order to count candidate 2-itemsets efficiently, GenMax needs additional memory space to perform the vertical-to-horizontal transformation. Thus, the available memory space becomes critical to the performance of both algorithms when the database is very large. As shown in Fig. 8, when the number of transactions was increased to 1500K and 2500K, MAFIA and GenMax could not finish after running 30,000 and 70,000 seconds, respectively, which are more than ten times of the execution time of max-miner.

Our tests demonstrated that max-miner is an efficient sequential algorithm, even though it is not the best for all cases. Its scalability is much better than other algorithms, like MAFIA and GenMax, in terms of the number of items and the number of transactions. This feature makes it a very good candidate for mining large transaction databases.

4.3 Improvement of PMM and DMM over count distribution (CD)

We ran PMM, DMM, and count distribution on different synthetic databases with different *minsup* values. If we define T_{CD}, T_{PMM} and T_{DMM} as the execution times of CD, PMM and DMM, respectively, then the speedup of PMM over CD is $T_{\text{CD}}/T_{\text{PMM}}$ and the speedup of DMM over CD is $T_{\text{CD}}/T_{\text{DMM}}$. In Table 3, the speedup of PMM is shown for different databases listed in the first column and for different values of *minsup* listed in the first row. Similarly, in Table 4, the speedup of DMM over CD is shown. In these experiments, all 8 nodes in our cluster system were used.

When *minsup* is high, PMM and DMM are comparable to or a little bit slower than count distribution. We also ran Apriori and max-miner for these cases, and found that max-miner doesn't show much improvement over Apriori, either. That is because the high *minsup* limits

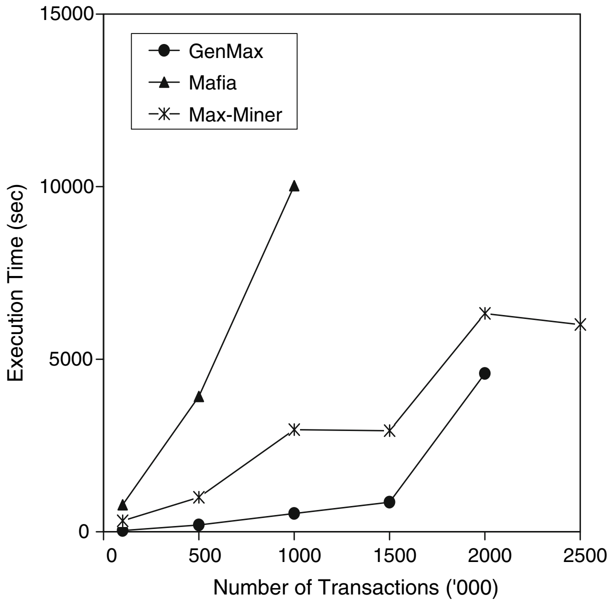


Fig. 8 Effect of the number of transactions (T40_I10_NI6K, minsup = 0.5%)

Table 3 Speedup of PMM over CD (8-node case)

Database	1%	0.75%	0.5%	0.4%	0.3%	0.25%	0.2%	0.15%	0.1%
T10_I02_D7852K	1.01	1.02	1.01	1.05	1.07	1.07	1.07	1.08	1.08
T20_I04_D4288K	1.04	1.02	1.04	1.08	1.19	1.30	1.41	1.64	1.66
T25_I06_D3504K	1.06	1.07	1.45	1.66	2.31	2.46	2.73	2.98	4.19
T30_I08_D2954K	1.05	1.97	2.79	5.02	5.29	5.82	6.41	16.80	21.34
T40_I10_D2256K	1.40	1.70	2.83	4.86	7.60	9.38	12.04	20.10	29.76

Table 4 Speedup of DMM over CD (8-node case)

Database	1%	0.75%	0.5%	0.4%	0.3%	0.25%	0.2%	0.15%	0.1%
T10_I02_D7852K	0.97	1.01	0.99	1.05	1.04	1.08	1.07	1.08	1.08
T20_I04_D4288K	0.99	1.03	1.04	1.07	1.18	1.26	1.32	1.54	1.61
T25_I06_D3504K	1.06	1.05	1.40	1.49	2.06	2.21	2.46	2.70	3.95
T30_I08_D2954K	0.95	1.96	2.65	3.57	4.79	4.73	5.09	14.30	19.89
T40_I10_D2256K	1.37	1.56	2.52	4.35	6.48	7.24	9.57	18.14	28.05

the number of frequent itemsets and the size of those frequent itemsets. Thus, the effect of look-ahead technique used by max-miner is not clearly shown, and naturally PMM and DMM have the same result.

