Mohammed J. Zaki
Rensselaer Polytechnic Institute

The author surveys the state of the art in parallel and distributed association-rule-mining algorithms and uncovers the field's challenges and open research problems. This survey can serve as a reference for both researchers and practitioners.

Parallel and Distributed Association Mining: A Survey

Since its inception, association rule mining has become one of the core data-mining tasks and has attracted tremendous interest among researchers and practitioners.[1] ARM is undirected or unsupervised data mining over variable-length data, and it produces clear, understandable results. It has an elegantly simple problem statement: to find the set of all subsets of items or attributes that frequently occur in many database records or transactions, and additionally, to extract rules on how a subset of items influences the presence of another subset.

The prototypical application of ARM is *market-basket analysis*, where the items represent products, and the records represent point-of-sales data at large grocery stores or department stores. An example rule might be, "90% of customers buying product A also buy product B." Other application domains for ARM include customer segmentation, catalog design, store layout, and telecommunication alarm prediction.

Although ARM has a simple statement, it is computationally and I/O intensive. Because data is increasing in terms of both the dimensions (number of items) and size (number of transactions), one of the main attributes needed in an ARM algorithm is scalability: the ability to handle massive data stores. Sequential algorithms cannot provide scalability, in terms of the data dimension, size, or runtime performance, for such large databases. Therefore, we must rely on high-performance parallel and distributed computing. This article surveys the different parallel and distributed ARM algorithms that have been proposed on various hardware platforms. Because of the astonishing amount of research in this area, I present the state of the art in ARM and identify the current open problems.

Problem statement and mining complexity

Association mining works as follows. Let I be a set of items and D a database of transactions, where each transaction has a unique identifier (*tid*) and contains a set of items called an *itemset*. An itemset with k items is called a k-itemset. The *support* of an itemset X, denoted $\sigma(X)$, is the number of transactions in which that itemset occurs as a subset. A k-subset is a k-length subset of an itemset. An itemset is frequent or large if its support is more than a user-specified *minimum support* (*min_sup*) value. F_k is the set of frequent k-itemsets. A frequent itemset is maximal if it is not a subset of any other frequent itemset.

1092-3063/99/$10.00 © 1999 IEEE

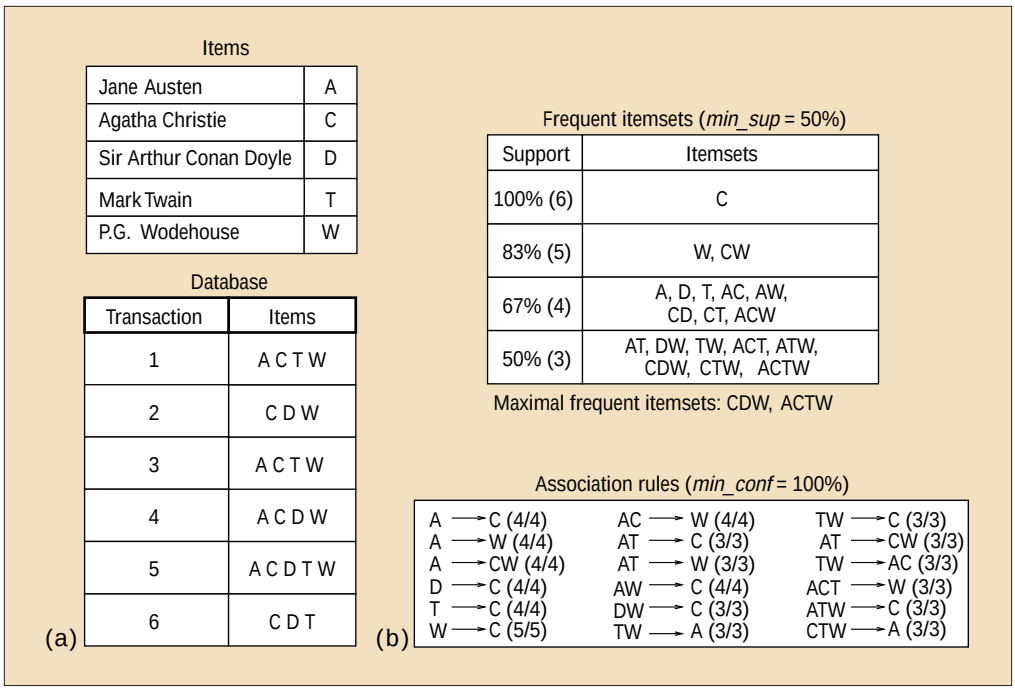

Figure 1. (a) Bookstore database; (b) frequent itemsets and strong rules.

An *association rule* is an expression $A \Rightarrow B$, where A and B are itemsets. The rule's support is the joint probability of a transaction containing both A and B, and is given as $\sigma(A \cup B)$. The confidence of the rule is the conditional probability that a transaction contains B, given that it contains A and is given as $\sigma(A \cup B)/\sigma(A)$. A rule is frequent if its support is greater than *min_sup* and strong if its confidence is more than a user-specified minimum confidence (*min_conf*).

Data mining involves generating all association rules in the database that have a support greater than *min_sup* (the rules are frequent) and that have a confidence greater than *min_conf* (the rules are strong). This task has two steps:

1. Find all frequent itemsets having minimum support. The search space for enumeration of all frequent itemsets is 2^m, which is exponential in m, the number of items. However, if we assume the transaction length has a bound, we can show that ARM is essentially linear in database size.
2. Generate strong rules having minimum confidence, from the frequent itemsets. We generate and test the confidence of all rules of the form $X \setminus Y \Rightarrow Y$, where $Y \subset X$ and X is frequent. Because we must consider each subset of X as the consequent, the rule-generation step's complexity is $O(r \cdot 2^l)$, where r is the number of frequent itemsets, and l is the longest frequent itemset.

Consider the example bookstore-sales database shown in Figure 1. There are five different items (names of authors the bookstore carries), $I = \{A, C, D, T, W\}$. The database comprises six customers who bought books by these authors. Figure 1 shows all the frequent itemsets contained in at least three customer transactions (*min_sup* = 50%). The figure also shows the set of all association rules with *min_conf* = 100%.

Sequential ARM algorithms

All parallel ARM algorithms examined in this article are based on their sequential counterparts. The design space for the sequential methods is composed of the following characteristics.

Bottom-up vs. hybrid search

The main observation in ARM is that the subset relation $\subseteq$ defines a partial order (in fact, a lattice) on the set of itemsets. The second observation is that the subset relation $\subseteq$ is monotonic with respect to the frequency $\sigma(\alpha)$. In other words, if β is a frequent itemset, then all subsets $\alpha \leq \beta$ are also frequent.

ARM algorithms differ in the manner in with they search the itemset lattice spanned by the subset relation. Most approaches use a level-wise or bottom-up search of the lattice to enumerate the frequent itemsets. If long frequent itemsets are expected, a pure top-down approach might be preferred. Some have proposed a hybrid search, which combines top-down and bottom-up approaches.

Complete vs. heuristic candidate generation

ARM algorithms can differ in the way they generate new candidates. A complete search, the dominant approach, guarantees that we can generate and test all frequent subsets. Here, complete doesn't mean exhaustive; we can use pruning to eliminate useless branches in the search space. Heuristic generation sacrifices completeness for the sake of speed. At each step, it only examines a limited number of "good" branches. Random search to locate the maximal frequent itemsets is also possible. Methods that can be used here include genetic algorithms and simulated annealing. Because of a strong emphasis on completeness, ARM literature has not given much attention to the last two methods.

