2009

Parallel Mining of Association Rules Using a Lattice Based Approach

Wessel Morant Thomas
Nova Southeastern University, wesselth@gmail.com

Parallel Mining of Association Rules Using a Lattice Based Approach

By

Wessel Thomas

A dissertation submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy
in
Computer Science

Graduate School of Computer and Information Sciences
Nova Southeastern University

2009

We hereby certify that this dissertation, submitted by Wessel M. Thomas, conforms to acceptable standards and is fully adequate in scope and quality to fulfill the dissertation requirements for the degree of Doctor of Philosophy.

Junping Sun, Ph.D. Chairperson of Dissertation Committee	Date
Mike Laszlo, Ph.D. Dissertation Committee Member	Date
James Cannady, Ph.D. Dissertation Committee Member	Date

Approved:

Edward Lieblein, Ph.D. Dean	Date

Graduate School of Computer and Information Sciences
Nova Southeastern University

2009

An Abstract of a Dissertation Submitted to Nova Southeastern University
in Partial Fulfillment of the Requirements for the Degree of Doctor of Philosophy

Parallel Mining of Association Rules Using a Lattice Based Approach

By
Wessel M. Thomas

2009

The discovery of interesting patterns from database transactions is one of the major problems in knowledge discovery in database. One such interesting pattern is the association rules extracted from these transactions. The goal of this research was to develop and implement a parallel algorithm for mining association rules. We implemented a parallel algorithm that used a lattice approach for mining association rules. The Dynamic Distributed Rule Mining (DDRM) is a lattice-based algorithm that partitions the lattice into sublattices to be assigned to processors for processing and identification of frequent itemsets. We implemented the DDRM using a dynamic load balancing approach to assign classes to processors for analysis of these classes in order to determine if there are any rules present in them.

Parallel algorithms are required for the mining of association rules due to the very large databases used to store the transactions. Some of the previous parallel algorithms are Count Distribution (CD), Data Distribution (DD), Candidate Distribution (CDD), Intelligent Data Distribution (IDD), and Hybrid Distribution (HD). However the costs associated with these algorithms are hash tree construction, hash tree traversal, communication overhead, input/output (I/O) cost and data movement respectively. These algorithms assign tasks to the processors using a static scheduling scheme. The main challenge for a static scheduling scheme is to determine the amount of time that will be needed to process each task. This information can then be used to compute the total time needed to process all the tasks and to divide these tasks among the processors so that an equal amount of tasks are assigned to each processor using processing time as the unit of measurement.

Experimental results show that DDRM utilizes the processors efficiently and performed better than the prefix-based and Partition algorithms that use a static approach to assign classes to the processors. The DDRM algorithm scales well and shows good speedup.

Acknowledgements

I thank my advisor, Professor Junping Sun for his extraordinary patience while modifying and editing the earlier drafts of this dissertation.

I would also like to thank all the members of my dissertation advisory committee for their comments and careful reading of the dissertation drafts, which resulted in the improvement of this research. In particular I thank Professor Michael J. Laszlo and Professor Jim Cannady. I thank Professor Barrington Chevannes for his comments and careful reading of this dissertation.

My graduate study at Nova Southeastern was beneficial and enjoyable. I thank all the professors and members of staff in the graduate School of Computing and Information Sciences for their contribution to my study.

Many thanks to Mr. Mark Powell and Mr. Joshua Morrison, Nova Southeastern University for the assistance given during the use of the Secure and Robust Distributed Information Systems (SARDIS) laboratory. I am also thankful to Mr. Colin Francis, University of Technology, Jamaica for the use of the multimedia laboratory.

I thank my wife Nadine, son Alex and daughter Leanne for their love, understanding, support and sacrifice. During the tough times they were always there to support and encourage me.

Table of Contents

List of Tables

Tables

List of Figures

Figures

Chapter 1

Introduction

1.1 Problem Statement and Goal

Many organizations are now finding it feasible economically to create ultra large databases of business and scientific data. This is made possible by the availability of inexpensive storage devices and developments in data capture technology (Agrawal & Shafer, 1996). Bar-code technology has made it possible to collect and store large amounts of sales data in retail organizations. The records associated with retail data are typically made up of transaction data and items bought in the transaction. These databases are viewed by organizations as important pieces of marketing infrastructure.

It is the desire of these organizations to institute information-driven marketing processes, managed by database technology, which will enable marketers to develop and implement customized marketing programs and strategies (Agrawal & Srikant, 1994). In order to accomplish the above these organizations are turning to the application of data mining technology to assist in the process of extracting valuable information from these large databases. It is recognized that new marketing strategies can be generated based on the extraction of previously unknown information from these large databases. Organizations are now using this data for the mining of association rules. A probabilistic statement such as 98% of customers that purchase tires and auto accessories also get automotive services done is an example of an association rule. It is a statement about the

co-occurrence of certain events in a database (Hand, Mannila, & Smyth, 2001). According to Agrawal and Srikant (1994) finding all such rules is valuable for cross marketing and attached mailing applications. In addition, applications such as catalogue design, add-on sales, store layout, and customer segmentation based on buying patterns, are important areas of application of mining of association rules.

The goal of data mining is the discovery of unknown patterns in large databases using efficient techniques to find these rules. Due to the large volume of data stored in these databases, considerable work has been done using serial algorithms. As the volume of data stored in these databases increases, the performances of the serial algorithms decrease due to the large volume of data that is being processed serially. However, according to Agrawal and Shafer (1996) it is clear that even with the development of fast serial algorithms, they are still limited due to the volume of data to be processed. It is therefore, necessary to use parallel algorithms for the task of mining of association rules. Parallel architectures are now affordable due to the significant progress made in networking, memory, and processor technologies. These technologies have made it possible to access and manipulate massive databases in a reasonable amount of time (Agrawal & Shafer, 1996).

In association rule mining, the database is scanned for interesting relationships in a given data set. Interestingness is measured by rule support and confidence. For example, milk $\Rightarrow$ bread [support = 5%, confidence = 70%]. Support of 5% means that 5% of all the transactions show that milk and bread are purchased together, and confidence of 70% shows that 70% of the customers purchasing milk also purchased bread. The goal of

mining association rules is to generate all association rules that have support and confidence greater than the user specified support and confidence, respectively.

In the original paper on the topic, mining of association rules can be divided into two steps (Agrawal, et al., 1993). In the first step the objective is to find all itemsets whose support is greater than the user specified minimum support (frequent itemsets). The second step uses the frequent itemsets to generate the desired rules. The first step requires more time and computation power than the second one. According to Zaki (2000) the search space for the discovery of all frequent associations in very large databases is exponential in the number of database attributes. In addition this is further complicated by I/O requirements for the millions of database objects.

The goal of this research was to develop and implement a parallel algorithm for the mining of association rules. The Dynamic Distributed Rule Mining (DDRM) algorithm uses a lattice to represent the search space for the generation of the frequent itemsets. DDRM partitions the search space and assigns each partition dynamically to the next available processor. An evaluation of the algorithm was carried out and its performance relative to the prefix-based algorithm proposed by Zaki (2000) with bottom-up search, which is a parallel algorithm for mining of association rules, was also determined.

According to Agrawal and Schafer (1996) because of the very large size of the databases needed to store the transactions used in the mining of association rules, parallel algorithms are required. Several parallel algorithms have been developed for the mining of association rules including Count Distribution (CD), Data Distribution (DD), Candidate Distribution (CDD), Intelligent Data Distribution (IDD), and Hybrid Distribution (HD). Agrawal and Shafer (1996) developed the CD, DD, and CDD

algorithms. The IDD and HD algorithms were both developed by Han, Karypis, and Kumar (2000). The IDD and HD algorithms have performed better than CD and DD. However, the cost associated with these algorithms includes hash tree construction, hash tree traversal, communication overhead, I/O operations, and the movement of data.

In DDRM there is no hash tree and the cost associated with I/O and communications are significantly reduced. It computes the frequent itemsets using an intersection operation in memory that requires no scanning of the database. This is different from the approach used by HD and IDD in which the database is scanned during the computation of the frequent itemsets.

The DDRM algorithm uses an equivalence operation to partition the search space lattice into sublattices to be assigned dynamically to processors for processing and identification of frequent itemsets. The Prefix-based algorithm uses a static approach to assign sublattices to the processors participating in the cluster. The system assigns a sublattice to each processor as it becomes available. Since the sublattices are assigned dynamically there will be a better utilization of the available processors. The partitioning of the lattice into sublattices can be controlled and used to determine the maximum size of a sublattice. If a sublattice is above the maximum size it will be partitioned into sublattices recursively until the size of each meets the required threshold. An outline of the approach is as follows:

1. Divide the database among the processors
2. All processors will contribute to the building of the tid-list
3. Generate the sublattices
4. Assign each sublattice to the next available processor

5. Update control processor with result
6. Generate rules

1.2 Relevance and Significance

Agrawal, et al. (1993) highlighted the issues associated with the generation of large itemsets during rule mining. They presented a template algorithm in which they addressed trade off between the number of passes and time wasted on processing itemsets that turned out to be small. They used an estimation procedure to determine what itemsets to measure in addition to two pruning procedures that prune detected itemsets that will not turn out to be large.

The Count Distribution (CD), Data Distribution (DD) and Candidate Distribution (CDD) algorithms were presented by Agrawal and Shafer (1996). These algorithms are parallel versions of the popular Apriori algorithm. CD, DD and CDD were designed for shared nothing systems. CDD incorporates detailed problem knowledge and removes processor dependence and synchronous communication from the process. However, this algorithm suffers from high communication overhead and the cost associated with the redistribution of the dataset. The performance of CDD is better than DD but not as good as CD. DD algorithm scales poorly and has a high communication cost; however DD exploits the aggregate memory of the multiprocessor better than CD. There is not a corresponding decrease in communication with decrease in computation.

CD reduces the communication overhead of DD significantly since it only broadcasts the candidate itemsets. Due to the fact that CD does not parallelize the computation of building the candidate hash tree, there is a bottleneck with a large number of processors.

CD scales linearly with the number of transactions. It was found to be the best of the three algorithms showing linear speedup and excellent scaleup and sizeup behavior.

The Intelligent Data Distribution (IDD) and the Hybrid Distribution (HD) algorithms were proposed by Han, et al. (2000) and seek to overcome some of the challenges of CD and DD. The IDD algorithm is similar to DD except that it uses a ring network. The HD algorithm combines CD and IDD to improve on the efficiency problem associated with IDD as the number of processors increases. IDD solves the communication problem of DD by using a ring-based all-to-all broadcast network. It eliminates the redundant work of DD by the use of a bit map and uses bin-packing to achieve equal distribution of the candidate itemsets. As more processors are added it becomes more difficult to balance the work with a smaller number of candidates. The hash tree is smaller for a smaller number of candidates and less computation work per transaction. HD inherits all the good features of IDD and reduces the amount of data movement.

Four hash-based algorithms for the parallel mining of association rules were presented by Shintani and Kitsuregawa (1996). These are the Non Partitioned Apriori (NPA), Simply Partitioned Apriori (SPA), Hashed Partitioned Apriori (HPA) and HPA with Extremely Large Itemset Duplication (HPA-ELD) algorithms. HPA-ELD was found to be faster than HPA in execution time and all four algorithms attained linearity for sizeup.

The Equivalence Class Transformation (ECLAT) algorithm is a localized algorithm for parallel mining of association rules and was presented by Zaki, Parthasarathy and Li (1997). ECLAT clusters related frequent itemsets and transactions. The work is distributed among the processors to facilitate the computations of frequent itemsets independently by each processor and uses a vertical data layout. The interconnection of

the processors allows a user-level application to write to the memory of remote nodes, resulting in fast user-level messages and low synchronization costs. ECLAT performed better than CD and reduces the high communication and I/O overhead.

Cheung, Han, Ng, Fu and Fu (1996) developed the Fast Distributed Mining (FDM) of association rules algorithm. In addition they also developed FDM with Local Pruning (FDM-LP), FDM with Local Upper Bound Pruning (FDM-LUP) and FDM with Local Pruning and Polling-Site-Pruning (FDM-LPP), which are based on different combinations of local and global pruning. A comparison of CD and FDM-LP based on candidate set, message size reduction, and execution reduction, shows FDM-LP as performing better.

The importance of data locality and reduction of false sharing was investigated by Parthasarathy, Zaki, and Li (1998). They presented three techniques for improving referencing locality and an additional three for reducing false sharing when processing the information in the hash tree. These techniques were designed for shared memory multiprocessors.