As *minsup* decreases, PMM and DMM begin to show more and more improvement in our tests. As shown in Tables 3 and 4, when the $|I|$ value of the database is large, such as 8 or 10, even if *minsup* is as high as 0.5%, PMM and DMM are faster than count distribution with a

Table 5 Comparison of synchronization requirement

Database	1%	0.75%	0.5%	0.4%	0.3%	0.2%	0.1%
T10_I02_D7852K	2:2:3	2:3:4	2:5:5	2:5:8	2:5:9	2:5:9	2:5:9
T20_I04_D4288K	2:4:6	2:6:6	2:8:9	2:7:9	2:9:9	2:8:11	3:9:11
T25_I06_D3504K	2:7:8	2:7:8	2:11:11	3:9:14	3:13:14	3:11:15	3:12:16
T30_I08_D2954K	2:6:6	2:12:12	3:13:15	3:13:16	4:11:16	4:13:16	4:17:18
T40_I10_D2256K	2:11:12	3:9:13	3:14:15	3:13:16	4:15:17	4:16:18	6:18:19

speedup above 2.5. It is because a large $|I|$ value results in large frequent itemsets (i.e., long patterns), which benefits PMM and DMM because more superset-frequency based pruning can be performed. If *minsup* is less than 0.25%, PMM and DMM outperform count distribution considerably. Comparing PMM with DMM, we found that the speedup of PMM over CD is a little higher than that of DMM in most cases. DMM uses the local and global mining phases to reduce the overall synchronization and communication requirement, but the global mining phase still needs several passes over the database and incurs some extra computation overhead. In our cluster system, since the communication speed between nodes is high, the benefit of reduced synchronization and communication overhead is not enough to offset the effect of extra passes during the global mining phase. However, this feature of DMM may be attractive to some distributed systems where the communication cost is relatively high.

4.4 Comparison of synchronization requirement

We compared the number of synchronizations needed between processing nodes in PMM, DMM and count distribution. In DMM, the local mining phase needs only one synchronization. So, the total number of synchronizations is the number of passes needed in the global mining phase plus one. Table 5 shows the comparison results. Here, we define S_{PMM}, S_{DMM}, and S_{CD} as the numbers of synchronizations needed in PMM, DMM and CD, respectively. The first row of the table lists various values of *minsup*, and the first column lists the names of databases. The values in each entry of the table represents $S_{\mathrm{DMM}}:S_{\mathrm{PMM}}:S_{\mathrm{CD}}$.

In most cases, PMM requires less number of synchronizations than count distribution, but the difference is small. As pointed out before, max-miner does not reduce the number of passes over the database much, compared to Apriori. As a straightforward parallel version of max-miner, PMM does not reduce the number of passes much, either. But DMM needed just two times of synchronization in the best cases. In other cases, the number of synchronizations needed for DMM was also much smaller than those of PMM and CD, mainly because DMM requires only one synchronization during the local mining phase.

4.5 Comparison of communication requirement

Like count distribution, all nodes running PMM have the same set of candidates in each pass. So, each node sends and receives the same amount of count information for the candidates. In DMM, nodes need to exchange two types of data: candidates and their counts. For the merging of local MFIs to construct the first global candidate set, each node performs $\log_2 N$ send and $\log_2 N$ receive operations when N processing nodes are used. Since the set of local MFIs in one node may be different from those in other nodes, the amount of data each node sends or receives varies at each communication step. In each global pass, all nodes have the same global candidate set and exchange the same count information in $\log_2 N$ steps. To make it simple, we computed the average amount of data each node sends and receives during the whole mining.

Let's consider the difference in the meaning of *candidates* of these three algorithms as the number of candidates determines how much data need to be exchanged between processing nodes during the mining. In count distribution, its candidates are the potential frequent itemsets generated as in Apriori. On the other hand, in PMM, candidates are the candidate groups defined exactly as in max-miner, where each candidate group contains a head (itemset) and a tail (itemset). For each candidate group, we can say there are two kinds of candidates: potential frequent itemsets, each of which contains the head and one of the tail items, and a potential maximal frequent itemset which is the union of the head and the tail. Thus, to exchange the count information for each candidate group, processing nodes need to exchange multiple integers for the two kinds of candidates. It may sound that we need more communication between processing nodes in PMM. However, in reality, PMM's look-ahead technique can reduce the communication requirement considerably, because if one potential maximal frequent itemset is found frequent early, then many candidates can be removed due to the superset-frequency based pruning. So, compared with count distribution, which does not use the superset-frequency based pruning, the total amount of count information exchanged in PMM is smaller in most cases.

In DMM, after the local mining phase, candidates involved in the communication are just the potential maximal frequent itemsets; i.e., all local MFIs and some of their subsets that can be global MFIs. Compared with the set of candidates in count distribution, the set of candidates in DMM is very small. Thus, DMM requires much less communication than count distribution even though DMM needs to merge the candidates first (after the local mining phase) and then exchange the count information (during the global mining phase).

When *minsup* is very low, count distribution tends to discover a large number of short frequent patterns, hence there are a large number of candidates in early passes. This results in very high communication overhead between nodes. PMM also generates a large number of candidate groups having short patterns as their heads, while many of them have a long tail. In practice, the look-ahead technique may not perform much pruning in early passes, because the tail of each candidate group is too long and may contain some items which make the union of the head and the whole tail infrequent. In later passes, the situation is getting better as the look-ahead technique takes effect. On the other hand, in DMM, the increase in the number of short patterns usually results in a small change in the number of local MFIs. Thus, even though a low *minsup* value may affect the local mining phase of DMM, it has a relatively small impact on the communication overhead during the global mining phase. Therefore, as *minsup* decreases, DMM performs better than count distribution and PMM in terms of communication requirement.

