Efficient Parallel Algorithms for Mining Associations *

Mahesh V. Joshi, Eui-Hong (Sam) Han, George Karypis, and Vipin Kumar

Department of Computer Science, University of Minnesota,
Minneapolis, MN 55455, USA
{mjoshi,han,karypis,kumar}@cs.umn.edu

Abstract. The problem of mining hidden associations present in the large amounts of data has seen widespread applications in many practical domains such as customer-oriented planning and marketing, telecommunication network monitoring, and analyzing data from scientific experiments. The combinatorial complexity of the problem has fascinated many researchers. Many elegant techniques, such as Apriori, have been developed to solve the problem on single-processor machines. However, most available datasets are becoming enormous in size. Also, their high dimensionality results in possibly large number of mined associations. This strongly motivates the need for efficient and scalable parallel algorithms. The design of such algorithms is challenging. In the chapter, we give a evolutionary and comparative review of many existing representative serial and parallel algorithms for discovering two kinds of associations. The first part of the chapter is devoted to the non-sequential associations, which utilize the relationships between events that happen together. The second part is devoted to the more general and potentially more useful sequential associations, which utilize the temporal or sequential relationships between events. It is shown that many existing algorithms actually belong to a few categories which are decided by the broader design strategies. Overall the focus of the chapter is to serve as a comprehensive account of the challenges and issues involved in effective parallel formulations of algorithms for discovering associations, and how various existing algorithms try to handle them.

* This work was supported by NSF grant ACI-9982274, by Army Research Office grant DA/DAAG55-98-1-0441, by Army High Performance Computing Research Center cooperative agreement number DAAH04-95-2-0003/contract number DAAH04-95-C-0008, the content of which does not necessarily reflect the position or the policy of the government, and no official endorsement should be inferred. Access to computing facilities was provided by AHPCRC, Minnesota Supercomputer Institute. Related papers are available via WWW at URL: http://www.cs.umn.edu/~kumar.

1 Introduction

One of the important problems in data mining [1] is discovering associations present in the data. Such problems arise in the data collected from scientific experiments, or monitoring of physical systems such as telecommunications networks, or from transactions at a supermarket. The problem was formulated originally in the context of the transaction data at supermarket. This *market basket* data, as it is popularly known, consists of transactions made by each customer. Each transaction contains items bought by the customer. The goal is to see if occurrence of certain items in a transaction can be used to deduce occurrence of other items, or in other words, to find associative relationships between items. If indeed such interesting relationships are found, then they can be put to various profitable uses such as shelf management, inventory management, etc. Thus, *association rules* were born [2]. Simply put, given a set of items, association rules predict the occurrence of some other set of items with certain degree of confidence. The goal is to discover *all* such *interesting* rules. This problem is far from trivial because of the exponential number of ways in which items can be grouped together and different ways in which one can define interestingness of a rule. Hence, much research effort has been put into formulating efficient solutions to the problem.

It is commonly agreed upon that the number of occurrences of a set of items in a given transaction database, called *support*, can be used to formulate the interestingness of association rules derived from it. A more formal definition of association rules will follow later in the chapter, but informally, the association rule discovery problem usually translates into finding all sets of items that satisfy a pre-specified minimum threshold on support, and then postprocessing them to find the interesting rules. Such itemsets are called *frequent*. In this chapter, we concentrate on the most time consuming operation in this discovery process, which is the discovery of frequent itemsets. Since usually the number of distinct items is large in transaction-based databases, the total number of potential itemsets satisfying the support threshold can be prohibitively large. The first algorithm that handled this problem of exponential explosion elegantly was the Apriori algorithm [3] . This algorithm used a very fundamental property of the support of itemsets: an itemset of size k can meet the minimum level of support only if all of its subsets also meet the minimum level of support. This property is used to systematically prune the search space of desired itemsets, by progressively increasing the length of the itemsets being discovered. Briefly, in an iteration k, all *candidate* k-itemsets (of length k) are formed such that all its $(k-1)$-subsets are frequent. The number of occurrences of these candidates are then counted in the transaction database. Efficient data structures are used to perform fast counting. Overall, the algorithm has been successful on a wide variety of transaction databases. Since its conception, many other algorithms [4–11] have emerged that improve upon the runtime, I/O, and scalability performance of the Apriori algorithm by various efficient means of pruning the itemset search space and counting the candidate occurrences in large databases. We describe serial Apriori algorithm in detail, and give a comparative review of many other representative serial algorithms.

Many practical applications of association rules involve huge transaction databases which contain a large number of distinct items. In such situations, the serial algorithms like Apriori running on single-processor machines may take unacceptably large times. This is despite of the algorithmic improvements proposed in many serial algorithms. The primary reasons are the memory, CPU speed, and I/O bandwidth limitations faced by single-processor machines. As an example, in the Apriori algorithm, if the number of candidate itemsets becomes too large, then they might not all fit in the main memory, and multiple database passes would be required within each iteration, incurring expensive I/O cost. This implies that, even with the highly effective pruning method of Apriori, the task of finding all association rules can require a lot of computational and memory resources, especially when the data is enormous and high dimensional (large number of distinct items). This is true of most of the other serial algorithms as well. This motivates the development of parallel formulations.

Computational work in association rule discovery consists of candidate generation and counting their occurrences, and the memory requirements come from storing the candidates generated. In order to extract concurrency, the computational work and the memory requirements need to distributed among all the available processors. In this chapter, we discuss the pros and cons of different work and memory distribution approaches by studying various parallel formulations of the Apriori-like algorithms in an evolutionary manner. Most existing parallel algorithms can be classified based on how the candidates are distributed among processors. We give details of the representative algorithms [12, 13, 5, 14, 15, 9], and briefly review few other parallelization strategies [16, 17].

The concept of association rules can be generalized and made more useful by observing another fact about transactions. All transactions have a timestamp associated with them; i.e. the time at which the transaction occurred. If this information can be put to use, one can find relationships such as "if an item A was bought by a customer, then he/she is likely to buy an item B in a few days time". The usefulness of this kind of rules gave birth to the problem of discovering *sequential patterns* or *sequential associations*.

In general, the data can be characterized in terms of objects and events happening on these objects. As an example, a customer can be an object and items bought by him/her can be the events. In experiments from molecular biology, an organism or its chromosome can be an object and its behavior observed under various conditions can form events. In a telecommunication network, switches can be objects and alarms happening on them can be events. The *events* happening in such data are related to each other via the temporal relationships of *together* and *before* (or *after*). The *association rules* utilize only the *together* part of the relationship. The concept was extended to the discovery of *sequential patterns* [18] or *episodes* [19], which take into account the sequential (*before*/*after*) relationship as well. The formulation in [18] was motivated by the supermarket transaction data, and the one in [19] was motivated by the telecommunication alarm data. A unified and generalized formulation of sequential associations is proposed in [20].

The sequential nature of the data, depicted by the *before/after* relationships, is important from the discovery point of view as it can be used to discover more powerful and predictive associations, but it is also important from the algorithmic point of view as it increases the complexity of the problem enormously. The total number of possible sequential associations is much larger than non-sequential associations. Various formulations and algorithms proposed so far [18, 19, 21, 22, 20], try to contain the complexity by imposing various temporal constraints, and by using the monotonicity of the support criterion as the number of events in the association increases. The enormity and high dimensionality of data can make these algorithms computationally very expensive, especially because of the more complex nature of sequential associations; and hence, the need for efficient parallel algorithms is even more as compared non-sequential associations. In many situations, the techniques used in parallel algorithms for discovering standard non-sequential associations can be extended easily to discover sequential associations. However, different issues and challenges arise due to the sequential nature of the associations and the way in which interesting associations are defined (counting strategies). In the final part of this chapter, we discuss all these issues and challenges, and a few parallel formulations for resolving them.

The rest of this chapter is organized as follows. Section 2 provides an overview of the serial algorithms for mining association rules. Section 3 describes parallel algorithms for finding association rules. Section 4 contains a description of a generalized formulation of sequential associations and parallel algorithms to discover them. Section 5 summarizes the chapter.

2 Serial algorithms for association rule discovery

2.1 Apriori Algorithm

Let T be the set of transactions where each transaction is a subset of the itemset I. Let C be a subset of I, then we define the *support count* of C with respect to T to be:

$$\sigma(C) = |\{t | t \in T, C \subseteq t\}|.$$

Thus $\sigma(C)$ is the number of transactions that contain C. For example, consider a set of transactions from supermarket as shown in Table 1. The items set I for these transactions is {Bread, Beer, Coke, Diaper, Milk}. The support count of {Diaper, Milk} is $\sigma(Diaper, Milk) = 3$, whereas $\sigma(Diaper, Milk, Beer) = 2$.

An *association rule* is an expression of the form $X \overset{s,\alpha}{\Longrightarrow} Y$, where $X \subseteq I$ and $Y \subseteq I$. The *support* s of the rule $X \overset{s,\alpha}{\Longrightarrow} Y$ is defined as $\sigma(X \cup Y)/|T|$, and the confidence α is defined as $\sigma(X \cup Y)/\sigma(X)$. For example, consider a rule {Diaper, Milk} $\Longrightarrow$ {Beer}, i.e. presence of diaper and milk in a transaction tends to indicate the presence of beer in the transaction. The support of this rule is $\sigma(Diaper, Milk, Beer)/5 = 40\%$. The confidence of this rule is

Table 1. Transactions from supermarket.

TID	Items
1	Bread, Coke, Milk
2	Beer, Bread
3	Beer, Coke, Diaper, Milk
4	Beer, Bread, Diaper, Milk
5	Coke, Diaper, Milk

$\sigma(Diaper, Milk, Beer)/\sigma(Diaper, Milk) = 66\%$. A rule that has a very high confidence (i.e., close to 1.0) is often very important, because it provides an accurate prediction on the association of the items in the rule. The support of a rule is also important, since it indicates how frequent the rule is in the transactions. Rules that have very small support are often uninteresting, since they do not describe significantly large populations. This is one of the reasons why most algorithms [3, 23, 5] disregard any rules that do not satisfy the minimum support condition specified by the user. This filtering due to the minimum required support is also critical in reducing the number of derived association rules to a manageable size. Note that the total number of possible rules is proportional to the number of subsets of the itemset I, which is $2^{|I|}$. Hence the filtering is absolutely necessary in most practical settings.

The task of discovering an association rule is to find all rules $X \overset{s,\alpha}{\Longrightarrow} Y$, such that s is greater than or equal to a given minimum support threshold and α is greater than or equal to a given minimum confidence threshold. The association rule discovery is composed of two steps. The first step is to discover all the frequent itemsets (candidate sets that have more support than the minimum support threshold specified). The second step is to generate association rules from these frequent itemsets. The computation of finding the frequent itemsets is much more expensive than finding the rules from these frequent itemsets. Hence in this chapter, we only focus on the first step.

A number of serial algorithms have been developed for discovering frequent itemsets. We will give a brief review of many of them later in section 2.2. The primary parallel algorithms discussed in this chapter are based on the *Apriori* algorithm [3]. We describe it briefly in the remainder of this section.

The high level structure of the *Apriori* algorithm is given in Figure 1. The *Apriori* algorithm consists of a number of passes. Initially F_1 contains all the items (i.e., item set of size one) that satisfy the minimum support requirement. During pass k, the algorithm finds the set of frequent itemsets F_k of size k that satisfy the minimum support requirement. The algorithm terminates when F_k is empty. In each pass, the algorithm first generates C_k, the candidate itemsets of size k. Function $apriori\_gen(F_{k-1})$ constructs C_k by extending frequent itemsets of size $k-1$. This ensures that all the subsets of size $k-1$ of a new candidate itemset are in F_{k-1}. Once the candidate itemsets are found, their frequencies are computed by counting how many transactions contain these candidate itemsets.

Finally, F_k is generated by pruning C_k to eliminate itemsets with frequencies smaller than the minimum support. The union of the frequent itemsets, $\bigcup F_k$, is the frequent itemsets from which we generate association rules.

```
F_1 = { frequent 1-itemsets } ;
for ( k = 2; F_{k-1} ≠ φ; k++ ) {
    C_k = apriori_gen(F_{k-1})
    for all transactions t ∈ T {
        subset(C_k, t)
    }
    F_k = {c ∈ C_k | c.count ≥ minsup}
}
Answer = ⋃ F_k
```

Fig. 1. Apriori Algorithm

Computing the counts of the candidate itemsets is the most computationally expensive step of the algorithm. One naive way to compute these counts is to perform string-matching of each transaction against each candidate itemset. A faster way of performing this operation is to use a candidate hash tree in which the candidate itemsets are hashed [3]. Here we explain this via an example to facilitate the discussions of parallel algorithms and their analysis.

