

Parallel Data Mining Algorithms for Association Rules and Clustering

Jianwei Li <i>Northwestern University</i>	1.1 Introduction.....	1-1
Ying Liu <i>DTKE Center and Grad. Univ. of CAS</i>	1.2 Parallel Association Rule Mining	1-2
Wei-keng Liao <i>Northwestern University</i>	<i>Apriori</i> -based Algorithms • Vertical Mining • Pattern-Growth Method • Mining by Bitmaps • Comparison	
Alok Choudhary <i>Northwestern University</i>	1.3 Parallel Clustering Algorithms	1-14
	Parallel <i>k-means</i> • Parallel Hierarchical Clustering • Parallel <i>HOP</i> : Clustering Spatial Data • Clustering High-Dimensional Data	
	1.4 Summary	1-22

1.1 Introduction

Volumes of data are exploding in both scientific and commercial domains. Data mining techniques that extract information from huge amount of data have become popular in many applications. Algorithms are designed to analyze those volumes of data automatically in efficient ways, so that users can grasp the intrinsic knowledge latent in the data without the need to manually look through the massive data itself. However, the performance of computer systems is improving at a slower rate compared to the increase in the demand for data mining applications. Recent trends suggest that the system performance has been improving at a rate of 10-15% per year, whereas the volume of data collected nearly doubles every year. As the data sizes increase, from gigabytes to terabytes or even larger, sequential data mining algorithms may not deliver results in a reasonable amount of time. Even worse, as a single processor alone may not have enough main memory to hold all the data, a lot of sequential algorithms could not handle large scale problems or have to process data out of core, further slowing down the process.

In recent years, there is an increasing interest in the research of parallel data mining algorithms. In parallel environment, by exploiting the vast aggregate main memory and processing power of parallel processors, parallel algorithms can have both the execution time and memory requirement issues well addressed. However, it is not trivial to parallelize existing algorithms to achieve good performance as well as scalability to massive data sets. First, it is crucial to design a good data organization and decomposition strategy so that workload can be evenly partitioned among all processes with minimal data dependence across them. Second, minimizing synchronization and/or communication overhead is important in order for the parallel algorithm to scale well as the number of processes

Database		Frequent Itemsets (minsup = 33%)	
TID	Items	k	Frequent Itemsets : support
1	f d b e	1	a:67% b:50% c:33% d:50% e:83% f:67%
2	f e b	2	ac:33% ad:33% ae:50% af:33% bd:33% be:33% bf:33% ce:33% cf:33% de:33% ef:67%
3	a d b	3	ace:33% acf:33% aef:33% bef:33% cef:33%
4	a e f c	4	acef:33%
5	a d e		
6	a c f e		

FIGURE 1.1: Example database and frequent itemsets.

increases. Workload balancing also needs to be carefully designed. Last, disk I/O cost must be minimized.

In this chapter, parallel algorithms for association rule mining and clustering are presented to demonstrate how parallel techniques can be efficiently applied to data mining applications.

1.2 Parallel Association Rule Mining

Association rule mining (ARM) is an important core data mining technique to discover patterns/rules among items in a large database of variable-length transactions. The goal of ARM is to identify groups of items that most often occur together. It is widely used in market-basket transaction data analysis, graph mining applications like substructure discovery in chemical compounds, pattern finding in web browsing, word occurrence analysis in text documents, and so on. The formal description of ARM can be found in [AIS93, AS94]. And most of the research focuses on the frequent itemset mining subproblem, i.e., finding all frequent itemsets each occurring at more than a minimum frequency (*minsup*) among all transactions. Figure 1.1 gives an example of mining all frequent itemsets with *minsup* = 33% from a given database. Well-known sequential algorithms include *Apriori* [AS94], *Eclat* [ZPOL97a], *FP-growth* [HPY00], and *D-CLUB* [LCJL06]. Parallelizations of these algorithms are discussed in this section, with many other algorithms surveyed in [Zak99].

1.2.1 *Apriori*-based Algorithms

Most of the parallel ARM algorithms are based on parallelization of *Apriori* that iteratively generates and tests candidate itemsets from length 1 to length k until no more frequent itemsets are found. These algorithms can be categorized into *Count Distribution*, *Data Distribution* and *Candidate Distribution* methods [AS96, HKK00]. The *Count Distribution* method follows a data-parallel strategy and statically partitions the database into horizontal partitions that are independently scanned for the local counts of all candidate itemsets on each process. At the end of each iteration, the local counts will be summed up across all processes into the global counts so that frequent itemsets can be found. The *Data Distribution* method attempts to utilize the aggregate main memory of parallel machines by partitioning both the database and the candidate itemsets. Since each candidate itemset is counted by only one process, all processes have to exchange database partitions during each iteration in order for each process to get the global counts of the assigned candidate itemsets. The *Candidate Distribution* method also partitions candidate itemsets but selectively replicates instead of partition-and-exchanging the database transactions, so that each process can

(a) Database Partitioning

TID	Items	Process Number
1	f d b e	P0
2	f e b	
3	a d b	P1
4	a e f c	
5	a d e	P2
6	a c f e	

(b) Mining Frequent 1-itemsets

Item	Local Counts			Global Count
	P0	P1	P2	
a	0	2	2	4
b	2	1	0	3
c	0	1	1	2
d	1	1	1	3
e	2	1	2	5
f	2	1	1	4

(c) Mining Frequent 2-itemsets

2-itemset	Local Counts			Global Count
	P0	P1	P2	
ab	0	1	0	1
ac	0	1	1	2
ad	0	1	1	2
ae	0	1	2	3
af	0	1	1	2
bc	0	0	0	0
bd	1	1	0	2
be	2	0	0	2
bf	2	0	0	2
cd	0	0	0	0
ce	0	1	1	2
cf	0	1	1	2
de	1	0	1	2
df	1	0	0	1
ef	2	1	1	4

(d) Mining Frequent 3-itemsets

3-itemset	Local Counts			Global Count
	P0	P1	P2	
ace	0	1	1	2
acf	0	1	1	2
ade	0	0	1	1
aef	0	1	1	2
bde	1	0	0	1
bef	2	0	0	2
cef	0	1	1	2

(e) Mining Frequent 4-itemsets

4-itemset	Local Counts			Global Count
	P0	P1	P2	
acef	0	1	1	2

FIGURE 1.2: Mining frequent itemsets in parallel using the *Count Distribution* algorithm with 3 processes. The itemset columns in (b), (c), (d) and (e) list the candidate itemsets. Itemsets that are found infrequent are grayed out.

proceed independently. Experiments show that the *Count Distribution* method exhibits better performance and scalability than the other two methods. The steps for the *Count Distribution* method are generalized as follows for distributed-memory multiprocessors.

- (1) Divide the database evenly into horizontal partitions among all processes;
- (2) Each process scans its local database partition to collect the local count of each item;
- (3) All processes exchange and sum up the local counts to get the global counts of all items and find frequent 1-itemsets;
- (4) Set level $k = 2$;
- (5) All processes generate candidate k -itemsets from the mined frequent $(k-1)$ -itemsets;
- (6) Each process scans its local database partition to collect the local count of each candidate k -itemset;
- (7) All processes exchange and sum up the local counts into the global counts of all candidate k -itemsets and find frequent k -itemsets among them;
- (8) Repeat (5) - (8) with $k = k + 1$ until no more frequent itemsets are found.

As an example, to mine all frequent itemsets in Figure 1.1, the *Count Distribution* algorithm needs to scan the database 4 times to count the occurrences of candidate 1-itemsets, 2-itemsets, 3-itemsets, and 4-itemsets respectively. As illustrated in Figure 1.2, the counting workload in each scan is distributed over 3 processes in such a way that each process scans only an assigned partition (1/3) of the whole database. The three processes proceed in parallel and each one counts the candidate itemsets locally from its assigned transactions. Summation of the local counts for one itemset generates the global count that is used to

determine the support of that itemset. The generation and counting of candidate itemsets is based on the same procedures as in *Apriori*.

In the *Count Distribution* algorithm, communication is minimized since only the counts are exchanged among the processes in each iteration, i.e., in step (3) and (7). In all other steps, each process works independently, relying only on its local database partition. However, since candidate and frequent itemsets are replicated on all processes, the aggregate memory is not utilized efficiently. Also, the replicated work of generating those candidate itemsets and selecting frequent ones among them on all processes can be very costly if there are too many such itemsets. In that case, the scalability will be greatly impaired when the number of processes increases. If on a shared-memory machine, since candidate/frequent itemsets and their global counts can be shared among all processes, only one copy of them needs to be kept. So the tasks of getting global counts, generating candidate itemsets and finding frequent ones among them, as in steps (3), (5) and (7), can be subdivided instead of being repeated among all processes. This actually leads to another algorithm, *CCPD* [ZOPL96], which works the same way as the *Count Distribution* algorithm in other steps. Nevertheless, both algorithms cannot avoid the expensive cost of database scan and inter-process synchronization per iteration.