All vs. maximal frequent itemset enumeration

ARM algorithms differ depending on whether they generate all frequent subsets or only the maximal ones. Identifying the maximal itemsets is the core task, because an additional database scan can generate all other subsets. Nevertheless, the majority of algorithms list all frequent itemsets.

Horizontal vs. vertical data layout

Most ARM algorithms assume a horizontal database layout, which stores each customer's *tid* along with the items contained in the transaction. Some methods also use a vertical database layout, associ-

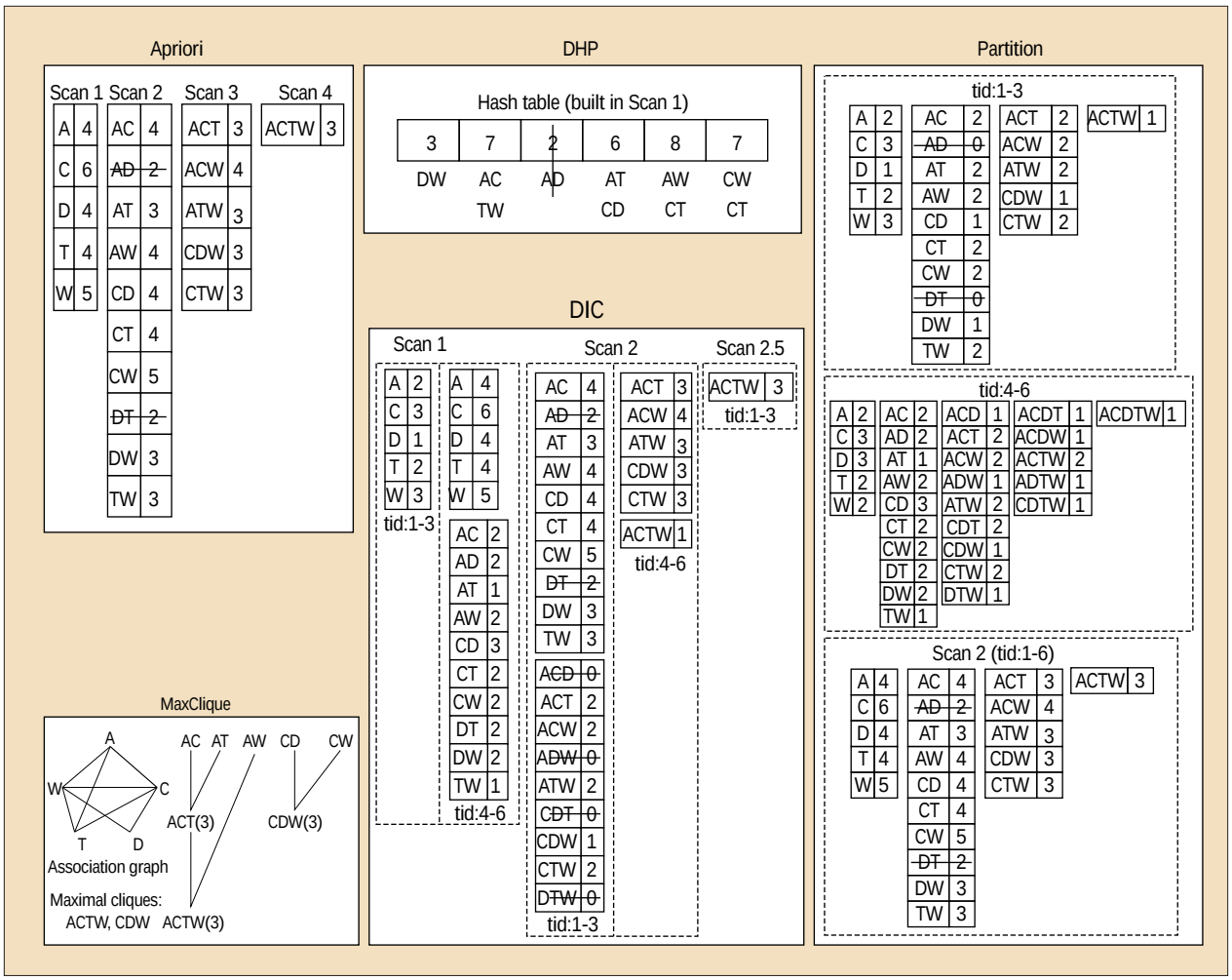

Figure 2. Sequential ARM algorithms. Apriori requires four scans for the example database. DHP builds a hash table (size = 6) during the first scan; we can immediately discard itemset *AD* because it cannot be frequent. In general, the hash table might discard many other 2-itemsets. DIC is shown with two database partitions (D1: tid 1–3 and D2: tid 4–6). DIC starts by counting the support of single items in D1, and generates new 2-itemsets before processing D2. While scanning D2, it counts both the single items and the new 2-itemsets, which completes scan 1. DIC generates new 3-itemsets from the 2-itemsets of D2, and then counts all current itemsets while scanning D1, and so on. In total, DIC requires 2.5 database scans. Partition (with two parts, D1 and D2) gathers locally frequent itemsets in each part, merges them, and then acquires their global support. MaxClique first builds an association graph from the frequent 2-itemsets and finds the maximal cliques in the graph (ACTW, CDW). These are the two equivalence classes, which MaxClique processes using hybrid search. Within a class, MaxClique keeps on extending an itemset until an infrequent itemset is found or no other extension is possible. MaxClique requires only three intersections to determine the maximal frequent itemsets.

ating with each item X its *tidlist*, which is a list of all *tids* containing the item.

While reading the following algorithm descriptions, refer to Figure 2, which shows how each method works on the example database from Figure 1.

Apriori

The Apriori algorithm by Rakesh Agrawal and colleagues has emerged as one of the best ARM algorithms.[1] It also serves as the base algorithm for most parallel algorithms. Apriori uses a complete, bottom-up search with a horizontal layout and enumerates all frequent itemsets. An iterative algorithm, Apriori counts itemsets of a specific length in a given database pass. The process starts by scanning all transactions in the database and computing the frequent items. Next, a set of potentially frequent *candidate* 2-itemsets is formed from the frequent items. Another database scan obtains their supports. The frequent 2-itemsets are retained for the next pass, and the process is repeated until all frequent itemsets have been enumerated. The algorithm has three main steps:

1. Generate candidates of length k from the frequent $(k-1)$ length itemsets, by a self-join on F_{k-1}. For example, for $F_2 = \{AC, AT, AW, CD, CT, CW, DW, TW\}$, we get $C_3 = \{ACT, ACW, ATW, CDT, CDW, CTW\}$.
2. Prune any candidate that has at least one infrequent subset. For example, *CDT* will be pruned because *DT* is not frequent.
3. Scan all transactions to obtain candidate supports. Apriori stores the candidates in a hash tree for fast support counting. In a hash tree, itemsets are stored in the leaves; internal nodes contain hash tables (hashed by items) to direct the search for a candidate.

Dynamic Hashing and Pruning

The DHP algorithm proposed by Jong Soo Park and colleagues extends the Apriori approach by using a hash table to precompute approximate support of 2-itemsets during the first iteration.[2] The second iteration need count only those candidates falling in hash cells with minimum support. This hash-table technique can successfully remove many candidate pairs that would eventually have become infrequent.

Partition

Ashok Savasere and others proposed the two-pass Partition algorithm,[3] which logically divides the horizontal database into nonoverlapping partitions. Each partition is read, and vertical tidlists (lists of all *tids* where the item appears) are formed for each item. Partition then generates all locally frequent itemsets through tidlist intersections. Locally frequent itemsets from all partitions merge to form a global candidate set. Partition then makes a second pass through all the partitions and obtains all candidates' global counts through tidlist intersections.