The Adaptive Parallel Mining (APM) algorithm for the parallel mining of association rules divides the database equally among the processors. APM was developed by Cheung, Hu and Xia (1998) and was designed for a shared-memory multiprocessors system. The APM Dynamic Itemset Counting (APM-DIC) and the APM Adaptive Intra-partition Internal Configuration (APM-IC) are variants of APM and were used to compare with the performance of CD. APM was found to be faster than CD.

Cheung and Xiao (1998) studied the effect of data skewness in parallel mining of association rules. They developed the Fast Parallel Mining (FPM) algorithm based on the

use of distributed and global pruning techniques. FPM is similar to CD but requires less bandwidth and has a simpler communication scheme. Cheung and Xiao (1998) developed a data skewness metric based on the use of entropy. For sizeup FPM was closer to the ideal than CD.

A parallel approach to the task of discovering association rules on a shared-nothing system has two major issues to be addressed. The first requires the development of an efficient way to exchange information among the processors. There is a reduction in the number of scans of the database. Secondly it is also necessary to address the issue of load balancing among the processors. These are important factors to be considered in the implementation of the DDRM algorithm.

1.3 Barriers and Issues

Mining of association rules is a challenge due to the size of the database used in this process. The availability of technology used to capture and store data has resulted in the creation of ultra large databases of business and scientific data (Agrawal & Shafer, 1996). Most of the algorithms proposed are based on serial designs. However, the databases used by these algorithms to mine association rules are often very large.

The performance of algorithms for the mining of association rules can be improved significantly if they are designed to execute in parallel rather than serially. Mining of association rules from databases of transactions is an important problem in data mining (Agrawal, et al., 1993). The computation of the frequencies of the occurrence of subsets of items is the most time consuming part of the process. Invariably, researchers in the area of association rule mining, concentrate mainly on this aspect of the problem. Agrawal and Srikant (1994) proposed a fast algorithm for mining of association rules.

Park, Chen, and Yu (1995) also proposed a fast algorithm for this task. A major limitation of these algorithms is the serial design approach used. Researchers in association rule mining are currently conducting research in developing parallel algorithms for the mining of association rules.

A major challenge for some of these algorithms including Count Distribution (CD), Data Distribution (DD), Intelligent Data Distribution (IDD) and Hybrid Distribution (HD), is the high overhead costs due to I/O and communications among the processors. These algorithms scan the database repeatedly and must exchange information on the frequent itemsets regularly (Agrawal & Schafer, 1996; Han, et al., 2000). This study proposes an algorithm that will address these issues as well as load balancing among the processors. In addition, the algorithm will improve on the execution time and processor utilization.

1.4 Summary

This chapter discussed the need for parallel solution to data mining problems. The mining of association rules requires the use of a parallel approach to improve on the execution time. An outline of the goal of implementing a parallel algorithm based on a lattice theoretic approach was presented. A brief review of the relevance and significance of data mining and the need for parallel algorithms in this area was also presented followed by an indication of some of the limitations and barriers related to the research.

Chapter 2

Review of the Literature

2.1 Historical Overview of the Theory and Research Literature

This chapter gives an overview of data mining approaches with emphasis on association rule mining. It discusses the theoretical issues associated with the mining of association rules.

2.1.1 Data Mining

Data mining is the science of extracting useful information from large data sets or databases. It is an interdisciplinary field involving the merging of ideas from statistics, machine learning, data management and databases, pattern recognition, artificial intelligence and other areas. It is a scientific discipline that is concerned with the analysis of observational data sets with the objective of finding unsuspected relationships and produces a summary of the data in novel ways that the owner can understand and use (Hand, et al., 2001).

According to Hand, et al. (2001) data mining originated in the artificial intelligence research field and is often set in the broader context of knowledge discovery in databases (KDD). The categories of KDD algorithms are classification, sequencing, and association. The input data are partitioned into disjoint groups such as decision tree or set of rules by classification algorithms. A sequencing algorithm is used to generate events

that are related in time. An example of events that are related in time is, an occurrence of events P and Q is usually followed by the occurrence of event R. Items that appear together based on a minimum frequency are extracted from transaction records by association algorithms (Carter & Hamilton, 1998).

Clustering is the process of grouping a set of objects into classes in which similar objects share the same cluster while being dissimilar to objects in other cluster. It facilitates the identification of dense and sparse regions, which makes it possible to discover the overall distribution patterns and interesting correlations among data attributes (Han & Kamber, 2001; Agrawal, Gehrke, Gunopulos, & Raghaven, 1998). There are five stages associated with KDD which are the selection of the target data, pre-processing the data, transforming them if necessary, performing data mining to extract patterns and relationships, and then interpreting and assessing the discovered structure. There are four steps involved in the extraction of patterns and relationships, which are the identification of the nature and structure of the representation, the choice of score function, the design of the algorithm that will optimize the score function, and the efficient implementation of the algorithm. In the Apriori algorithm these steps can be identified as the structure, which is association rules; the score function, which is based on support and accuracy; the search method, which is breadth-first with pruning; and the data management technique, which is linear scans.

2.1.2 Data Mining Tasks

Hand, et al. (2001) gave the following categorization of data mining tasks corresponding to the different objectives of the analyst. Exploratory data analysis (EDA) uses visual and interactive techniques to explore the data without any clear idea of what

to look for. In descriptive modeling the goal is to describe all of the data and may include the overall probability distribution of the data (density estimation), cluster analysis and segmentation, and dependency modeling. Predictive modeling (classification and regression) uses the model to predict the value of one variable from the known values of other variables. The key distinction between prediction and description is that the objective of prediction is a unique variable while there is no single variable central to the model of descriptive problems.

In discovering patterns and rules the concern is the detection of patterns. Finding combinations of items that occur in transaction databases has been addressed using algorithmic techniques based on association rules. The use of a pattern of interest to find similar patterns in the data set is referred to as *retrieve by content*.

2.2 The Theory and Research Literature Specific to Data Mining

2.2.1 Association Rule Mining

Association rule mining searches for interesting relationships among items in a given data set. An association rule is a simple probabilistic statement about the co-occurrence of certain events in a database, and is particularly applicable to sparse transaction data sets (Han & Kamber, 2001; Hand, et al., 2001). According to Hand, et al. (2001) association algorithms find all rules satisfying the frequency and accuracy thresholds. Low thresholds result in the generation of many rules with the possibility of some of them being trivial to the user. One of the challenges in data mining is to develop methods for selecting potentially interesting rules from the large set of rules generated by the system.

Agrawal, Imielinski and Swami (1993) first introduced the problem of mining association rules, which can be stated as follows:

Let $I = \{i_1, i_2, \ldots, i_n\}$ be a set of items and D be a set of transactions where each transaction T has a unique identifier called its *TID* and consists of a set of items such that $T \subseteq I$. An itemset is a set of items. An itemset with k items is called a k-itemset. An itemset is maximal if it is not a subset of any other itemset. An association rule is an implication of the form $X \Rightarrow Y$, where $X \subseteq I$, $Y \subseteq I$, and $X \cap Y = \varnothing$. The support s of the rule is the percentage of transactions in D that contains $A \cup B$. The confidence c of the rule is the percentage of transactions in D containing A that also contains B. These can be expressed in probability terms as $P(A \cup B)$ and $P(A|B)$ respectively (Han & Kamber, 2001; Shintani & Kitsuregawa, 1998; Megiddo & Srikant, 1998; Srikant, Vu, & Agrawal, 1997; Bayardo Jr., Agrawal, & Gunopulos, 1999). An itemset is frequent if its support is more than a user specified minimum support (*min_sup*) value. The goal of mining association rules is to generate all association rules that have support and confidence greater than the user specified support and confidence, respectively.

The first step in the mining of association rules requires the identification of all frequent itemsets with each of these itemsets occurring with a frequency no less than the minimum support count. In the second step the frequent itemsets are used to generate a set of strong association rules that satisfy both minimum support and minimum confidence. The second step is the easier of the two steps and can be accomplished by finding all non-empty subsets of every frequent itemset l. For every such subset a, output a rule of the form $a \Rightarrow (l - a)$ if the ratio of support (l) to support (a) is at least the minimum required (Han & Kamber, 2001).

There are five components that are associated with data mining algorithms for association rules, which are task, structure, score function, search method and data management technique. The task is to describe the association between variables and the structure is probabilistic association rules. The score function based on thresholds on accuracy and support and the search method is breath-first with pruning. The data management technique is multiple linear scans (Hand, Mannila, & Smyth, 2001).

2.2.1.1 Classification of Association Rules

According to Han and Kamber (2001) association rules are classified into four categories, which are types of values handled in a rule; dimension of the data; levels of abstraction involved in the rule; and the extension to association mining. The type of values handled in the rule refers to Boolean and quantitative association rules. In Boolean association rules the objective is to identify the presence or absence of association between items. A quantitative association rule partitions quantitative values for items into intervals. In a single-dimension association rule the items reference one dimension only. When the items reference two or more dimensions it is said to be multi-dimensional. Consider the rule set Age (X, "30...39") $\Rightarrow$ buys (X, "Laptop") and Age (X, "30...39") $\Rightarrow$ buys (X, "Computer"). Here the two items laptop and computers are at two different levels of abstraction. The rule set is said to be a multilevel association rules. If the rule in a set does not reference items at different levels of abstraction the set is said to be single-level association rules. Correlation analysis is one possible extension of association mining. In this extension the presence or absence of correlation between items is established. Mining of max patterns and frequent closed itemsets are also possible extensions. A max pattern is a frequent pattern p, such that any proper super pattern of p

is not frequent. A frequent closed itemset is where an itemset *c* is closed if there exists no proper superset of *c*, *c*', such that every transaction containing *c* also contains *c*'. Max patterns and frequent closed itemsets can be used to reduce the number of frequent itemsets generated in mining (Han & Kamber, 2001).

2.2.2 Apriori Algorithm

This algorithm is influential in mining frequent itemsets for Boolean association rules. The name of the algorithm is based on the fact that the algorithm uses prior knowledge of frequent itemset properties (Agrawal, Imielinski and Swami, 1993; Han & Kamber, 2001). Apriori is a serial algorithm that has a smaller computational complexity when compared with other serial algorithms (Han, Karypis & Kumar, 2000). The outline of the algorithm is shown in Figure 2.2.1.

(1) L_1 = find_frequent_1_itemsets (D);
(2) for (k = 2; $L_{k-1} \neq \emptyset$; k++){
(3) C_k = apriori_gen(L_{k-1}, *min_sup*);
(4) for each transaction $t \in D$ {// scan D for counts
(5) C_t = subset(C_k, t); // get the subset of t that are candidates
(6) for each candidate $c \in C_t$
(7) *c.count*++;
(8) }
(9) $L_k = \{c \in C_k \mid c.count \geq min\_sup\}$
(10) }
(11) return $L = \cup_k L_k$;

procedure apriori_gen(L_{k-1}: frequent (k-1)-*itemsets*; *min_sup*: minimum support threshold)

(1) for each itemset $l_1 \in L_{k-1}$
(2) for each itemset $l_2 \in L_{k-1}$
(3) if($l_1[1] = l_2[1]$)^($l_1[2] = l_2[2]$)^…^($l_1[k\text{-}2]=l_2[k\text{-}2]$)^($l_1[k\text{-}1]=l_2[k\text{-}1]$) then{
(4) $c = l_1 \bowtie l_2$; // join step: generate candidates
(5) if has_infrequent_subset(c, L_{k-1}) then
(6) delete c; // prune step: remove unfruitful candidate
(7) else add c to C_k;
(8) }
(9) return C_k

procedure has_infrequent_subset(c: candidate k-itemsets; L_{k-1}: frequent (k-1)-itemsets);
// use prior knowledge
(1) for each (k-1)-subset s of c
(2) if $s \notin L_{k-1}$ then
(3) return TRUE
(4) return FALSE

Figure 2.2.1 Apriori Algorithm
(Agrawal, Imielinski and Swami, 1993; Han & Kamber, 2001)

The following example illustrates the use of the Apriori Algorithm to mine the association rules from transaction database shown below.

Database *D*

TID	Items
100	I1 I3 I4
200	I2 I3 I5
300	I1 I2 I3 I5
400	I2 I5

STEPS

1. In the first iteration, each item is a member of the set of candidate 1-itemsets, C_1. The algorithm scans all the transactions in *D* in order to count the number of occurrences of each item.