1.2.2 Vertical Mining

To better utilize the aggregate computing resources of parallel machines, a localized algorithm [ZPL97] based on parallelization of *Eclat* was proposed and exhibited excellent scalability. It makes use of a vertical data layout by transforming the horizontal database transactions into vertical tid-lists of itemsets. By name, the tid-list of an itemset is a sorted list of ID's for all transactions that contain the itemset. Frequent k -itemsets are organized into disjoint equivalence classes by common $(k-1)$ -prefixes, so that candidate $(k+1)$ -itemsets can be generated by joining pairs of frequent k -itemsets from the same classes. The support of a candidate itemset can then be computed simply by intersecting the tid-lists of the two component subsets. Task parallelism is employed by dividing the mining tasks for different classes of itemsets among the available processes. The equivalence classes of all frequent 2-itemsets are assigned to processes and the associated tid-lists are distributed accordingly. Each process then mines frequent itemsets generated from its assigned equivalence classes independently, by scanning and intersecting the local tid-lists. The steps for the parallel *Eclat* algorithm are presented below for distributed-memory multiprocessors.

- (1) Divide the database evenly into horizontal partitions among all processes;
- (2) Each process scans its local database partition to collect the counts for all 1-itemsets and 2-itemsets;
- (3) All processes exchange and sum up the local counts to get the global counts of all 1-itemsets and 2-itemsets, and find frequent ones among them;
- (4) Partition frequent 2-itemsets into equivalence classes by prefixes;
- (5) Assign the equivalence classes to processes;
- (6) Each process transforms its local database partition into vertical tid-lists for all frequent 2-itemsets;
- (7) Each process exchanges the local tid-lists with other processes to get the global ones for the assigned equivalence classes;
- (8) For each assigned equivalence class on each process, recursively mine all frequent itemsets by joining pairs of itemsets from the same equivalence class and intersecting their corresponding tid-lists.

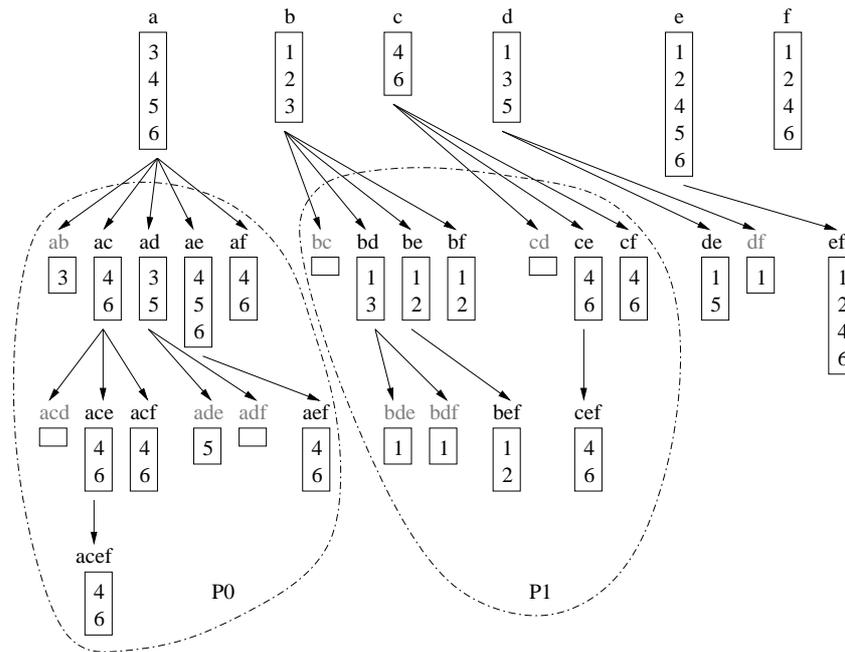


FIGURE 1.3: Mining frequent itemsets using the parallel *Eclat* algorithm. Each itemset is associated with its tid-list. Itemsets that are found infrequent are grayed out.

Step (1) through (3) works in a similar way as in the *Count Distribution* algorithm. In step (5), the scheduling of the equivalence classes on different processes needs to be carefully designed in a manner of minimizing the workload imbalance. One simple approach would be to estimate the workload for each class and assign the classes in turn in descending workload order to the least loaded process. Since all pairs of itemsets from one equivalence class will be computed to mine deeper level itemsets, $\binom{s}{2}$ can be used as the estimated workload for an equivalence class of s itemsets. Other task scheduling mechanisms can also be applied once available. Steps (6) and (7) construct the tid-lists for all frequent 2-itemsets in parallel. As each process scans only one horizontal partition of the database, it gets a partial list of transaction ID's for each itemset. Concatenating the partial lists of an itemset from all processes will generate the global tid-list covering all the transactions. In many cases, the number of frequent 2-itemsets can be so large that assembling all their tid-lists may be very costly in both processing time and memory usage. As an alternative, tid-lists of frequent items can be constructed instead and selectively replicated on all processes so that each process has the tid-lists of all the member items in the assigned equivalence classes. However, this requires generating the tid-list of a frequent 2-itemset on the fly in the later mining process, by intersecting the tid-lists of the two element items. Step (8) is the asynchronous phase where each process mines frequent itemsets independently from each of the assigned equivalence classes, relying only on the local tid-lists. Computing on each equivalence class usually generates a number of child equivalence classes that will be computed recursively.

Taking Figure 1.1 as an example, Figure 1.3 illustrates how the algorithm mines all frequent itemsets from one class to the next using the intersection of tid-lists. The frequent 2-itemsets are organized into 5 equivalence classes that are assigned to 2 processes. Process

P0 will be in charge of the further mining task for one equivalence class, $\{ac, ad, ae, af\}$, while process P1 will be in charge of two, $\{bd, be, bf\}$ and $\{ce, cf\}$. The rightmost classes, $\{de\}$ and $\{ef\}$, do not have any further mining task associated. The two processes then proceed in parallel, without any data dependence across them. For example, to mine the itemset “*ace*”, process P0 only needs to join the two itemsets, “*ac*” and “*ae*”, and intersect their tid-lists, “46” and “456”, to get the result tid-list “46”. At the same time, process P1 can be independently mining the itemset “*bef*” from “*be*”, “*bf*” and their associated tid-lists that are locally available.

There are four variations of parallel *Eclat* - *ParEclat*, *ParMaxEclat*, *ParClique*, and *ParMaxClique* - as discussed in [ZPOL97b]. All of them are similar in parallelization and only differ in the itemset clustering techniques and itemset lattice traversing strategies. *ParEclat* and *ParMaxEclat* use prefix-based classes to cluster itemsets, and adopt bottom-up and hybrid search strategies respectively to traverse the itemset lattice. *ParClique* and *ParMaxClique* use smaller clique-based itemset clusters, with bottom-up and hybrid lattice search, respectively.

Unlike the *Apriori*-based algorithms that need to scan the database as many times as the maximum length of frequent itemsets, the *Eclat*-based algorithms scan the database only three times and significantly reduces the disk I/O cost. Most importantly, the dependence among processes is decoupled right in the beginning so that no communication or synchronization is required in the major asynchronous phase. The major communication cost comes from the exchange of local tid-lists across all processes when the global tid-lists are set up. This one time cost can be amortized by later iterations. For better parallelism, however, the number of processes should be much less than that of equivalence classes (or cliques) for frequent 2-itemsets, so that task assignment granularity can be relatively fine to avoid workload imbalance. Also, more effective workload estimation functions and better task scheduling or workload balancing strategies are needed in order to guarantee balanced workload for various cases.

1.2.3 Pattern-Growth Method

In contrast to the previous itemset generation-and-test approaches, the pattern-growth method derives frequent itemsets directly from the database without the costly generation and test of a large number of candidate itemsets. The detailed design is explained in the *FP-growth* algorithm. Basically, it makes use of a novel frequent-pattern tree (FP-tree) structure where the repetitive transactions are compacted. Transaction itemsets are organized in that frequency-ordered prefix tree such that they share common prefix part as much as possible, and re-occurrences of items/itemsets are automatically counted. Then the FP-tree is traversed to mine all frequent patterns (itemsets). A partitioning-based, divide-and-conquer strategy is used to decompose the mining task into a set of smaller subtasks for mining confined patterns in the so-called conditional pattern bases. The conditional pattern base for each item is simply a small database of counted patterns that co-occur with the item. That small database is transformed into a conditional FP-tree that can be processed recursively.