Figure 2 shows one example of the candidate hash tree with candidates of size 3. The internal nodes of the hash tree have hash tables that contain links to child nodes. The leaf nodes contain the candidate itemsets. A hash tree of candidate itemsets is constructed as follows. Initially, the hash tree contains only a root node, which is a leaf node containing no candidate itemset. When each candidate itemset is generated, the items in the set are stored in sorted order. Note that since C_1 and F_1 are created in sorted order, each candidate set is generated in sorted order without any need for explicit sorting. Each candidate itemset is inserted into the hash tree by hashing each successive item at the internal nodes and then following the links in the hash table. Once a leaf is reached, the candidate itemset is inserted at the leaf if the total number of candidate itemsets are less than the maximum allowed. If the total number of candidate itemsets at the leaf exceeds the maximum allowed and the depth of the leaf is less than k, the leaf node is converted into an internal node and child nodes are created for the new internal node. The candidate itemsets are distributed to the child nodes according to the hash values of the items. For example, the candidate item set $\{1\ 2\ 4\}$ is inserted by hashing item 1 at the root to reach the left child node of the root, hashing item 2 at that node to reach the middle child node, hashing item 3 to reach the left child node which is a leaf node.

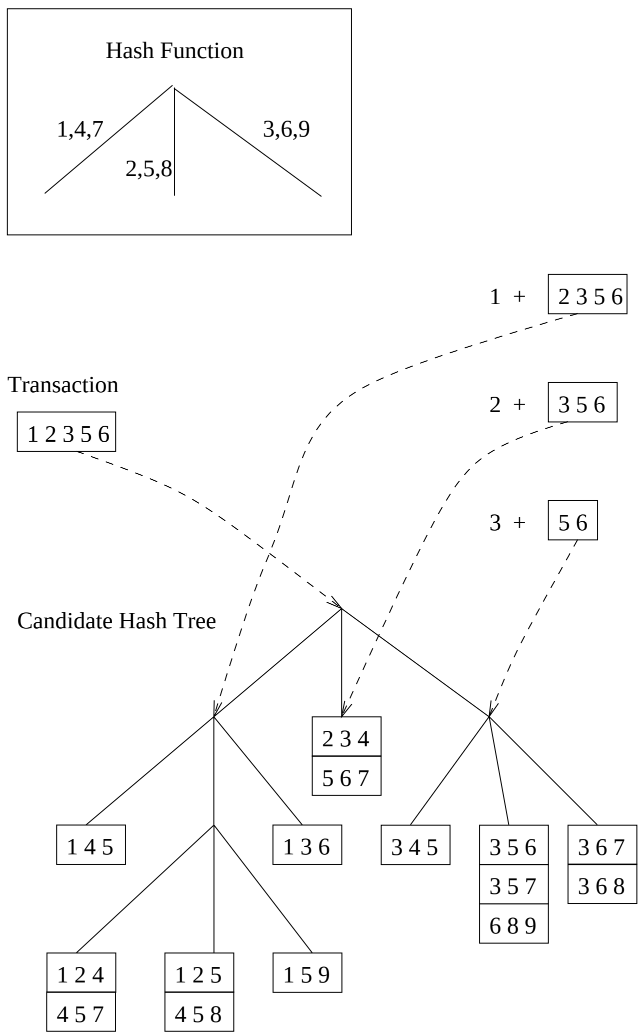


Fig. 2. Subset operation on the root of a candidate hash tree.

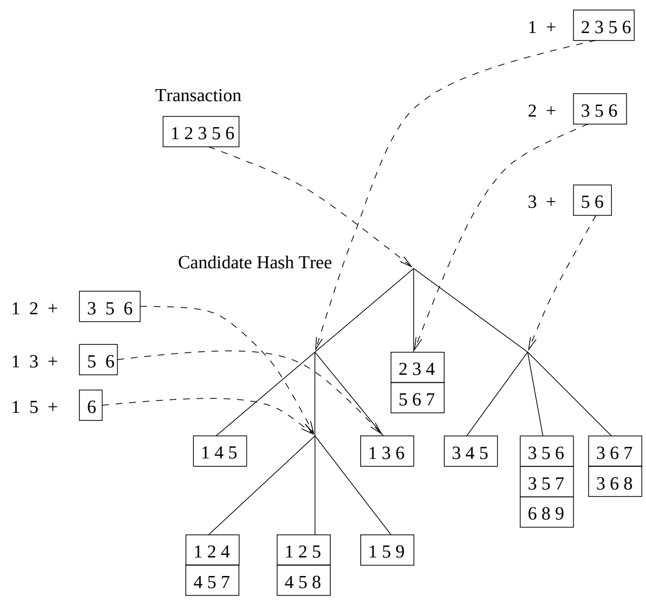


Fig. 3. Subset operation on the left most subtree of the root of a candidate hash tree.

The *subset* function traverses the hash tree from the root with every item in a transaction as a possible starting item of a candidate. In the next level of the tree, all the items of the transaction following the starting item are hashed. This is done recursively until a leaf is reached. At this time, all the candidates at the leaf are checked against the transaction and their counts are updated accordingly. Figure 2 shows the subset operation at the first level of the tree with transaction $\{1\ 2\ 3\ 5\ 6\}$. The item 1 is hashed to the left child node of the root and the following transaction $\{2\ 3\ 5\ 6\}$ is applied recursively to the left child node. The item 2 is hashed to the middle child node of the root and the whole transaction is checked against two candidate itemsets in the middle child node. Then item 3 is hashed to the right child node of the root and the following transaction $\{5\ 6\}$ is applied recursively to the right child node. Figure 3 shows the subset operation on the left child node of the root. Here the items 2 and 5 are hashed to the middle child node and the following transactions $\{3\ 5\ 6\}$ and $\{6\}$ respectively are applied recursively to the middle child node. The item 3 is hashed to the right child node and the remaining transaction $\{5\ 6\}$ is applied recursively to the right child node.

As stated earlier, the runtime for the entire algorithm is dominated by the counting process encoded in the *subset* function. More precisely, according to the analysis presented in [13], at level k of the algorithm, the computation time required per transaction for visiting the hash tree is proportional to N_{C_k}, the number of candidate k-itemsets present in a transaction, and the expected number of distinct leaf nodes visited by the transaction. It is shown that as the number of leaf nodes in hash tree grows larger, the runtime gets dominated more by N_{C_k}.

2.2 Other Serial Algorithms

In the previous subsection, we described Apriori, one of the first and most popular algorithms for generating frequent itemsets. There are many other algorithms proposed after the conception of Apriori. We will briefly describe some representative algorithms from the lot, namely DHP [4], Tree Projection algorithms [9, 10], PARTITION [5], the sampling-based algorithms [6], a family of algorithms proposed in [7], the DIC algorithm [8], and the FP-tree based algorithm [11].

All the algorithms use the monotone property of the itemset support in some way. As stated earlier, this property implies that a k-itemset is frequent only if all of its $(k-1)$-subitemsets are frequent. The sets of items can be visualized to form a lattice. Essentially, all the algorithms traverse this itemset lattice. Different ways of using the monotone property result in different ways of traversal, and that reflects in the performance. Another dimension where algorithms differ is the way they handle the transaction database; i.e. how many passes they make over the entire database and how they reduce the size of the processed database in each pass. With these points in mind, we present a comparative summary of all the algorithms.

A class of algorithms generate candidate k-itemsets from frequent $(k-1)$-itemsets. These are called level-wise algorithms. The Apriori, DHP, and breadth-first Tree Projection algorithms make a pass over the entire database at every level of the algorithm. They differ in the ways they optimize on the number of candidates generated, and the ways that make the counting phase efficient.

DHP (direct hashing and pruning) algorithm improves upon the Apriori algorithm in two ways. First, it reduces the candidate space by looking ahead in the transactions for potentially frequent $(k+1)$-itemsets while counting candidate k-itemsets. This is achieved by hashing all potentially frequent $(k+1)$-subsets of each transaction to a common hash table, and using this hash table to prune some $(k+1)$ candidates without counting them. The algorithm, however, must balance the trade-off between the size of the hash table and its effectiveness in aggressive pruning. The second factor which allows DHP to improve upon Apriori, is its idea of *transaction trimming*. While counting at level k, each item in a transaction is checked for whether it appears in at least k different candidate k-itemsets. If it does not, then it will not be present in any subsequent candidate j-itemsets $(j > k)$, and hence it can be removed from the transaction. Similarly, while preparing the hash table at level k, if an item does not appear in any of the $(k+1)$-itemsets being hashed, then it can be removed from the transaction. If the hashing scheme is effective in pruning many candidates at an early level, then this transaction trimming scheme reduces the active transaction database size substantially, which in turn can reduce the computation time spent per transaction.

Tree Projection algorithms achieve candidate space pruning as well as counting efficiency by combining a novel idea of representing the candidates in a lexicographic tree structure with a way of reducing the transaction database size in every pass by *projecting* the transactions onto this lexicographic tree. The lexicographic tree is an alternative systematic representation of the itemset lattice. Each node in the tree is associated with an itemset and a set of its possible extensions. A node can be extended only by an item that is lexicographically larger and appears as an extension of the its parent. A list of *active items* is kept at each node. Also, the extensions of each node are marked active or inactive. The active item list is used to project a transaction onto the node, and this projected transaction needs to flow down only the active extensions. The idea is, only those items in a transaction percolate down the tree that can potentially be useful in extending the tree by one more level. With every pass of the algorithm, many extensions become progressively inactive, which in turn results in reduction of active item list sizes at all nodes. This yields the algorithm its efficiency in counting phase as well as helps it in possibly pruning the candidate space more aggressively as compared to Apriori or DHP. The concept of projection can be thought of as a more generalized form of transaction trimming used in DHP. Also, the concept of active items and active extensions effectively render the lexicographic tree as a compact, dynamic version of the hash-tree structures used in Apriori.

The PARTITION and sampling-based algorithms [6] are level-wise, but only on a small portion of the entire database. In fact, use of smaller subsets of database allows them to optimize the database performance by making at most two passes over the entire database.

PARTITION algorithm takes the idea of support monotonicity further. It partitions the database into multiple parts, and observes that if an itemset is frequent in the entire database then it is frequent in at least one of the partitions, when the frequency is computed relative to the partition size. This observation is used to prune the potential frequent itemsets by counting the candidates in smaller local partitions. A level-wise algorithm is employed to generate all *locally frequent* itemsets. All such itemsets are gathered and their global counts are collected in a second pass over the database. Thus, only two database passes are needed. In order to achieve true gain in performance, the algorithm has to minimize the effect of data skew across partitions by randomizing the partitioning scheme. It also has to take care of the trade-off between the partition size and number of partitions. Finding locally large itemsets in smaller partitions is quick, but the lower amount of information available in smaller partitions also tends to give rise to many false positives because the support is counted with respect their small size. The PARTITION algorithm has one more novel feature as compared to Apriori, which can potentially accelerate the counting phase. It uses vertical data layout in which instead of storing a list of items for each transaction (horizontal layout as used in Apriori), it stores the *tid-list* of transaction ids for each item. It is made sure that the size of each partition is such that all the required tid-lists in a partition fit in the main memory. This allows the itemset support counting to be performed efficiently by intersecting the tid-lists of its individual items.

The sampling-based algorithms proposed in [6] use a randomly sampled partition of the database to find locally frequent itemsets in that partition. The gain in performance is possible due to the less amount of data that the algorithms work on, making it attractive for large databases. However, in order to ensure the completeness of the frequent itemsets discovered, the algorithm has to do several things. First, it has to reduce the support threshold used for discovering frequent sets in the sampled data. This is done with the hope of capturing most of the actual frequent itemsets. Despite of this reduction in support threshold (which cannot be reduced below certain level), some itemsets will be missing. The algorithm has a novel systematic strategy of checking for all the missing itemsets. It introduces a concept of *negative border* of the locally frequent itemsets. This border is formed by all minimal small itemsets; i.e., the sets which are infrequent but all their subsets are frequent. Locally frequent sets and the sets in their negative border are counted in the entire database, and these global counts are used to see if any true frequent itemsets are lost by sampling. Since the algorithm is based on a random sample, the authors present a probabilistic analysis that relates the sample size, the limit on lowering support threshold, and accuracy that can be achieved.

The class of non-level-wise algorithms consists of the hybrid lattice traversal technique proposed in [7], the DIC algorithm, and the depth-first version of the tree projection algorithm [10]. Like PARTITION and sampling-based algorithms, the design goal for these algorithms is reduction in the number of passes made over the entire database. However, the major point of difference is their itemset lattice traversal technique. Instead of a level-wise (or breadth-first) traversal, they interleave the depth-first and breadth-first searches with the database passes. In other words, the candidate generation and candidate counting phases are interleaved. The guiding factor is the search for either the maximal frequent itemsets [7] or the minimal infrequent itemsets [8].