C_1

Itemset	Support Count
{I1}	2
{I2}	3
{I3}	3
{I4}	1
{I5}	3

2. Generate the support count using a minimum transaction support count of 2. Therefore the set of frequent 1-itemsets L_1 consists of candidate 1-itemsets satisfying minimum support.

L_1

Itemset	Support Count
{I1}	2
{I2}	3
{I3}	3
{I5}	3

3. To discover the set of frequent 2-itemsets, L_2 the algorithm uses $L_1 \bowtie L_1$ to generate a candidate set of 2-itemsets, C_2 consisting of $\binom{4}{2}$ 2-itemsets.

C_2

Itemset
{I1, I2}
{I1, I3}
{I1, I5}
{I2, I3}
{I2, I5}
{I3, I5}

4. The transactions in the database D are scanned and the support count of each candidate itemset in C_2 is generated

C_2

Itemset	Support Count
{I1, I2}	1
{I1, I3}	2
{I1, I5}	1
{I2, I3}	2
{I2, I5}	3
{I3, I5}	2

5. The set of frequent 2-itemsets, L_2, is then determined and consists of those candidate 2-itemsets in C_2 having minimum support.

L_2

Itemset	Support Count
{I1, I3}	2
{I2, I3}	2
{I2, I5}	3
{I3, I5}	2

6. The generation of the set of candidate 3-itemsets, C_3 which is as follows:

Join: $C_3 = L_2 \bowtie L_2$

= {{I1, I3}, {I2, I3}, {I2, I5}, {I3, I5}} $\bowtie$ {{I1, I3}, {I2, I3}, {I2, I5}, {I3, I5}}

= {I2, I3, I5}

C_3 = {I2, I3, I5}

The subsets of C_3 are {I2, I3}, {I2, I5} and {I3, I5} and they are all frequent so there is no pruning.

Therefore C_3 = {I2, I3, I5}.

Generate count of each candidate in C_3.

C_3

Itemset	Support Count
{I2, I3, I5}	2

7. Compare candidate support count with maximum support to generate L_3.

L_3

Itemset	Support Count
{I2, I3, I5}	2

8. The algorithm uses $L_3 \bowtie L_3$ to generate a candidate set of 4-itemsets, C_4.

$C_4 = \varnothing$ and the algorithm terminates.

Generating Association rules

Once the frequent itemsets from transactions in a database *D* have been found, it is straightforward to generate strong association rules from them (where strong association rules satisfy both minimum support and minimum confidence). The confidence is given by:

Confidence $(A \Rightarrow B) = P(A|B) = (support\_count(A \cup B))/(support\_count(A))$, where *support_count* $(A \cup B)$ is the number of transactions containing the itemsets $A \cup B$, and *support_count*(A) is the number of transactions containing the itemset A.

Based on this equation, association rules can be generated as follows:

For each frequent itemset *i*, generate all non-empty subset of *i*.

For every non-empty subset *s* of *i*, output the rule "$s \Rightarrow (i - s)$" if the confidence of this rule is greater than or equal to the maximum confidence threshold (Han & Kamber, 2001).

In the example above the frequent itemset i = {I2, I3, I5}. The nonempty subsets of i are {I2, I3}, {I2, I5}, {I3, I5}, {I2}, {I3} and {I5}.

The resulting association rules are as follows:

$I2 \wedge I3 \Rightarrow I5$ *confidence* = 2/2 = 100%

$I2 \wedge I5 \Rightarrow I3$ *confidence* = 2/3 = 67%

$I3 \wedge I5 \Rightarrow I2$ *confidence* = 2/2 = 100%

$I2 \Rightarrow I3 \wedge I5$ *confidence* = 2/3 = 67%

$I3 \Rightarrow I2 \wedge I5$ *confidence* = 2/3 = 67%

$I5 \Rightarrow I2 \wedge I3$ *confidence* = 2/3 = 67%

The confidence threshold will determine the rules for output.

If minimum confidence were set at 70% we would output the following rules:

$I2 \wedge I3 \Rightarrow I5$ *confidence* = 2/2 = 100%

$I3 \wedge I5 \Rightarrow I2$ *confidence* = 2/2 = 100%

Hash Tree

One method used to improve the counting of the itemsets by Apriori based algorithms is a hash tree. The hash tree identifies the items to be counted efficiently and reduces the time taken to count the candidate itemsets. One approach to counting the itemsets is to compare the items in each transaction against all the candidate itemsets. This is a time consuming activity, which is significantly improved by the use of a hash tree (Han, Karypis, & Kumar, 2000).

The candidate itemsets to be counted using a hash tree are shown in Table 2.2.1. Before we can count these itemsets a hash tree is implemented for these candidate itemsets. In Figure 2.2.2 we build a hash tree to count 3-itemsets. The hash function is that itemsets starting with 1, 4, or 7 hashes to the left child, itemsets starting with 2, 5, or 8 hashes to the middle child and itemsets starting with 3 or 6 hashes to the right child.

The hash function $H(x)$ is defined as follows:

$$H(x) = \begin{cases} L & \text{if } (x \bmod 3) = 1 \\ M & \text{if } (x \bmod 3) = 2 \\ R & \text{if } (x \bmod 3) = 0 \end{cases}$$

Where x is the first item in the itemset

L, M, and R represent the left, middle and right child respectively.

The maximum number of itemsets that can be stored in a bucket is 3. Leaf nodes contain itemsets that hashed to those nodes.

Consider a transaction with the items 1, 5, 6, 7, 8. We first hash at the root with item 1 which takes us to the left child, at the next node we hash on 5 which takes us to the middle child, we then hash on 6 which takes us to the right child. We are now at a leaf node. We check the transaction against the items in the leaf node and there is no match. We return to the level above where we hash on 7, which takes us to the left node. This is also a leaf node so we compare its contents against the transaction and there is no match. We return to the level above and hash on 8, which takes us to the middle node. The middle node is a leaf and there is also a match with 1, 5, 8 so we increase the count for candidate itemset 1, 5, 8. At this point we have checked all itemsets starting with 1, 5, we now need to check for itemsets starting 1, 6. We next go back up to level 2 of the hash tree where we hash on 6 which takes us to the right node which is a leaf node. We also found a match for 1, 6, 8 and increment the count for this candidate itemset. We go back up to the next level and hash on 7 which takes us to the left node which is also a leaf node and there is also a match for 1, 7, 8. The count for candidate 1, 7, 8 is incremented. At

this point we have identified all the itemsets starting with 1, 6. This is repeated for the remaining items in the transaction.

The next step is to identify all those itemsets starting with 5. We then go back to level 1, the root node and hash on 5, which takes us to the middle node, and the process is repeated as outlined above. It is clear that the hash tree is an efficient approach to identify the frequent itemsets in a transaction. The final count of candidate itemsets after processing the transaction is shown in Table 2.2.2.

Table 2.2.1 Candidate Itemsets in Hash Tree

Candidate Itemsets	**Count**
{1 2 4}	0
{1 2 7}	0
{1 3 7}	0
{1 4 5}	0
{1 4 6}	0
{1 5 8}	0
{1 6 8}	0
{1 7 8}	0
{2 3 4}	0
{2 4 5}	0
{2 4 6}	0
{2 5 6}	0
{3 5 7}	0
{2 5 8}	0
{2 6 7}	0
{2 7 8}	0
{3 5 7}	0
{3 6 8}	0
{4 5 8}	0
{6 7 8}	0
{7 8 9}	0

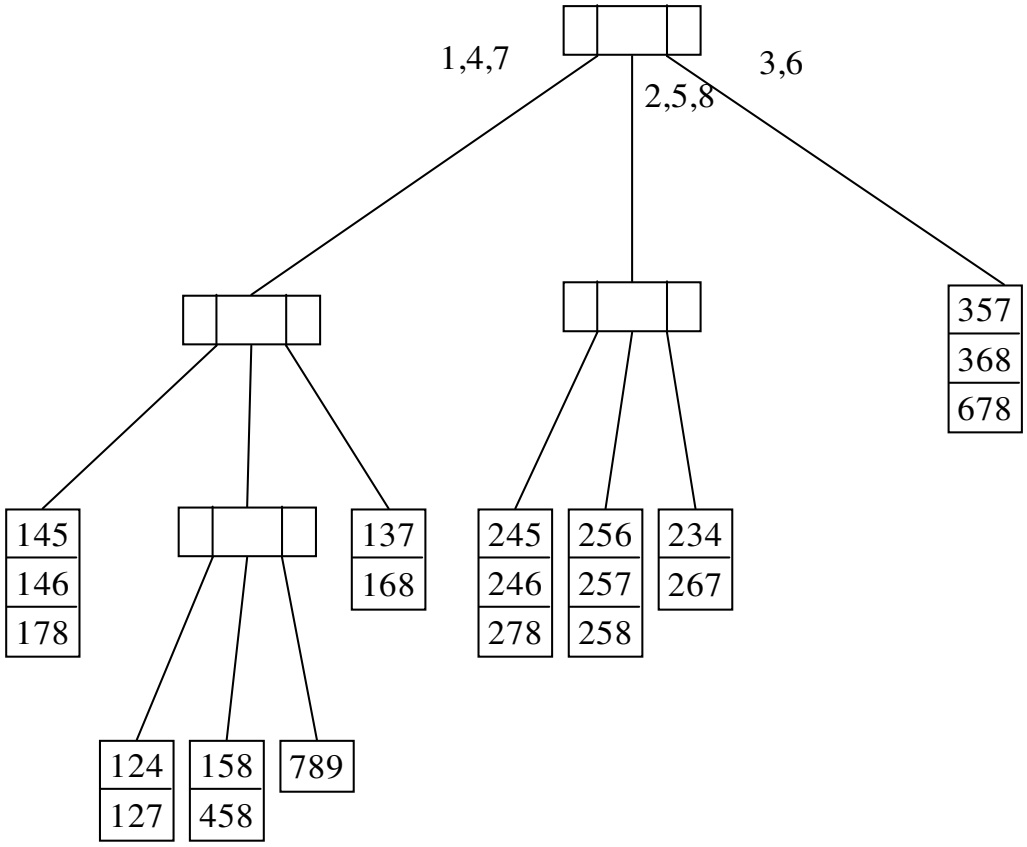


Figure 2.2.2 Hash Tree of Candidate 3-Itemsets

Table 2.2.2 Count of Candidate Itemsets in Hash Tree

Candidate Itemsets	**Count**
{1 2 4}	0
{1 2 7}	0
{1 3 7}	0
{1 4 5}	0
{1 4 6}	0
{1 5 8}	1
{1 6 8}	1
{1 7 8}	1
{2 3 4}	0
{2 4 5}	0
{2 4 6}	0
{2 5 6}	0
{3 5 7}	0
{2 5 8}	0
{2 6 7}	0
{2 7 8}	0
{3 5 7}	0
{3 6 8}	0
{4 5 8}	0
{6 7 8}	1
{7 8 9}	0

2.2.3 Database Organization

The database of transactions shown in Table 2.2.3 can be considered as a Boolean relational table as shown in Table 2.2.4. The database can be physically organized horizontally as shown in Table 2.2.3 or vertically as shown in Table 2.2. 5. The horizontal organization consists of a set of pairs (transaction ID, itemset), where transaction ID is the transaction number and itemset is the set of items bought in that transaction. The vertical organization consists of a set of pairs (item, transaction list), where item is an item bought and transaction list is the set of transactions in which the item was bought.

Table 2.2.3 Transaction Database

TID	List of Items
100	I1 I2 I5
200	I2 I4
300	I2 I3
400	I1 I2 I4
500	I1 I3
600	I2 I3
700	I1 I3
800	I1 I2 I3 I5
900	I1 I2 I3

Table 2.2.4 Boolean Representation of Transaction Database

TID	I1	I2	I3	I4	I5
100	1	1	0	0	1
200	0	1	0	1	0
300	0	1	1	0	0
400	1	1	0	1	0
500	1	0	1	0	0
600	0	1	1	0	0
700	1	0	1	0	0
800	1	1	1	0	0

Table 2.2.5 Vertical View of Transaction Database

I1	**I2**	**I3**	**I4**	**I5**
T100	T100	T300	T200	T100
T400	T200	T500	T400	T800
T500	T300	T600		
T700	T400	T700		
T800	T600	T800		
T900	T800	T900		
	T900			

2.2.4 Parallel Processing

Parallel processing is the concurrent manipulation of data elements belonging to one or more processes solving a single problem. Pipelining and parallelism are normally used to achieve concurrency. Pipelining divides the computation of a task into a number of steps, while parallelism is the use of multiple resources to increase concurrency. Pipelined computation is divided into a number of steps called segments or stages. Each segment is assigned a part of the computation to be carried out and the output of one segment serves as input to the next segment.