Due to the complicated and dynamic structure of the FP-tree, it may not be practical to construct a single FP-tree in parallel for the whole database. However, multiple FP-trees can be easily built in parallel for different partitions of transactions. And conditional pattern bases can still be collected and transformed into conditional FP-trees for all frequent items. Thereafter, since each conditional FP-tree can be processed independently, task parallelism can be achieved by assigning the conditional FP-trees of all frequent items to different processes as in [ZEHL01, PK03]. In general, the *FP-growth* algorithm can be parallelized

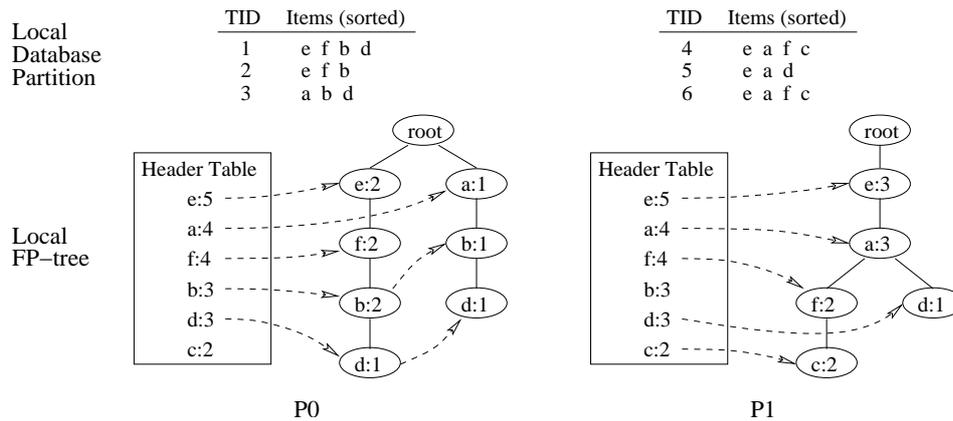


FIGURE 1.4: Construction of local FP-trees from local database partitions over 2 processes. In the transactions and header tables, items are sorted by frequencies in descending order.

in the following steps, assuming distributed-memory multiprocessors.

- (1) Divide the database evenly into horizontal partitions among all processes;
- (2) Scan the database in parallel by partitions and mine all frequent items;
- (3) Each process constructs a local FP-tree from its local database partition with respect to the global frequent items (items are sorted by frequencies in descending order within each scanned transaction);
- (4) From the local FP-tree, each process generates local conditional pattern bases for all frequent items;
- (5) Assign frequent items (hence their associated conditional FP-trees) to processes;
- (6) For each frequent item, all its local conditional pattern bases are accumulated and transformed into the conditional FP-tree on the designated process;
- (7) Each process recursively traverses each of the assigned conditional FP-trees to mine frequent itemsets in the presence of the given item.

Like its sequential version, the parallel algorithm also proceeds in two stages. Step (1) through (3) is the first stage to construct the multiple local FP-trees from the database transactions, Using the transactions in the local database partition, each process can build its own FP-tree independently. For each transaction, global frequent items are selected and sorted by frequency in descending order, and then fed to the local FP-tree as follows. Starting from the root of the tree, check if the first item exists as one of the children of the root. If it exists then increase the counter for this node, or else add a new child node under root for this item with 1 count. Then, taking the current item node as the new temporary root, repeat the same procedure for the next item in the sorted transaction. The nodes of each item are linked together with the head in the header table. Figure 1.4 shows the parallel construction of the multiple local FP-trees on 2 processes for the example database in Figure 1.1.

The second stage is to mine all frequent itemsets from the FP-trees, as in step (4) to (7). The mining process starts with a bottom-up traversal of the local FP-trees to generate the

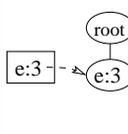
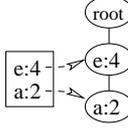
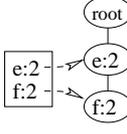
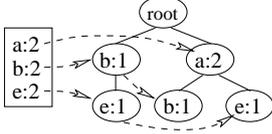
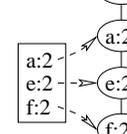
Process Number	P0				P1		
Item	e:5	a:4	f:4	b:3	d:3	c:2	
Conditional Pattern Base		(e:3)	(e:2) (e:2, a:2)	(e:2, f:2) (a:1)	(e:1, f:1, b:1) (a:1, b:1) (e:1, a:1)	(e:2, a:2, f:2)	
Conditional FP-tree							
Mined Frequent Itemsets	e:5	a:4 ea:3	f:4 ef:4 af:2 eaf:2	b:3 eb:2 fb:2 efb:2	d:3 ed:2 ad:2 bd:2	c:2 ac:2 ec:2 fc:2 afc:2 efc:2 aec:2 aefc:2	

FIGURE 1.5: Mining frequent itemsets from conditional pattern bases and conditional FP-trees in parallel.

conditional pattern bases starting from their respective items in the header tables. Each entry of a conditional pattern base is a list of items that precede a certain item on a path of a FP-tree up to the root, with the count of each item set to be that of the considered item node along that path. The assignment of items among processes will be based on some workload estimation heuristic, which is part of ongoing research. For simplicity, the size of the conditional pattern base can be used to estimate the workload associated with an item, and items in the header table can be assigned to processes consecutively. Transforming the conditional pattern bases into conditional FP-trees is no different than constructing FP-trees from database transactions, except that the counter increment is the exact count collected for each item instead of 1. For each frequent item, as each local FP-tree will derive only part of the conditional pattern base, building the global conditional FP-tree needs the accumulation of local conditional pattern bases from all processes. Then a call to the recursive *FP-growth* procedure on each conditional FP-tree will generate all the conditional patterns on the designated process independently. If on a shared-memory machine, since the multiple FP-trees can be made accessible to all processes, the conditional pattern base and conditional FP-tree can be generated on the fly for the designated process to mine the conditional patterns, for one frequent item after another. This can largely reduce memory usage by not generating all conditional pattern bases and conditional FP-trees for all frequent items at one time. Figure 1.5 gives the conditional pattern bases and conditional FP-trees for all frequent items. They are assigned to the two processes. Process P0 computes on the conditional FP-trees for items *a*, *f* and *b*, while process P1 does those for *d* and *c*. Item *e* has an empty conditional pattern base and does not have any further mining task associated. Frequent itemsets derived from the conditional FP-tree of each item are listed as the conditional patterns mined for the item.

In parallel *FP-growth*, since all the transaction information is compacted in the FP-trees, no more database scan is needed once the trees are built. So the disk I/O is minimized by scanning the original database only twice. The major communication/synchronization overhead lies in the exchange of local conditional pattern bases across all processes. Since the repetitive patterns are already merged, the total size of the conditional pattern bases

is usually much smaller than the original database, resulting in relatively low communication/synchronization cost.

1.2.4 Mining by Bitmaps

All previous algorithms work on ID-based data that is either organized as variable-length records or linked by complicated structures. The tedious one-by-one search/match operations and the irregular layout of data easily become the hurdle for higher performance which can otherwise be achieved as in fast scientific computing over well-organized matrices or multi-dimensional arrays.

Based on *D-CLUB*, a parallel bitmap-based algorithm *PD-CLUB* can be developed to mine all frequent itemsets by parallel adaptive refinement of clustered bitmaps using a differential mining technique. It clusters the database into distributed association bitmaps, applies a differential technique to digest and remove common patterns, and then independently mines each remaining tiny bitmaps directly through fast aggregate bit operations in parallel. The bitmaps are well organized into rectangular two-dimensional matrices and adaptively refined in regions that necessitate further computation.

The basic idea of parallelization behind *PD-CLUB* is to dynamically cluster the itemsets with their associated bit-vectors, and divide the task of mining all frequent itemsets into smaller ones, each to mine a cluster of frequent itemsets. Then the subtasks are assigned to different processes and accomplished independently in parallel. A dynamic load balancing strategy can be used to reassign clusters from overloaded processes to free ones. The detailed explanation of the algorithm will be based on a number of new definitions listed below.

- **FI-cluster** - A FI-cluster is an ordered set of frequent itemsets. Starting from the FI-cluster (\mathcal{C}_0) of all frequent 1-itemsets (sorted by supports in ascending order), other FI-clusters can be defined recursively as follows: from an existing FI-cluster, joining one itemset with each of the succeeding ones also generates a FI-cluster if only frequent itemsets are collected from the results. Itemsets can be reordered in the generated FI-cluster.
- **Bit-vector** - For a given database of d transactions, each itemset is associated with a bit-vector, where one bit corresponds to each transaction and is set to 1 iff the itemset is contained in that transaction.
- **Clustered Bitmap** - For each FI-cluster, the bit-vectors of the frequent itemsets are also clustered. Laying out these vertical bit-vectors side by side along their itemsets in the FI-cluster will generate a two-dimensional bitmap, called the clustered bitmap of the FI-cluster.
- **dCLUB** - In the clustered bitmap of a FI-cluster, the following patterned bit rows are to be removed: *e-rows* (each with all 0's), *a-rows* (each with only one 1), *p-rows* (each with zero or more leading 0's followed by trailing 1's), *o-rows* (each with only one 0), and *c-rows* (each with zero or more leading 1's followed by trailing 0's). The remaining rows with different bits mixed disorderly form the differential clustered bitmap (*dCLUB*) of the FI-cluster.

Recursively generating FI-clusters from the initial \mathcal{C}_0 results in a cluster tree that covers all frequent itemsets exactly once. The root of the tree is \mathcal{C}_0 , each node is a FI-cluster (or simply a cluster), and the connection between two FI-clusters denotes the generation relationship. Taking the frequent itemsets in Figure 1.1 as an example, Figure 1.6(a) shows the cluster tree of all frequent itemsets. For instance, FI-cluster $\{caf, cae\}$ is generated from $\{ca, cf, ce\}$ by joining itemset “ca” with “cf” and “ce” respectively.