The lattice traversal algorithms proposed in [7] use a vertical layout (similar to PARTITION). One pass is made over the database to generate the item tid-lists. After that, no more passes are required over the database. Only the tid-lists need to be scanned. A novel feature of all their algorithms is that they are seeded by an itemset clustering method. The clustering allows them to identify close approximations to the potentially maximal itemsets. This may substantially prune the candidate search space by dividing the original itemset lattice into smaller sublattices formed only by items belonging to same cluster. They propose three different approaches to traverse these smaller itemset sublattices. The bottom-up approach does a breadth-first traversal of the lattice starting with the 2-itemsets. This is similar to the level-wise algorithms. But it faces a problem of generating all the subsets of frequent itemsets. The top-down approach starts with potentially maximal itemsets given by the clustering, and goes down the lattice until all the maximal frequent itemsets are found. This approach faces the problem of costly multi-way intersections of tid-lists as well as it suffers from the approximate nature of clusters. A hybrid approach combines the good features of both approaches, and is shown to be better than the two. Although it is true that the entire database is scanned once, there are several passes made over the individual tid-lists. The main performance gain achieved may be attributed to their clustering scheme to prune the search space clubbed with an underlying assumption that the tid-lists for individual items or 2-itemsets are not very large.

The DIC algorithm is a recent non-level-wise algorithm which is actually closer to the sampling-based algorithms. Instead of using a random sample of the database and potentially losing some frequent item-sets, it proposes a systematic search of the database that reduces the number of database passes to some number between 1 and the total number of passes that would be made by a level-wise algorithm. Unlike level-wise algorithms which count only k-itemsets in one pass of the algorithm, DIC starts counting longer itemsets after some fixed intervals during a given database pass. For example, in a database of 10000 transactions, it starts computing 1-itemsets at first transaction, then some 2-itemsets start getting counted after M=1000 transactions, some 3-itemsets start getting counted after 2*M=2000 transactions, and so on. The value of M can be changed. Each itemset that the algorithm decides to count, gets counted in each transaction. The algorithm keeps track of potential frequent itemsets and potential minimal small itemsets. The counting starts only for those itemsets whose subsets have been found frequent in the data visited so far. Essentially the amount of lattice traversed by the algorithm is same as that by a level-wise algorithm, but the dynamic nature of counting the itemsets gives the algorithm a flexibility to reduce database passes. The crucial factor for its performance is the ability to identify frequent subsets of a given itemset early enough so that the itemset starts getting counted early. Ideally if the probability of seeing a given itemset in any fraction of transactions is the same, then DIC performs very well. However, if the dataset is not *homogeneous*, then the performance would suffer. The authors of DIC identify this problem and propose some remedies such as randomization and relaxing the support threshold.

The depth-first version of the tree projection algorithm [10] generates the lexicographic tree in a depth-first manner. The crucial factor for its performance is that the entire transaction database needs to fit in the memory, which is not very practical for many transaction databases. Hence, we will not review it in detail here.

Finally, we briefly review a class of algorithms [24, 11] that choose a radically different approach to discover frequent itemsets. These algorithms do not involve generation of potential candidates. The algorithm based on *FP-trees* [11] uses a compact trie-like representation of the transaction database that is used to directly infer the frequent itemsets involving a given frequent item. This compact representation is achieved using the data structure called frequent pattern tree (FP-tree), which is a data structure based on set-enumeration tree formed using frequency-ordered 1-itemsets. It is constrained using the given transaction database in the following manner. Each transaction is transformed to a frequency-ordered set of items and is mapped to the set-enumeration tree. The counts of items on the path it gets mapped to are incremented by one. All the occurrences of a given item are linked across the tree. Once FP-tree is constructed, for each item, the algorithm finds all the frequent itemsets having that item as the last item (in frequency-order). This is achieved by using the prefix paths to all the occurrences of that item in the tree. A systematic recursive decomposition of the prefix paths yields all the desired frequent itemsets. If this process is mapped to a lattice traversal process, then the algorithm essentially traverses the lattice in a top-down fashion (i.e. going from longer itemsets down to smaller itemsets), starting with the itemset formed by all the frequent items in the union of the items occurring in the prefix paths. However, its recursion process breaks the lattice down into only the interesting sublattices driven by the increasingly smaller FP-trees. This authors show their algorithm to be an order of magnitude faster than the Apriori algorithm and considerably faster than the Tree Projection algorithm.

A related algorithm proposed in [24], also uses the compact trie [25] representation of the transaction database, to directly infer the frequent associations. However, unlike FP-tree, which encodes the entire transaction database into a trie-like structure, their algorithm constructs a trie only out of those subsets of a transaction that contains less than a pre-specified number of items. This was motivated by their observation that the largest frequent itemsets do not contain more than 8-10 items. Once the trie is constructed, they use all the subsets present in the trie as potential frequent sets. However, unlike FP-tree based algorithm, they do not give a systematic algorithm for inferring actual frequent itemsets based on support.

This concludes our survey of the representative serial algorithms for computing frequent itemsets.

3 Parallel Formulations

The enormity and high dimensionality of datasets typically available as input to the problem of association rule discovery, makes it an ideal problem for solving on multiple processors in parallel. The primary reasons are the memory and CPU speed limitations faced by single processors. Despite of many recent improved approaches to compute all frequent itemsets, the sheer amount of computational work that needs to be done for large and high dimensional problems results in prohibitively large runtimes on single processors. Thus, it is critical to design efficient parallel algorithms to do the task. Another reason for designing parallel algorithms comes from the fact that many transaction databases are already available in parallel databases or they are distributed at multiple sites to begin with. The cost of of bringing them all to one site or one computer for serial discovery of association rules can be prohibitively expensive.

In the process of association rule discovery, the first part of finding frequent itemsets is much more expensive as compared to the second part of finding rules from these frequent itemsets. Hence, we concentrate on parallel algorithms for frequent itemset discovery. We reviewed many different serial algorithms in previous subsection. Except for a few, most of these algorithms involve generation of candidate itemsets and counting them in the transaction database, especially the level-wise algorithms such as Apriori. First, we present possible parallel formulations of such algorithms and map the existing parallel algorithms to these formulations. In the end, we review parallel formulations of some non-level-wise algorithms.

3.1 Parallel Formulations of level-wise Algorithms

The computational work in level-wise algorithms can be viewed to consist of two parts: the effort spent in generating the candidates and the effort spent in counting them. In order to distribute this work among processors, multiple possibilities emerge depending on how the transactions and candidate itemsets are assigned to processors. The need for parallel algorithms comes from the transaction database being too large (enormity of the database), or possible number of frequent itemsets being too large (because of high dimensionality of the database), or both. Correspondingly, in order to achieve concurrency, either the candidates need to be counted in parallel, or they need to be generated in parallel, or both phases need to be done in parallel.

We assume that the transaction database is too large to be replicated among all processors. For most practical problems in data mining, this is a fair or rather necessary assumption. Usually, the transactions are distributed among processors equally. Given this, the issue becomes how to distribute the candidates among processors such that their counting and generation is effectively parallelized. There are two possibilities. One is to replicate the candidates on all processors and the other is to avoid replication. In the following we review in detail various algorithms based on these possibilities. The discussion takes into account the issues of minimizing parallelization overheads, extracting concurrency, and utilizing the total available memory effectively.

Replicating Candidate Itemsets One possible way to parallelize is to simply replicate the candidate generation process on all the processors, and parallelize the counting process. Here are a few representative algorithms that take this approach.

- **Count Distribution (CD):** In this parallel formulation of Apriori algorithm, proposed in [26], each processor computes how many times all the candidates appear in the locally stored transactions. This is done by building the entire hash tree that corresponds to all the candidates and then performing a single pass over the locally stored transactions to collect the counts. The global counts of the candidates are computed by summing these individual counts using a global reduction operation [27]. This algorithm is illustrated in Figure 4. Note that since each processor needs to build a hash tree for all the candidates, these hash trees are identical at each processor. Thus, excluding the global reduction, each processor in the *CD* algorithm executes the serial *Apriori* algorithm on the locally stored transactions. The

NPA (Non-Partitioned Apriori) algorithm, proposed in [15], is also identical to this CD algorithm.
This algorithm has been empirically shown to scale linearly with the number of transactions [26]. A detailed scalability analysis is presented by [13]. Given N number of transactions and P number of processors, if M is the total number of candidates that get generated, then they show that the parallel runtime of the algorithm is $T_s/P + O(M)$, where T_s is the serial runtime of the algorithm. The $O(M)$ term comes from the hash tree construction and global reduction of counts. This indicates that the algorithm is scalable in number of transactions, however it does not parallelize the computation of building the candidate hash tree. This step becomes a bottleneck with large number of processors. Furthermore, if the number of candidates is large, then the hash tree does not fit into the main memory. In this case, this algorithm has to partition the hash tree and compute the counts by scanning the database multiple times, once for each partition of the hash tree. The cost of extra database scanning can be expensive in the machines with slow I/O system. Note that the number of candidates increases if either the number of distinct items in the database increases or if the minimum support level of the association rules decreases. Thus the *CD* algorithm is effective for small number of distinct items and a high minimum support level.

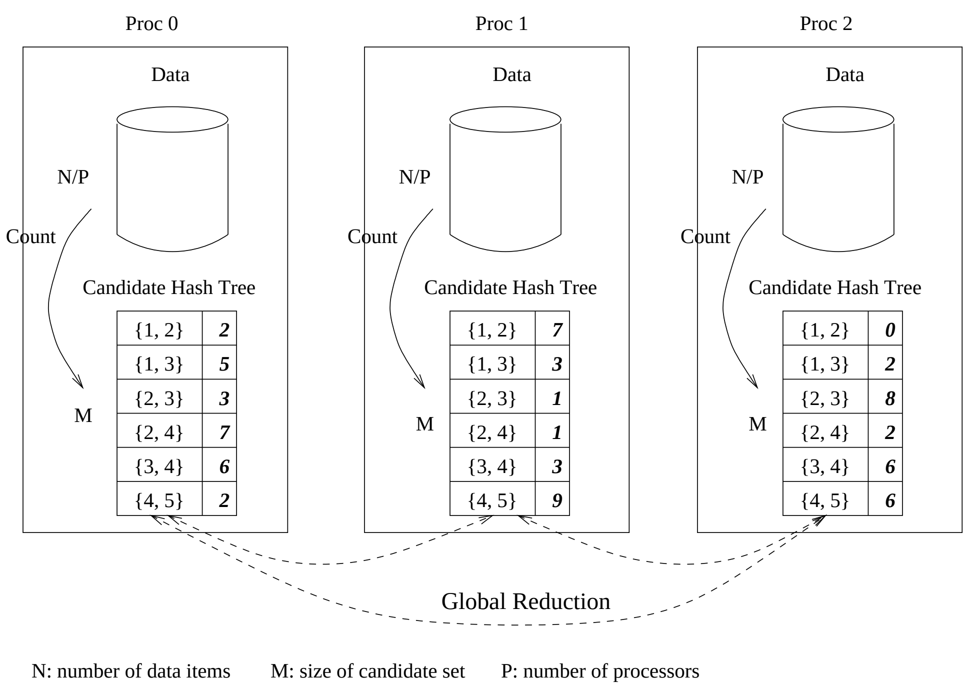


Fig. 4. Count Distribution (CD) Algorithm

- **Parallel PARTITION algorithm:** The parallel formulation of the serial PARTITION algorithm has been given in [5]. The serial algorithm has inherent parallelism in it as far as processing of each partition is concerned. The algorithm is very similar to the count distribution algorithm, in that the data is distributed and the candidate set is replicated among processors. The difference is that the frequent itemsets are counted in four stages. In the first stage, each processor discovers *locally frequent* itemsets assuming that its local data is the entire database. Next, these itemsets are exchanged among processors, forming the global candidate set. In the third stage, local counts for these candidates are computed by scanning the local data again. Finally, a communication operation is performed to add up the local counts to get the global counts for the candidates, from which globally frequent itemsets can be determined. In this algorithm, the size of the candidate set generated in second stage is dependent on the size of local datasets and skew in the data. It could potentially be bigger than the candidate set in CD because of false positives, and hence can cause the algorithm to lose its main purpose of achieving efficiency by pruning based on local counts. As in the serial case, the vertical data layout used in parallel PARTITION can make the counting phase efficient, and allows it to avoid multiple scans of the local database.

- **PDM Algorithm:** Another parallel algorithm which is based on the serial Apriori-like algorithm is PDM [14], which is a parallel formulation of the DHP [4] algorithm. The approach to parallelization is very much similar to the CD algorithm. The difference is in the fact that DHP differs from Apriori in its use of hash tables to look ahead into the potential candidates of next phase. The phase of candidate generation from frequent k-itemsets in parallelized in PDM by using a parallel nested loop join algorithm, where each processor generates only a small subset of entire candidate set. These sets are exchanged by all nodes to generate global candidate set similar to CD. The crucial point in the parallel formulation of DHP is the construction of the hash table in parallel. Since the hash table is used in the subsequent candidate generation pass to prune the candidates, a global copy of the hash table should be available to all the processors. While counting k-itemsets, the hash table stores the counts of $k+1$-itemsets appearing in transactions. Since the transactions are partitioned across processors, each processor will have the counts due to local transactions. A simple approach of gathering global counts for each location in the hash table is to do a global exchange of all local hash tables. The potential of requiring a large hash table size, especially for 2-itemsets, makes this simple approach inefficient. The paper proposes an optimization over this by simply observing the fact that not all entries in the local hash tables need to be exchanged with other processors. An entry in the global hash table will be greater than support threshold, s, only if at least one processor has its corresponding local entry greater than s/p, where p is the number of processors. This fact is used to determine which entries should be exchanged using global broadcast. Rest of the entries in the hash table are exchanged using a clue-and-poll procedure which reduces the amount of communication. Since the same hash table and the entire candidate set is available to all the processors, the transaction trimming feature of DHP algorithm is easily maintained in PDM as well. Each processor tries to reduce the size of transactions in its local partition. Overall, PDM is much similar to CD. But, effective parallelization of hash table construction, the possible advantages gained by a good hashing function, and the transaction trimming might give PDM an edge over CD.