In an ideal parallel system the following are true: (1) linear speedup: Four times as much hardware can perform the task in one quarter the time, and (2) linear scaleup: four times as much hardware can perform four times as large a task in the same elapsed time (DeWitt & Gray, 1992). If a job is executed on a small system and a larger system, the speedup that is obtained from the larger system is defined as:

$$Speedup = \frac{small\_system\_elapsed\_time}{larger\_syetem\_elapsed\_time}$$

If an *N*-times large or more expensive system yields a speedup of *N* it is said to be linear. This metric holds the problem size constant while it grows the system. Scaleup refers to the ability of an *N*-times larger system to perform an *N*-times larger job in the same elapsed time as the original system (DeWitt & Gray, 1992).

$$Scaleup = \frac{small\_system\_elapsed\_time\_on\_small\_problem}{larger\_system\_elapsed\_time\_on\_larger\_problem}$$

A linear scaleup has a value of 1 since executing a problem that is twice as large on a system that is twice as large as the original system will take the same time to execute as

the time taken by the original problem on the original system. Three major challenges to speedup and scaleup are startup, interference and skew. The time taken to startup thousands of processors can dominate the computations. The accessing of shared resources by processes can cause interference when these processes try to access a shared resource. The average size of each step decreases as the number of parallel steps increases and may result in a variance that is in excess of the mean. Increased parallelism will improve the elapsed time only slightly where the variance dominates the mean (DeWitt & Gray, 1992). A large grain size will increase speedup since it reduces the frequency of synchronization.

If a portion of the algorithm must be executed sequentially by one of the *p* processors, then the remaining *p*-1 processors must wait for the sequential portion to complete before they resume, this implies synchronization among the processors. Contention for single resource limits the speedup possible. The workload must be balanced among processors. Static decomposition assumes that the tasks and their precedence relations are known before execution. Dynamic decomposition assumes that tasks are generated during program execution.

In a distributed environment the practical implications of communication overhead, the effect of the underlying architecture, and the dynamic behaviour of the system are issues that contribute to the complexity of a distributed environment (Zaki, 2000b).

Scalable Systems

Ideally increasing the number of processors should produce a corresponding increase in the processing power of the machine, and there should be no upper limit to the number of processors used. An ideal system should not have global memory, as you cannot put an

unbounded number of processors close to a global memory. It is therefore necessary to limit access to global memory due to the fact that performance suffers as processors are put farther and farther from memory. In order to keep the communications cost low it is necessary to limit communications to processors that are close together.

Parallel Data Mining

Tightly coupled systems are generally associated with parallel data mining (PDM). These systems include distributed memory machines (DMM), shared memory machines also known as symmetric multiprocessors (SMP), and clusters of SMP workstations (Zaki, 2000b). Distributed data mining (DDM) is based on loosely coupled systems including sites that are geographically distributed over a wide area network. PDM and DDM differs significantly in scale, data distribution and communication costs (Zaki, 2000b).

According to Zaki (2000b) the main challenges associated with parallel and distributed data mining are minimization of communications, load balancing, synchronization, disk I/O minimization and decomposition and layout of the data. The partitioning of the task and data together with the type of memory system will affect the design space for parallel systems. In distributed and shared memory systems synchronization is implicit in message passing with DMMs. It is therefore necessary to optimize communications. In shared memory machines (SMP) locks and barriers are used for synchronization. I/O is of importance for SMP machines. Data decomposition is important for distributed systems. The objective is to have optimal decomposition among the processors and to minimize communications. Algorithms based on a distributed shared-nothing memory are designed based on the reduction of communication, pruning of candidate sets and partition of the

candidate sets across the distributed memory. This is known as a *level-wise* approach as developed in Apriori. In the *level-wise* approach the computation cost generally peaks in the second iteration and decreases in the subsequent iterations due to reduction in the size of the candidate itemsets. Two options for reducing the cost of the *level-wise* approach are the reduction in the number of rounds of scanning the database and reduction in the number of candidate itemsets especially in iteration 2 (Zaki, 2000b).

There are two approaches to the implementation of data mining, which are task and data parallelism. One approach is to divide the data among several processors with each one performing the same set of operations on the data assigned to it. This approach is referred to as data parallelism. In the second approach the processors perform different operations independently but have access to entire database. This is known as task parallelism. A hybrid combines both approaches (Zaki, 2000b).

2.2.5 Partitioning of Candidate and Data

There are generally two approaches associated with parallel and distributed data mining methods. These can be described in terms of the computation and data partitioning methods used. The database can be shared in a shared-memory or shared disk architecture. The database can also be partitioned among the available nodes in a distributed memory architecture (Zaki, 2000b).

The candidate set can also be shared, replicated or partitioned among the nodes. In the shared approach a single copy of the candidate set is evaluated by all nodes. In the partitioned approach each processor is responsible for the computations associated with a specific set of candidate itemsets. The candidate itemsets are replicated on all processors where they are evaluated locally and then merged to generate the global results.

Replicated or Shared Candidates, Partitioned Database

In the replicated or shared candidates and partitioned database approach the database is partitioned into equal sizes among the processors and the candidate itemsets replicated across all processors. Parallel algorithms based on Apriori that use this approach compute the frequency of the candidate sets in the database at each processor during each iteration. The information at each processor is broadcast to all other processors for the computation of global counts. Some of the algorithms based on this approach are Count Distribution (CD), Fast Distributed Mining (FDM) (Cheung, et al., 1996), and Non Partition Apriori (NPA) (Shintani, & Kitsuregawa, 1996). This approach reduces the communication cost since it exchanges frequency counts only at the end of each iteration. However by replicating the candidates they fail to use the aggregate system memory that is available. Cheung, Hu and Xia (1998) implemented Adaptive Parallel Mining (APM) that is based on Dynamic Itemset Counting (DIC). The candidate set is shared among processors and updated asynchronously.

Partitioned Candidate, Partitioned Database

Three Apriori based algorithms that use this approach are Data Distribution (DD), Simply Partitioned Apriori (SPA) (Shintani, & Kitsuregawa, 1996) and Intelligent Data Distribution (IDD). The main advantage of this approach is the utilization of the aggregate memory. The main disadvantage is the need to scan the partitions of other processors; this is accomplished by exchanging the partitions at each iteration.

Partitioned Candidates, Selectively Replicated or Shared Database

Shintani and Kitsuregawa (1996) implemented the Hashed Partitioned Apriori

(HPA) and HPA with Extremely Large Itemset Duplication (HPA-ELD) algorithms that used this approach. In this approach the database on each processor is selectively replicated on each processor and each processor also evaluates a specific set of candidate itemsets.

2.2.6 Parallel and Distributed Algorithms

Several parallel association algorithms have been designed based on the Apriori algorithm. Park, et al. (1995) implemented the Direct Hashing and Pruning (DHP) algorithm, which was later, used in a number of parallel implementations. Zaki, Parthasarathy, Ogihara and Li (1997) used an approach based on equivalence class to implement four new parallel algorithms.

Agrawal and Shafer (1996) presented three parallel algorithms for mining association rules. These algorithms are the Count Distribution (CD), Data Distribution (DD), and Candidate Distribution (CDD) and are based on the Apriori serial algorithm used for the mining of association rules. The CD algorithm substitutes redundant computations in parallel on otherwise idle processors for communications overhead. Each processor keeps a copy of the complete candidate itemsets which it updates using the locally stored database. This copy is then broadcast to all other processors to be used for the final count. The CD algorithm does not exploit the total available memory and so it counts the same number of candidates in one pass as the serial algorithm.

The DD algorithm is designed to exploit the total memory available as the number of processors increase. The candidate itemset is divided among the *N* processors. On an *N*-processor configuration a candidate set that would require *N* passes in CD can be counted in one pass in DD. It is very expensive for every processor to broadcast the locally stored

data to every other processors. In the CD and DD algorithms data tuples and candidate itemsets are partitioned merely to equally divide the work. They require all processors to be connected and all information gathered before they can proceed on to the next pass. These constraints are eliminated in the CDD algorithm.

The CDD algorithm partitions both the data and the candidates in such a way, that each processor may proceed to the next pass independent of the other processors. Depending on the quality of the itemset partitioning, parts of the database may have to be replicated on several processors. Following candidate distribution, the processors work independently. Each processor counts only the portion of candidate itemset assigned to it. The pruning of the local candidate set is the only step that requires a processor to get information from other processors. This information is sent asynchronously making it possible for the processors to proceed without complete pruning information. It then uses the late arriving pruning information in subsequent stages. The results of tests on the performance of CD, DD and CDD show CD to be the best of the three with linear speedup and excellent scaleup behaviors.

Two algorithms proposed by Han, et al. (2000) for the parallel mining of association rules are the Intelligent Data Distribution (IDD) algorithm and the Hybrid Distribution (HD) algorithm. The main difference between IDD and CD is the use of a ring network to connect the processors in IDD. In IDD the portion of the transactions stored at each processor is sent to the other processors using a point-to-point communication between neighbors resulting in the elimination of any communication contention among processors. IDD partitions the candidate itemset among the *N* processors in such a way

that each processor gets itemsets that begin only with a subset of all possible items. Load balancing is achieved by using a special partitioning algorithm based on bin packing.

The HD algorithm is a combination of CD and IDD and improves on the inefficiency problem associated with IDD as the number of processor increases. The *N* processors are divided into *G* equal size groups, each containing *N*/*G* processors. The transactions are then divided among the groups treating each group as a single processor. In the HD algorithm the CD algorithm is executed as if there were only *N*/*G* processors. Within each group the candidate itemsets are partitioned among the processors and IDD used to compute the counts. IDD implements the process of building the hash tree in parallel and is scalable as the size of the candidate set increases and it also utilizes memory more effectively. HD combines the good qualities of CD and IDD. It achieves better load balancing than IDD since the candidate set is partitioned into fewer buckets (Han, et al., 2000).

Cheung, et al. (1996) presented the Fast Distributed Mining of association rules (FDM) algorithm. In this algorithm interesting relationships between locally large sets and globally large sets are explored to generate a smaller set of candidate sets at each iteration. Some candidate sets are also pruned away using local and global pruning techniques and only $O(n)$ messages are passed to determine whether or not a candidate set is large.

Three variations of FDM based on different combinations of local and global pruning are the FDM with Local Pruning (FDM-LP), FDM with Local and Upper Bound Pruning (FDM-LUP) and FDM with Local Pruning and Polling-Site-Pruning (FDM-LPP). These algorithms make use of the properties related to large itemsets in a distributed

environment. One such property is that every globally large itemset must be locally large at some site(s). Pruning is done both locally at each site and globally using information from all the sites. These two techniques can be combined to form different pruning strategies. FDM also uses a count polling technique to ensure that only $O(n)$ messages are needed for every candidate itemset in all cases, where n is the number of processors.

The Adaptive Parallel Mining (APM) algorithm for mining association rules divides the database into n logical partitions, where n is the number of processors. It is based on the shared memory machines (SMP) architecture and uses dynamic candidate generation technique to generate the common candidates asynchronously. Processors communicate through shared variables.

One variant of the APM algorithm uses the Dynamic Itemset Counting (DIC), which was developed by Brin, Motwani, Ullman, and Tsur (1997) and is referred to as APM-DIC. The APM-AIC is a second variant of APM that uses an adaptive interval configuration (AIC), which was designed to address the exponential growth of the candidate sets associated with DIC. APM was found to be faster than CD, when compared to CD the gain for APM-DIC was insignificant (Cheung, et al., 1998).

Cheung & Xiao (1998) investigated the effect of data skewness on parallel mining of association rules using the Fast Parallel Mining (FPM) algorithm based on the use of distributed and global pruning techniques. This algorithm is similar to the Count Distribution (CD) algorithm but requires less bandwidth and has a simpler communication scheme. The effectiveness of these pruning techniques depends on the itemset distribution that can be captured as data skewness. The speedup of the algorithm

was found to be super linear and when compared with CD the response time was found to be significantly faster.