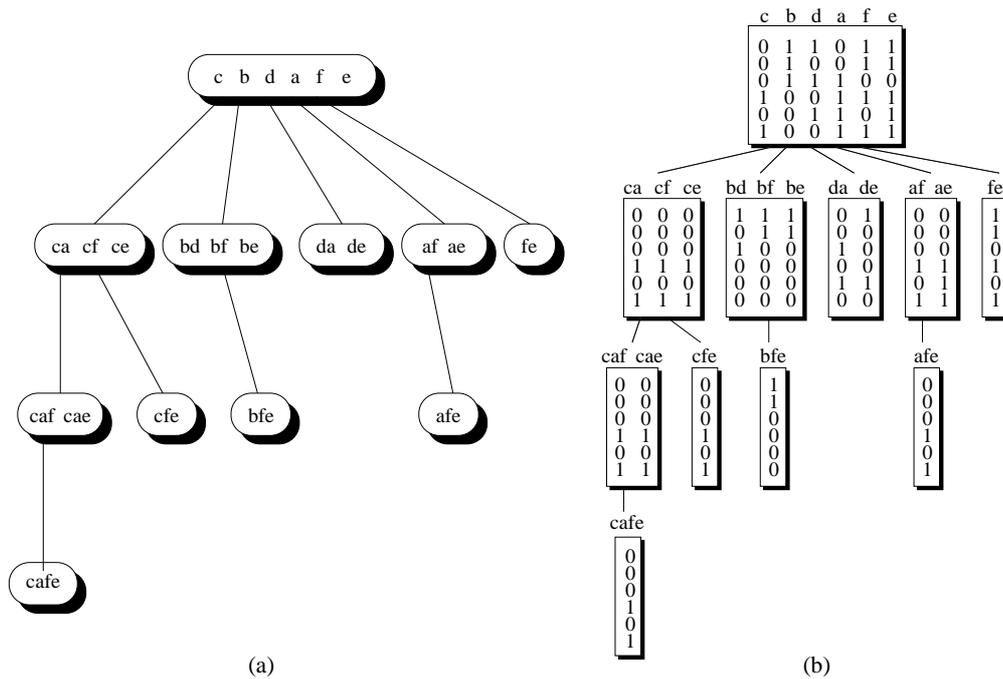


FIGURE 1.6: Mining frequent itemsets by clustered bitmaps. (a) Cluster tree of frequent itemsets. (b) Clustered bitmaps.

Given the initial FI-cluster \mathcal{C}_0 , all frequent itemsets can be generated by traversing the cluster tree top down. Bit-vectors are used to compute the supports of the generated itemsets. The count of an itemset in the database is equal to the number of 1's contained in its bit-vector. Since the bitwise *AND* of the bit-vectors for two itemsets results in the bit-vector of the joined itemset, the clustered bitmap of a given FI-cluster sufficiently contains the count information for all the itemsets and their combinations. So a subtree of FI-clusters can be independently mined from the root FI-cluster and its clustered bitmap, by generating the progeny FI-clusters and their associated clustered bitmaps as follows. When pairs of itemsets in a parent FI-cluster are joined to generate a child FI-cluster as described in the definition, the corresponding bit-vectors from the parent's clustered bitmap are also operated via bitwise *AND* to form the child's clustered bitmap. Figure 1.6(b) shows how the clustered bitmaps are bound with their FI-clusters so that all frequent itemsets can be mined with supports computed along the cluster tree hierarchy.

In most cases, the cluster bitmaps may be too big to be processed efficiently, as they could be very sparse and contain too many obvious patterns as in *e-rows*, *a-rows*, *p-rows*, *o-rows* and *c-rows*. So *dCLUB*'s are used in place of clustered bitmaps. To make *e-rows* and *p-rows* take the majority in a clustered bitmap, the itemsets in the FI-cluster can be reordered by ascending counts of 1's in their bit columns. With most of the sparse as well as dense rows removed, the size of the bitmap can be cut down by several orders of magnitude. Generating *dCLUB*'s along the cluster tree hierarchy is basically the same as doing clustered bitmaps, except that those patterned rows are to be removed. Removed rows are digested and turned into partial supports of the itemsets to be mined, through a number of propagation counters that can carry over from parent to children clusters. The

support of an itemset is then computed by combining the count obtained from the *dCLUB* and that from the propagation counters.

In practice, the algorithm starts by building the *dCLUB*'s for the level-2 clusters (FI-clusters of frequent 2-itemsets) from the original database. It then recursively refines each of the *dCLUB*'s along the cluster tree hierarchy, via bitwise *AND* of the corresponding bit columns. After each refinement, only selected rows and columns of the result bitmap are to be further refined. Selection of rows is achieved by the removal of those patterned rows, while selection of columns is by retaining only the columns for frequent itemsets. That gives the result *dCLUB* actually so that it can be recursively refined. Propagation counters are incrementally accumulated along the traversing paths of the cluster tree when the patterned rows are digested and removed. The *dCLUB*'s are organized into a two-dimensional matrix of integers, where each bit column is grouped into a number of 32-bit integers. So generating children *dCLUB*'s from their parents is performed by fast aggregate bit operations in arrays. Since the FI-clusters are closely bound to their *dCLUB*'s and the associated propagation counters, refinement of the *dCLUB*'s will directly generate the frequent itemsets with exact supports. The refinement stops where the *dCLUB*'s become empty, and all frequent itemsets in the subtree rooted at the corresponding FI-cluster can then be inferred, with supports calculated directly from the propagation counters.

The following steps outline the spirit of the *PD-CLUB* algorithm for shared-memory multiprocessors.

- (1) Scan the database in parallel by horizontal partitions and mine frequent 1-itemsets and 2-itemsets by clusters;
- (2) For clusters of frequent 2-itemsets, build their partial *dCLUB*'s over the local database partition on each process, recording local propagation counters at the same time;
- (3) Combine the partial *dCLUB*'s into global ones for the level-2 clusters, and sum up the local propagation counters to get the global ones for each of the itemsets;
- (4) Sort the level-2 clusters by estimated workloads in descending order;
- (5) Assign each level-2 cluster in turn to one of the free processes and recursively refine its *dCLUB* to mine all frequent itemsets in the FI-cluster subtree rooted at that cluster.

In step (1), the count distribution method is used in mining frequent 1-itemsets and 2-itemsets. The *dCLUB*'s and propagation counters for the level-2 clusters are initially set up in steps (2) and (3). The subsequent mining tasks are dynamically scheduled on multiple processes, as in steps (4) and (5). The granularity of task assignment is based on the workload associated with mining the whole FI-cluster subtree rooted at each of the level-2 clusters. The workload for each cluster can be estimated as the size of the *dCLUB* that is to be refined. Due to the mining independence between subtrees belonging to different branches of the cluster tree, each of the assigned tasks can be independently performed on the designated process. Each of the subtrees is mined recursively in a depth-first way on a single process to better utilize the cache locality. Since tasks are assigned in a manner from coarse to fine grain, workload balance can be fairly well kept among all processes. However, it may happen that there are not enough clusters to be assigned to some free processes while others are busy with some extra work. To address this issue, a fine tune measure can be taken in step (5) to reassign branches of clusters to be mined on a busy process to a free one. It works as follows. The initial tasks from step (4) are added to an assignment queue and assigned to processes as usual. Whenever a process finds the queue empty after

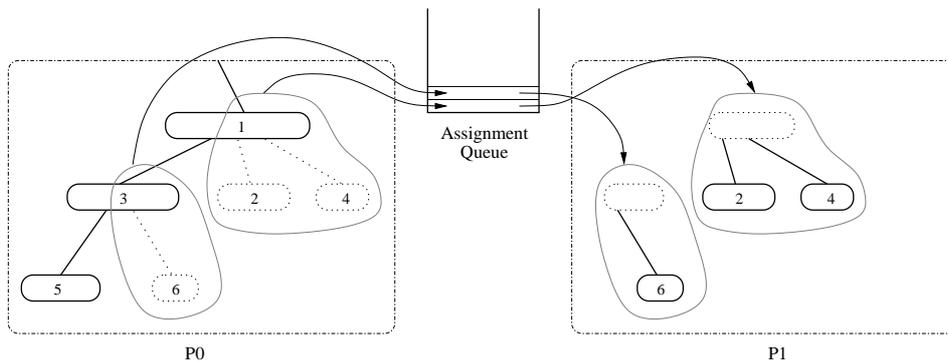


FIGURE 1.7: Dynamic cluster reassignment. Extra cluster subtrees originally to be mined by process P0 (which is overloaded) are reassigned for the free process P1 to mine, via a global assignment queue. Clusters in solid lines are mined locally, while those in dotted lines are mined on remote processes. Clusters are numbered by the order in which they are generated in full.

it generates a new cluster, it goes on to mine only the leftmost subtree of the cluster. The further mining task for the right part of that cluster is added to the queue so that it can be reassigned for some free process to mine all other subtrees. After the queue becomes filled again, all processes settle back to independent mining of their assigned cluster subtrees in full in a depth-first way. Figure 1.7 shows such an example of cluster reassignment among 2 processes. At some moment when the assignment queue is empty, process P1 becomes free while process P0 is mining cluster 1. After generating cluster 1, P0 adds the right part of it to the queue and continues to mine only its left subtree, i.e., clusters 3, 5 and 6. At the same time, P1 gets the new assignment to mine clusters 2 and 4, and the queue becomes empty again. Then cluster 2 and 3 are generated successively on P1 and P0. Cluster 2 does not have subtrees while cluster 3 has some. So P1 continues mining cluster 4, and P0 moves forward to mine cluster 5, adding right part of cluster 3 to the queue for P1 to mine cluster 6 later. P0 and P1 do approximately equal amounts of work and finish roughly at the same time.