SEAR and Spear

Andreas Mueller's Sequential Efficient Association Rules algorithm is identical to Apriori, except that SEAR stores candidates in a prefix tree instead of a hash tree.[4] In a prefix tree (also called a trie), each edge is labeled by items; common prefixes are represented by tree branches, and the unique suffixes are stored at the leaves. Also, SEAR uses a pass-bundling optimization, where it generates candidates for multiple passes if the candidates will fit in memory.

The Spear algorithm is similar to SEAR, but it uses the Partition technique; it is the non-tidlist version of Partition. Spear uses the horizontal data format, but makes two scans: first it gathers potentially frequent itemsets, then it obtains their global support.

Mueller's objectives were to evaluate the intrinsic benefits of partitioning irrespective of the data format used. He concluded that partitioning did not help, because of high overhead of processing multiple partitions and because of the many locally frequent but globally infrequent itemsets found by partitioning. SEAR was the winner, because it also performed pass-bundling.

Dynamic Itemset Counting

The DIC algorithm proposed by Sergey Brin and others is a generalization of Apriori.[5] The database is divided into p equal-sized partitions so that each partition fits in memory. For partition 1, DIC gathers the supports of single items. Items found to be locally frequent (only in this partition) generate candidate 2-itemsets. Then DIC reads partition 2 and obtains supports for all current candidates—that is, the single items and the candidate 2-itemsets. This process repeats for the remaining partitions. DIC starts counting candidate k-itemsets while processing partition k in the first database scan. After the last partition p has been processed, the processing wraps around to partition 1 again. A candidate's global support is known once the processing wraps around the database and reaches the partition where it was first generated.

DIC is effective in reducing the number of database scans if most partitions are homogeneous (have similar frequent itemset distributions). If data is not homogeneous, DIC might generate many false positives (itemsets that are locally frequent but not globally frequent) and scan the database more than Apriori does. DIC proposes a random partitioning technique to reduce the data–partition skew.

Eclat, MaxEclat, Clique, and MaxClique

A completely different design characterizes the equivalence class-based algorithms proposed by my colleagues and me.[6] The simplest is Eclat; the best is MaxClique. These methods use a vertical database format, complete search, and a mix of bottom-up and hybrid search, and they generate a mix of maximal and nonmaximal frequent itemsets.

The main advantage of using a vertical format is that we can determine the support of any k-itemset by simply intersecting the tidlists of the lexicographically first two $(k-1)$-length subsets that share a common prefix (the generating itemsets). These methods break the large search space into small, independent, manageable chunks. These chunks can be processed in memory through prefix- or clique-based equivalence classes; the clique-based approach produces much smaller classes. Each class is independent in that it has complete information for generating all frequent itemsets that share the same prefix.

Among the four algorithms proposed, Eclat uses prefix-based classes and bottom-up search, MaxEclat uses prefix-based classes and hybrid search, Clique uses clique-based classes and bottom-up search, and MaxClique uses clique-based classes and hybrid search. The best approach was MaxClique, which outperformed Apriori and Partition by more than an order of magnitude and Eclat by a factor of 2 or more.

Table 1 presents a summary of the major differences among all the algorithms reviewed thus far.

Parallel ARM algorithms

Researchers expect parallelism to relieve current ARM methods from the sequential bottleneck, providing scalability to massive data sets and improving response time. Achieving good performance on today's multiprocessor systems is not trivial. The main challenges include synchronization and communication minimization, workload balancing, finding good data layout and data decomposition, and disk I/O minimization (which is especially important for ARM). The parallel design space spans three main components: the hardware platform, the type of parallelism, and the load-balancing strategy

DISTRIBUTED VS. SHARED MEMORY SYSTEMS

Two dominant approaches for using multiple processors have emerged: distributed memory (where each processor has a private memory) and shared memory (where all processors access common memory). A shared-memory (SMP) architecture has many desirable properties. Each processor has direct and equal access to all the system's memory. Parallel programs are easy to implement on such a system.

Table 1. Algorithm characteristics. K denotes the size of the longest frequent itemset. C_2 array optimization uses a 2D array to count candidate 2-itemsets rather than using hash trees or prefix trees.

Algorithm	Database layout	Data structure	Search	Enumeration	Optimizations	Number of database scans
Apriori	Horizontal	Hash tree	Bottom-up	All	C_2 array	K
DHP	Horizontal	Hash tree	Bottom-up	All	C_2 array and hash table	K
Partition	Vertical	None	Bottom-up	All	C_2 array and partitioning	2
SEAR	Horizontal	Prefix tree	Bottom-up	All	Pass-bundling	K
Spear	Horizontal	Prefix tree	Bottom-up	All	Partitioning	2
DIC	Horizontal	Prefix tree	Bottom-up	All	Count multiple lengths per scan	$\leq K$
Eclat	Vertical	None	Bottom-up	All	C_2 array and prefix classes	≥ 3
MaxEclat	Vertical	None	Hybrid	Maximal and nonmaximal	C_2 array and prefix classes	≥ 3
Clique	Vertical	None	Bottom-up	All	C_2 array and clique classes	≥ 3
MaxClique	Vertical	None	Hybrid	Maximal and nonmaximal	C_2 array and clique classes	≥ 3

A different approach to multiprocessing is to build a system from many units, each containing a processor and memory. In a distributed-memory (DMM) architecture, each processor has its own local memory, which only that processor can access directly. For a processor to access data in the local memory of another processor, message passing must send a copy of the desired data elements from one processor to the other. Although a shared-memory architecture offers programming simplicity, a common bus's finite bandwidth can limit scalability. A distributed-memory, message-passing architecture cures the scalability problem by eliminating the bus, but at the expense of programming simplicity.

A third, very popular, paradigm combines the best of the distributed- and shared-memory approaches. Included in this paradigm are hardware- or software-distributed shared-memory systems. These systems distribute the physical memory among the nodes but provide a shared global address space on each processor. The hardware or software ensures cache coherence; so locally cached data always reflects any processor's latest modification. Clusters of SMP workstations (Clumps) are also part of this mixed paradigm. Clumps necessitate a hierarchical parallelism approach, with SMP primitives used in a node and message passing used among the SMP nodes.

The performance-optimization objectives for distributed-memory machines versus shared-memory systems depend on the underlying architecture. In DMMs, synchronization is implicit in message passing, so the goal becomes communication optimization. For SMPs, synchronization stems from locks and barriers, and the goal is to minimize these points. Data decomposition is very important for distributed memory, but not for shared memory. While parallel I/O comes free in DMMs, it can be problematic for SMP machines, which typically serialize I/O. The main challenge for obtaining good performance on DMMs is finding a good data decomposition among the nodes and minimizing communication.

For SMPs, the objective is to achieve good data locality. This means we must maximize access to local cache and avoid or reduce false sharing. That is, we need to minimize the Ping-Pong effect, where multiple processors might be trying to modify different variables that coincidentally reside on the same cache line. For today's nonuniform memory access (NUMA) hybrid and hierarchical machines (clusters of SMPs), the optimization parameters draw from both the DMM and SMP paradigms.

Data vs. task parallelism

Task and data parallelism are the two main paradigms for exploiting algorithm parallelism. For ARM, data parallelism corresponds to the case where the database is partitioned among P processors—logically partitioned for SMPs, physically for DMMs. Each processor works on its local partition of the database but performs the same computation of counting support for the global candidate itemsets. Task parallelism corresponds to the case where the processors perform different computations independently, such as counting a disjoint set of candidates, but have or need access to the entire database. SMPs have access to the entire data, but for DMMs, the process of accessing the database can involve selective replication or explicit communication of the local portions. Hybrid parallelism, which combines both task and data parallelism, is also possible and perhaps desirable for exploiting all available parallelism in ARM methods.