– **Count Distributed Tree Projection algorithm:** This formulation proposed in [9] is based on the CD algorithm described above. Identical lexicographic tree, upon which the tree projection algorithms are based, is built on each processor and counts are communicated at every level. As with CD, this parallel formulation works well only if the lexicographic tree fits in memory, and its scalability with number of candidates is poor.

Partitioning Candidate Itemsets Given the problems possibly encountered because of replication of candidates, an alternative approach would be to partition the candidates among processors. However, many issues arise regarding how to partition them and how to effectively parallelize counting for given partitioning. Following algorithms handle these issues. DD algorithm discussed first makes a simple yet weak effort to parallelize. The next algorithm, IDD, improves upon it greatly. A few other algorithms, inspired by IDD, are also described in the end.

– **Data Distribution (DD):** This algorithm [26] addresses the memory problem of the *CD* algorithm by partitioning the candidate item-sets among the processors. This partitioning is done in a round robin fashion. Each processor is responsible for computing the counts of its locally stored subset of the candidate item-sets for all the transactions in the database. In order to do that, each processor needs to scan the portions of the transactions assigned to the other processors as well as its locally stored portion of the transactions. In the *DD* algorithm, this is done by having each processor receive the portions of the transactions stored in the other processors as follows. Each processor allocates P buffers (each one page long and one for each processor). At processor P_i, the i^{th} buffer is used to store transactions from the locally stored database and the remaining buffers are used to store transactions from the other processors. Now each processor P_i checks the P buffers to see which one contains data. Let l be this buffer (ties are broken in favor of buffers of other processors and ties among buffers of other processors are broken arbitrarily). The processor processes the transactions in this buffer and updates the counts of its own candidate subset. If this buffer corresponds to the buffer that stores local transactions (i.e., $l = i$), then it is sent to all the other processors (via asynchronous sends), and a new page is read from the local database. If this buffer corresponds to a buffer that stores transactions from another processor (i.e., $l \neq i$), then it is cleared and this buffer is marked available for next asynchronous receive from any other processors.

This continues until every processor has processed all the transactions. Having computed the counts of its candidate item-sets, each processor finds the frequent item-sets from its candidate item-set and these frequent item-sets are sent to every other processor using an all-to-all broadcast operation [27]. Figure 5 shows the high level operations of the algorithm. Note that each processor has a different set of candidates in the candidate hash tree.
The SPA (Simply Partitioned Apriori) algorithm, proposed in [15], is identical to DD. It partitions the candidates among processors in a round robin manner. Each transaction is broadcast to all the processors so as to generate a global count for *all* the candidates.

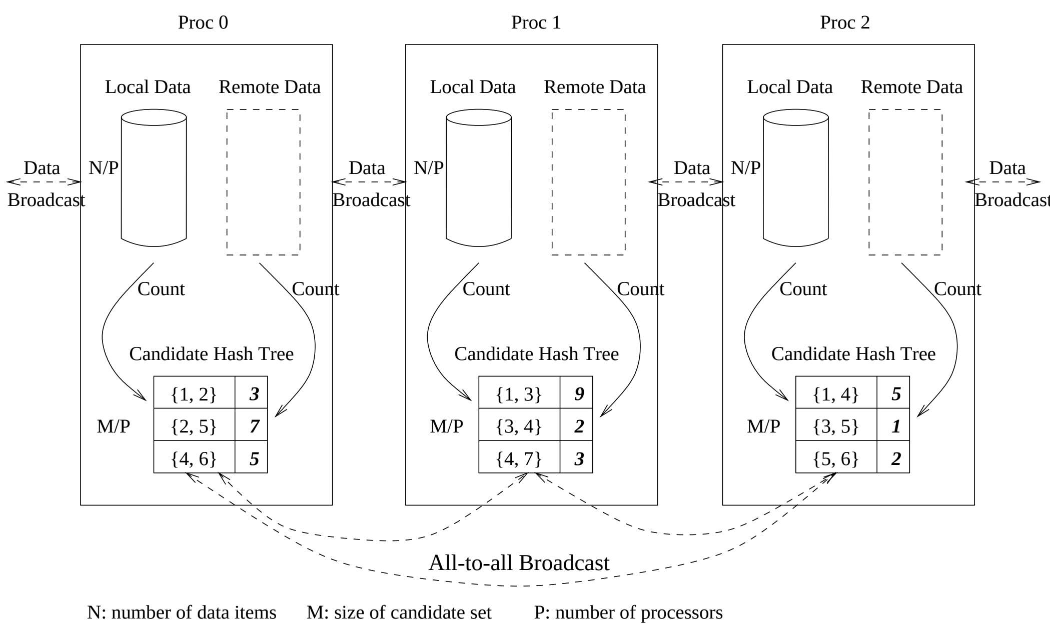


Fig. 5. Data Distribution (DD) Algorithm

The DD algorithm exploits the total available memory better than *CD*, as it partitions the candidate set among processors. As the number of processors increases, the number of candidates that the algorithm can handle also increases. However, as reported in [26], the performance of this algorithm is significantly worse than the *CD* algorithm. The run time of this algorithm is 10 to 20 times more than that of the *CD* algorithm on 16 processors [26]. The problem lies with the communication pattern of the algorithm and the redundant work that is performed in processing all the transactions.
The communication pattern of this algorithm causes three problems. First, during each pass of the algorithm each processor sends to all the other processors the portion of the database that resides locally. In particular, each

processor reads the locally stored portion of the database one page at a time and sends it to all the other processors by issuing $P-1$ send operations. Similarly, each processor issues a receive operation from each other processor in order to receive these pages. If the interconnection network of the underlying parallel computer is fully connected (i.e., there is a direct link between all pairs of processors) and each processor can receive data on all incoming links simultaneously, then this communication pattern will lead to a very good performance. In particular, if $O(N/P)$ is the size of the database assigned locally to each processor, the amount of time spent in the communication will be $O(N/P)$. However, even on the parallel computer with fully connected network, if each processor can receive data from (or send data to) only one other processor at a time, then the communication will be $O(N)$. On all realistic parallel computers, the processors are connected via a sparser networks (such as 2D, 3D or hypercube) and a processor can receive data from (or send data to) only one other processor at a time. On such machines, this communication pattern will take significantly more than $O(N)$ time because of contention within the network.

Second, in architectures without asynchronous communication support and with finite number of communication buffers in each processor, the proposed all-to-all communication scheme causes processors to idle. For instance, consider the case when one processor finishes its operation on local data and sends the buffer to all other processors. Now if the communication buffer of any receiving processors is full and the outgoing communication buffers are full, then the send operation is blocked.

Third, if we look at the size of the candidate sets as a function of the number of passes of the algorithm, we see that in the first few passes, the size of the candidate sets increases and after that it decreases. In particular, during the last several passes of the algorithm, there are only a small number of items in the candidate sets. However, each processor in the *DD* algorithm still sends the locally stored portions of the database to all the other processors. Thus, even though the computation decreases, the amount of communication remains the same.

The redundant work is introduced due to the fact that every processor has to process every single transaction in the database. In *CD* (see Figure 4), only N/P transactions go through each hash tree of M candidates, whereas in *DD* (see Figure 5), all N transactions have to go through each hash tree of M/P candidates. Although, the number of candidates stored at each processor has been reduced by a factor of P, the amount of computation performed for each transaction has not been proportionally reduced. According to the analysis presented in [13], in general, the amount of work per transaction will go down by a factor much smaller than P.

The detailed analysis of parallel runtime is given in [13], according to which the algorithm is not scalable with respect to number of transactions, but it scales well with respect to number of candidates.

- **Intelligent Data Distribution (IDD):** This algorithm was proposed in [28]. It solves the problems of the *DD* algorithm. First, in *IDD*, the locally stored portions of the database are sent to all the other processors by using a ring-based all-to-all broadcast described in [27]. Compared to *DD*, where all the processors send data to all other processors, IDD performs only a point-to-point communication between neighbors, thus eliminating any communication contention that DD algorithm faces. Thus, the all-to-all broadcast operation takes $O(N)$ time on *any* parallel architecture that can be embedded in a ring. Furthermore, if the time to process a buffer does not vary much, then there is little time lost in idling. Also, when it is implemented using asynchronous communication operations, the computation and communication operations can be overlapped.
  Second problem of DD that IDD improves upon is that of redundant work. In order to eliminate the redundant work due to the partitioning of the candidate item-sets, IDD finds a fast way to check whether a given transaction can potentially contain any of the candidates stored at each processor. This cannot be done by partitioning C_k in a round-robin fashion. However, if C_k is partitioned among processors in such a way that each processor gets item-sets that begin only with a subset of all possible items, then the items of a transaction can be checked against this subset to determine if the hash tree contains candidates starting with these items. The hash tree is traversed with only the items in the transaction that belong to this subset. Thus, the redundant work problem of *DD* is solved by the intelligent partitioning of C_k.
  These points can be understood better by looking at Figure 6, which shows the high level picture of the algorithm. In this example, Processor 0 has all the candidates starting with items 1 and 7, Processor 1 has all the candidates starting with 2 and 5, and so on. Each processor keeps the first items of the candidates it has in a bit-map. In the *Apriori* algorithm, at the root level of hash tree, every item in a transaction is hashed and checked against the hash tree. However, in IDD, at the root level, each processor filters every item of the transaction by checking against the bit-map to see if the processor contains candidates starting with that item of the transaction. If the processor does not contain the candidates starting with that item, the processing steps involved with that item as the first item in the candidate can be skipped. This reduces the amount of transaction data that has to go through the hash tree; thus, reducing the computation. For example, let $\{1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\}$ be a transaction that processor 0 is processing in the *subset* function discussed in Section 2.1. At the top level of the hash tree, processor 0 will only proceed with items 1 and 7 (i.e., 1 + 2 3 4 5 6 7 8 and 7 + 8). When the page containing this transaction is shifted to processor 1, this processor will only process items starting with 2 and 5 (i.e., 2 + 3 4 5 6 7 8 and 5 + 6 7 8). Figure 7 shows how this scheme works when a processor contains only those candidate item-sets that start with 1, 3 and 5.
  Thus for each transaction in the database, IDD partitions the amount of work to be performed among processors, thus eliminating most of the re-

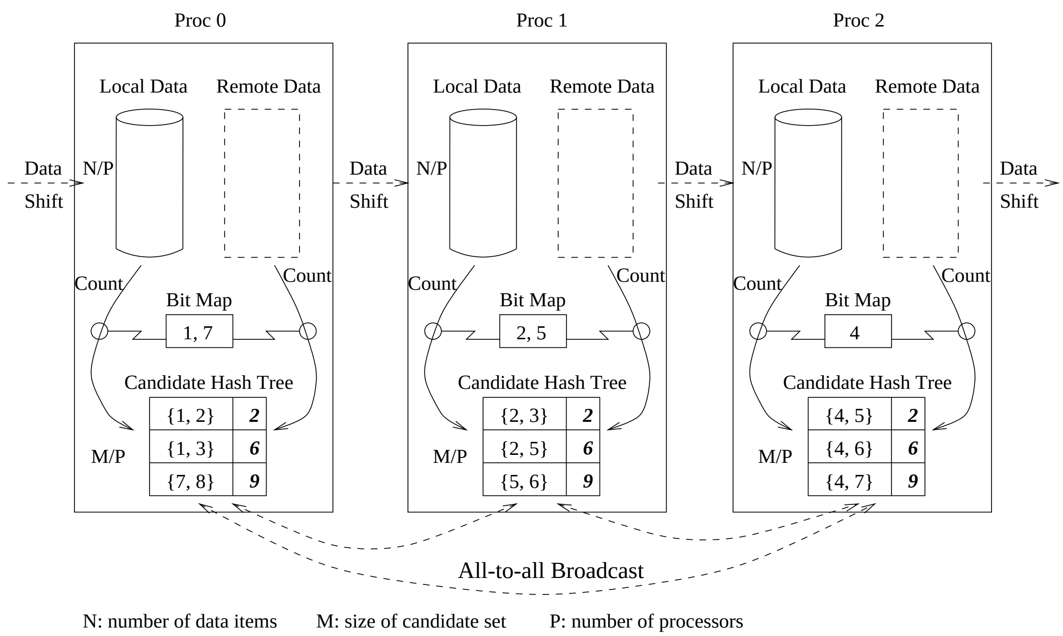


Fig. 6. Intelligent Data Distribution (IDD) Algorithm

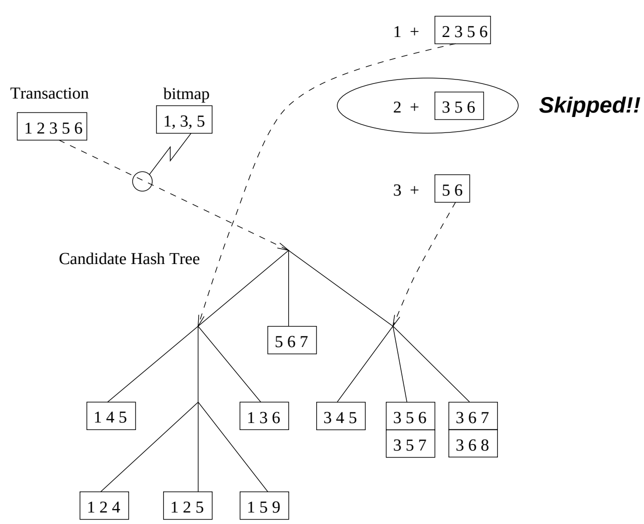


Fig. 7. Subset operation on the root of a candidate hash tree in IDD.