For distributed-memory multiprocessors, due to the expensive synchronization and communication cost, the algorithm can use a static task scheduling strategy instead to assign tasks as equally as possible at the very beginning. Then each process can perform their tasks independently without communicating or being synchronized with other processes. The basic steps can be expressed as follows.

- (1) Scan the database in parallel by horizontal partitions and mine frequent 1-itemsets and 2-itemsets by clusters;
- (2) For clusters of frequent 2-itemsets, build their partial *dCLUB*'s over the local partition of database on each process;
- (3) Initially assign the level-2 clusters to processes;
- (4) For the assigned clusters on each process, combine the partial *dCLUB*'s from all processes to get the global ones;
- (5) Each process recursively refines the *dCLUB* to mine all frequent itemsets for each assigned cluster.

The first two steps work the same way as previously. In step (3), the mining tasks for the cluster subtrees rooted at the level-2 clusters are pre-scheduled on all processes. A greedy scheduling algorithm can be used to sort the level-2 clusters by estimated workloads in descending order and then assign the clusters in turn to the least loaded process. Communication is needed in step (4) for all processes to exchange partial *dCLUB*'s. The global *dCLUB*s for the level-2 clusters are constructed only on the designated processes so that the refinement of each *dCLUB* in step (5) can be performed independently. For a workload balancing purpose, cluster reassignment is possible by sending a FI-cluster, its *dCLUB* and the associated propagation counters as a unit from one process to another. Then the cluster subtree rooted at that FI-cluster can be mined on the second process instead of on the originally designated one. However, since the tasks are already pre-scheduled, the ongoing processes need a synchronization mechanism to detect the moment when cluster reassignment is needed and release some tasks for rescheduling. A communication mechanism is also needed for the source process to send data to the destination process. A dedicated communication thread can be added to each process for such purposes.

1.2.5 Comparison

In general performance, experiments show that *D-CLUB* performs the best, followed by *FP-growth* and *Eclat*, with *Apriori* doing the worst. Similar performance ranking holds for their parallel versions. However, each of them has its own advantages and disadvantages.

Among all of the parallel ARM algorithms, the *Apriori*-based algorithms are the most widely used because of the simplicity and easy implementation. Also association rules can be directly generated on the way of itemset mining, because all the subset information is already computed when candidate itemsets are generated. These algorithms scale well with the number of transactions, but may have trouble handling too many items and/or numerous patterns as in dense databases. For example, in the *Count Distribution* method, if the number of candidate/frequent itemsets grows beyond what can be held in the main memory of each processor, the algorithm can not work well no matter how many processors are added. The performance of these algorithms is dragged behind mainly by the slow itemset counting procedure that repeatedly searches the profuse itemsets against the large amount of transactions.

The *Eclat*-based algorithms have the advantage of fast support computing through tid-list intersection. By independent task parallelism, they gain very good speedups on distributed-memory multiprocessors. The main drawback of these algorithms is that they need to generate and redistribute the vertical tid-lists of which the total size is comparable to that of the original database. Also, for a long frequent itemset, the major common parts of the tid-lists are repeatedly intersected for all its subsets. To alleviate this situation, diffset optimization [ZG03] has been proposed to track only the changes in tid-lists instead of keeping the entire tid-lists through iterations so that it can significantly reduce the amount of data to be computed.

Parallel *FP-growth* handles dense databases very efficiently and scales particularly well with the number of transactions, benefiting from the fact that repeated or partially repeated transactions will be merged into paths of the FP-trees any way. However, this benefit does not increase accordingly with the number of processes, because multiple FP-trees for different sets of transactions are purely redundant. The benefit is also very limited for sparse databases with a small number of patterns scattered. The algorithm can handle a large number of items by just assigning them to multiple processes, without worrying about the exponentially large space of item/itemset combinations.

The *PD-CLUB* algorithm is self-adaptive to the database properties, and can handle both

dense and sparse databases very efficiently. With the data size and representation fundamentally improved in the differential clustered bitmaps, the mining computation is also substantially reduced and simplified into fast aggregate bit operations in arrays. Compared to parallel *Eclat*, the *dCLUB*'s used in *PD-CLUB* have much smaller sizes than the tid-lists or even the diffsets, which results in much less communication cost when the *dCLUB*'s need to be exchanged among processes. The independent task parallelism plus the dynamic workload balancing mechanism gives the algorithm near linear speedups on multiple processes.

1.3 Parallel Clustering Algorithms

Clustering is to group data objects into classes of similar objects based on their attributes. Each class, called a cluster, consists of objects that are similar between themselves and dissimilar to objects in other classes. The dissimilarity or distance between objects is measured by the given attributes that describe each of the objects. As an unsupervised learning method, clustering is widely used in many applications, such as pattern recognition, image processing, gene expression data analysis, market research, and so on.

Existing clustering algorithms can be categorized into partitioning, hierarchical, density-based, grid-based and model-based methods [HK00], each generating very different clusters for various applications. Representative algorithms are introduced and their parallelizations are studied in this section.

1.3.1 Parallel *k-means*

As a partitioning method, the *k-means* algorithm [Mac67] takes the input parameter, k , and partitions a set of n objects into k clusters with high intra-cluster similarity and low inter-cluster similarity. It starts by randomly selecting k objects as the initial cluster centroids. Each object is assigned to its nearest cluster based on the distance between the object and the cluster centroid. It then computes the new centroid (or mean) for each cluster. This process is repeated until the sum of squared-error (*SSE*) for all objects converges. The *SSE* is computed by summing up all the squared distances, one between each object and its nearest cluster centroid.

In parallel *k-means* [DM00], data parallelism is used to divide the workload evenly among all processes. Data objects are statically partitioned into blocks of equal sizes, one for each process. Since the main computation is to compute and compare the distances between each object and the cluster centroids, each process can compute on its own partition of data objects independently if the k cluster centroids are maintained on all processes. The algorithm is summarized in the following steps.

- (1) Partition the data objects evenly among all processes;
- (2) Select k objects as the initial cluster centroids;
- (3) Each process assigns each object in its local partition to the nearest cluster, computes the *SSE* for all local objects, and sums up local objects belonging to each cluster;
- (4) All processes exchange and sum up the local *SSE*'s to get the global *SSE* for all objects and compute the new cluster centroids;
- (5) Repeat (3) - (5) until no change in the global *SSE*.

The algorithm is proposed on distributed-memory multiprocessors but works similarly on

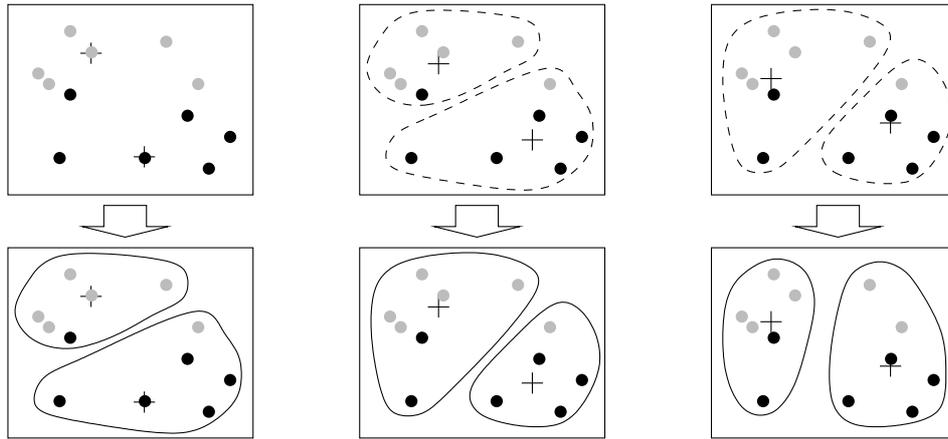


FIGURE 1.8: k -means clustering ($k = 2$) on two processes. One process computes on the objects in gray color, and the other is in charge of those in black. Cluster centroids, marked by '+', are maintained on both processes. Circles in solid lines denote the cluster formation in each iteration, based on the current cluster centroids. Dashed circles mark the previous cluster formation that is used to compute new cluster centroids.

shared-memory systems as well. Step (3) is the major computation step where clusters are formed. Objects belonging to one cluster may be distributed over multiple processes. In step (4), each of the new cluster centroid is computed in the same way as the global SSE for all objects, except that the summational result will be further divided by the count of objects in the cluster in order to get the mean. Figure 1.8 shows an example of k -means clustering for $k = 2$. The data objects are partitioned among 2 processes and the two clusters are identified through three iterations.