Static vs. dynamic load balancing

Static load balancing initially partitions work among the processors using a heuristic cost function; no subsequent data or computation movement is available to correct load imbalances resulting from ARM algorithms' dynamic nature. Dynamic load balancing seeks to address this by taking work from heavily loaded processors and reassigning it to lightly loaded ones. Computation movement also entails data movement, because the processor responsible for a computational task needs the data associated with that task. Dynamic load balancing thus

incurs additional costs for work and data movement, and also for the mechanism used to detect whether there is an imbalance. However, dynamic load balancing is essential if there is a large load imbalance or if the load changes with time.

Dynamic load balancing is especially important in multiuser environments with transient loads and in heterogeneous platforms, which have different processor and network speeds. These kinds of environments include parallel servers and heterogeneous clusters, metaclusters, and superclusters (the so-called grid platforms that are becoming common today). All extant ARM algorithms use only static load balancing that is inherent in the initial partitioning of the database among available nodes. This is because they assume a dedicated, homogeneous environment.

Figure 3 shows where each parallel ARM method falls in the design space. DMMs form the dominant platform, and a mix of data- and task- parallel approaches have been explored. However, all schemes use static load balancing (or very limited dynamic load balancing).

The main design issues in DMMs are minimizing communication and evenly distributing data for good load balancing. The distributed-memory ARM algorithms that I now consider assume the database is partitioned among all the processors in equal-sized blocks residing on each processor's local disk.

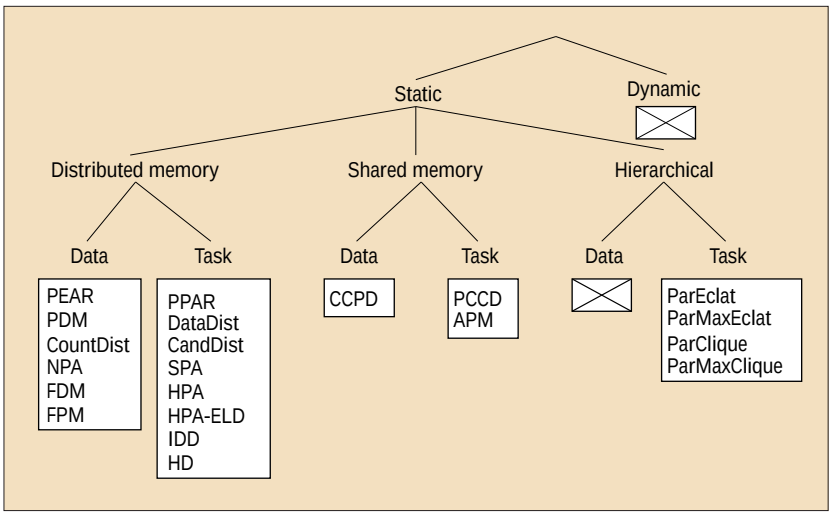

Figure 3. Parallel association-rule-mining (ARM) algorithms. The taxonomy shows where each method fits in the design space, organized by load-balancing strategy, architecture, and parallelism used. For example, Count the word Distribution uses static balancing, distributed memory, and data parallelism.

SEAR- and SPEAR-based

Andreas Mueller proposed some of the first parallel ARM methods,[4] built atop his sequential methods, which were based on Apriori and Partition. PEAR is the parallel version of SEAR. In each iteration, every processor generates a candidate prefix tree from the global frequent itemsets of the previous pass. Each processor has the entire copy of the same candidate set. Each node then gathers local supports, followed by a sum reduction to obtain global supports on each processor.

Partitioned Parallel Association Rules is based on Spear. In fact, PPAR is the parallelization suggested, but not implemented, by Partition's authors, with the exception that PPAR uses the horizontal data format. PPAR works as follows. Each processor gathers the locally frequent itemsets of all sizes in one pass over their local database (which also may be partitioned into local chunks). PPAR broadcasts all potentially frequent itemsets to other processors. Then each processor gathers the counts of these global candidates in the second local pass. Finally, a broadcast obtains the global frequent itemsets. Experiments on a 16-node IBM SP2 DMM showed that PEAR always outperformed PPAR. This is because PEAR uses pass-bundling, whereas PPAR might unnecessarily generate many candidates that end up infrequent.

DHP-based

The PDM algorithm by Jong Soo Park and his colleagues is based on DHP.[2,7] In PDM, each processor generates the local supports of 1-itemsets and approximate counts for the 2-itemsets with a hash table. An all-to-all broadcast of local counts obtains the global counts for 1-itemsets. Because the 2-itemset hash table can be very large, directly exchanging the counts through an all-to-all broadcast can be expensive. Park and his colleagues use an optimized method that exchanges only the cells that are guaranteed to be frequent. However, this method requires two rounds of communication. For the second pass, PDM generates local candidates using the global 2-itemset hash table. Only the second pass uses hash tables; subsequent passes generate candidates directly from F_{k-1} (as in Apriori) .

Candidates are generated in parallel. Each processor generates its own local set, which is exchanged through an all-to-all broadcast to construct the global candidate set. Next, PDM obtains the local counts for all candidates and exchanges them among all processors to determine the globally frequent itemsets. The stage is now set for the next iteration. PDM has several limitations. First, it parallelizes the candidate generation at the cost of an all-to-all broadcast to construct the entire candidate set. The communication costs might render this parallelization ineffective.

Park and his colleagues presented only simulation results on an IBM-SP2-type distributed-memory machine, so assessing the practical impact of their optimizations is difficult.

Apriori-based

Many parallel algorithms use Apriori as the base method, because of its success in the sequential setting.

Count, Data, and Candidate Distribution

Rakesh Agrawal and John Shafer,[8] from the group that developed Apriori, have proposed three parallel algorithms. Their target machine was a 32-node IBM SP2 DMM.

The Count Distribution algorithm is a simple parallelization of Apriori. All processors generate the entire candidate hash tree from F_{k-1}. Each processor can thus independently get partial supports of the candidates from its local database partition. Next, the algorithm does a sum reduction