dundant work of *DD*. Note that both the judicious partitioning of the hash tree (indirectly caused by the partitioning of candidate item-set) and the filtering step are required to eliminate this redundant work.
The intelligent partitioning of the candidate set used in *IDD* brings up the issue of load balancing. One of the criteria of a good partitioning involved here is to have an equal number of candidates in all the processors. This gives about the same size hash tree in all the processors and thus provides good load balancing among processors. Note that in the *DD* algorithm, this was accomplished by distributing candidates in a round robin fashion. This does not give any guarantees of load balance. Even in IDD, a naive method for assigning candidates to processors can lead to a significant load imbalance. For instance, consider a database with 100 distinct items numbered from 1 to 100 and that the database transactions have more data items numbered with 1 to 50. Let the candidates be partitioned between two processors. If all the candidates starting with items 1 to 50 are assigned to processor P_0 and all candidates starting with items 51 to 100 to processor P_1, then there would be more work for processor P_0.
To achieve an equal distribution of the candidate item-sets, the authors of IDD use a partitioning algorithm that is based on bin-packing [29]. For each item, they first compute the number of candidate item-sets starting with this particular item. Note that at this time they do not actually store the candidate item-sets, but they just store the number of candidate item-sets starting with each item. Then a bin-packing algorithm is used to partition these items in P buckets such that the sum of numbers of the candidate item-sets starting with these items in each bucket are roughly equal. Once the location of each candidate item-set is determined, then each processor locally regenerates and stores candidate item-sets that are assigned to this processor. Note that bin-packing is used per pass of the algorithm and the amount of time spent on bin-packing is minor compared to the overall runtime. Figure 6 shows the partitioned candidate hash tree and its corresponding bitmaps in each processor.
Note that this scheme will not be able to achieve an equal distribution of candidates if there are too many candidate itemsets starting with the same item. For example, if there are more than M/P candidates starting with the same item, then one processor containing candidates starting with this item will have more than M/P candidates even if no other candidates are assigned to it. This problem gets more serious with increasing P. One way of handling this problem is to partition candidate item sets based on more than the first items of the candidate item sets. In this approach, whenever the number of candidates starting with one particular item is greater than the threshold, this item set is further partitioned using the second item of the candidate item sets.
Note that the equal assignment of candidates to the processors does not guarantee the perfect load balance among processors. This is because the cost of traversal and checking at the leaf node are determined not only by the size and shape of the candidate hash tree, but also by the actual items

in the transactions. However, in the experiments, authors [28] have observed a reasonably good correlation between the size of candidate sets and the amount of work done by each processor. For example, with 4 processors, the load imbalance was 1.3% in terms of the number of candidate sets, which translated into 5.4% load imbalance in the actual computation time. With 8 processors, load imbalance was 2.3% in the number of candidate sets, and this resulted in 9.4% load imbalance in the computation time. Since the effect of transactions on the work load cannot be easily estimated in advance, IDD scheme only ensures that each processor has roughly equal number of candidate itemsets in the local hash tree.

A detailed analysis of the load balancing issues and scalability of IDD is given in [13]. In summary, IDD has the flexibility of minimizing the data movement cost by overlapping the counting computation with data communication. Moreover, it does not perform any redundant computation as in DD, which makes it more scalable than DD with respect to number of transactions, and it is scalable with respect to the number of candidates.

- **HPA Algorithm:** The HPA (Hash Partitioned Apriori) algorithm, given in [15], is similar in spirit to the IDD algorithm. It tries to reduce the communication overhead of sending each transaction to every processor. It assigns the candidates to processors using a hash function, which determines which processor the candidate would go to. In the counting phase, if candidate k-itemsets are being counted, then each transaction in local database is first processed to find all the k-itemsets present in the transaction. Each such itemset is hashed using the same hash function as used for partitioning the candidates to derive the destination processor, and is sent to that processor. This partitioning due to hashing function can be considered similar to the mechanism of partitioning candidates in IDD, but unlike IDD, HPA does not give any guarantees of load balance achieved because of its hashing-based candidate distribution.

- **Intelligent Data Distributed Tree Projection algorithm:** This formulation proposed in [9] is based on the IDD algorithm described above. The lexicographic tree, upon which the tree projection algorithms are based, is distributed among different processors based on the first item in the tree. Using the active item lists at the root of each of the processor's lexicographic tree, only relevant transactions can be communicated to a given processor. This can further save on the communication overhead.

Hybrid Approach: Partial Replication of Candidate Itemsets We saw two approaches: pure replication of candidates and pure partitioning with no replication. However, according to analyses of these approaches, especially for CD and IDD, it can be seen that each approach has some issues regarding scalability. In particular, CD is scalable with respect to number of transaction because of replicated candidate sets, whereas IDD is scalable with respect to number of candidates. This hybrid approach is essentially an attempt to see if two approaches can be combined via partial replication of candidates, to achieve better scalability than both. In the following, we discuss some algorithms that have been able to do this successfully.

– **HD (Hybrid Distribution) Algorithm:** The *IDD* algorithm exploits the total system memory by partitioning the candidate set among all processors. The average number of candidates assigned to each processor is M/P, where M is the number of total candidates. As more processors are used, the number of candidates assigned to each processor decreases. This has two implications. First, with fewer number of candidates per processor, it is much more difficult to balance the work. Second, the smaller number of candidates gives a smaller hash tree and less computation work per transaction. Eventually the amount of computation may become less than the communication involved. This would be more evident in the later passes of the algorithm as the hash tree size further decreases dramatically. This reduces overall efficiency of the parallel algorithm. This will be an even more serious problem in a system that cannot perform asynchronous communication.
  The *Hybrid Distribution* (*HD*) algorithm addresses the above problem by combining the *CD* and the *IDD* algorithms in the following way. Consider a P-processor system in which the processors are split into G equal size groups, each containing P/G processors. In the *HD* algorithm, we execute the *CD* algorithm as if there were only P/G processors. That is, we partition the transactions of the database into P/G parts each of size $N/(P/G)$, and assign the task of computing the counts of the candidate set C_k for each subset of the transactions to each one of these groups of processors. Within each group, these counts are computed using the *IDD* algorithm. That is, the transactions and the candidate set C_k are partitioned among the processors of each group, so that each processor gets roughly $|C_k|/G$ candidate item-sets and N/P transactions. Now, each group of processors computes the counts using the *IDD* algorithm, and the overall counts are computing by performing a reduction operation among the P/G groups of processors.
  The *HD* algorithm can be better visualized if we think of the processors as being arranged in a two dimensional grid of G rows and P/G columns. The transactions are partitioned equally among the P processors. The candidate set C_k is partitioned among the processors of each column of this grid. This partitioning of C_k is identical for each column of processors; i.e., the processors along each row of the grid get the same subset of C_k. Figure 8 illustrates the *HD* algorithm for a 3×4 grid of processors. In this example, the *HD* algorithm executes the *CD* algorithm as if there were only 4 processors, where the 4 processors correspond to the 4 processor columns. That is, the database transactions are partitioned in 4 parts, and each one of these 4 hypothetical processors computes the local counts of all the candidate item-sets. Then the global counts can be computed by performing the global reduction operation discussed in Section 3.1. However, since each one of these hypothetical processors is made up of 3 processors, the computation

of local counts of the candidate item-sets in a hypothetical processor requires the computation of the counts of the candidate item-sets on the database transactions sitting on the 3 processors. This operation is performed by executing the *IDD* algorithm within each of 4 hypothetical processors. This is shown in the step 1 of Figure 8. Note that processors in the same row have exactly the same candidates, and candidate sets along the each column partition the total candidate set. At the end of this operation, each processor has complete count of its local candidates for all the transactions located in the processors of the same column (i.e., of a hypothetical processor). Now a reduction operation is performed along the rows such that all processors in each row have the sum of the counts for the candidates in the same row. At this point, the count associated with each candidate item-set corresponds to the entire database of transactions. Now each processor finds frequent item-sets by dropping all those candidate item-sets whose frequency is less than the threshold for minimum support. These candidate item-sets are shown as shaded in Figure 8(b). In the next step, each processor performs all-to-all broadcast operation along the columns of the processor mesh. At this point, all the processors have the frequent sets and are ready to proceed to the next pass.

The *HD* algorithm determines the configuration of the processor grid dynamically. In particular, the *HD* algorithm partitions the candidate set into a big enough section and assign a group of processors to each partition. Let m be a user specified threshold. If the total number of candidates M is less than m, then the *HD* algorithm makes G equal to 1, which means that the *CD* algorithm is run on all the processors. Otherwise G is set to $\lceil M/m \rceil$.

The *HD* algorithm inherits all the good features of the *IDD* algorithm. It also provides good load balance and enough computation work by maintaining minimum number of candidates per processor. At the same time, the amount of data movement in this algorithm has been cut down to $1/G$ of the *IDD*. A detailed parallel runtime analysis of HD is given in [13]. It shows that HD is scalable with respect to both number of transactions and number of candidates. The analysis also proves the necessary conditions under which HD can outperform CD. Detailed experimental results which compare CD, DD, IDD, and HD formulations of Apriori algorithm are given in [28]. HD is shown to be faster and more scalable as compared to the other algorithms.

– **HPA-ELD algorithm:** The paper [15] that proposed the HPA algorithm, proposes another algorithm called HPA-ELD (Hash-Partitioned Apriori with Extremely Large Itemsets Duplication). This algorithm reduces the communication required by HPA, by using partial replication of candidates. It first sorts the itemsets based on their frequency of appearance and replicates the most frequently occurring itemsets over all processors. For the replicated candidates, NPA (or CD) algorithm is used to collect global counts. For the rest, HPA algorithm is used. Because of the replication of most frequent itemsets, HPA-ELD is less sensitive to the data skew. Also, it also utilizes the local processor memory efficiently in case of relatively small size of candidate

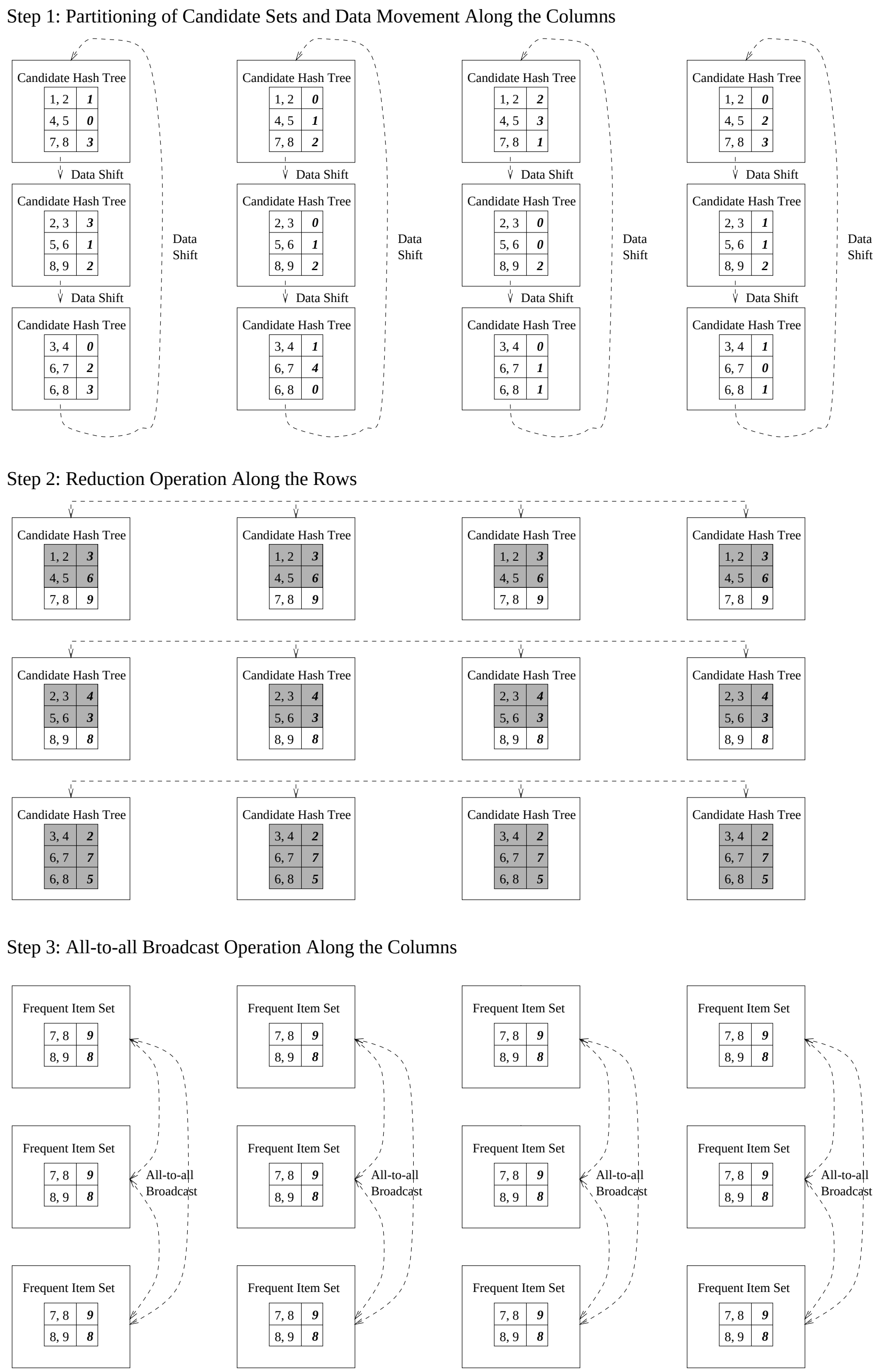


Fig. 8. Hybrid Distribution (HD) Algorithm in 3×4 Processor Mesh ($G = 3, P = 12$)