We implemented two versions of count distribution: one is using the n-cube communication, and the other is using the all-to-all communication. For count distribution, based on the number of candidates generated in each pass, we can compute the amount of data each node sends and receives. Theoretically, if there are N nodes and m candidates generated in the whole mining process, each node needs to send and receive total $2m(N-1)$ integers when the all-to-all communication is used, and $2m \log_2 N$ integers with the n-cube communication.

We compared the average amount of data each node communicates with others when we executed PMM, DMM and count distribution on the T30_I08_D2954K database with various values of *minsup*, and the results are shown in Fig. 9.

As shown in Fig. 9, the communication overhead of DMM is much lower than that of count distribution, while PMM has almost the same level of communication overhead with

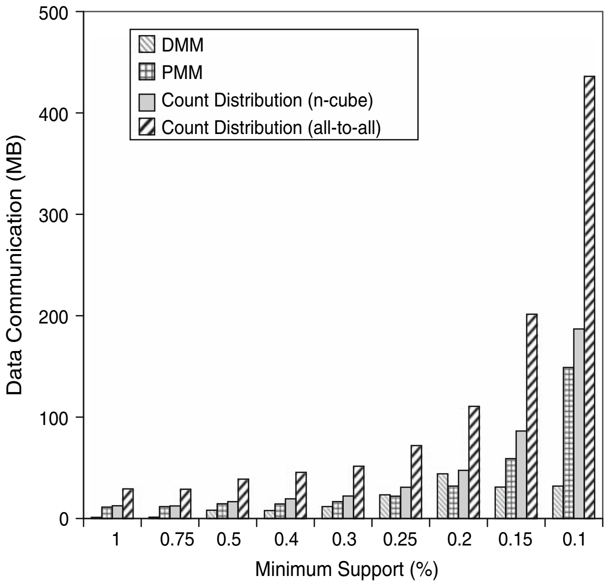


Fig. 9 Comparison of communication requirement

count distribution. Even though DMM needs to exchange candidates at the end of the local mining phase and some candidates may consist of many items, the total amount of data to be transferred is still relatively small, because PMM and count distribution must exchange the count information for much larger candidate sets. DMM's overall communication overhead is lower than that of PMM in most cases, especially when *minsup* is very low, like 0.15% or 0.1%. Compared with count distribution using the all-to-all communication scheme, both DMM and PMM demonstrate a big improvement in communication for all cases. Here, we'd like to emphasize that the advantage of DMM in communication requirement comes from its much smaller size of candidate sets and the *n*-cube communication scheme. DMM is very suitable for the distributed computing environment where the communication cost is critical.

4.6 Sensitivity analysis of PMM and DMM

In this section, we analyze the characteristics of the PMM and DMM algorithms in three aspects: scaleup, speedup and sizeup. All tests were performed with *minsup* of 0.25%.

4.6.1 Scaleup

We evaluated how PMM and DMM perform when more processors are available for the mining of larger databases. A set of experiments were performed on the databases listed in Table 6. These databases were generated by duplicating transactions according to the number of nodes used in the mining.

We used 2-node, 4-node and 8-node configurations of the cluster, and the total number of transactions was selected depending on how many nodes were used for the mining. For example, when PMM and DMM were running on the T10_I02_D7852K database using 2 nodes, we partitioned the database into two parts. When 4 nodes were used, we partitioned the database into four parts, one for each node, and then duplicated the local partition once at each node. So, as the number of nodes was increased from 2 to 4, the number of transactions

Table 6 Databases for scaleup tests

Name	$\|D_2\|$ (360 MB)	$\|D_4\|$ (720 MB)	$\|D_8\|$ (1,440 MB)
T10_I02_D7852K	7852K	15704K	31408K
T20_I04_D4288K	4288K	8576K	17152K
T25_I06_D3504K	3504K	7008K	14016K
T30_I08_D2954K	2954K	5908K	11816K
T40_I10_D2256K	2256K	4512K	9024K

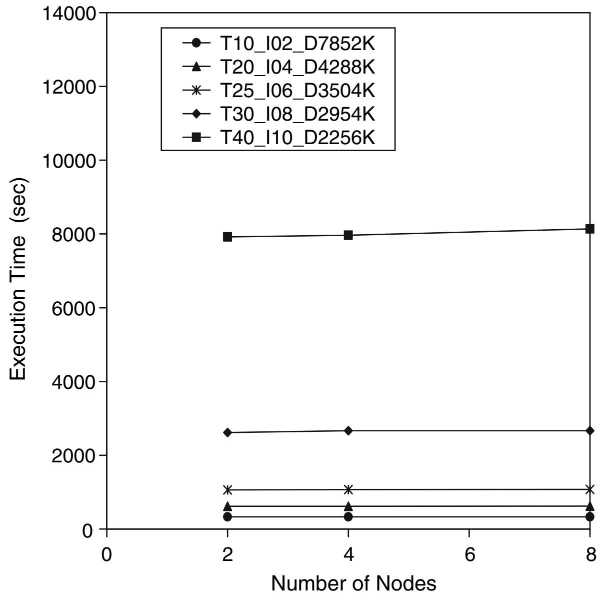


Fig. 10 Scaleup of PMM

at each node was not changed. In Table 6, $|D_2|$, $|D_4|$, and $|D_8|$ represent the total number of transactions processed by 2-node, 4-node, and 8-node systems, respectively. They were increased linearly with the number of nodes in the system.