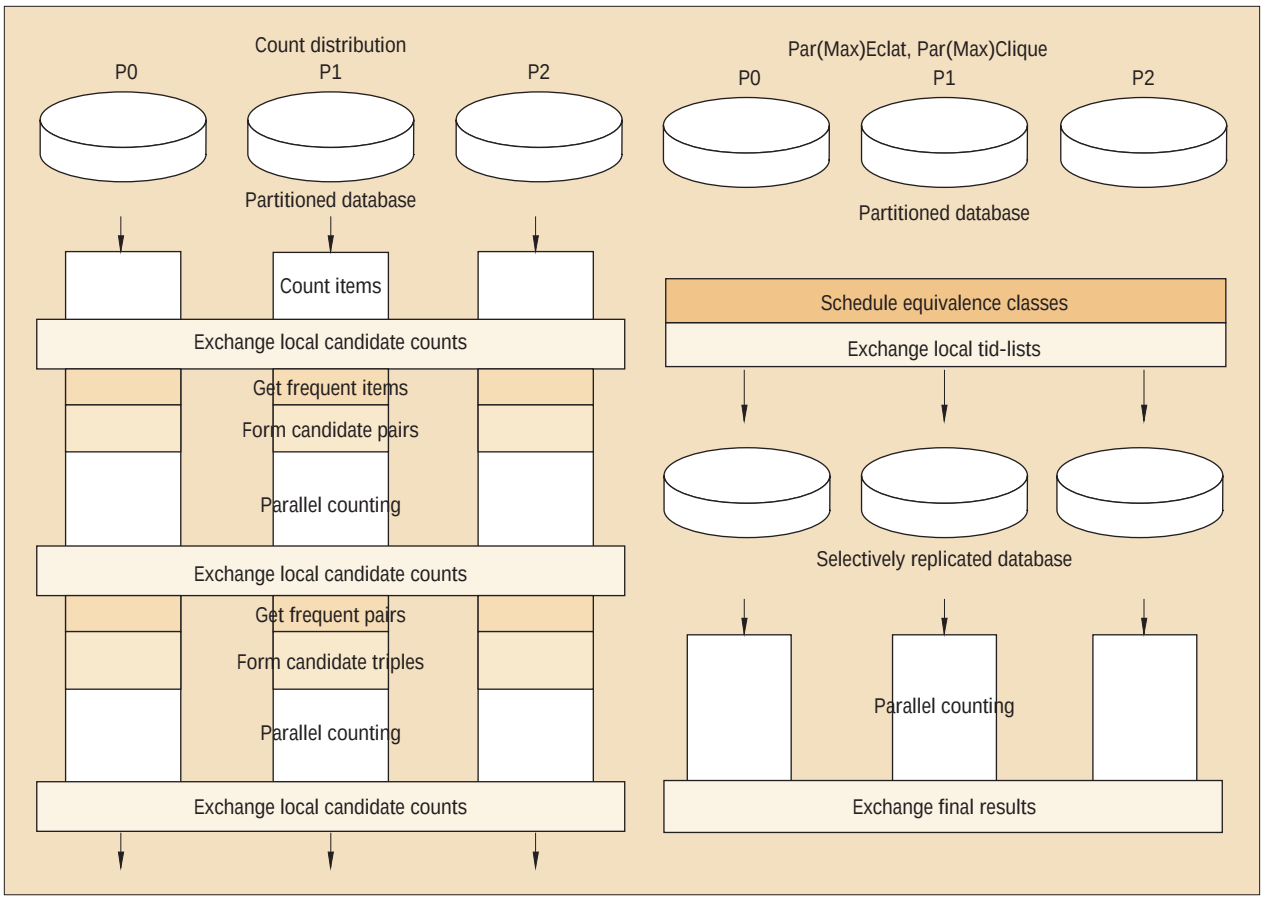

Figure 4. Count Distribution vs. Par(Max)Eclat and Par(Max)Clique. Count Distribution is iterative, like Apriori; in each iteration, it generates potential candidates and gathers their local support in parallel, followed by a sum-reduction. After the setup phase, Par(Max)Eclat/Clique asynchronously computes the frequent itemsets in parallel.

to obtain the global counts by exchanging local counts with all other processors. Rather than merging different hash trees, the algorithm needs to communicate only partial counts, because each processor has a copy of the entire tree. Once the global F_k has been determined, each processor builds the entire candidate C_{k+1} in parallel, and repeats the process until all frequent itemsets are found. Figure 4 shows an illustration of count distribution. This algorithm minimizes communication, because only the counts are exchanged among the processors. However, because the algorithm replicates the entire hash tree on each processor, it doesn't use the aggregate system memory effectively.

The Data Distribution algorithm uses the total system memory by generating disjoint candidate sets on each processor. However, to generate the global support, each processor must scan the entire database (its local partition and all remote partitions) in all iterations. Hence, this algorithm suffers from high communication overhead and performs poorly when compared to Count Distribution.

The Candidate Distribution algorithm partitions the candidates during iteration l, so that each processor can generate disjoint candidates independent of other processors. The partitioning uses a heuristic based on support, so that each processor gets an equal amount of work. At the same time, the database is selectively replicated so that a processor can generate global counts independently. The choice of the redistribution pass involves a tradeoff between decoupling processor dependence as soon as possible and waiting until sufficient load balance can be achieved. In Agrawal and Shafer's experiments, the repartitioning was done in the fourth pass. After that, the only dependence a processor had on other processors is for pruning the candidates. Each processor asynchronously broadcasts the locally frequent set to other processors during each iteration. If this pruning information arrives in time, it is used; otherwise, it is saved for the next iteration. Each processor must still scan its local data once per iteration. Even though it uses problem-specific information, Candidate Distribution performs worse than Count Distribution, because it pays the cost of redistributing the database while scanning the local database partition repeatedly.

Non Partitioned, Simply Partitioned, and Hash-Partitioned Apriori

Independently, Takahiko Shintani and Masaru Kitsuregawa proposed four Apriori-based parallel algorithms, which are very similar to the three discussed above.[9] Their target machine was a 64-node Fujitsu AP1000DDV DMM.

Non Partitioned Apriori is essentially the same as Count Distribution, except that the sum reduction occurs on one master processor. Simply Partitioned Apriori is the same as Data Distribution.

Hash-Partitioned Apriori is similar to Candidate Distribution. Each processor generates candidates from the previous level's frequent set and applies a hash function to determine a home processor for that candidate. If a processor is the home for a candidate, it inserts the candidate in the local hash tree; otherwise, it discards the candidate. For counting, HPA, unlike

Candidate Distribution, does not selectively replicate the database; each processor generates a *k*-subset for every local transaction, calculates the destination processor, and communicates that subset to the processor. The home processor is responsible for incrementing the counts using the local database and any messages sent by other processors.

Shintani and Kitsuregawa also propose a variant of HPA called HPA-ELD (HPA with extremely large itemsets duplication). The motivation is that even though we might partition candidates equally among processors, some candidates are more frequent than others. Therefore, their home processors will consequently be loaded, while others will have a light load. HPA-ELD addresses this by replicating the extremely frequent itemsets on all processors and processing them using the NPA scheme (a limited form of dynamic load balancing). Thus, no subsets are communicated for these candidates. Local counts are obtained, followed by a sum reduction for their global support.

Shintani and Kitsuregawa experimentally confirmed that HPA-ELD outperforms the other approaches. However, they used SPA, HPA, and HPA-ELD only for the second iteration, while they performed the remaining passes using non partitioned Apriori. This suggests that the best approach is a hybrid one: use HPA-ELD as long as candidates do not fit in memory and then switch to non partitioned Apriori. This makes sense, because non partitioned Apriori and Count Distribution require the least amount of communication.

Intelligent Data Distribution and Hybrid Distribution

Eui-Hong Han and his colleagues have proposed two ARM methods based on Data Distribution.[10] Their platform is a 128-node Cray T3D DMM. They observe that Data Distribution uses an expensive all-to-all broadcast to send local database portions to every other processor. Furthermore, although Data Distribution divides the candidates equally among the processors, it fails to divide the work done on each transaction. That is, it still generates a subset of the transaction and determines whether the hash tree contains that subset.

In Intelligent Data Distribution, Han and his colleagues use a linear-time, ring-based, all-to-all broadcast for communication.[10] Second, they switch to Count Distribution once the candidates fit in memory. Third, instead of a round-robin candidate partitioning, they perform a single-item, prefix-based partitioning. Before processing a transaction, they make sure that it contains the relevant prefixes. If not, the transaction can be discarded. The entire database is still communicated, but a transaction might not be processed if it does not contain relevant items.

The Hybrid Distribution combines Count Distribution and Intelligent Data Distribution. It partitions the P processors into G equal-sized groups, where each group is considered a superprocessor. Count Distribution is used among the G superprocessors, while the P/G processors in a group use Intelligent Data Distribution. The database is horizontally partitioned among the G superprocessors, and the candidates are partitioned among the P/G processors in a group. Additionally, Hybrid Distribution adjusts the number of groups dynamically for each pass. The advantages of Hybrid Distribution are that it reduces database communication costs by $1/G$ and that it tries to keep processors busy, especially during later iterations. Han and his colleagues' experiments showed that while Hybrid Distribution has the same performance as Count Distribution, it can handle much larger databases.