itemsets. This replication of highly frequent itemsets to all processors is similar in spirit to the HD algorithm. However, HD replicates some candidates on a small number of processors, instead of on all processors. According to the performance comparisons presented in [15], HPA-ELD performs better for the smaller support thresholds, whereas NPA performs better for large support thresholds. However, as with HPA, the performance of HPA-ELD is critically dependent on the hashing scheme, and the paper does not provide any theoretical results regarding the scalability of the algorithm.

3.2 Other Parallel Formulations

Along with the parallel formulations of level-wise algorithms, presented in previous subsections, many other schemes have been proposed in the literature so far[5, 14, 16, 17, 15]. This section reviews these formulations in a comparative manner.

Parallel formulation, DMA, designed specifically for distributed databases is described in [17]. It uses an idea of pruning based on local count. The founding principle of DMA is similar to that of PARTITION: a globally frequent itemset (when support is counted with respect to the entire database) has to be frequent in at least one of the processors (when support is counted with respect to the local database). DMA uses this principle to compute *heavy* itemsets at each site. These are the itemsets which are frequent locally as well as globally. The $k+1$-candidates are locally generated using the local heavy k-itemsets instead of using the globally frequent k-itemsets. Use of heavy itemsets can generate much smaller number of candidates overall, when compared to the CD algorithm which uses global frequent k-itemsets. Local counts for these candidates are measured by scanning the database once. The candidates which are not locally frequent are pruned away and the remaining candidates are communicated to all other processors. Each processor measures the local count for each candidate received from remote processors, and sends it back to the processor who requested it. Adding up local and remote counts, each processor determines which of the candidates are globally frequent and forms the local heavy set. Local heavy sets are exchanged by a broadcast operation to find global frequent sets. The algorithm proposes communication optimizations by assigning each candidate a host site for the purpose of collecting its remote counts. In this form, DMA is similar in nature to the DD algorithm where candidates as well as the data are distributed across processors.

It should be noted that DMA uses horizontal data layout similar to CD and DD, unlike the vertical data layout used in PARTITION. If implemented naively, DMA would need to two passes over database in each iteration over k, one for counting candidates generated from local heavy itemsets, and second for candidates received from remote processors. The paper identifies this and proposes an optimization for making only a single scan by generating all the candidates that would be generated at all remote sites, and collecting counts for these along with the locally generated candidates. This optimization brings the algorithm closer to the CD algorithm, except that the candidate set generated in DMA could be potentially much smaller than the one in CD (because of the use of heavy itemsets). In the performance results shown in the paper, DMA performs better than CD, mainly because of the reduction in the number of candidates generated. It should be noted that although DMA uses the same principle as PARTITION, its sensitivity to the problems of small partition size and data skew is less than PARTITION. This is because PARTITION, in an effort of reducing the database scans, generates *all* locally frequent itemsets in its first scan of the database. It does not have the flexibility of interleaving the global information with local information in every iteration over k. This causes it to generate many false positives which need to be counted in the second pass over the database.

The last set of algorithms that we will discuss here is the parallel formulations of the itemset-clustering based lattice traversal algorithms given in [7]. As described in section 2.2, these algorithms try to find potential maximal frequent itemsets by pruning the search space of itemsets. This pruning is achieved by finding clusters of related items, using either the equivalence class method or the hypergraph clique method. Each cluster corresponds to a potential maximal itemset. Such itemsets form disjoint sublattices of the entire itemset lattice. The idea behind the parallel formulations given in [16] is essentially to identify such sublattices and assign them to different processors so that the processing of each sublattice can be done entirely independently. The algorithms try to achieve load balance by estimating the work needed for each sublattice and determining the number of sublattices going to each processor. To achieve independent processing of each sublattice, the algorithms bring all the transaction data required for that sublattice to the processor assigned to process the sublattice. Remember that these algorithms use vertical data layout for efficient counting of candidates.

These parallel algorithms have the same advantages that are enjoyed by their serial counterparts, specifically those of doing at most two database scans and performing efficient counting by simple tid-list intersection. Along with these, the parallel formulations have the advantage of reducing communication overhead involved in communicating candidates or counts. But, these algorithms have limitations also. First, they have to pay the cost of replicating parts of the database across multiple processors. Second, the amount of concurrency that the algorithm can achieve depends entirely on the quality of clusters it can find, and on the transaction dataset. If the number of clusters is very few, then the algorithm may not fully utilize the total number of processors available, thus making it unscalable to larger number of processors. In the worst case, the algorithm may reduce to serial algorithm with a single processor working on the entire problem because of lack of multiple maximal potential itemsets. The hypergraph clique based clustering can be used avoid such worst case scenarios. But, clique based techniques tend to become expensive based on how dense the hypergraph gets, which in turn depends on the nature of transactions and the support threshold level. Another possibility where these algorithms can become expensive is when the number of clusters is such that the items appearing in different clusters have a large overlap. In such cases, the algorithm may end up replicating a large part of the database to all the processors. As an aside, the idea of itemset clustering using equivalence classes used in these algorithms is similar to the Candidate Distribution algorithm of[12], which assigns candidates to processors based on their equivalence classes.

4 Bringing in the Sequential Relationships

The data collected from scientific experiments, or monitoring of physical systems such as telecommunications networks, or from transactions at a supermarket, have inherent sequential nature to them. Sequential nature means that the events occurring in such data are related to each other by relationships of the form *before* (or *after*) and *together*. The concept of item-sets and association rules discussed so far takes into account only the *together* part of the relationship, the information provided by the *before/after* relationships is ignored. This information could be very valuable in finding more interesting patterns hidden in the data, which could be useful for many purposes such as prediction of events or identification of better sequential rules that characterize different parts of the data.

In this section, we discuss the concept of *sequential associations*, more commonly known as *sequential patterns*, and algorithms to discover them.

4.1 Generalized Sequential Associations: Definition

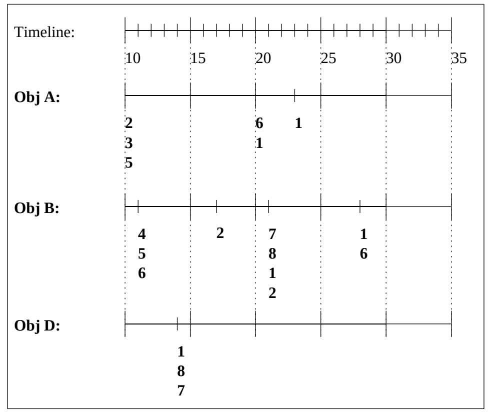
Sequential associations are defined in the context of an input sequence data characterized by three columns: *object*, *timestamp*, and *events*. Each row records occurrences of events on an object at a particular time. An example is shown in Figure 9(a). Alternative way to look at the input data is in terms of the time-line representations of all objects as illustrated in Figure 9(b). Note that the term *timestamp* is used here as a generic term to denote a measure of sequential (or temporal) dimension.

Various definitions of *object* and *events* can be used, depending on what kind of information one is looking for. For example, in one formulation, object can be a telecommunication switch, and event can be an alarm type occurring on the switch. With this, the sequences discovered will indicate interesting patterns of occurrences of alarm types occurring at a switch. In another formulation, object can be a *day*, and event can be a switch or a pair of switch and type of the alarm occurring on it. This will give interesting sequential relations between different switches or switch-alarm type pairs over a day.

Given this input data, the goal is to discover associations or patterns of the form given in Figure 10. A pattern is essentially a sequence of sets of events, which conform to the given *timing constraints*. As an example, the sequential pattern (A) (C,B) (D), encodes an *interesting* fact that event D occurs after an event-set (C,B), which in turn occurs after event A. The occurrences of events in a sequential pattern are governed by the following timing constraints:

Object	timestamp	events
A	10	2, 3, 5
A	20	6, 1
A	23	1
B	11	4, 5, 6
B	17	2
B	21	7, 8, 1, 2
B	28	1, 6
D	14	1, 8, 7

(a)



(b)

Fig. 9. Example Input Data: (a) Flat representation, (b) Timeline Representation

- **Maximum Span**(ms): The maximum allowed time difference between the latest and earliest occurrences of events in the *entire* sequence.
- **Event-set Window Size**(ws): The maximum allowed time difference between the latest and earliest occurrences of events in any *event-set*.
- **Maximum Gap**(xg): The maximum allowed time difference between the latest occurrence of an event in an event-set and the earliest occurrence of an event in its immediately preceding event-set.
- **Minimum Gap**(ng): The minimum required time difference between the earliest occurrence of an event in an event-set and the latest occurrence of an event in its immediately preceding event-set.

We assume the interestingness of a sequence to be defined based on how many times it occurs in the input data; i.e. its support. If the support is greater than a user-specified *support threshold*, then the sequence is called *frequent* or *interesting*. The the number of occurrences of a sequence can be computed in many ways, which are illustrated using the example shown in in Figure 11(a). The method COBJ counts at most one occurrence of a sequence for every object, as long as it is found within the given timing constraints. In the example, (1)(2) has two occurrences, one for each object. This method may not capture the sequences which are exhibited many times within a single object, which could really determine its interestingness. In the method CWIN, the support of a sequence is equal to the number of span-size windows it appears in. Each span-size window has a duration of ms, and consecutive windows have an overlap of $ms - 1$ units. Windows can span across a single object; i.e., no window can span across multiple objects. The support is added over all objects to get final support for a sequence. As shown in Figure 11(b), sequence (1)(2) has support of 3 for Object A, because it occurs in windows starting at time-points 0, 1, and 2. For object B, it occurs in 5 windows, hence the total support is 8. In other counting methods, instead of counting the span-windows, actual occurrences of a sequence are counted. Two options CDIST and CDIST_O are illustrated in Figure 11(c) and Figure 11(d), respectively. In CDIST, an event-timestamp pair is considered at most once in counting occurrences of a given sequence. So, there is only 1 occurrence of (1)(2) for Object A in the example, because there is no corresponding event 2's occurrence for event 1@2, 2@4 was used up in first occurrence. In CDIST_O, the occurrences are counted such that each new occurrence found has at least one different event-timestamp pair than previously found occurrences. So, (1)(2) has 3 occurrences for object B, and total of 5 occurrences, using this method.

The choice of which counting method to use is dependent on the problem and the judgment of the person using the discovery tool. For the purpose of our discussion in this paper, we will assume the method depicted in part (b), which counts the number of span-windows, because it is fairly general in the way it assigns interestingness to a sequence (especially when compared to method in part (a)).