Figures 10 and 11 show the execution times of PMM and DMM on the five databases, respectively. Both PMM and DMM scale very well and maintain the execution time almost constant as we increase the database size and the number of nodes. PMM performed a little better than DMM because its small increase in execution time, as the number of nodes increases, is mainly due to more communication between nodes. On the other hand, DMM is affected by another factor. As we repartitioned the database for more nodes, the local mining result at each node may become different. This resulted in a little more change in the performance of DMM from one system configuration to another. Overall, both PMM and DMM showed a good scaleup property.

4.6.2 Speedup

We measured the speedup of PMM and DMM as the number of processing nodes was increased while the database size remained the same. For the databases listed in Table 2, we

Fig. 11 Scaleup of DMM

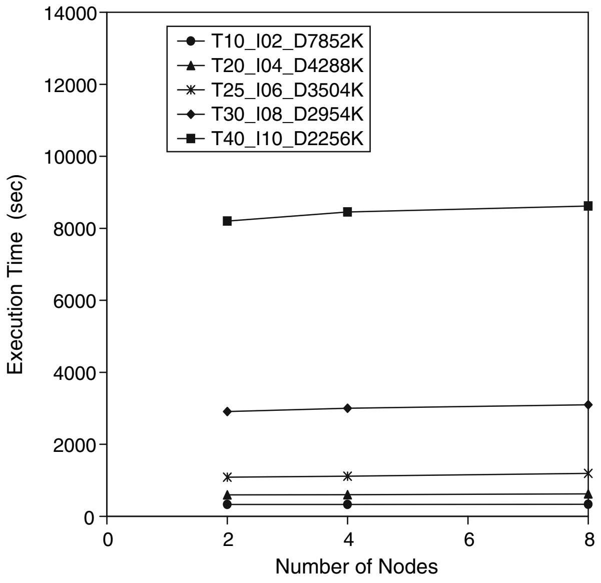
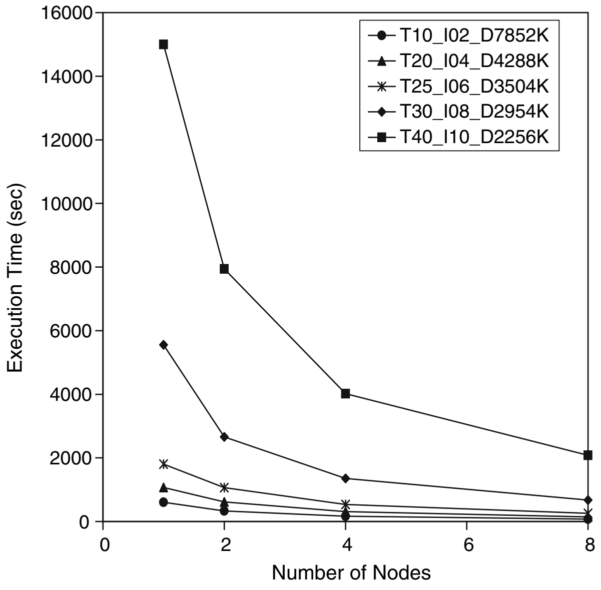


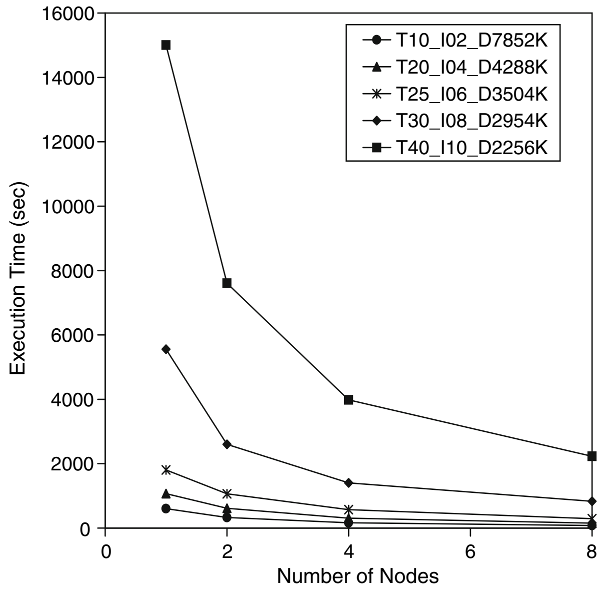
Fig. 12 Speedup of PMM



kept the same database size of 360 MB, but the database was partitioned into 2, 4, and 8 parts when the number of nodes were 2, 4, and 8, respectively.

Figures 12 and 13 show the execution times of PMM and DMM on the 2-node, 4-node, and 8-node systems. To demonstrate the speedup, we also ran the sequential max-miner for each database on a single node. As the number of nodes was doubled, the execution times of PMM and DMM were reduced by about 40 to 50%. In PMM, all the nodes generate and count the same set of candidates in each pass, no matter how many nodes are used. Thus, PMM

Fig. 13 Speedup of DMM



shows an almost linear speedup. Even though DMM may not achieve the linear speedup, it still shows a good speedup.