Fast Distributed

David Cheung and his colleagues proposed the Fast Distributed Mining (FDM) algorithm for ARM.[11] The main difference between parallel and distributed data mining is the interconnection network latency and bandwidth. In distributed mining, we assume that the network is much slower. Apart from this distinction, the difference between the two is becoming blurred. For a slow network, any variants of Data Distribution, which essentially communicate the entire database in each iteration, are not practical, given the communication costs. Because Count Distribution has the lowest communication cost, it is an ideal base method to build upon in a distributed environment

FDM builds on Count Distribution, and proposes new techniques to reduce the number of candidates considered for counting. In this way, it also minimizes communication. FDM assumes that the database is horizontally partitioned among the distributed sites. Because any globally frequent itemset must be locally frequent at a site, the only candidates a site has to consider are the ones generated from the ones both globally frequent and locally frequent at that site (denoted as GL_i for site i). For example, out of all frequent items $F_1 = \{A, B, C, D, E\}$, let $GL_1 = \{A, B, C\}$ and $GL_2 = \{C, D, E\}$. The first site then considers only the candidates $CG_1 = \{AB, AC, BC\}$ and $CG_2 = \{CD, CE, DE\}$. Instead of these six candidates, Count Distribution would generate

$$\binom{5}{2} = 10$$

candidates. FDM also suggests three optimizations: local pruning, global pruning, and count polling.

The FDM-LP algorithm uses local pruning and count polling. Each site generates candidates using the GL_i from all sites and assigns a home site for each candidate. Then, each site computes the local support for all candidates. Next comes the local pruning step: remove any itemset X that is not locally frequent at the current site, because if X is globally frequent, then it must occur at some other site.

The next step forms the count-polling optimization. Each home site requests, for all candidates assigned to it, local counts from all other sites and computes their global support. The home site then broadcasts the global supports to all other sites. At the end, each site has the globally frequent set, and a new iteration may begin. Recall that Count Distribution broadcasts the local counts of all candidates to everyone else, whereas FDM sends it to only one home site per candidate. Thus, FDM requires far less communication, and local pruning cuts it down even more.

Another optimization that Cheung and

his colleagues suggest is global pruning. Rather than sending only the global supports for the frequent itemsets, they also send their local supports in each partition at the end of iteration $k-1$. For the next iteration k, if X is a candidate, the local supports of all its $k-1$ subsets are available. We can place an upper bound on the support of X at site i as

$$UB(X) = X.\sup_i + \sum_{j=1, j\neq i}^{s} ub_j(X)$$

where s is the number of sites, and $ub_j(X)$ is the minimum local support of any $k-1$ subset of X at site j (the upper bound on the local support of X at site j). If $UB(X) < min\_sup$, we can discard X from consideration. Cheung and his colleagues evaluated FDM on a cluster of six workstations connected with a 10-Mbyte Ethernet LAN. Their experiments, tested only for local pruning with count polling, showed a reduction of 75% to 90% in the candidate set size on each site, and a reduction of 85% to 90% in the message size.

Fast Parallel Mining

David Cheung and Yongqiao Xiao recently proposed a parallel version of FDM, called Fast Parallel Mining.[12] The problem with FDM's polling mechanism is that it requires two rounds of messages in each iteration: one for computing the global supports and one for broadcasting the frequent itemsets. This two-round scheme can degrade performance in a parallel setting. FPM generates fewer candidates and retains the local and global pruning steps. But instead of count polling and subsequent broadcast of frequent itemsets, it simply broadcasts local supports to all processors.

The more interesting aspect of this work is a metric Cheung and Xiao define for data skewness (the distribution of itemsets among the various partitions). For an itemset X, let $pX(i)$ denote the probability that X occurs in partition i. The entropy of X is given as

$$H(X) = -\sum_{i}^{n} pX(i)\log\left(pX(i)\right)$$

The entropy measures the distribution of the local support counts of X among all partitions. The skewness of an itemset X is given as

$$S(X) = (H_{max} - H(X))/\ H_{max}$$

where $H_{max} = \log(n)$ for n partitions. $S(X)$ is 0 if X has equal support in all partitions, and 1 if X occurs in only one partition. A database's total data skewness is the sum of the skew of all itemsets weighted by their supports. In practice, we need consider only the frequent itemsets' skew. Cheung and Xiao's experiments on a 32-node IBM SP2 indicate that FPM can outperform Count Distribution by a factor of 3 for data sets with a very high skew, and by a factor of 1.5 for low-skew data.

SHARED-MEMORY MACHINES

The main design issues in shared-memory systems concern minimization and elimination of false sharing and maintenance of good data locality. SMP platforms have not received widespread attention in parallel ARM literature. However, with the advent of multiprocessor desktops and Clumps (for example, nodes in an IBM SP2 can consist of eight-way SMPs), SMP platforms are becoming increasingly important.

Apriori-based

One of the first algorithms targeting SMP machines was Common Candidate Partitioned Database, which my colleagues and I proposed.[13] As the name suggests, CCPD uses a data-parallel approach. The database is logically partitioned into equal-sized chunks, and all the processors synchronously process a global or common-candidate hash tree.

CCPD parallelizes candidate generation. Each processor generates a disjoint candidate subset, leading to a good computational division. To build the hash tree in parallel, CCPD associates a lock with each leaf node. When a processor wants to insert a candidate into the tree, it starts at the root, and successively hashes on the items until it reaches a leaf. It then acquires the lock and inserts the candidate. With this locking mechanism, each processor can insert itemsets in different parts of the hash tree in parallel. For support counting, each processor computes frequency from its logical partition.

We also proposed additional optimizations, such as short-circuited join and hash-tree balancing. Short-circuited join propagates bit markers up the hash tree to avoid processing subtrees already processed earlier. We used a new hash function for hash-tree balancing, because a simple *mod* (for two integers, *a mod b* returns the remainder of *a* divided by *b*) function can lead to skewed trees. A balanced tree speeds up processing, because it has a shorter height. We evaluated CCPD's performance on a 12-node SGI Power Challenge. CCPD obtained reasonable speedup, but the serial I/O was detrimental to performance.

We also implemented the Partitioned Candidate Common Database algorithm, where the processors construct disjoint candidate trees and scan the entire database for candidate supports. However, the I/O overhead and disk contention for PCCD was unacceptable, resulting in slowdowns on more than one processor.

In recent work, we proposed memory-placement optimizations to speed up CCPD. We showed that, because of the nature of hashing, the candidate hash tree has very poor data locality. Furthermore, a common tree can lead to false sharing in the support-counting phase. We proposed mechanisms and policies to control the hash tree's memory layout based on the access patterns in support counting. Our schemes ensure that the nodes most likely to be accessed in a sequence lie close in physical memory as well. This leads to good locality. We also proposed an effective privatization mechanism, where each processor collects counts in a local array, followed by a sum reduction for reducing false sharing. Experiments on a 12-node SGI Challenge showed improvements of 50% to 60% over the base case.

DIC-based

Cheung and colleagues have proposed the Asynchronous Parallel Mining algorithm, which is based on DIC.[14] APM uses FDM's global-pruning technique to decrease the size of candidate 2-itemsets. This pruning is most effective when there

is high data skew among the partitions. However, DIC requires that the partitions be homogeneous (as explained earlier).

APM addresses this problem by treating the first iteration separately. APM logically divides the database into many small, equal-sized virtual partitions. The number of virtual partitions l is independent of the number of processors p, but usually $l \geq p$. Let m be the number of items. APM gathers the local counts of the m items in each partition. This forms an $l \times m$ data set, with l item support vectors in an m-dimensional space. APM groups these l vectors into k clusters, maximizing intercluster distance and minimizing intracluster distance. Thus, the k clusters or partitions are as skewed as possible, and they are used to generate a small set of candidate 2-itemsets.