The importance of data locality and reduction of false sharing in modern shared memory machines (SMP) due to the increasing gap between processor and memory subsystem performance was highlighted by Parthasarathy, et al. (1998). The Common Candidate Partitioned Database (CCPD) is a shared-memory algorithm, which is based on the Apriori algorithm. In CCPD the candidate itemsets are stored in a hash tree to facilitate fast support counting. The candidate hash tree is common, but the database is split logically among the processors. New candidates are generated and inserted in parallel. It uses a lock to guarantee mutual exclusion.

Three techniques for improving reference locality are Simple Placement Policy (SPP), Localized Placement Policy (LPP) and Global Placement Policy (GPP) (Parthasarathy, et al., 1998). In SPP all the different hash tree building blocks are allocated memory from a single region and do not rely on any special placement of the blocks based on traversal order. Three possible variants of this technique are common regions, individual regions, and grouped regions. The LPP scheme groups related data structures together using local access information present in a single routine. GPP utilizes knowledge of order of traversal of the entire hash tree to place hash tree building blocks in memory so that structures are arranged in the order of access in the same cache line in most cases.

Shared memory systems suffer from false sharing, which occurs when two different shared variables are located in the same cache block. This results in the exchange of the block between the two processors even though they are accessing different variables. Three techniques for reducing false sharing are Padding and Aligning, Segregate Read-

Only Data and Privatize (and Reduce) (Parthasarathy, et al., 1998). Padding and Aligning places unrelated read-write data on separate cache lines and results in a significant loss of locality and high memory space overhead. In Segregate Read-Only Data locks and counters (read-write data) are separated from the itemset (read-only data) and eliminates the possibility of falsely sharing read-only data. The Privatize (and Reduce) scheme makes a private copy of the data that will be used locally so as to avoid false sharing with operations on that data and was combined with the global placement policy and was given the name Local Counter Array-Global Placement Policy (LCA-GPP). When compared with the Common Candidate Partitioned Database (CCPD) shared memory algorithm Simple Placement Policy (SPP) did extremely well due to its simplicity.

The Non Partitioned Apriori (NPA), Simply Partitioned Apriori (SPA), Hashed Partitioned Apriori (HPA) and HPA with Extremely Large Itemset Duplication (HPA-ELD) are parallel algorithms for mining association rules on shared nothing parallel machines. In NPA the candidate itemsets are copied among all the processors. If the processor is unable to hold all the candidate itemsets in memory, the candidate itemsets are partitioned into fragments, each of which fits in the memory of the processor. In this case there is repeated scanning of the database to generate support counts. SPA, HPA and HPA-ELD partitioned the candidate itemsets over the memory space of all the processors. HPA-ELD replicates candidates with high support on all processors in order to reduce communications among the processors (Shintani & Kitsuregawa, 1996).

The disk I/O cost for NPA is high and no transaction data are exchanged among the processors in the second phase. SPA exploits the aggregate memory of the system by partitioning the candidate itemsets equally over the memory space of all the processors.

The I/O cost of SPA is low but the communication cost is high. HPA partitions the candidate itemsets among the processors using a hash function, which eliminates the need to broadcast all the transaction data. HPA has low I/O and communication costs. HPA-ELD utilizes the total system memory by copying some of the itemsets. It selects the most frequently occurring itemsets and copies them over the processors so that all the memory space is used which helps to reduce communication among the processors. These frequently occurring itemsets are counted locally, at all the processors. HPA-ELD has low I/O and communication costs and is also capable of skew handling (Shintani & Kitsuregawa, 1996).

Zaki, et al. (1997) highlighted the limitations of current parallel algorithms such as Count Distribution, Data Distribution, and Candidate Distribution. These algorithms make repeated passes over the disk-resident database partition incurring high I/O overheads. In addition there is an exchange of count of candidates at the remote database partitions during each iteration. The Equivalence Class Transformation (ECLAT) algorithm, proposed by Zaki, et al. (1997) is a parallel algorithm that clusters related frequent itemsets and transactions. The interconnection of the processors allows a user-level application to write the memory of remote nodes, which makes it possible to have very fast user-level messages and low synchronization costs. ECLAT clusters related groups of itemsets using equivalence class partitioning while clustering transactions using the vertical database layout technique. The performance of ECLAT was found to be better than that of the Count Distribution algorithm.

2.2.7 Current State and Existing Methodologies

Several parallel algorithms have been proposed for the mining of association rules including Count Distribution (CD), Data Distribution (DD), Intelligent Data Distribution (IDD) and Hybrid Distribution (HD). The transaction database shown in Table 2.2.6 will be used to illustrate examples of these algorithms.

Table 2.2.6 Transaction Database

TID	List of Items
100	I1 I2 I5
200	I2 I4
300	I2 I3
400	I1 I2 I4
500	I1 I3
600	I2 I3
700	I1 I3
800	I1 I2 I3 I5
900	I1 I2 I3

2.2.7.1 Count Distribution (CD) Algorithm.

CD divides the database among the processors and stores all the candidates at each processor. The entire hash tree is stored at each processor, in addition each processor counts how many times each candidate itemset appears in the transactions stored in local memory. The global counts of the candidates are computed by summing all of the local counts at each processor. Each processor executes the serial Apriori algorithm on the locally stored transactions.

The main drawback with CD is that the building of the hash tree is not done in a parallel manner. The problem with this approach is that with a large number of processors this step can be a bottleneck and secondly if the number of candidates is large, the hash tree may be too big to fit in memory making it necessary to partition it. In this case the local transactions must be read once for each partition. This can be expensive on machines with slow I/O systems. The CD algorithm is effective for small number of distinct items and a high minimum support.

Count Distribution Algorithm (Agrawal & Shafer, 1996).

The first pass is special. For all other passes $k > 1$, the algorithm works as follows:

1. Each processor P^i generates the complete C_k, using the complete frequent itemset L_{k-1} created at the end of pass k-1. Observe that since each processor has identical L_{k-1}, they will be generating identical C_k.
2. Processor P^i makes a pass over its data partition D^i and develops local support counts for candidates in C_k.
3. Processor P^i exchanges local C_k counts with all other processors to develop global C_k counts. Processors are forced to synchronize in this step.
4. Each processor P^i now computes L_k from C_k.
5. Each processor P^i independently makes the decision to terminate or continue to the next pass. The decision will be identical as the processors all have identical L_k.

An illustration of the algorithm is shown in Figure 2.2.3 (Agrawal & Shafer, 1996).

Computation Using CD

The following example is an illustration of how the CD algorithm works using the sample database in Table 2.2.6. In this example we use a minimum transaction support count of 2.

The database will be divided among the 4 processors as shown in Figure 2.2.4. The computations for the 3-itemsets are shown in Figure 2.2.5 and Figure 2.2.6. The frequent 3-itemsets are shown in Figure 2.2.6. However, only {I1, I2, I3} has met the required minimum transaction support of 2 and will be used to generate the rules.

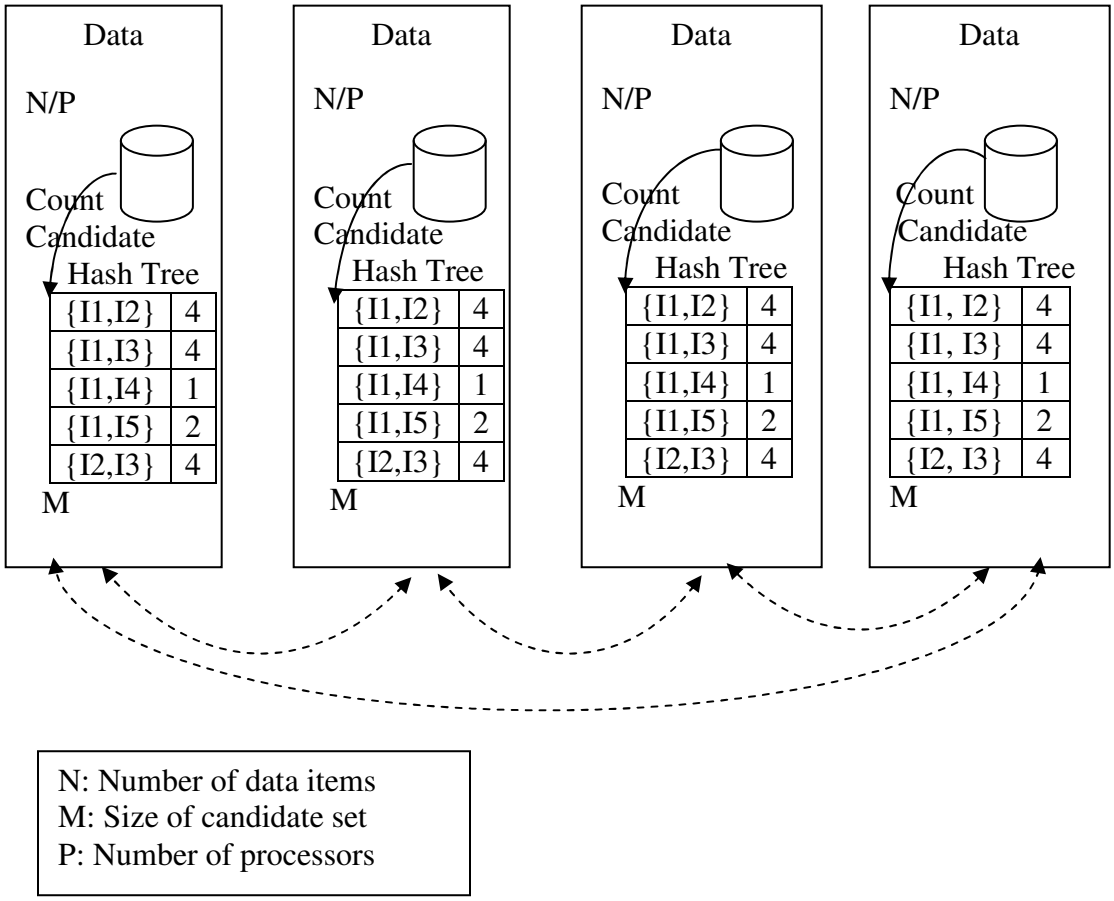


Figure 2.2.3 Count Distribution (CD) Algorithm (Agrawal & Shafer, 1996).