In parallel k -means, the workloads per iteration are fairly well balanced between processes, which results in linear speedups when the number of data objects is large enough. Between iterations, there is a small communication/synchronization overhead for all processes to exchange the local SSE 's and the local member object summations for each cluster. The algorithm needs a full scan of the data objects in each iteration. For large disk-resident data sets, having more processors could result in super-linear speedups as each data partition then may be small enough to fit in the main memory, avoiding disk I/O except in the first iteration.

1.3.2 Parallel Hierarchical Clustering

Hierarchical clustering algorithms are usually applied to bioinformatics procedures such as grouping of genes and proteins with similar structure, reconstruction of evolutionary trees, gene expression analysis, etc. An agglomerative approach is commonly used to recursively merge pairs of closest objects or clusters into new clusters until all objects are merged into one cluster or until a termination condition is satisfied. The distance between two clusters can be determined by single link, average link, complete link or centroid-based metrics. The single link metric uses the minimum distance between each pair of inter-cluster objects, average link uses the average distance, and complete link uses the maximum. The centroid-based metric uses the distance between the cluster centroids. Figure 1.9 gives an example

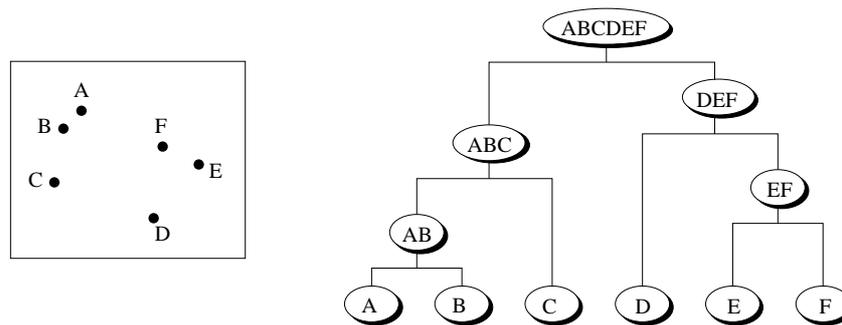


FIGURE 1.9: Example of hierarchical clustering.

of agglomerative hierarchical clustering using the single link metric. The dendrogram on the right shows which clusters are merged at each step.

The hierarchical merge of clusters can be parallelized by assigning clusters to different processes so that each process will be in charge of a disjoint subset of clusters, as in [Ols95]. When two clusters are merged, they are released from the owner processes, and the new cluster will be assigned to the least loaded process. The basic parallel algorithm for agglomerative hierarchical clustering takes the following steps.

- (1) Initially treat each data object as a cluster and assign clusters among all processes evenly so that each process is responsible for a disjoint subset of clusters;
- (2) Calculate the distance between each pair of clusters in parallel, maintaining a nearest neighbor for each cluster;
- (3) All processes synchronously find the closest pair of clusters, agglomerate them into a new cluster, and assign it to the least loaded process;
- (4) Calculate the distance between the new cluster and each of the remaining old clusters and update the nearest neighbor for each cluster on the responsible process;
- (5) Repeat (3) - (5) until there is only one cluster remaining or certain termination conditions are satisfied.

The algorithm works for both shared and distributed memory multiprocessors, though the latter case requires all processes to exchange data objects in the initialization step of (2). It is crucial to design an appropriate data structure for the parallel computation of the all-pair distances between clusters. For steps (2), (3) and (4), a two-dimensional array can be used to store distances between any two clusters. It is distributed over all processes such that each one is responsible for those rows corresponding to its assigned clusters. Similarly, the nearest neighbor of each cluster and the associated distance are maintained in a one-dimensional array, with the responsibility divided among processes accordingly. The closest pair of clusters can then be easily determined based on the nearest neighbor distances. Step (3) is a synchronous phase so that all processes know which two clusters are merged. After two clusters are merged into a new cluster, each process only updates the assigned rows to get the distances from the assigned clusters to the new cluster. Specifically, the distance from cluster i to the new cluster is computed from the distances between cluster i and the two old clusters that are merged, e.g., by taking the less value when the single link metric

		Intercluster Distances					Nearest Neighbor	
		A	B	C	D	E		F
P0	A		10	38	60	60	41	B: 10
	B	10		26	60	62	44	A: 10
	C	38	26		50	66	50	B: 26
P1	D	60	60	50		38	39	E: 38
	E	60	62	66	38		20	F: 20
	F	41	44	50	39	20		E: 20

(a) Before Merge

		Intercluster Distances					Nearest Neighbor
		AB	C	D	E	F	
P0	AB		26	60	60	41	C: 26
	C	26		50	66	50	AB: 26
P1	D	60	50		38	39	E: 38
	E	60	66	38		20	F: 20
	F	41	50	39	20		E: 20

(b) After Merge

FIGURE 1.10: Compute/update inter-cluster distances and nearest neighbors in parallel over 2 process. From (a) to (b), clusters “A” and “B” are merged into a new cluster “AB”. In (b), only entries in bold font require new computation or update.

is used. The nearest neighbor information is then recomputed, taking into account the newly computed distances. The assignment of the new cluster requires estimation of the workload on each process. For simplicity, the number of assigned clusters can be used as the estimated workload, which is pretty effective. Taking Figure 1.9 as an example, Figure 1.10 illustrates how the all-pair distances between clusters and the neighbor information are divided for 2 process to compute in parallel, before and after the first merge of clusters. The new cluster “AB” is assigned to process P0.

By task parallelism and employing a dynamic workload balancing strategy, this algorithm can achieve good speedups when the number of processes is much less than that of clusters. However, there is some communication and/or synchronization overhead in every step of cluster agglomeration, because all processes have to obtain the global information to find the minimum distance between clusters and also have to keep every process informed after the new cluster has been formed. The algorithm assumes the data structures being able to be kept in main memory so that it scans the data set only once.

1.3.3 Parallel HOP: Clustering Spatial Data

HOP [EH98], a density-based clustering algorithm proposed in astrophysics, identifies groups of particles in N-body simulations. It first constructs a KD tree by recursively bisecting the particles along the longest axis so that nearby particles reside in the same sub-domain. Then it estimates the density of each particle by its N_{dens} nearest neighbors that can be efficiently found by traverse of the KD tree. Each particle is associated to its densest neighbor within its N_{hop} nearest neighbors. A particle can hop to the next densest particle and continues hopping until it reaches a particle that is its own densest neighbor. Finally, *HOP* derives clusters from groups that consist of particles associated to the same densest neighbor. Groups are merged if they share a sufficiently dense boundary, according to some given density threshold. Particles whose densities are less than the density threshold are excluded from groups.

Besides the cosmological N-body problem, *HOP* may find its application in other fields, such as molecular biology, geology and astronomy, where large spatial data sets are to be processed with similar clustering or neighbor finding procedures.

To parallelize the *HOP* clustering process, the key idea is to distribute the data particles across all processes evenly with proper data placement so that the workloads are balanced and communication cost for remote data access is minimized. The following steps explain the

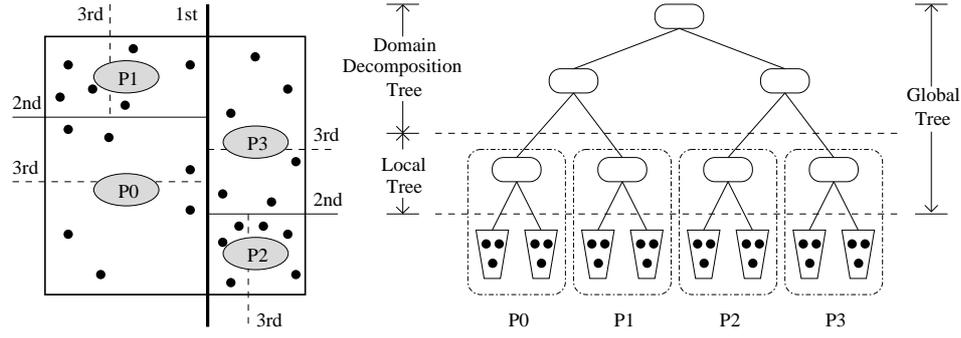


FIGURE 1.11: Two-dimensional KD tree distributed over 4 processes. Each process contains 6 particles. Bucket size is 3 and the global tree has 3 levels. Local tree can be built concurrently without communication. Every process maintains the same copy of the global tree.

parallel *HOP* algorithm [LLC03] in detail, assuming distributed-memory multiprocessors.