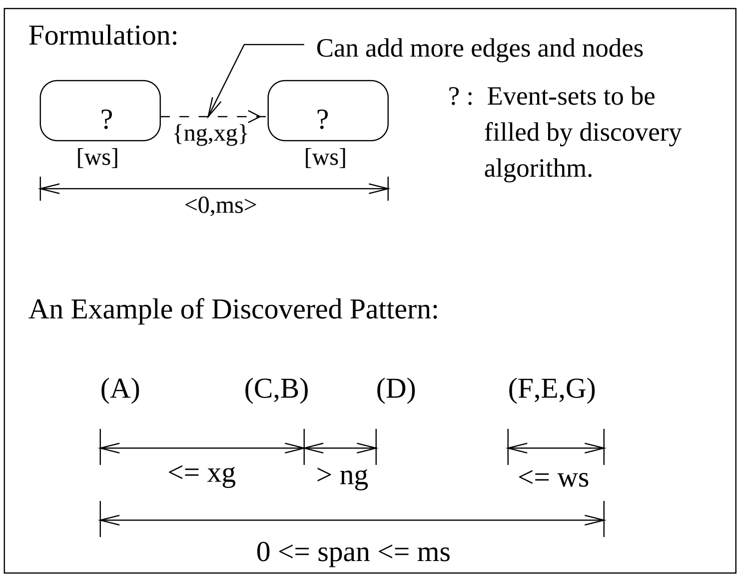


Fig. 10. Generalized Formulation of Sequential Patterns

The definition of sequential association presented above is a special case of the generalized universal sequential patterns described in [20]. It combines the notions of generalized sequential patterns (GSP) proposed in [21] and episodes proposed in [30]. These notions are actually the special cases of the generalized sequential associations presented above. If maximum span constraint is considered ineffective ($ms \rightarrow \infty$) and COBJ method is used for counting, then the formulation is identical to GSP. If constraints $ws < 0$, $xg \geq ms$, and $ng = 0$ are used along with the CWIN counting method, then the formulation is equiv-

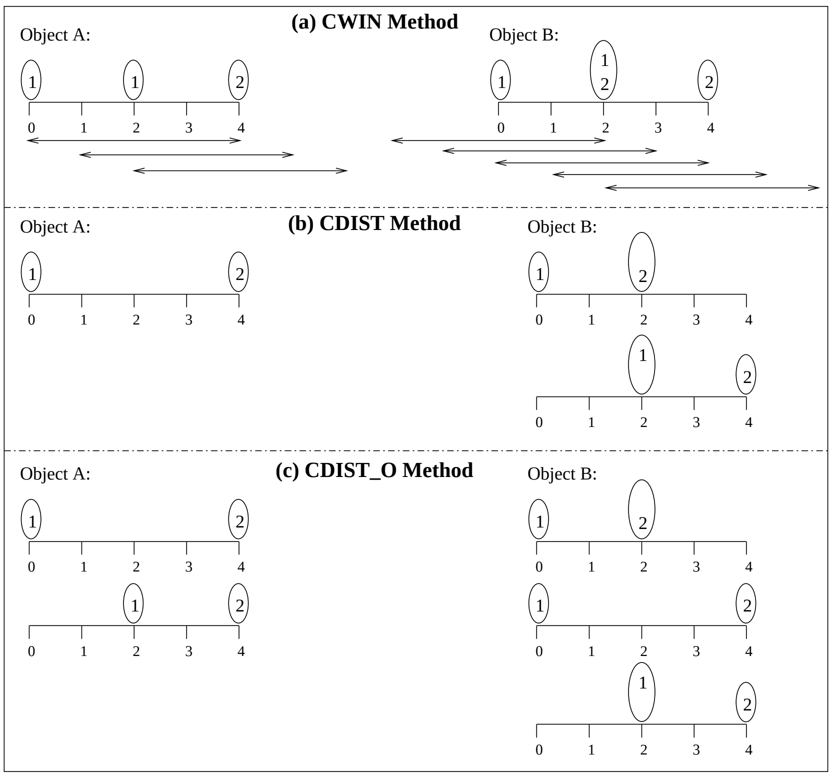


Fig. 11. Illustration of Methods of Counting Support

alent to the *serial episodes* of [30]. If constraints $ws == ms$, $xg \geq ms$, and $ng \geq ms$ are used along with the CWIN counting method, then the formulation is equivalent to the *parallel episodes* of [30]. In summary, the formulation of generalized sequential associations given above is fairly general for a wide variety of sequential data.

4.2 Serial algorithms for Sequential Associations

The complexity of discovering frequent sequences is much more than the complexity of mining non-sequential associations. To get an idea, the maximum number of sequences having k events is $O(m^k 2^{k-1})$, where m is the total number of events in the input data. Compare this to the $\binom{m}{k}$ possible item-sets of size k. Using the definition of interestingness of a sequence, and the timing constraints imposed on the events occurring in a sequence, many of these sequences can be pruned. But in order to contain the computational complexity, the search space needs to be traversed in a manner that searches only those sequences that would potentially satisfy both the support and timing constraints. The GSP algorithm given in [21] addresses this issue by building frequent sequences level-wise. Like apriori, it makes use of the monotonicity property of the support. The frequent sequences having $k-1$ events can be used to build a *candidate* sequence having k events, such that all its $(k-1)$-subsequences are frequent. The algorithm also takes into account the timing constraints relevant to the formulation of [21]. This algorithm has been modified in [31] to handle the generalized sequential associations described in section 4.1. The main modification is to take into account the multiple counting strategies which are driven primarily by the maximum span (ms) constraint. Especially when counting strategies other than COBJ are used, entire timeline of each object needs to be scanned to count all occurrences of every candidate. Data structures such as hash tree can be used to quickly find the candidates that may exist in a given timeline, but such structures will be helpful only for the *first* occurrence of a candidate. The rest of the occurrences need to be found by scanning the entire remaining timeline. A detailed description of how the algorithm works using hash tree structures is given in [31].

4.3 Parallel Formulation: Issues, Challenges, and Some Solutions

If the input sequence data has following features, then serial[1] algorithms briefly described in previous subsection face severe limitations.

- Enormity; i.e., large number of objects and/or large time-lines for many objects. Serial algorithms would take a very long time to in the counting phase for such datasets.

[1] The terms serial and sequential should not be confused. Traditionally, sequential and serial are both used to describe algorithms that would run on single processor machines. Here, we use the term serial to represent such algorithms, and reserve the term sequential to indicate the temporal or sequential nature of the input data

– High dimensionality; i.e., large number of events. The number of candidates generated for such datasets will be very large; hence, either they may not fit in the memory available for a single processor, or they would make the hash tree data structures act counter-productively if their size and structure is not optimally managed.

This motivates the need for parallel formulations. In this section, we will briefly discuss the issues and research challenges involved in developing effective parallel formulations of sequential pattern discovery algorithm.

The parallel formulation should be able to divide two entities among processors. One is the computational work and other is the memory requirement. These should be divided such that the time and memory limitations faced by serial algorithms could be minimized, and it should be possible to achieve this with as little overhead as possible. In parallel formulations, the overheads come mainly from load imbalance (causing idling of processors) and the communication required to co-ordinate the computations performed by different processors.

The computational load in sequential pattern discovery algorithm consists of candidate generation and counting of candidates. The memory requirements come from storing the input datasets and the candidates generated. Depending on how the candidates and object time-lines are distributed among processors, different parallel algorithms are possible.

In the following, we describe several parallel formulations given in [31] that take into account the generalized nature of sequential patterns. The intention is to bring out the challenges involved in designing effective parallel formulations. In the first category of algorithms, called EVE (event distribution), input data is distributed among processors and the candidate set is replicated on all the processors. The candidate generation phase is done serially on all the processors. Three different variations of EVE algorithm are discussed to cater to different scenarios emerging depending on the number of objects, the length of the time-lines in terms of the number of events happening on them, and the value of ms. The second category of algorithms, called EVECAN (event and candidate distribution), distributes events as well as candidates among processors, to overcome some of the problems that EVE might face.

EVE-S: Simple Event Distribution Algorithm For shorter time-lines and relatively large number of objects, the input data is distributed such that the total number of event points is as evenly distributed as possible within the constraint that a processor gets the entire timeline of every object allocated to it. It is embarrassingly parallel as far as counting phase is concerned, except for the final communication operation required to accumulate the candidate counts. EVE-S is illustrated in Figure 12. This algorithm is essentially an extension of the CD algorithm for discovering non-sequential associations, except that the transactions are replaced with more generic objects[2]. A similar algorithm called NPSPM (non-partitioned sequential pattern mining) is proposed by [15]. They

[2] objects can be thought of as a time-ordered collection of transactions

assume the restricted GSP[21] formulation of sequential patterns. Also, they cater only to the supermarket transaction scenario, which indeed is fitting for the EVE-S algorithm also, because usually object timelines contain small number of transactions, each in turn consisting of small number of events (which are items in this case).

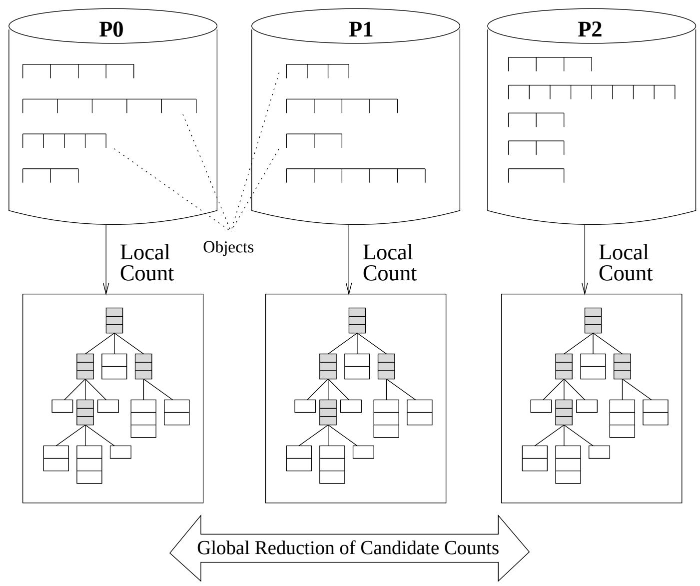


Fig. 12. Illustration of EVE-S algorithm.

EVE-R: Event distribution with partial data replication This formulation is designed for the scenario in which there are relatively small number of objects (less than the number of processors), each object has a large timeline, and the span value (ms) is relatively small. The input data is distributed as follows. The timeline for each object is split across different processors such that the total number of events assigned to different processors is similar. Note that the sequence occurrences are computed in span-size windows. We assume that the span value is small such that no span window spans across more than two processors. But, still each processor will have some span-windows that do not have sufficient data to declare the occurrence of an sequence in them. This is resolved in EVE-R by gathering such missing data from neighboring processors. Each processor gathers data that is required to process the last span-window beginning on that processor. This is illustrated in Figure 13. Since we assume that span-windows do not straddle more than two processors, just the neighbor-to-neighbor communication is sufficient. Once every span-window is complete on all processors, each processor processes only those span-windows which begin at the events points originally assigned to it. For example, processor P0 processes windows that begin at time instances 0, 1, 2, and 3, whereas processor P1 will process windows that begin at 4, 5, 6, and 7. By distributing the event points equitably, load balance can be achieved. As in EVE-S algorithm, the occurrences are collected by a global communication (reduction) operation, in the end.

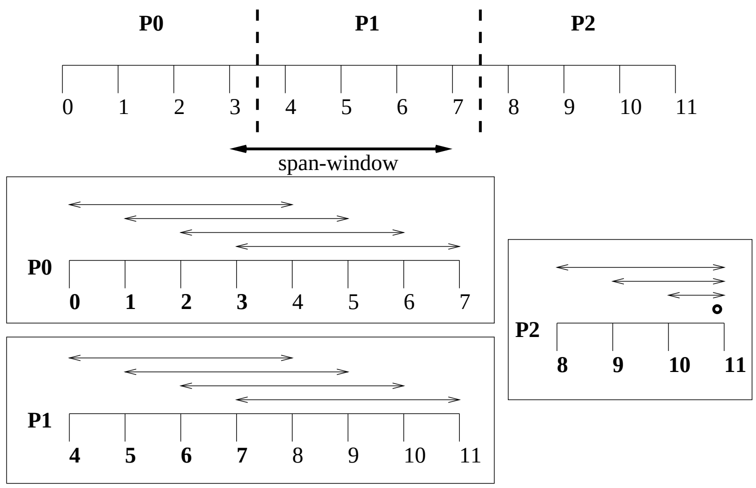


Fig. 13. Illustration of EVE-R algorithm.

EVE-C: Complex Event Distribution Algorithm This formulation depicts the most complex scenario as far as distribution of the counting workload is concerned. This happens when there are small number of objects, each object has a large timeline, and the span value is large such that after splitting the object time-lines across processors, the span-windows straddle more than two processors. There are two ways to handle this.

One way is to replicate the data across processors such that no processor has any incomplete or partial span-window. This is the same idea used in EVE-R, what makes it different is the fact that the amount of replication can become very large in this case. So, if processors do not have enough disk space to hold the entire replicated information, this approach may not be feasible. Even when there is enough disk space available on each processor, the replication of data may result in a lot of replication of work. The details are given in [31], but to summarize, when data is replicated, there is trade-off between the approach of replicating the work with no communication cost (except for the data replication cost), and the approach of avoiding work replication by paying the extra cost of communicating the candidate occurrences.