APM now prepares to apply DIC in parallel. The idea is to divide the database into p homogeneous partitions. Each processor independently applies DIC to its local partition. However, there is a shared prefix tree among all processors, which is built asynchronously. APM stops when all processors have processed all candidates, whether generated by themselves or others, and when no new candidates are generated. To apply DIC on its partitions, each processor must divide its local partition into r subpartitions. Furthermore, DIC requires that both the p interprocessor partitions and the r intraprocessor partitions be as homogeneous as possible. APM ensures that the p partitions are homogeneous by assigning the virtual partitions from each of the k clusters of the first pass in a round-robin manner among the p processors. Thus, each processor gets an equal mix of virtual partitions from separate clusters, resulting in homogeneous processor partitions.

To get intraprocessor partition homogeneity, APM performs a secondary k-clustering. That is, they group the r partitions into k clusters, and again assign elements from each of the k clusters to the r partitions in a round-robin manner. Experiments on a 12-node Sun Enterprise 4000 SMP indicate that APM outperforms a Count Distribution/CCPD-type algorithm by a factor of four to five. An interesting trade-off in APM is that although data skewness is good for global pruning, it is detrimental to workload balance.

Hierarchical systems

A hierarchical system has both distributed- and shared-memory components—for example, a cluster of SMP workstations. Hierarchical systems are becoming increasingly popular today, especially with the advent of multiprocessor desktops and recent advances in high-speed networks. These clusters provide scalability and performance comparable to expensive machines but at an attractive cost. In fact, even the distributed-memory machines such as IBM SP2 can have eight-way SMP nodes. Another example is the SGI Origin NUMA hardware-distributed shared-memory system. In a hierarchical system, we have to optimize internode communication and data decomposition and optimize intranode data locality and false sharing for each SMP node.

Eclat-based

My colleagues and I have proposed four algorithms—ParEclat, ParMaxEclat, ParClique, and ParMaxClique—that target hierarchical systems.[15] All four are based on their sequential counterparts.[6]

In the discussion below, I refer to a p-way SMP node as a host, and I assume that there are n hosts, for a total of np processors in the system. These methods assume that the database is in the vertical format and partitioned among the hosts so that each host gets an entire tidlist for a single item. The total length of local tidlists is roughly equal on all hosts.

All four algorithms have a similar parallelization and differ only in search strategy and equivalence class-decomposition technique. Figure 3 contrasts these methods against the Count-Distribution algorithm. Each of these four algorithms has three main phases:

- the initialization phase, which performs computation and data partitioning;
- the asynchronous phase, where each processor independently generates frequent itemsets; and
- the reduction phase, which aggregates final results.

In the initialization phase, the master host generates prefix or Clique-based equivalence classes, using the frequent 2-itemsets. A greedy algorithm then schedules these classes among all available processors. Each class has a weight based on its cardinality. The greedy scheduling algorithm then sorts the classes by weight and assigns the largest class to the processor with the least total weight, repeating the process for each class in sorted order. After parent class scheduling, tidlists are selectively replicated on each host, so all item tidlists that are part of an assigned class on a processor are available on the host's local disk. Only the hosts take part in this communication.

In the asynchronous phase, each processor has available the classes assigned to it, and the tidlists for all items. Thus, each processor can independently generate all frequent itemsets from its classes. No communication or synchronization is required. Furthermore, all available system memory is used; no in-memory hash or prefix trees are needed. Only simple intersection operations are required for itemset enumeration.

The four algorithms differ depending on the decomposition and search strategy used. ParEclat and ParMaxEclat use prefix-based classes, but they use bottom-up and hybrid search, respectively. ParClique and ParMaxClique use smaller clique-based classes, with bottom-up and hybrid-lattice search, respectively. We experimented on a 32-processor Digital Alpha cluster, with eight four-way SMP hosts, connected by the fast Digital Memory-Channel network. Comparisons with a hierarchical implementation of Count Distribution/CCPD showed orders-of-magnitude improvements of ParMaxClique over Count Distribution.

Summary of parallel algorithms

Table 2 shows the essential differences among the different methods reviewed earlier and groups together related algorithms. As you can see, there are only a handful of distinct paradigms. The other algorithms propose optimizations over these basic cases. For example, PEAR, PDM, NPA, FDM, FPM, and CCPD are

Table 2. Parallel-algorithm characteristics.

Algorithm	Characteristics
Count Distribution	Apriori-based
PEAR	Candidate prefix tree
PDM	Hash table for 2-itemsets, parallel candidate generation
NPA	Only master does sum reduction
FDM	Local and global pruning, count polling
FPM	Local and global pruning, skewness handling
CCPD	Shared memory
Data Distribution	Exchange full database per iteration
SPA	Same as Data Distribution
IDD	Ring-based broadcast, item-based candidate partitioning
PCCD	Shared memory (logical database exchange)
Hybrid Distribution	Combines Count and Data Distribution
Candidate Distribution	Selectively replicated database, asynchronous
HPA	No database replication, exchange itemsets
HPA-ELD	Replicate frequent itemsets
ParEclat	Eclat-based, asynchronous, hierarchical
ParMaxEclat	MaxEclat-based, asynchronous, hierarchical
ParClique	Clique-based, asynchronous, hierarchical
ParMaxClique	MaxClique-based, asynchronous, hierarchical
APM	DIC-based, shared memory, asynchronous
PPAR	Partition-based, horizontal database

all similar to Count Distribution. Likewise, SPA, IDD, and PCCD are similar to Data Distribution, whereas HPA and HPA-ELD are similar to Candidate Distribution. Hybrid Distribution combines Count and Data Distribution techniques. ParEclat, ParMaxEclat, ParClique, and ParMaxClique are all based on their sequential counterparts. Finally, APM is based on the sequential DIC method, and PPAR is based on Partition. Thus, these parallel methods share the same complexity and properties of the sequential algorithms on which they are based (see Table 1).

Open problems

Despite recent advances in high-performance ARM algorithms, several problems still need serious and immediate attention.

High dimensionality

Current ARM methods can handle only a few thousand dimensions or items. The problem is as follows. The second iteration, which counts the frequency of all 2-itemsets, essentially has quadratic complexity. The reason is, we must consider all item pairs, and no pruning is possible at this stage. In general, the algorithms' complexity might not be linear in the number of dimensions. We need new parallel methods that scale with the dimensions. Some possible solutions include methods that enumerate only maximal patterns, those that use hash-based pruning to reduce the candidate itemsets (especially 2-itemsets)[7] and those that use global pruning.[11]

Large size

Databases continue to increase in size. Current methods can handle data in the tens-of-gigabytes range. Current ARM algorithms do not appear to be suitable for the terabyte range. Even a single scan for these databases is considered expensive. Most algorithms are iterative and scan data multiple times; hence, they are not really scalable. Mining all frequent itemsets in a single pass is an open problem. Another factor limiting the scalability of most current algorithms is the in-memory candidate hash tree or prefix tree. For large databases, the candidates will certainly not fit in aggregate system memory. This means candidates must be written out to disk and divided into partitions small enough to be processed in memory—entailing further data scans. For vertical-format approaches, we must consider cases where even a single tidlist doesn't fit in memory. Techniques from parallel-join algorithms offer a possible solution.

Data location

Today's large-scale data sets are usually logically and physically distributed. Organizations that are geographically distributed need a decentralized approach to ARM. The issues concerning modern organizations are not only the size of the data to be mined, but also its distributed nature. ARM data may be horizontally partitioned (where different sites have different transactions) or vertically partitioned (where different sites have different items). Most current work concerns only the horizontal-partitioning approach.