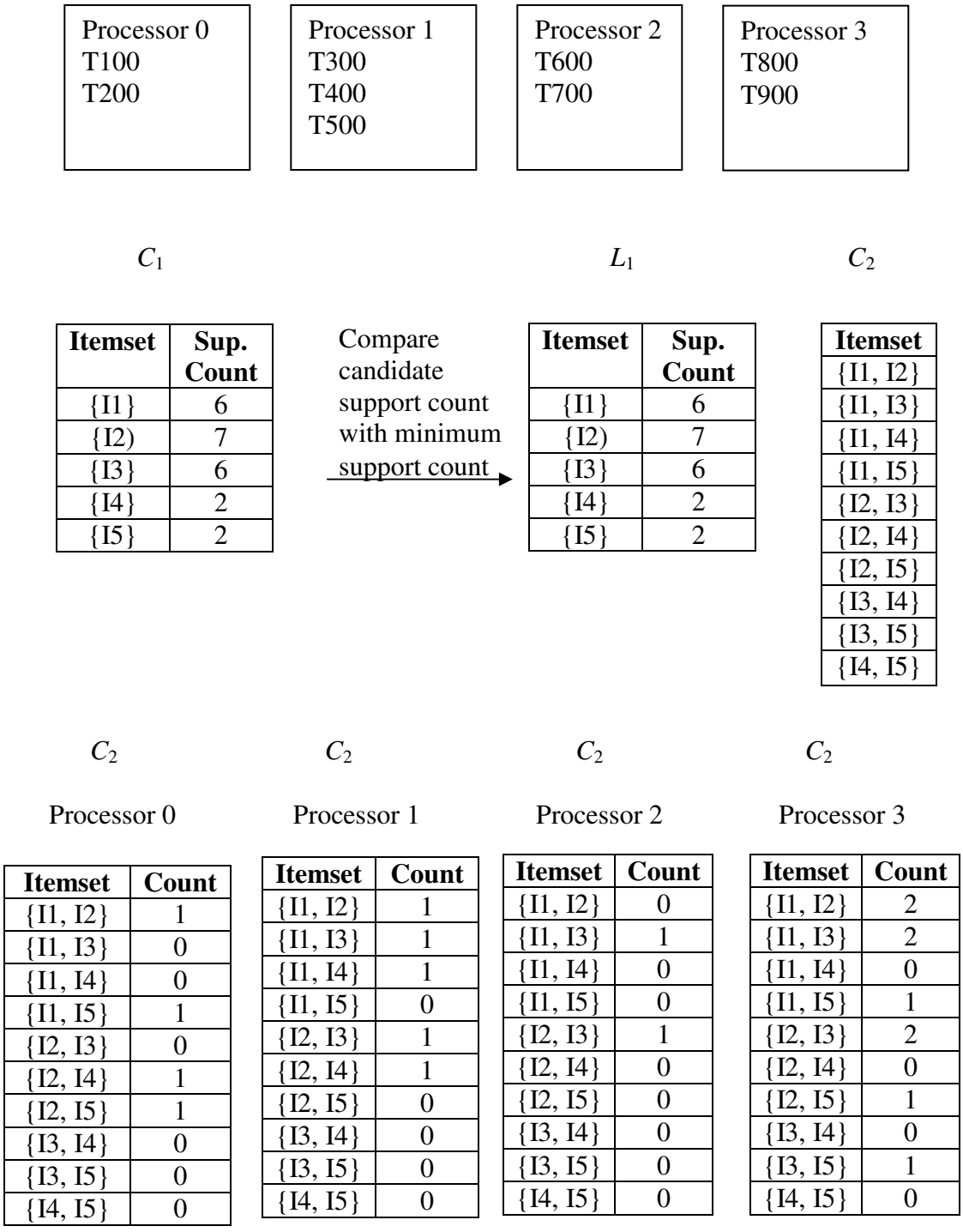


Figure 2.2.4 Local Count for CD

C_2 Processor 0

Itemset	Count
{I1,I2}	4
{I1,I3}	4
{I1,I4}	1
{I1,I5}	2
{I2,I3}	4
{I2,I4}	2
{I2,I5}	2
{I3,I4}	0
{I3,I5}	1
{I4,I5}	0

C_2 Processor 1

Itemset	Count
{I1,I2}	4
{I1,I3}	4
{I1,I4}	1
{I1,I5}	2
{I2,I3}	4
{I2,I4}	2
{I2,I5}	2
{I3,I4}	0
{I3,I5}	1
{I4,I5}	0

C_2 Processor 2

Itemset	Count
{I1,I2}	4
{I1,I3}	4
{I1,I4}	1
{I1,I5}	2
{I2,I3}	4
{I2,I4}	2
{I2,I5}	2
{I3,I4}	0
{I3, I5}	1
{I4, I5}	0

C_2 Processor 3

Itemset	Count
{I1, I2}	4
{I1, I3}	4
{I1, I4}	1
{I1, I5}	2
{I2, I3}	4
{I2, I4}	2
{I2, I5}	2
{I3, I4}	0
{I3, I5}	1
{I4, I5}	0

L_2 Processor 0

Itemset	Count
{I1, 2}	4
{I1,I3}	4
{I1,I5}	2
{I2,I3}	4
{I2,I4}	2
{I2,I5}	2

C_3

Itemset
{I1, I2, I3}
{I1, I2, I5}

Figure 2.2.5 Count After Global Reduction for CD

Processor 0

Itemset	Count
{I1, I2, I3}	0
{I1, I3, I5}	0

Processor 1

Itemset	Count
{I1, I2, I3}	0
{I1, I3, I5}	0

Processor 2

Itemset	Count
{I1, I2, I3}	0
{I1, I3, I5}	0

Processor 3

Itemset	Count
{I1, I2, I3}	2
{I1, I3, I5}	1

Figure 2.2.6 Local Count of 3-Itemset for CD

The count for each candidate is shown below:

Itemset	Count
{I1, I2, I3}	2
{I1, I3, I5}	1

The algorithm terminates at this point and the frequent itemsets used to generate the rules.

2.2.7.2 Data Distribution (DD) Algorithm

DD partitions the candidate itemsets among the processors in a round-robin fashion. Each processor is now responsible for computing the count of the locally stored subset of the candidate itemsets for all the transactions in the database. Since each processor is assigned a specific subset of the candidate itemsets it is now necessary to scan the rest of the transactions stored in the memory of the other processors in addition to the locally assigned transactions. After computing the count of its candidate itemsets, each processor finds the frequent itemsets from the local candidate itemsets and sends these to all other processors.

Total available memory is better utilized since the candidate itemsets are partitioned among p processors. However this algorithm was found to be slower than the Count Distribution (CD) algorithm. The communication pattern of this algorithm causes three problems. First, for each pass of the algorithm each processor sends to all the other processors the portion of the database that resides locally. Each processor reads the locally stored portion of the database one page at a time and sends it to all the other processors by issuing $p - 1$ send operations. Similarly, each processor issues a receive operation from each other processor in order to receive these pages. If the interconnection network of the underlying parallel computer is fully connected and each processor can

receive data on all incoming links simultaneously, then this communication pattern will lead to a very good performance. An illustration of the algorithm is shown in Figure 2.2.7.

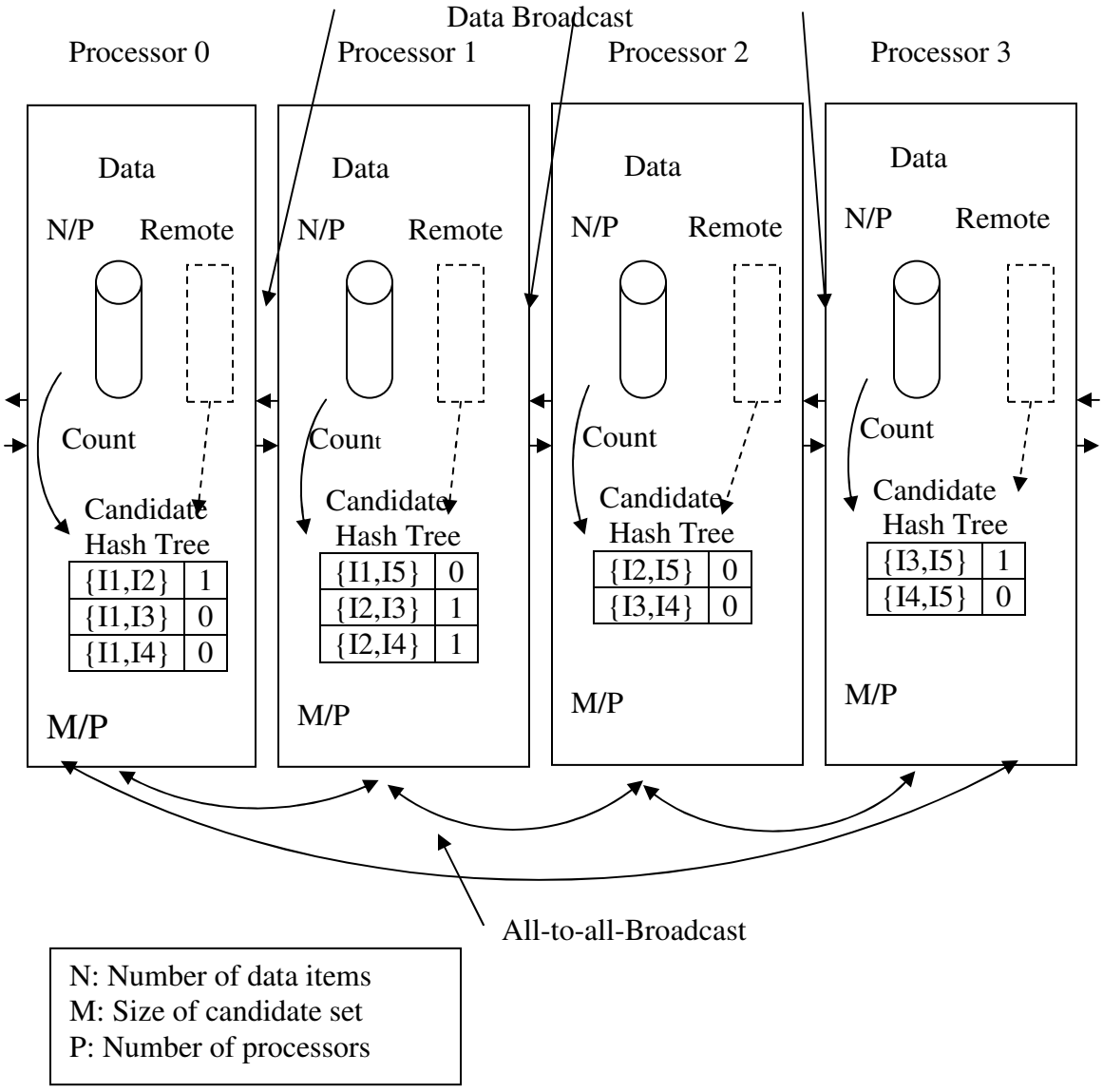


Figure 2.2.7 Data Distribution (DD) Algorithm (Agrawal & Shafer, 1996).

Data Distribution (Agrawal & Shafer, 1996).

Pass 1: Same as CD.

Pass $k > 1$:

1. Processor P^i generates C_k from L_{k-1}. It retains only 1/Nth of the itemsets forming the candidate subset C_k^i that it will count. Which 1/N itemsets are retained is determined by the processor id and can be computed without communicating with other processors. Itemsets are assigned in a round-robin fashion. The C_k^i sets are all disjoint and the union of all C_k^i sets is the original C_k.
2. Processor P^i develops support counts for the itemsets in its local candidate set C_k^i using both local data pages and data pages received from other processors.
3. At the end of the pass over the data, each processor P^i calculates L_k^i using the local C_k^i. Again, all L_k^i sets are disjoint and the union of all L_k^i sets is L_k.
4. Processors exchange L_k^i so that every processor has the complete L_k for generating C_{k+1} for the next pass. This step requires processors to synchronize. Having obtained the complete L_k, each processor can independently (but identically) decide whether to terminate or continue on to the next pass.

Computation Using DD

The following is an illustration of how the DD algorithm works using the sample database in Table 2.2.6. The database and itemsets are divided among the four processors as shown in Figure 2.2.8. Figure 2.2.9 shows the count of itemsets at each processor after one complete cycle.

In Figure 2.2.8 the database is divided among the four processors with transactions T100 and T 200 assigned to processor 0, transactions T300, T400 and T500 assigned to processor 1, transactions T600 and T700 assigned to processor 2 and transactions T800 and T900 assigned to processor 3. The candidate 2-itemsets are also divided among the processors with processor 0 assigned to {I1, I2}, {I1, I3} and {I1, I4}, processor 1 assigned to

{I1, I5}, {I2, I3}, and {I2, I4}, processor 2 assigned to {I2, I5} and {I3, I4} and processor 3 assigned to {I3, I5} and {I4, I5}. The count of the 2-itemsets assigned to each processor is first calculated using the transactions assigned to each processor as shown in Figure 2.2.8.

The top row of Figure 2.2.9 shows transactions 800 and 900, which are assigned, to processor 3 being used by processor 0 to update the count for each itemset assigned to it. Similarly transactions 100 and 200, which are assigned to processor 0, are being used by processor 1 to update the count of each itemset assigned to it. This is also the situation with the remaining processors in the top row. The results from the processing of these transactions can be seen by comparing the counts shown in Figure 2.2.8 with those shown in Figure 2.2.9.

The second row of Figure 2.2.9 shows transactions T600 and T700, which are assigned, to processor 2 being used by processor 0 to update the count for each itemset assigned to it while processor 1 is updating its count of itemsets using transactions T800 and T900. The transactions are shifted until every processor updates its count using transactions from all other processors.

After identifying the frequent itemsets assigned to it each processor then sends this information to all other processors to determine all frequent 2-itemsets. The process then repeats itself with the generation of candidate 3-itemsets. It is at this stage that all processors will decide independently whether to terminate or to go on to the next pass of the algorithm.