When PMM and DMM are executed on the T40_I10_D2256K database using 2 nodes, the execution time of DMM is less than that of PMM. This is because, when the number of nodes is small, the data distribution characteristic of each data partition is very similar to that of the whole database. So, after the local mining phase, the initial global candidate set would be similar to the set of global MFIs. As a result, during the global mining phase, the communication and synchronization overhead is low.

4.6.3 Sizeup

For the sizeup test, we fixed the system to the 8-node configuration, and distributed each database listed in Table 2 to the 8 nodes. Then, we increased the local database size at each node from 45 to 225 MB by duplicating the initial database partition allocated to the node. Thus, the data distribution characteristics remained the same as the local database size was increased. This is different from the scaleup and speedup tests, where the database repartitioning was performed when the number of nodes was increased. The performance of DMM is affected by the database repartitioning to some extent, although it is usually very small. On the other hand, the database repartitioning does not affect the performance of PMM. During the sizeup test, the local mining result of DMM is not changed at all at each node.

The results shown in Figs. 14 and 15 indicate that both PMM and DMM have a very good sizeup property. Since increasing the size of local database did not affect the local mining results of DMM and PMM at each node, the total execution time increased just due to more disk I/O and computation cost which scaled almost linearly with the sizeup.

4.7 Performance evaluation on real datasets

In our tests on large synthetic databases, both DMM and PMM have demonstrated much better performance than count distribution (CD). Compared with PMM, DMM has better

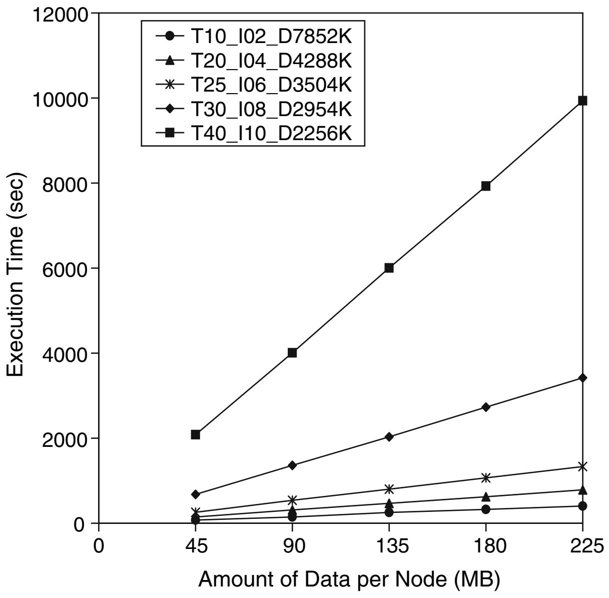


Fig. 14 Sizeup of PMM

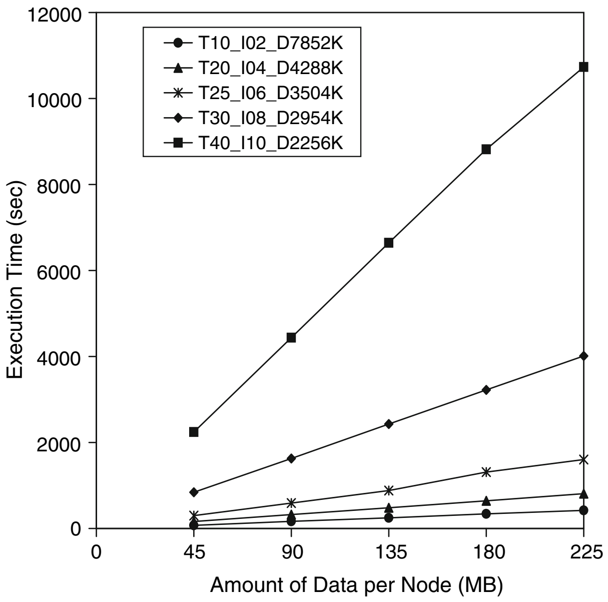
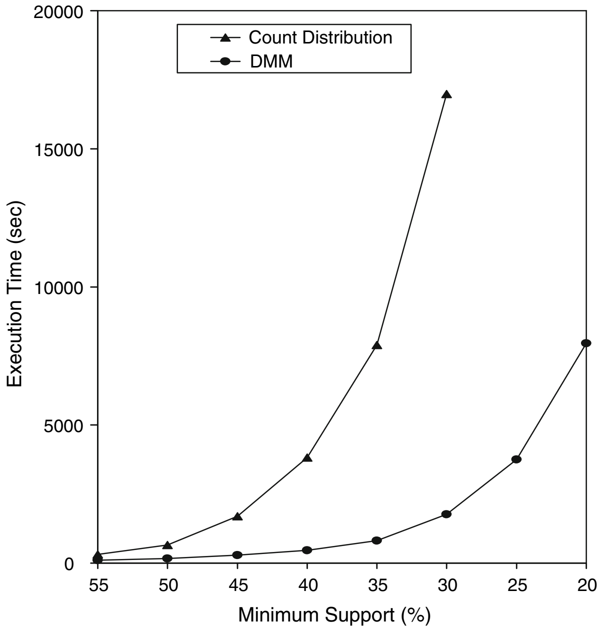


Fig. 15 Sizeup of DMM

efficiency in communication and synchronization. Another big advantage of DMM is that it can use any sequential maximal frequent itemset mining algorithm during its local mining phase. Thus, we tested DMM further using some of the real datasets available at the Frequent Itemset Mining Implementations (FIMI) repository [10]. Here, we present the test results on two real datasets, *accidents** and *pumsb**. They are obtained by duplicating the most commonly used datasets, *accidents* and *pumsb*, multiple times, respectively.