Data skew

One of the problems adversely affecting load balancing in ARM algorithms is sensitivity to data skew. Most methods partition the database horizontally in equal-sized blocks. However, the number of frequent itemsets generated from each block can be heavily skewed. That is, although one block might contribute many frequent itemsets, the other might have very few, implying that the processor responsible for the latter block will be idle most of the time. Randomizing the blocks is one solution, but it is not adequate, given ARM's dynamic and interactive nature. The user might specify different support levels, or the user might be interested in only a particular set of items.

Another kind of data skew depends on whether itemsets are frequent in many or only a few blocks. We have seen that most algorithms require low data skewness for good load balancing. But a high skew is needed to apply global pruning and cut down the candidate set. Both of these opposing needs must be reconciled, and the effect of skewness on different algorithms needs to be studied further.

Dynamic load balancing

All extant algorithms use only a static load-balancing scheme based on the initial data decomposition, and they assume a homogeneous, dedicated environment. This is far from reality. A typical parallel database server has multiple users and transient loads. This calls for an investigation of dynamic load-balancing schemes for ARM. Dynamic load balancing is also crucial in a heterogeneous environment. Such an environment might include metaclusters and superclusters, with machines ranging from ordinary workstations to supercomputers.

RULE DISCOVERY

The main focus of the current parallel methods has been frequent itemset discovery. The rule-generation phase has received almost no attention. The reason was the assumption that there were only a few frequent itemsets, which led researchers to believe rule generation was cheap. This assumption doesn't hold for massive data sets. We can extract literally millions of frequent itemsets. The complexity of the rule-generation step is $0(r \cdot 2^l)$, where r is the number of frequent itemsets and l is the longest frequent pattern. This is not a trivial problem if r is large, even if we assume l is bounded. Parallel methods are needed to efficiently enumerate all strong rules.

PARALLEL DATABASE-MANAGEMENT SYSTEMS AND FILE SYSTEMS

All the results reported, thus far, partition the database, mainly horizontally, among the different processors by hand. No one has studied using a parallel file system for managing the partitioned database, along with the accompanying striping and layout issues. Recently, we've witnessed increasing emphasis on tight database integration of ARM, but it has been confined to sequential approaches.

GENERALIZATIONS OF ASSOCIATION RULES

The ARM problem we have considered in this article is in fact the binary-association problem: an item is either present or absent from a transaction. We can also think about the general case where the quantity of the items bought is also considered. This problem is called quantitative-association mining. In general, we can have items that take values from a continuous domain (called numeric attributes) or items that take a finite number of nonnumeric values (called categorical attributes). Applying ARM to such data typically requires *binning* or *discretizing* the continuous attributes into ranges. Then we must form new items for each range of the numeric attributes or for each value of the categorical attributes.

Another extension of ARM is called generalized-association mining. Here, the items are at the leaf levels in a hierarchy or taxonomy of items. The goal is to discover association rules involving concepts at multiple (and mixed) levels, from the primitive item level to the root of the hierarchy. The computational complexity for both these generalizations of binary associations is significantly greater. Therefore, parallel computing is crucial for obtaining good performance.

ALTHOUGH I PLACED the problems discussed above within the association framework, they are equally relevant for any practical parallel or distributed data-mining system. Such systems are still in their infancy, and a lot of exciting work remains to be done in system design, implementation, and deployment. I hope that this article will serve as a reference for the state of the art for both researchers and practitioners interested in building parallel and distributed ARM systems.

References

1. R. Agrawal et al., "Fast Discovery of Association Rules," *Advances in Knowledge Discovery and Data Mining*, U. Fayyad et al., eds., AAAI Press, Menlo Park, Calif., 1996, pp. 307–328.
2. J.S. Park, M. Chen, and P.S. Yu, "An Effective Hash Based Algorithm for Mining Association Rules," *Proc. ACM SIGMOD Conf.*, ACM Press, New York, 1995, pp. 175–186.
3. A. Savasere, E. Omiecinski, and S. Navathe, "An Efficient Algorithm for Mining Association Rules in Large Databases," *Proc. 21st Int'l Conf. Very Large Databases.*, Morgan Kaufmann, San Francisco, 1995, pp. 432–444.
4. A. Mueller, *Fast Sequential and Parallel Algorithms for Association Rule Mining: A Comparison*, Tech. Report CS-TR-3515, Univ. of Maryland, College Park, Md., 1995.
5. S. Brin et al., "Dynamic Itemset Counting and Implication Rules for Market Basket Data," *Proc. ACM SIGMOD Conf. Management of Data*, ACM Press, New York, 1997, pp. 255–264.
6. M.J. Zaki et al., "New Algorithms for Fast Discovery of Association Rules," *Proc. 3rd Int'l Conf. Knowledge Discovery and Data Mining*, AAAI Press, Menlo Park, Calif., 1997, pp. 283–286.
7. J.S. Park, M. Chen, and P.S. Yu, "Efficient Parallel Data Mining for Association Rules," *Proc. ACM Int'l Conf. Information and Knowledge Management*, ACM Press, New York, 1995, pp. 31–36.
8. R. Agrawal and J. Shafer, "Parallel Mining of Association Rules," *IEEE Trans. Knowledge and Data Eng.*, Vol. 8, No. 6, Dec. 1996, pp. 962–969.
9. T. Shintani and M. Kitsuregawa, "Hash Based Parallel Algorithms for Mining Association Rules," *Proc. 4th Int'l Conf. Parallel and Distributed Information Systems*, IEEE Computer Soc. Press, Los Alamitos, Calif., 1996, pp. 19–30.
10. E.H. Han, G. Karypis, and V. Kumar, "Scalable Parallel Data Mining for Association Rules," *Proc. ACM Conf. Management of Data*, ACM Press, New York, 1997, pp. 277–288.
11. D. Cheung et al., "A Fast Distributed Algorithm for Mining Association Rules," *Proc. 4th Int'l Conf. Parallel and Distributed Information Systems*, IEEE Computer Soc. Press, Los Alamitos, Calif., 1996, pp. 31–42.
12. D. Cheung and Y. Xiao, "Effect of Data Skewness in Parallel Mining of Association Rules," *Proc. Pacific-Asia Conf. Knowledge Discovery and Data Mining*, *Lecture Notes in Computer Science*, Vol. 1394, Springer-Verlag, New York, 1998, pp. 48–60.
13. M.J. Zaki et al., "Parallel Data Mining for Association Rules on Shared-Memory Multiprocessors," *Proc. Supercomputing '96*, IEEE Computer Soc. Press, Los Alamitos, Calif., 1996.
14. D. Cheung, K. Hu, and S. Xia, "Asynchronous Parallel Algorithm for Mining Association Rules on Shared-Memory Multiprocessors," *Proc. 10th ACM Symp. Parallel Algorithms and Architectures*, ACM Press, New York, 1998, pp. 279–288.
15. M.J. Zaki et al., "Parallel Algorithms for Fast Discovery of Association Rules," *Data Mining and Knowledge Discovery: An Int'l J.*, Vol. 1, No. 4, Dec. 1997, pp. 343–373.

Mohammed J. Zaki is an assistant professor of computer science at Rensselaer Polytechnic Institute. His research interests focus on developing efficient, scalable, parallel, and interactive algorithms for various data-mining and knowledge-discovery tasks. He received his BS in computer science from Angelo State University, Texas. He received his MS and PhD in computer science, both from the University of Rochester. Contact him at the Computer Science Dept., Rensselaer Polytechnic Inst., Troy, NY 12180; zaki@cs.rpi.edu; www.cs.rpi.edu/~zaki.