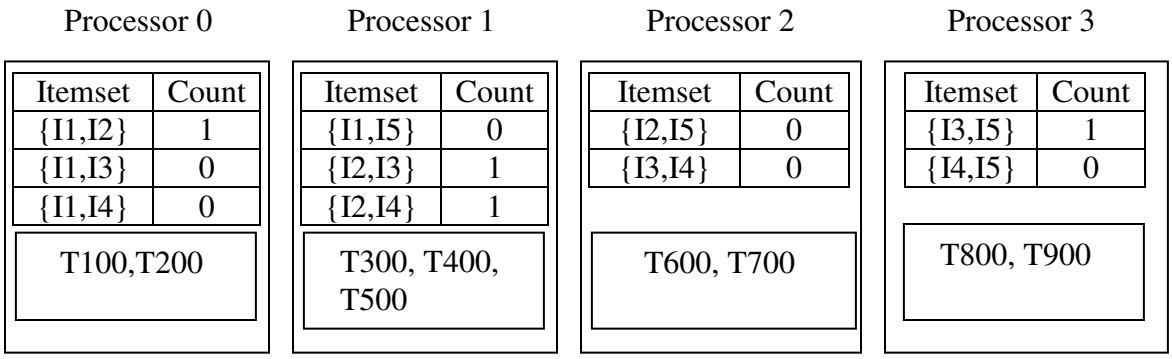


Figure 2.2.8 Count After Assigning Partitions to Processors for DD

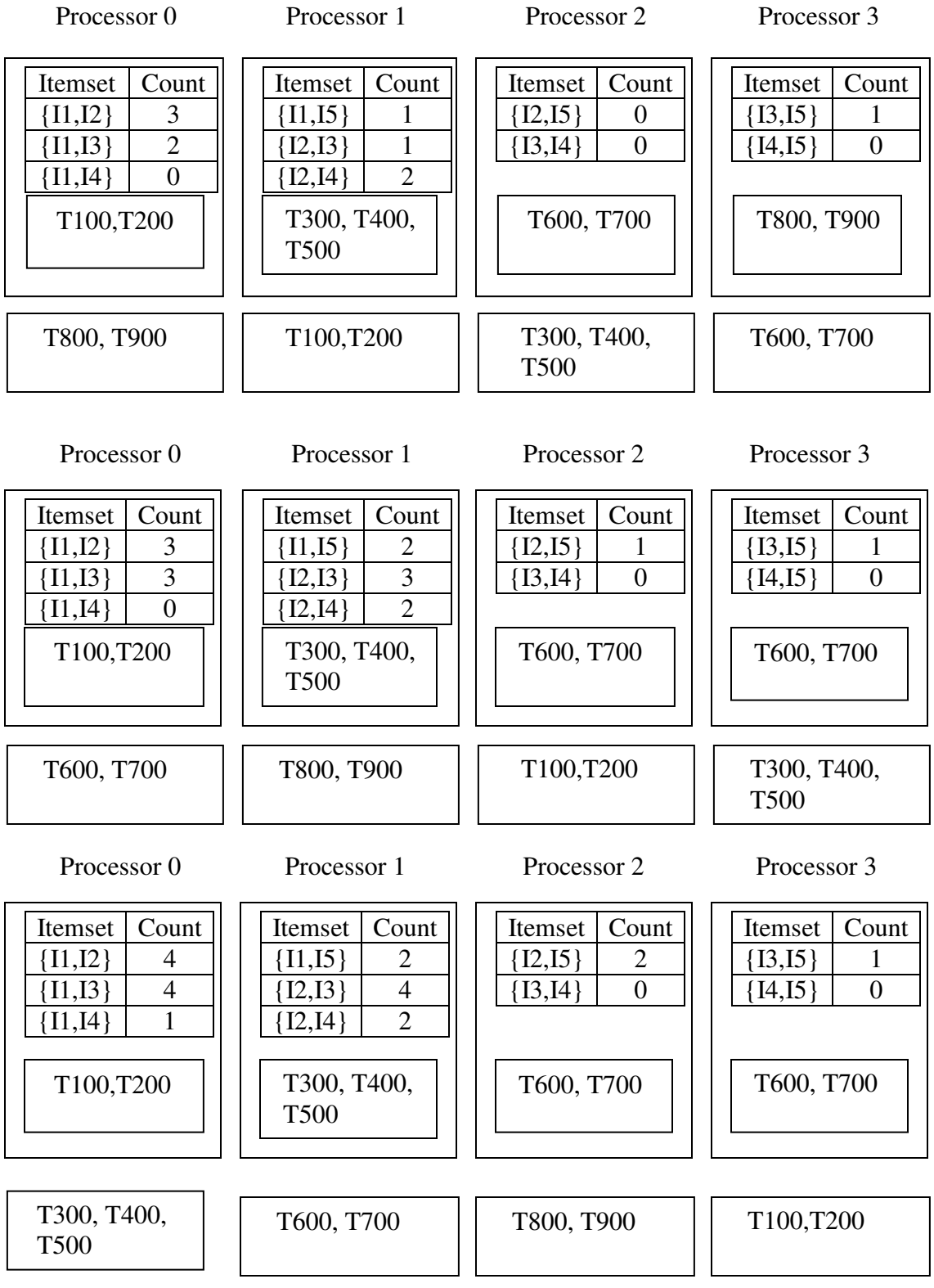


Figure 2.2.9 Count After Complete Cycle for DD

2.2.7.3 Intelligent Data Distribution (IDD) Algorithm

In the IDD algorithm the candidate itemset is partitioned among the processors. The database of transactions is also divided equally among the processors. The locally stored portions of the database are sent to all the other processors using a ring-based network. The ring network eliminates the contention problem that is associated with the DD algorithm. The pseudo code used for the movement of data is shown in Figure 2.2.10 (Han, et al., 2000).

In order to reduce the redundant work due to the partitioning of the candidate itemsets, it is partitioned in such a way that each processor gets itemsets that begin only with a subset of all possible itemsets. The transactions are then checked against this subset to determine if the hash tree contains candidates starting with these items. The hash tree is then traversed only with items in the transaction that belong to this subset, thereby eliminating the redundant work problem of DD.

An illustration of the IDD algorithm is shown in Figure 2.2.11 (Han, et al., 2000).

```
while (!done)
  FillBuffer(fd, Sbuf);
  for (k = 0; k < P-1; ++k) {
      /* send/receive data in non-blocking pipeline */
     MPI_Irecv(Rbuf, left);
     MPI_Isend(Dbuf, right);

     /* process transactions in Sbuf and update hash tree */
     Subset(Htree, Sbuf);

        MPI_Waitall();

        /* swap two buffers */
    tmp = Sbuf;
    Sbuf = Rbuf;
    Rbuf = tmp;
   }
   /* process transactions in Sbuf and update hash tree */
   Subset(Htree, Sbuf);
}
```

Figure 2.2.10 Pseudo Code for Data Movements for IDD (Han et al., 2000).

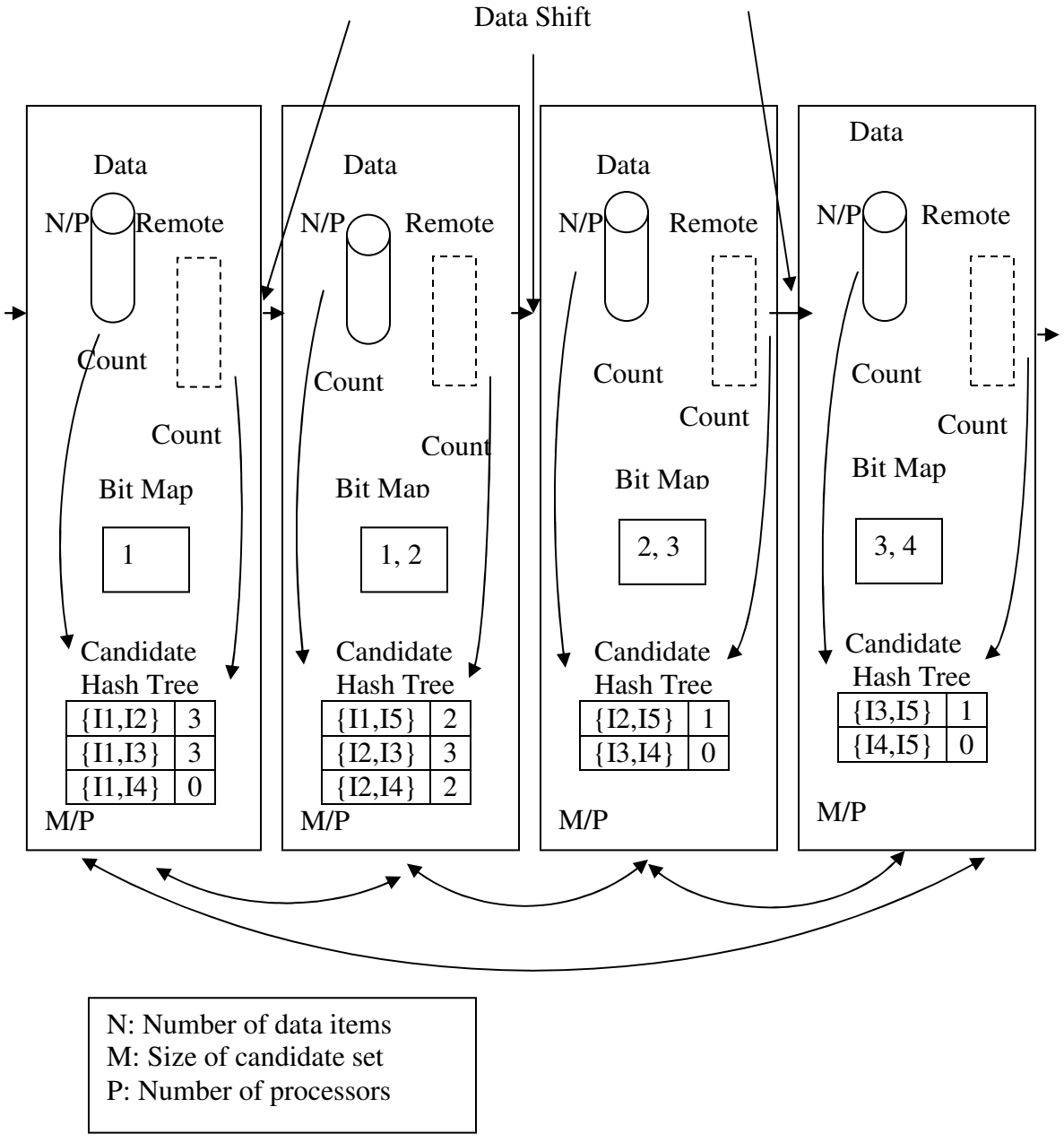


Figure 2.2.11 Intelligent Data Distribution (IDD) (Han, Karypis, & Kumar, 2000).

The IDD uses a bit-map at each processor to store the first item of the candidates assigned to the processor. Each processor filters every item of the transaction by checking against the bit-map to see if the processor contains candidates starting with that item of the transaction. This reduces the number of transaction data that has to go through the hash tree resulting in a reduction of the number of computations.

A fundamental requirement of this algorithm is good load balancing. In this case one of the criteria of a good partitioning algorithm is that there are an equal number of candidates in all the processors. This will result in the same size hash tree in all the processors. A round-robin partitioning technique is not likely to result in good load balancing.

The IDD algorithm uses bin-packing to partition the candidate itemsets. For each item, the number of candidate itemsets starting with it is computed. The algorithm only stores the number of items and not the itemset starting with an item. The system then uses bin packing to partition these items into P buckets such that the sum of number of the candidate itemsets starting with these items in each bucket are approximately equal. Each processor then regenerates and stores the candidate itemsets that are assigned to it.

It is important to note that equal assignment of candidates to the processors does not guarantee the perfect load balance among processors. This is due to the fact that the cost of traversal and checking at the leaf node are determined not only by the size and shape of the candidate hash tree, but also by the actual items in the transactions. Since it is difficult to estimate the effect of transactions on the workload in advance, the scheme is designed to target the equal distribution of candidates among the processors.

Computation using IDD

The following example is an illustration of how the IDD algorithm works using the sample database in Table 2.2.6. The database and itemsets will be divided among the four processors as shown in Figure 2.2.12. The movement of local data among the processors and the count of itemsets after one cycle are shown in Figure 2.2.12 and Figure 2.2.13. In Figure 2.2.12 the database is divided among the four processors with transactions 100 and 200 assigned to processor 0, transactions 300, 400 and 500 assigned to processor 1, transactions 600 and 700 assigned to processor 2 and transactions 800 and 900 assigned to processor 3. The candidate 2-itemsets are also divided among the processors with processor 0 assigned to {I1, I2}, {I1, I3} and {I1, I4}, processor 1 assigned to {I1, I5}, {I2, I3}, and {I2, I4}, processor 2 assigned to {I2, I5} and {I3, I4} and processor 3 assigned to {I3, I5} and {I4, I5}. The count of the 2-itemsets assigned to each processor is first calculated using the transactions assigned to each processor as shown in Figure 2.2.12.