Fig. 16 Performance on *accidents** dataset



The *accidents* dataset [11] is obtained from the National Institute of Statistics (NIS), and it contains traffic accident data in a region of Belgium during 1991–2000. There are 340,184 transactions and over 500 items in *accidents*. The *pumsb* dataset is produced from the Public Use Microdata Sample (PUMS) census data. The *pumsb* dataset contains 49,046 transactions and over 7,000 items. A large number of large frequent itemsets can be mined for even relatively high *minsup* values. However, these two real datasets are small. The *accidents* dataset is about 45 MB, and the *pumsb* dataset is about 16 MB in binary format.

In parallel mining, we partition a dataset and then concurrently process different parts of the dataset to obtain performance speedup. However, if the dataset is too small, the performance gain from concurrent processing is limited because the total workload is not big enough for the processors to offset the overhead introduced by the parallelization, such as communication, synchronization, and unbalanced workload. Another practical problem in the parallel mining of a small dataset is that some data partitions may have quite different data distributions compared to that of the original dataset. Then, the nodes mining those data partitions would generate many false candidates, so that the cost to clean up the false candidates could be too high. Thus, we duplicated *accidents* dataset four times and *pumsb* dataset eight times, respectively. The resulting datasets are denoted as *accidents** and *pumsb** in this paper. The *accidents** dataset is about 180 MB, and the *pumsb** dataset is about 120 MB, but note that their data distributions are exactly the same as those of *accidents* and *pumsb*, respectively.

Figures 16 and 17 show the execution times of DMM and count distribution (CD) on *accidents** and *pumsb** datasets for various *minsup* values. Unlike DMM, the execution time of CD increases very quickly as *minsup* is decreased. These two real datasets are much more dense than the synthetic databases we used. For example, the average size of maximal frequent itemsets in *accidents** is 9.9 when *minsup* is 20%, and it is 9.4 in *pumsb** when *minsup* is 70%. On the other hand, in the synthetic database T40_I10_D2256K, the average size of

Fig. 17 Performance on *pumsb** dataset

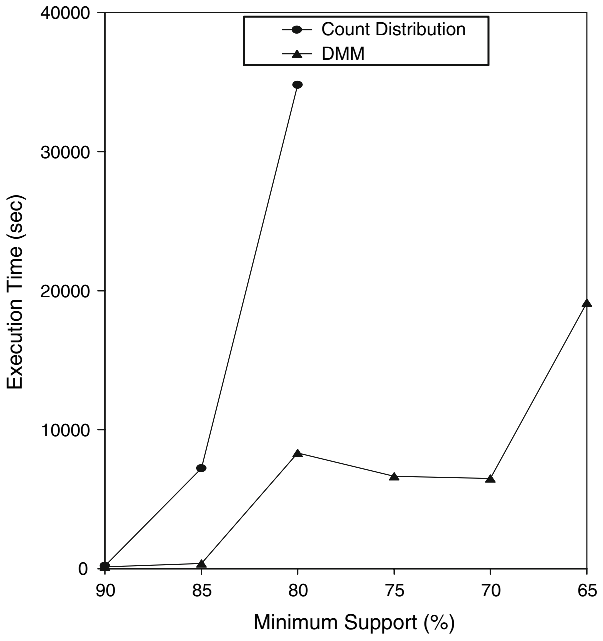
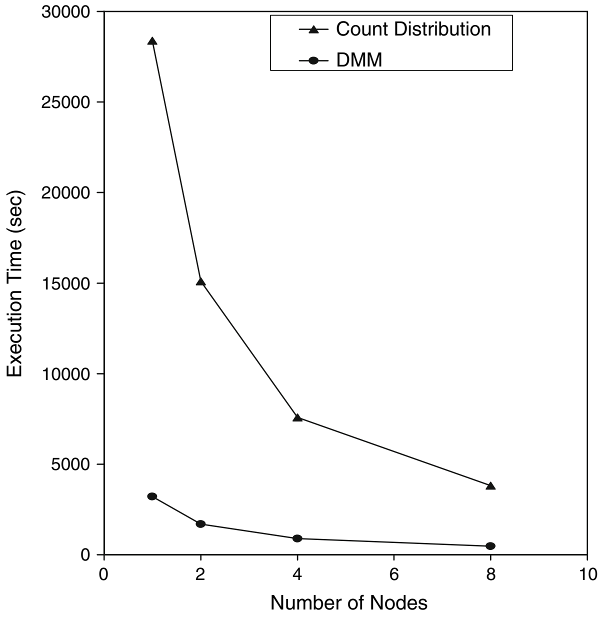


Fig. 18 Speedup on *accidents** (minsup = 40%)



maximal frequent itemsets is less than 7.0 for all *minsup* values used. As there are many long frequent patterns in *accidents** and *pumsb**, CD needs to enumerate a large number of their subsets, so that it could not finish many of these tests within 12 h. Thus, these tests demonstrate that the speedup of DMM over CD can be even better for the real dense datasets.