The second way to handle this is not to replicate the data. Now, two kinds of situations need to be handled. In first situation, those occurrences that are found completely on a single processor might contribute to span-windows that begin on other processors. Care should be taken to avoid the double counting, which as shown in [31] requires communication of ranges of occurrences of candidates between processors. Second situation is when some occurrences cannot be declared to occur in some span-windows because there may not be sufficient data available on a single processor. This scenario actually gives rise to the most complex method of parallelizing the counting process. The details are given in [31], but the key idea is that only partial occurrences of candidates can be found by each processor. This partial work needs to be communicated to other processors to search for complete occurrences. First issue is amount of concurrency that can be achieved in this process. This can be handled by breaking down the granularity of computation and doing asynchronous communications. The second and more serious issue comes from the nature sequential association discovery problem, in which each span-size window has a potential to support exponential number of sequences. Hence, the amount of partial work that needs to be transferred can quickly become large. In summary, this approach of avoiding replication of data can become very expensive.

Thus depending on the scenario, there is a trade-off between the cost of replicating and storing the data and the cost of communicating large amount partial work among processors. A detailed discussion is given in [31].

Event and Candidate Distribution (EVECAN) Algorithm In the set of EVE algorithms described above, it is assumed that the candidates are replicated over all the processors. This may not be desirable when the number of candidates is very large, and for the complexity of sequential patterns, such scenarios are not uncommon. Large number of candidates results in two things. The set of candidates may not fit in the memory of a processor, in which case they need to be counted in parts. This involves multiple I/O passes over the disk for counting the candidates. Secondly, EVE algorithm builds candidates serially on all processors, thus losing out on extracting the possible concurrency. The amount of time spent in generating the large number of candidates can be significantly large.

These issues are addressed in the second formulation, called EVECAN (event and candidate distribution) [31]. In this algorithm, the input data is partitioned similar to EVE. But, now the candidates are also distributed. They are stored in a distributed hash table. The hashing criterion is designed to maintain equal number of candidates on all processors. One simple hash function can be based on the lexicographical ordering of candidates and splitting them among processors such that all candidates assigned to one processor have a common prefix sequence. The non-local candidates required for the join-and-prune phase are obtained using the scalable communication structure of the parallel hashing paradigm introduced in [32]. Now since all the processors must count all the candidates, there are two options. In the first option, the candidates are kept stationary at processors and a local hash tree access structure is built for these candidates. The input data is circulated among processors in a fashion similar to that of the round-robin scheme proposed for IDD algorithm of [28]. But this option may work only when the span value is small, in which case we will circulate the span-windows. For large span-values, it could be very expensive to send all the span-windows to all the processors. In such cases, second option can be used, which is to move around the candidates in a round robin fashion. In both the options, a hash function is used to do a relatively quick search of whether a span-window can contain the candidates stored at that processor. Figure 14 pictorially depicts the EVECAN algorithm.

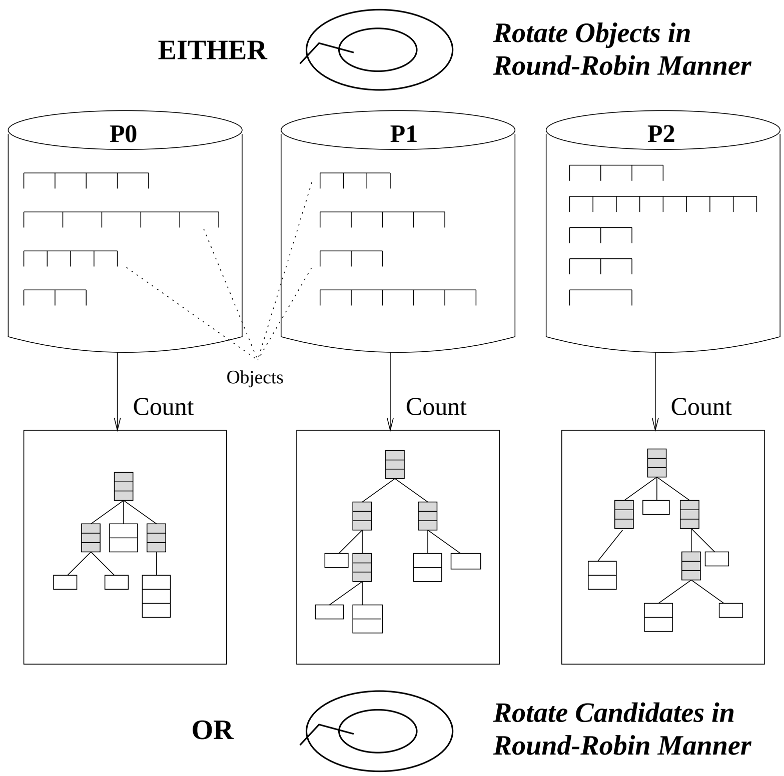


Fig. 14. Illustration of EVECAN algorithm for parallel discovery of generalized sequential associations.

Another set of parallel algorithms SPSPM (simple partitioned sequential pattern mining) and HPSPM (hash partitioned sequential pattern mining) are given in [33]. These are also based on distribution of objects as well as candidates. However, these algorithms assume the sequential pattern format given in [21]; hence, their algorithms do not have notion of span (ms), and they count only one occurrences of a sequence in a given object's timeline (COBJ counting method). Also they assume a market transaction type of dataset, in which the object time-lines are usually very short. SPSPM algorithm distributes the candidates in a simple round-robin manner, whereas HPSPM distributes candidates in a more intelligent manner using hash functions. These are straight-forward extensions of the SPA and HPA algorithms [15] for parallel discovery of non-sequential associations. The counting in SPSPM is performed in a way similar to the DD algorithm for non-sequential associations, where every object's timeline is sent to every processor. HPSPM, in k^{th} iteration, generates all k-sequences present in each object's timeline and hashes them using the same hash function as was used for hashing the candidates to distribute them among processors. Each sequence is sent to the processor it hashes to, and is used to search for the list of candidates stored there. The HPSPM algorithm is shown to perform better than the rest two, but it also faces severe limitations when the object time-lines are very large, and when it is extended to use the counting method used in the generalized sequential pattern formulation. These are precisely the issues that the EVE and EVECAN formulations take into account.

5 Summary

In this chapter, we presented a evolutionary and comparative review of many existing algorithms for solving a very popular and important problem of mining associations from data. We considered the traditional non-sequential associations which originated from the transaction or market basket kind of data as well as the more generalized sequential association formulation which is useful to wider variety of datasets in real world. The chapter mainly elaborates on various design issues involved in parallel formulations of association discovery algorithms, and how existing parallel algorithms map to only a few categories of formulations. In the process, a comprehensive survey of many existing serial algorithms is also given. Although many parallel (and serial) algorithms exist today, no single algorithm is superior to all the rest, and the research in the discovery of associations remains active. Overall, this chapter is intended to serve as a comprehensive account of existing parallel methods of mining non-sequential as well as sequential associations with respect to the design issues and different parallelization strategies.

References

1. Chen, M., Han, J., Yu, P.: Data mining: An overview from database perspective. IEEE Transactions on Knowledge and Data Eng. **8** (1996) 866–883
2. Agrawal, R., Imielinski, T., Swami, A.: Mining association rules between sets of items in large databases. In: Proc. of 1993 ACM-SIGMOD Int. Conf. on Management of Data, Washington, D.C. (1993)
3. Agrawal, R., Srikant, R.: Fast algorithms for mining association rules. In: Proc. of the 20th VLDB Conference, Santiago, Chile (1994) 487–499
4. Park, J., Chen, M., Yu, P.: An effective hash-based algorithm for mining association rules. In: Proc. of 1995 ACM-SIGMOD Int. Conf. on Management of Data. (1995)

5. Savasere, A., Omiecinski, E., Navathe, S.: An efficient algorithm for mining association rules in large databases. In: Proc. of the 21st VLDB Conference, Zurich, Switzerland (1995) 432–443
6. Toivonen, H.: Sampling large databases for association rules. In: Proc. of the 22nd VLDB Conference. (1996)
7. Zaki, M.J., Parthasarathy, S., Ogihara, M., Li, W.: New algorithms for fast discovery of association rules. In: Proc. of the Third Int'l Conference on Knowledge Discovery and Data Mining. (1997)
8. Brin, S., Motwani, R., Ullman, J.D., Tsur, S.: Dynamic itemset counting and implication rules for market basket data. In: Proc. of 1997 ACM-SIGMOD Int. Conf. on Management of Data, Tucson, Arizona (1997) 255–264
9. Agarwal, R.C., Aggarwal, C., Prasad, V.V.V.: A tree projection algorithm for generation of frequent itemsets. Journal of Parallel and Distributed Computing (Special Issue on High Performance Data Mining) (2000)
10. Agarwal, R.C., Aggarwal, C., Prasad, V.V.V.: Depth-first generation of large itemsets for association rules. Technical Report RC-21538, IBM Research Division (1999)
11. Han, J., Pei, J., Yin, Y.: Mining frequent patterns without candidate generation. Technical Report CMPT99-12, School of Computing Science, Simon Fraser University (1999)
12. Agrawal, R., Shafer, J.: Parallel mining of association rules: Design, implementation and experience. Technical Report RJ10004, IBM Research Division, Almaden Research Center (1996)
13. Han, E., Karypis, G., Kumar, V.: Scalable parallel data mining for association rules. IEEE Transactions on Knowledge and Data Eng. (1999)
14. Park, J., Chen, M., Yu, P.: Efficient parallel data mining for association rules. In: Proceedings of the 4th Int'l Conf. on Information and Knowledge Management. (1995)
15. Shintani, T., Kitsuregawa, M.: Hash based parallel algorithms for mining association rules. In: Proc. of the Conference on Parallel and Distributed Information Systems. (1996)
16. Zaki, M.J., Parthasarathy, S., Ogihara, M., Li, W.: New parallel algorithms for fast discovery of association rules. Data Mining and Knowledge Discovery: An International Journal **1** (1997)
17. Cheung, D., Ng, V., Fu, A., Fu, Y.: Efficient mining of association rules in distributed databases. IEEE Transactions on Knowledge and Data Eng. **8** (1996) 911–922
18. Agrawal, R., Srikant, R.: Mining sequential patterns. In: Proc. of the Int'l Conference on Data Engineering (ICDE), Taipei, Taiwan (1996)
19. Mannila, H., Toivonen, H., Verkamo, A.I.: Discovering frequent episodes in sequences. In: Proc. of the First Int'l Conference on Knowledge Discovery and Data Mining, Montreal, Quebec (1995) 210–215
20. Joshi, M.V., Karypis, G., Kumar, V.: Universal formulation of sequential patterns. Technical Report TR 99-021, Department of Computer Science, University of Minnesota, Minneapolis (1999)
21. Srikant, R., Agrawal, R.: Mining sequential patterns: Generalizations and performance improvements. In: Proc. of the Fifth Int'l Conference on Extending Database Technology, Avignon, France (1996)
22. Bettini, C., Wang, X.S., Jajodia, S.: Testing complex temporal relationships involving multiple granularities and its application to data mining. In: Proc. of ACM PODS'96, Montreal (1996) 68–78

23. Houtsma, M.A.W., Swami, A.N.: Set-oriented mining for association rules in relational databases. In: Proc. of the 11th Int'l Conf. on Data Eng., Taipei, Taiwan (1995) 25–33
24. Amir, A., Feldman, R., Kashi, R.: A new and versatile method for association generation. In Komorowski, H.J., Zytkow, J.M., eds.: Proceedings of Principles of Data Mining and Knowledge Discovery, First European Symposium (PKDD'97). Lecture Notes in Computer Science. Volume 1263. Springer, Trondheim, Norway (1997) 221–231
25. Sedgewick, R.: Algorithms. Second edn. Addison-Wesley (1988)
26. Agrawal, R., Shafer, J.: Parallel mining of association rules. IEEE Transactions on Knowledge and Data Eng. **8** (1996) 962–969
27. Kumar, V., Grama, A., Gupta, A., Karypis, G.: Introduction to Parallel Computing: Algorithm Design and Analysis. Benjamin Cummings/ Addison Wesley, Redwod City (1994)
28. Han, E., Karypis, G., Kumar, V.: Scalable parallel data mining for association rules. In: Proc. of 1997 ACM-SIGMOD Int. Conf. on Management of Data, Tucson, Arizona (1997)
29. Papadimitriou, C.H., Steiglitz, K.: Combinatorial Optimization: Algorithms and Complexity. Prentice-Hall, Englewood Cliffs, NJ (1982)
30. Mannila, H., Toivonen, H., Verkamo, A.I.: Discovery of frequent episodes in event sequences. Technical Report C-1997-15, Department of Computer Science, University of Helsinki, Finland (1997)
31. Joshi, M.V., Karypis, G., Kumar, V.: Parallel algorithms for mining sequential associations: Issues and challenges. Technical Report under preparation, Department of Computer Science, University of Minnesota, Minneapolis (1999)
32. Joshi, M.V., Karypis, G., Kumar, V.: ScalParC: A new scalable and efficient parallel classification algorithm for mining large datasets. In: Proc. of the 12th International Parallel Processing Symposium, Orlando, Florida (1998)
33. Shintani, T., Kitsuregawa, M.: Mining algorithms for sequential patterns in parallel: Hash based approach. In: Research and Development in Knowledge Discovery and Data Mining: Second Pacific-Asia Conference (PAKDD'98), Melbourne, Australia (1998) 283–294