- (1) **Constructing a Distributed KD Tree:** The particles are initially distributed among all processes in blocks of equal sizes. Starting from the root-node of the KD tree, the algorithm first determines the longest axis d and then finds the median value m of all particles' d coordinates in parallel. The whole spatial domain is bisected into two sub-domains by m . Particles are exchanged between processes such that the particles whose d coordinates are greater than m go to one sub-domain and the rest of the particles to the other one. Therefore, an equal number of particles are maintained in each sub-domain after the bisection. This procedure is repeated recursively in every sub-domain till the number of sub-domains is equal to the number of processes. Then, each process continues to build its own local tree within its domain until the desired bucket size (number of particles in each leaf) is reached. Note that inter-process communication is not required in the construction of the local trees.

A copy of the whole tree is maintained on every process so that the communication overhead incurred at performing search domain intersection test with the remote local trees at the stages 2 and 3 can be reduced. Therefore, at the end of this stage, local trees are broadcasted to all processes. As shown in Figure 1.11, the root-node of the KD tree represents the entire simulation domain while each of the rest tree nodes represent a rectangular sub-domain of its parent node. The information contained in a non-leaf tree node includes the aggregated mass, center of mass, number of particles, and domain boundaries. When the KD tree is completed, particles are divided into spatially-closed regions of approximately equal number. The advantage of using a KD tree is not only its simplicity but also the balanced data distribution.

- (2) **Generating Density:** The density of a particle is estimated by its N_{dens} nearest neighbors, where N_{dens} is a user-specified parameter. Since it is possible that some of the N_{dens} neighbors of a particle are owned by remote processes, communication is required to access non-local neighbor particles at this stage. One effective approach is, for each particle, to perform an intersection test by traversing the global tree with a given initial search radius r , while keeping track

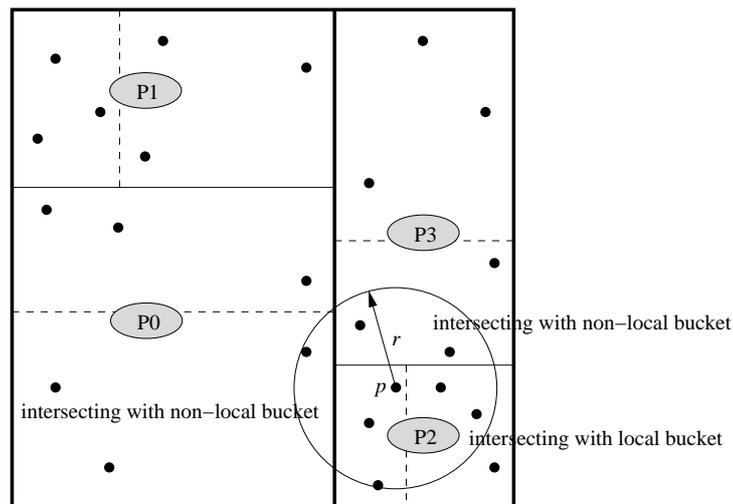


FIGURE 1.12: Intersection test for particle p on process P2 for $N_{dens} = 7$. The neighborhood is a spherical region with a search radius r . The neighborhood search domain of p intersects with the sub-domains of P0 and P3.

of the non-local intersected buckets, as shown in Figure 1.12. If the total number of particles in all intersected buckets is less than N_{dens} , the intersection test is re-performed with a larger radius. Once tree walking is completed for all local particles, all the remote buckets containing the potential neighbors are obtained through communication. Note that there is only one communication request to each remote process to gather the intersected buckets. No further communication is necessary when searching for its N_{dens} nearest neighbors. Since the KD tree displays the value of spatial locality, particle neighbors are most likely located in the same or nearby buckets. According to the experimental results, the communication volume is only 10%-20% of the total number of particles. However, with highly irregular particle distribution, communication costs may increase.

To calculate the density for particle p , the algorithm uses a PQ tree (priority queue) [CLR90] to maintain a sorted list of particles that are currently the N_{dens} nearest neighbors. The root of the PQ tree contains the neighbor farthest from p . If a new neighbor whose distance to p is shorter than the root, replace the root with the second farthest one and update the PQ tree. Finally, the particles remained in the PQ tree are the N_{dens} nearest neighbors of p .

- (3) **Hopping:** This stage first associates each particle to its highest density neighbor among its N_{hop} nearest neighbors that are already stored in the PQ tree generated at the previous stage. Each particle, then, hops to the highest density neighbor of its associated neighbor. Hopping to remote particles is performed by first keeping track of all the remote particles and then by making a communication request to the owner processes. This procedure may repeat several times until all the needed non-local particles are already stored locally. Since the hopping is in density increasing order, the convergence is guaranteed.
- (4) **Grouping:** Particles linked to the same densest particle are defined as a group. However, some groups should be merged or refined according to the chosen den-

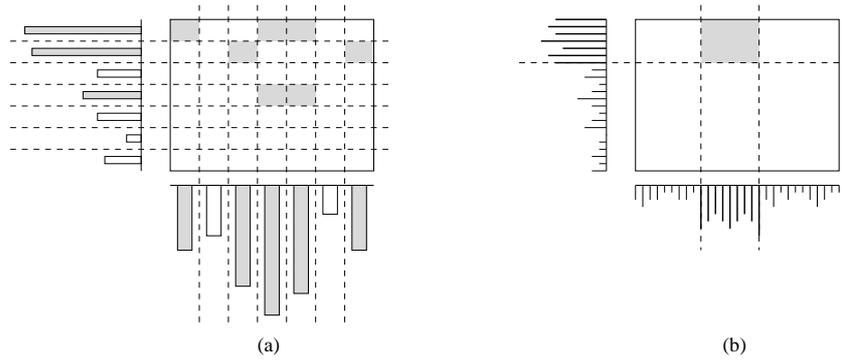


FIGURE 1.13: Identify dense units in two dimensional grids. (a) uses a uniform grid size, while (b) uses adaptive grid sizes by merging fine bins into adaptive grid bins in each dimension. Histograms are built for all dimensions.

sity thresholds. Thus, every process first builds a boundary matrix for the groups constructed from its local particles and then exchanges the boundary matrix among all processes. Particles whose densities are less than a given threshold are excluded from groups and two groups are merged if their boundary satisfies some given thresholds.

The parallel *HOP* clustering algorithm distributes particle data evenly among all processes to guarantee balanced workload. It scans the data set only once and stores all the particle data in the distributed KD tree over multiple processes. This data structure helps to minimize inter-process communication as well as improves the neighbor search efficiency, as the spatially closed particles are usually located in the same buckets or a very few neighbor buckets. The communication cost comes from the particle redistribution during the KD tree construction and the remote particle access in the neighbor-based density generation. Experiments showed that it gained good speedups on a number of different parallel machines.

1.3.4 Clustering High-Dimensional Data

For high-dimensional data, grid-based clustering algorithms are usually used. *CLIQUE* [AGGR98] is one of such algorithms. It partitions the n -dimensional data space into nonoverlapping rectangular units of uniform size, identifying the dense units among these. Based on a user-specified global density threshold, it iteratively searches dense units in the subspaces from 1-dimension through k -dimension until no more dense units are found. The generation of candidate subspaces is based on the *Apriori property* used in association rule mining. The dense units are then examined to form clusters.

The *pMAFIA* algorithm [NGC01] improves *CLIQUE* by using adaptive grid sizes. The domain of each dimension is partitioned into variable sized adaptive grid bins that capture the data distribution. Also variable density thresholds are used, one for each bin. Adaptive dense units are then found in all possible subspaces. A unit is identified as dense if its population is greater than the density thresholds of all the bins that form the unit. Each dense unit of dimension d can be specified by the d dimensions and their corresponding d bin indices. Figure 1.13 illustrates the dense unit identification for both uniform grid size

and adaptive grid sizes.

PMAFIA is one of the first algorithms that demonstrate a parallelization of subspace clustering for high-dimensional large-scale data sets. Targeting distributed-memory multiprocessors, it makes use of both data and task parallelism. The major steps are listed below.

- (1) Partition the data objects evenly among all processes;
- (2) For each dimension, by dividing the domain into fine bins, each process scans the data objects in its local partition and builds a local histogram independently;
- (3) All processes exchange and sum up the local histograms to get the global one for each dimension;
- (4) Determine adaptive intervals using the global histogram in each dimension and set the density threshold for each interval;
- (5) Each process finds candidate dense units of current dimensionality (initially 1) and scans data objects in its local partition to populate the candidate dense units;
- (6) All processes exchange and sum up the local populations to get the global one for each candidate dense unit;
- (7) Identify dense units and build their data structures;
- (8) Increase the dimensionality and repeat (5) - (8) until no more dense units are found;
- (9) Generate clusters from identified dense units.