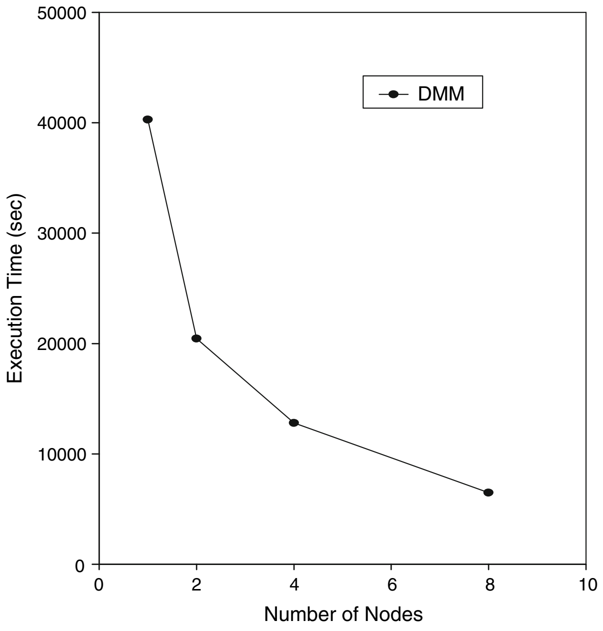


Fig. 19 Speedup on *pumsb** (minsup = 70%)

We also compared the speedup of DMM and CD by testing them on 1-node, 2-node, 4-node, and 8-node configurations in our cluster. Figures 18 and 19 show the results of mining accidents* and pumsb* for *minsup* of 40% and 70%, respectively. For the 1-node case, the sequential max-miner and Apriori algorithms were used. DMM achieves a good performance speedup: on accidents*, it is 1.90, 3.61, and 6.89 for 2-node, 4-node and 8-node cases, respectively. On *pumsb**, the corresponding speedup is 1.97, 3.15, and 6.21, respectively. In our tests, CD could not finish the mining on *pumsb** within 12 h for *minsup* of 70%, thus it is not shown in Fig. 19.

5 Conclusions and future work

In this paper, we proposed two new parallel maximal frequent itemset (MFI) mining algorithms, named parallel max-miner (PMM) and distributed max-miner (DMM), for shared-nothing multiprocessor systems. PMM is based on the sequential max-miner algorithm and avoids enumerating all potential frequent itemsets. DMM is another parallel version of max-miner, and it requires low synchronization and communication overhead. In DMM, max-miner is applied on each database partition during the local mining phase. Only one synchronization is needed at the end of this phase to construct the initial global candidate set. In the global mining phase, a top-down search is performed on the candidate set, and a prefix tree is used to count the candidates with different length efficiently. Usually, just a few passes are needed to find all global maximal frequent itemsets. Thus, DMM largely reduces the number of synchronizations required between the processing nodes.

We implemented DMM, PMM and count distribution on a cluster of Linux workstations. Compared with count distribution, our algorithms show a great improvement when some frequent itemsets are large (i.e., long patterns). A cube-based communication scheme is employed by both PMM and DMM for efficient communication between nodes. For DMM, global support estimation, subset-infrequency based pruning, and superset-frequency based pruning are used to reduce the size of global candidate set.

The global mining phase of DMM can be used together with any sequential local mining algorithm as well as with the max-miner algorithm. Both PMM and DMM also have very good properties in terms of speedup, scaleup and sizeup, as evidenced by the corresponding test results.

There are some research topics that can be investigated for our DMM algorithm. First, data skewness is an important factor affecting the overall performance of DMM. It can make one processing node produce too many local MFIs as global candidates. To avoid this problem, we randomly distribute the transactions among nodes, but there is no guarantee. Another possible method is clustering the transactions in the whole database based on their itemsets first so that similar transactions will be in the same cluster, and then randomly distributing the transactions within each cluster across all the processing nodes. As a result, the workload of the processing nodes would be more uniform. Several algorithms have been proposed for the clustering of transactions in databases [8,13,25,26], and they can be used to improve the performance of DMM.

Second, we chose the max-miner algorithm to perform the local mining phase. It needs multiple passes over the database partition to find the local MFIs. Compared to MaxClique and MaxEclat algorithms [27,29], max-miner can produce a better set of candidates for the global mining phase, but it may need more processing time for the local mining phase. Combining max-miner with some approximation technique may be a practical solution for the trade-off between the quality of the initial global candidate set and the processing time. Third, in the global mining phase, some exceptionally large candidates in the initial candidate set may require many passes to check their subsets. If we can efficiently test some subsets of those long patterns, it can help us break the long patterns into shorter ones early, so that the number of passes in the global mining phase is reduced.

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

References

1. Agarwal RC, Aggarwal CC, Prasad VVV (2000) Depth first generation of long patterns. In: Proceedings of the 6th ACM SIGKDD international conference on knowledge discovery and data mining, pp 108–118
2. Agrawal R, Srikant R (1994) Fast algorithms for mining association rules. In: Proceedings of the 20th VLDB conference, pp 487–499
3. Agrawal R, Shafer JC (1996) Parallel mining of association rules. IEEE Trans Knowl Data Eng 8(6):962–969
4. Bayardo RJ (1998) Efficient mining long patterns from databases. In: Proceedings of ACM SIGMOD international conference on management of data, pp 85–93
5. Brin S, Motwani R, Ullman J, Tsur S (1997) Dynamic itemset counting and implication rules for market basket data. In: Proceedings of ACM SIGMOD international conference on management of data, pp 255–264
6. Burdick D, Calimlim M, Gehrke J (2001) MAFIA: a maximal frequent itemset algorithm for transaction databases. In: Proceedings of international conference on data engineering, pp 443–452
7. Cheung DW, Han J, Ng V, Fu AW, Fu Y (1996) A fast distributed algorithm for mining association rules. In: Proceedings of the 4th international conference on parallel and distributed information systems, pp 31–43
8. 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

9. Chung SM, Yang J (1996) A parallel distributive join algorithm for cube-connected multiprocessors. IEEE Trans Parallel Distrib Systems 7(2):127–137
10. Frequent Itemset Mining Implementations (FIMI) Repository (2004) http://fimi.cs.helsinki.fi/fimi04/
11. Geurts K, Wets G, Brijs T, Vanhoof K (2003) Profiling high frequency accident locations using association rules. In: Proceedings of the 82nd annual meeting of the transportation research board, Washington, p 18
12. Gouda K, Zaki MJ (2001) Efficiently mining maximal frequent itemsets. In: Proceedings of the 1st IEEE international conference on data mining, pp 163–170
13. Guha S, Rastogi R, Shim K (1999) ROCK: a robust clustering algorithm for categorical attributes. In: Proceedings of international conference on data engineering, pp 512–521
14. Han EH, Karypis G, Kumar V (2000) Scalable parallel data mining for association rules. IEEE Trans Knowl Data Eng 12(3):337–352
15. Han J, Pei J, Yin Y (2000) Mining frequent patterns without candidate generation. In: Proceedings of ACM SIGMOD international conference on management of data, pp 1–12
16. Holt JD, Chung SM (2001) Multipass algorithms for mining association rules in text databases. Knowl Inf Systems 3(2):168–183
17. Holt JD, Chung SM (2002) Mining association rules using inverted hashing and pruning. Inf Process Lett 83(4):211–220
18. Lin D, Kedem ZM (2002) Pincer-Search: an efficient algorithm for discovering the maximal frequent set. IEEE Trans Knowl Data Eng 14(3):553–566
19. Snir M, Otto S, Huss-Lederman S, Walker D, Dongarra J (1996) MPI: the complete reference. MIT Press, Cambridge
20. Park JS, Chen MS, Yu PS (1995) Efficient parallel mining for association rules. In: Proceedings of international conference on information and knowledge management, pp 31–36
21. 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
22. Pasquier N, Bastide Y, Taouil R, Lakhal L (1999) Discovering frequent closed itemsets for association rules. In: Proceedings of international conference on database theory, LNCS, vol 1540, pp 398–416
23. Savasere A, Omiecinski E, Navathe S (1995) An efficient algorithm for mining association rules in large databases. In: Proceedings of the 21st VLDB conference, pp 432–444
24. Thakur R, Rabenseifner R, Gropp W (2000) Optimization of collective communication operations in MPICH. Int J High Perform Comput Appl 19(1):49–66
25. Wang K, Xu C, Liu B (1999) Clustering transactions using large items. In: Proceedings of international conference on information and knowledge management, pp 483–490
26. Yang Y, Guan X, You J (2002) CLOPE: a fast and effective clustering algorithm for transactional data. In: Proceedings of the 8th ACM SIGKDD international conference on knowledge discovery and data mining, pp 682–687
27. Zaki MJ, Parthasarathy S, Ogihara M, Li W (1997) New algorithms for fast discovery of association rules. Computer Science Department Technical Rep. #651, University of Rochester
28. Zaki MJ, Parthasarathy S, Ogihara M, Li W (1997) Parallel algorithms for fast discovery of association rules. Data Mining Knowl Discovery 1(4):343–373
29. Zaki MJ (2000) Scalable algorithms for association mining. IEEE Trans Knowl Data Eng 12(3):372–390
30. Zaki MJ (2000) Generating non-redundant association rules. In: Proceedings of the 6th ACM SIGKDD international conference on knowledge discovery and data mining, pp 34–43

Author biographies

Soon M. Chung received the B.S. degree in Electronic Engineering from Seoul National University, Korea, in 1979, the M.S. degree in Electrical Engineering from Korea Advanced Institute of Science and Technology, Korea, in 1981, and the Ph.D. degree in Computer Engineering from Syracuse University, Syracuse, New York, in 1990. He is currently a professor in the Department of Computer Science and Engineering at Wright State University, Dayton, Ohio. His research interests include database, data mining, Grid computing, text mining, XML, and parallel and distributed processing.

Congnan Luo received the B.E. degree in Computer Science from Tsinghua University, P.R. China, in 1997, the M.S. degree in Computer Science from the Institute of Software, Chinese Academy of Sciences, Beijing, P.R. China, in 2000, and the Ph.D. degree in Computer Science and Engineering from Wright State University, Dayton, Ohio, in 2006. Currently he is a technical staff at the Teradata Corporation in San Diego, CA, and his research interests include data mining, machine learning, and databases.