The algorithm spends most of its time in making repeated passes over the data objects and finding out the dense units among the candidate dense units formed from 1-dimensional to k -dimensional subspaces until no more dense units are found. Building the histograms in step (2) and populating the candidate dense units in step (5) are performed by data parallelism such that each process scans only a partition of data objects to compute the local histograms and local populations independently. After each of the independent steps, the local values are collected from all processes and add up to the global values, as in steps (3) and (6) respectively. In step (4), all processes perform the same adaptive grid computation from the global histograms they gathered. The adaptive grid bins in each dimension are then generated, each considered to be a candidate dense unit of dimension 1. Candidate dense units from subspaces of higher dimensions need to be generated in step (5). Similar to the candidate subspace generation procedure in *CLIQUE*, *pMAFIA* generates candidate dense units in any dimension k by combining dense units of dimension $k-1$ such that they share any $k-2$ dimensions. Since each pair of the dense units needs to be checked for possible intersection into a candidate dense unit, the amount of work is known and the task can be easily subdivided among all processes such that each one intersects an equal number of pairs. Task parallelism is also used in step (7) to identify dense units from the populated candidates such that each process scans an equal number of candidate dense units.

pMAFIA is designed for disk-resident data sets and scans the data as many times as the maximum dimension of the dense units. By data parallelism, each partition of data is stored in the local disk once it is retrieved from the remote shared disk by a process. That way, subsequent data accesses can see a much larger I/O bandwidth. It uses fine bins to build the histogram in each dimension, which results in large sizes of histograms and could add to the communication cost for all processes to exchange local histograms. Compared to the major

time spent in populating the candidate dense units, this communication overhead can be negligible. Also, by using adaptive grids that automatically capture the data distribution, the number of candidate dense units is minimized, so is the communication cost in the exchange of their local populations among all processes. Experiments showed that it could achieve near linear speedups.

1.4 Summary

As data accumulates in bulk volumes and goes beyond the processing power of single-processor machines, parallel data mining techniques become more and more important for scientists as well as business decision-makers to extract concise, insightful knowledge from the collected information in an acceptable amount of time. In this chapter, various algorithms for parallel association rule mining and clustering are studied, spanning distributed and shared memory systems, data and task parallelism, static and dynamic workload balancing strategies, and so on. Scalability, inter-process communication/synchronization, workload balance, and I/O issues are discussed for these algorithms. The key factors that affect the parallelism varies for different problems or even different algorithms of the same problem. For example, due to the dynamic nature of association rule mining, the workload balance is a big issue for many algorithms that use static task scheduling mechanisms. And for the density-based parallel *HOP* clustering algorithms, the focus of effort should be put on minimizing the data dependence across processes. By efficiently utilizing the aggregate computing resources of parallel processors and minimizing the inter-process communication/synchronization overhead, high-performance parallel data mining algorithms can be designed to handle massive data sets.

While extensive research has been done in this field, a lot of new exciting work is being explored for future development. With the rapid growth of distributed computing systems, the research on distributed data mining has become very active. For example, the emerging pervasive computing environment is becoming more and more popular, where each ubiquitous device is a resource-constrained distributed computing device. Data mining research in such systems is still in its infancy but most algorithm design can be theoretically based on existing parallel data mining algorithms. This chapter can serve as a reference for the state of the art for both researchers and practitioners who are interested in building parallel and distributed data mining systems.

References

References

- [AGGR98] Rakesh Agrawal, Johannes Gehrke, Dimitrios Gunopulos, and Prabhakar Raghavan. Automatic subspace clustering of high dimensional data for data mining applications. In *Proc. of the ACM SIGMOD Int'l Conf. on Management of Data*, pages 94–105, June 1998.
- [AIS93] Rakesh Agrawal, Tomasz Imielinski, and Arun N. Swami. Mining association rules between sets of items in large databases. In *Proc. of the ACM SIGMOD Int'l Conf. on Management of Data*, pages 207–216, May 1993.
- [AS94] Rakesh Agrawal and Ramakrishnan Srikant. Fast algorithms for mining association rules. In *Proc. of the 20th Int'l Conf. on Very Large Databases*, pages 487–499, September 1994.

- [AS96] Rakesh Agrawal and John C. Shafer. Parallel mining of association rules. *IEEE Trans. on Knowledge and Data Engineering*, 8(6):962–969, December 1996.
- [CLR90] Thomas T. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to algorithms*. MIT Press, Cambridge, MA, USA, June 1990.
- [DM00] Inderjit S. Dhillon and Dharmendra S. Modha. A data-clustering algorithm on distributed memory multiprocessors. *Large-Scale Parallel Data Mining, Lecture Notes in Artificial Intelligence*, 1759:245–260, 2000.
- [EH98] Daniel J. Eisenstein and Piet Hut. Hop: A new group-finding algorithm for n-body simulations. *Journal of Astrophysics*, 498:137–142, 1998.
- [HK00] Jiawei Han and Micheline Kamber. *Data mining: concepts and techniques*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, August 2000.
- [HKK00] Eui-Hong Han, George Karypis, and Vipin Kumar. Scalable parallel data mining for association rules. *IEEE Trans. on Knowledge and Data Engineering*, 12(3):337–352, 2000.
- [HPY00] Jiawei Han, Jian Pei, and Yiwen Yin. Mining frequent patterns without candidate generation. In *Proc. of the ACM SIGMOD Int'l Conf. on Management of Data*, pages 1–12, May 2000.
- [LCJL06] Jianwei Li, Alok Choudhary, Nan Jiang, and Wei-keng Liao. Mining frequent patterns by differential refinement of clustered bitmaps. In *Proc. of the SIAM Int'l Conf. on Data Mining*, April 2006.
- [LLC03] Ying Liu, Wei-keng Liao, and Alok Choudhary. Design and evaluation of a parallel HOP clustering algorithm for cosmological simulation. In *Proc. of the 17th Int'l Parallel and Distributed Processing Symposium*, April 2003.
- [Mac67] James B. MacQueen. Some methods for classification and analysis of multivariate observations. In *Proc. of the 5th Berkeley Symposium on Mathematical Statistics and Probability*, volume 1, pages 281–297, 1967.
- [NGC01] Harsha Nagesh, Sanjay Goil, and Alok Choudhary. Parallel algorithms for clustering high-dimensional large-scale datasets. In Robert Grossman, Chandrika Kamath, Philip Kegelmeyer, Vipin Kumar, and Raju Namburu, editors, *Data Mining for Scientific and Engineering Applications*, pages 335–356. Kluwer Academic Publishers, 2001.
- [Ols95] Clark F. Olson. Parallel algorithms for hierarchical clustering. *Parallel Computing*, 21:1313–1325, 1995.
- [PK03] Iko Pramudiono and Masaru Kitsuregawa. Tree structure based parallel frequent pattern mining on PC cluster. In *Proc. of the 14th Int'l Conf. on Database and Expert Systems Applications*, pages 537–547, September 2003.
- [Zak99] Mohammed J. Zaki. Parallel and distributed association mining: A survey. *IEEE Concurrency*, 7(4):14–25, 1999.
- [ZEHL01] Osmar R. Zaiane, Mohammad El-Hajj, and Paul Lu. Fast parallel association rule mining without candidacy generation. In *Proc. of the IEEE Int'l Conf. on Data Mining*, November 2001.
- [ZG03] Mohammed J. Zaki and Karam Gouda. Fast vertical mining using diffsets. In *Proc. of the ACM SIGKDD Int'l Conf. on Knowledge Discovery and Data Mining*, pages 326–335, August 2003.
- [ZOPL96] Mohammed J. Zaki, Mitsunori Ogihara, Srinivasan Parthasarathy, and Wei Li. Parallel data mining for association rules on shared-memory multi-processors. In *Proc. of the ACM/IEEE Conf. on Supercomputing*, November 1996.
- [ZPL97] Mohammed J. Zaki, Srinivasan Parthasarathy, and Wei Li. A localized algorithm for parallel association mining. In *Proc. of the 9th ACM Symposium on Parallel Algorithms and Architectures*, pages 321–330, June 1997.

- [ZPOL97a] Mohammed J. Zaki, Srinivasan Parthasarathy, Mitsunori Ogihara, and Wei Li. New algorithms for fast discovery of association rules. Technical Report TR651, University of Rochester, July 1997.
- [ZPOL97b] Mohammed J. Zaki, Srinivasan Parthasarathy, Mitsunori Ogihara, and Wei Li. Parallel algorithms for discovery of association rules. *Data Mining and Knowledge Discovery: An International Journal, special issue on Scalable High-Performance Computing for KDD*, 1(4):343–373, December 1997.

Index

Association Rule Mining, 1-2–1-14

- Apriori* algorithm, 1-2
- Count Distribution*, 1-2
- Eclat* algorithm, 1-4
- FP-growth* algorithm, 1-6
- PD-CLUB* algorithm, 1-9
- bit-vector, 1-9
- clustered bitmap, 1-9
- conditional pattern base, 1-6
- FP-tree, 1-6
- frequent itemset, 1-2
- tid-list, 1-4

Clustering, 1-14–1-22

- k-means* algorithm, 1-14
- CLIQUE* algorithm, 1-20
- HOP* algorithm, 1-17
- pMAFIA* algorithm, 1-20
- hierarchical clustering, 1-15

KD tree, 1-17