The bit map used for processor 0 is 1, which means that only transactions with item 1 in it will be processed by this processor. Similarly the bit map for processor 1 is 1 and 2, which means that only transactions that contain these items will be processed by processor 1.

The second row of Figure 2.2.12 shows the count of 2-itemsets following the movement of transactions among the processors. The count of 2-itemsets at processor 0 is updated after processing transactions 800 and 900, which are assigned to processor 3. Similarly the count of itemsets at processor 1 is updated after processing transactions that are assigned to processor 0. In the IDD the bit map is used to filter transactions that do

not contain items in the bit map for a given processor. Looking at processor 0 it is obvious that transactions 200, 300 and 600 will not be passed through the hash tree as they do not contain item 1. These eliminate unnecessary traversal of the hash tree. Every processor will process the transactions stored at all other processors in order to update the count of itemsets assigned to it.

After identifying the frequent itemsets assigned to it each processor then sends this information to all other processors to determine all frequent 2-itemsets. The process then repeats itself with the generation of candidate 3-itemsets. It is at this stage that all processors will decide independently whether to terminate or to go on to the next pass of the algorithm.

Figure 2.2.13 shows the count of 2-itemsets assigned to each processor at the completion of processing the transactions assigned to all other processors. This information is then exchanged among the processors for the commencement of the next cycle to determine the 3-itemsets.

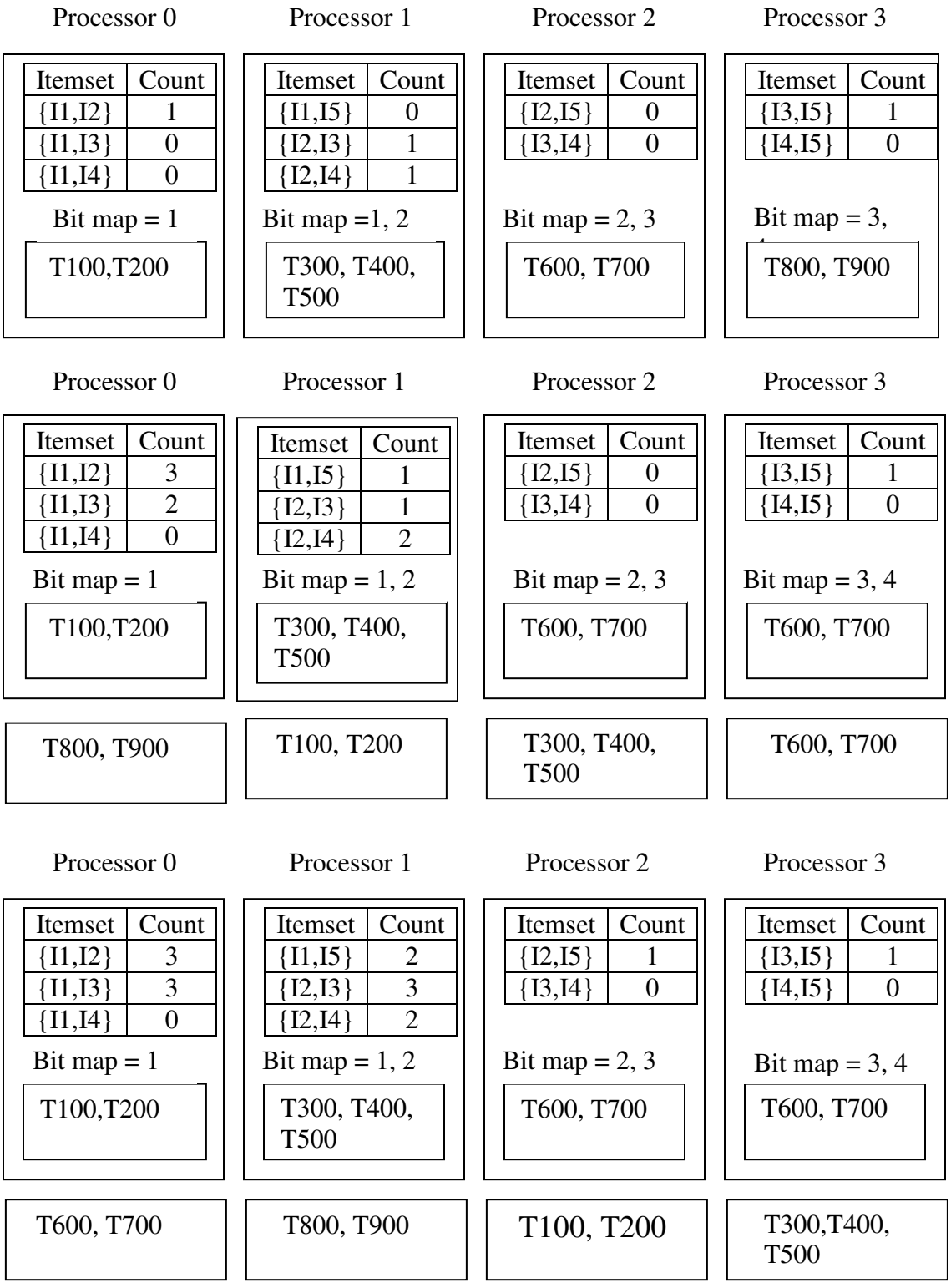


Figure 2.2.12 Movement of Local Data Among Processors for IDD

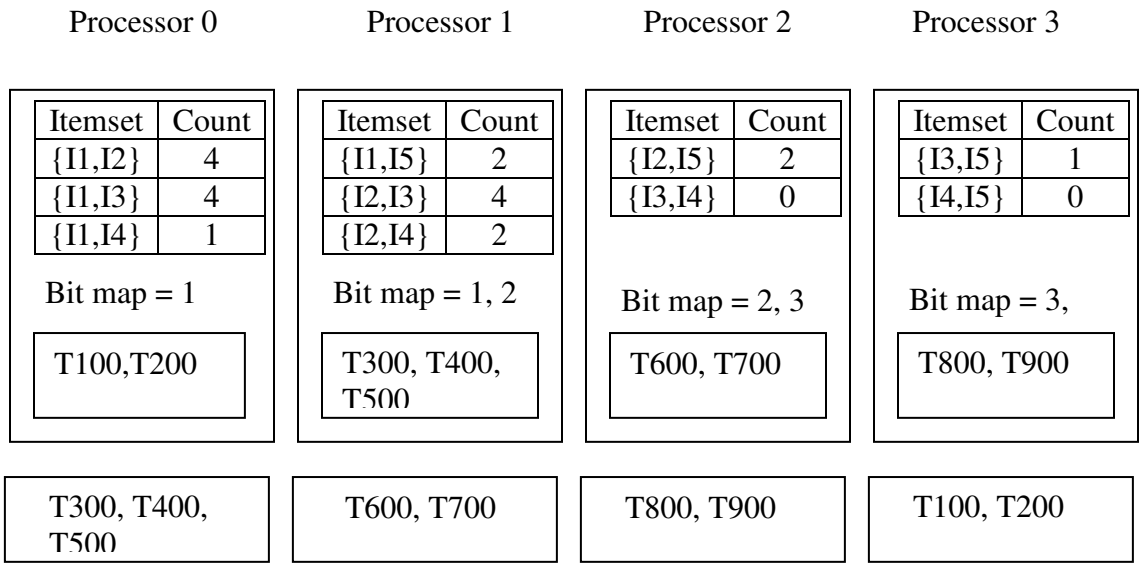


Figure 2.2.13 Count of Itemsets After one Cycle for IDD

2.2.7.4 Hybrid Distribution (HD) Algorithm

As more processors are added in IDD, the number of candidates assigned to each processor decreases. A reduction in the number of candidates per processor makes it more difficult to balance the work. In addition the smaller number of candidates gives a smaller hash tree and less computation work per transaction. It is possible for the amount of computation to be less than the communication involved. This is more easily seen in the latter passes of the algorithm as the hash tree size further decreases. The overall efficiency of the parallel algorithm will be reduced. This can be a serious problem in a system that cannot perform asynchronous communication.

The problems associated with the IDD are addressed by combining the CD and IDD algorithms to form the HD algorithm. In this approach a P-processor system is split into G equal groups, each containing P/G processors. The database transactions are partitioned into P/G parts each of size $N/(P/G)$. The computation of the candidate set C_k for each subset of the transactions is assigned to each one of the P/G processors. IDD is then used to compute the counts within each group. By applying IDD within each group the transactions and the candidate set C_k are partitioned among the processors of each group, so that each processor gets roughly $|C_k|/G$ candidate itemsets and N/P transactions. The overall count is computed by performing a reduction operation among the P/G groups of processors.

Figure 2.2.14 to Figure 2.2.16 show the steps used by HD to compute the frequent itemsets (Han, et al., 2000). It can be visualized as consisting of G rows and P/G columns. The transactions are partitioned equally among the P processors. The candidate

set C_k is partitioned among the processors of each column. All the processors in a row get the same subset of C_k. The CD algorithm is executed in Figure 2.2.16 as if there were only four processors since there are four columns. The database transactions are partitioned in four parts and each one of these hypothetical processors computes the local counts of all the candidate itemsets. The global counts are then computed using the global reduction operation. The computation of local counts of the candidate itemsets in a hypothetical processor requires the computation of the counts of the candidate itemsets on the database transactions stored on the three processors. The IDD algorithm is executed within each of the four hypothetical processors in order to perform this operation. Each processor has complete count of its local candidates for all the transactions located in the processors of the same column. A reduction operation is then performed along the rows such that all processors in each row have the sum of the counts for the candidates in the same row. The count associated with each candidate itemset corresponds to the entire database of transactions. Each processor will now find frequent itemsets and drops all candidate itemsets with frequency less than the threshold for minimum support. These are shown in Step 2 (Figure 2.2.15). In Step 3 (Figure 2.2.16) each processor performs an all-to-all broadcast operation along the columns of the processor mesh. The processors are now ready to proceed to the next pass.

The HD algorithm partitions the candidate set into a big enough section and assigns a group of processors to each partition. If m is a user specified threshold and the total number of candidates M is less than m, then the HD algorithm makes G equal to 1, which means that the CD algorithm is run on all the processors. Otherwise G is set to $\lceil M/n \rceil$.

Computation Using Hybrid

The HD algorithm inherits all the good features of the IDD algorithm. It also provides good load balance and enough computation work by maintaining minimum number of candidates per processor. The amount of data movement has been cut down to 1/*G* of the IDD.

An illustration of the algorithm using the sample database in Table 2.2.7 is shown in the set of figures starting from Figure 2.2.17 to Figure 2.2.21. In Figure 2.2.17 there are 12 processors divided into four equal groups each consisting of three processors. The database transactions are partitioned into 3 parts each of size 4. The HD algorithm executes CD as if there were only 3 processors. In this case the 3 processors correspond to the 3 columns. The database of transactions is therefore divided into 3 parts where each part is assigned to each column. We can view a column as a hypothetical processor that will use the portion of database assigned to it to compute the counts for the candidate itemsets as is done in CD. A global count is then accomplished by a global reduction operation.

Each column consists of 4 processors and HD uses IDD to compute the counts of the candidate itemsets at each of these 4 processors. In Figure 2.2.17 the candidate items are partitioned among the four groups. From Figure 2.2.17 it can be seen that all the candidates in a group (row) are the same for the processors in that row. The transactions assigned to a column are then divided among the processors in the column. In our example there are 12 transactions, 12 processors and 4 groups. We assign 4 transactions to each column. Within each column there are 4 processors and we divide the 4

transactions among the 4 processors resulting in 1 transaction for each processor as shown in Figure 2.2.17.

Figure 2.2.17 also shows the count for candidate itemsets assigned to each processor using the transaction database that is assigned to each. Figure 2.2.18 to Figure 2.2.20 show the use of IDD to compute the count for candidate itemsets along each column. There is a change in the count of itemset for the processor at row1 and column 1 in Figure 2.2.18 as a result of processing transaction 400, which is assigned to the processor at row 4 column 1. This is also true for a number of other processors.

Figure 2.2.21 shows the use of CD to compute the counts for all the itemsets assigned to each group. It can be observed in Figure 2.2.21 that the counts at each processor in a row are the same for all the processors. The next step in HD, which is not shown, is a broadcast of all the frequent itemsets to all processors.

Step 1: Partitioning of Candidate Sets and Data Movement Along the Columns

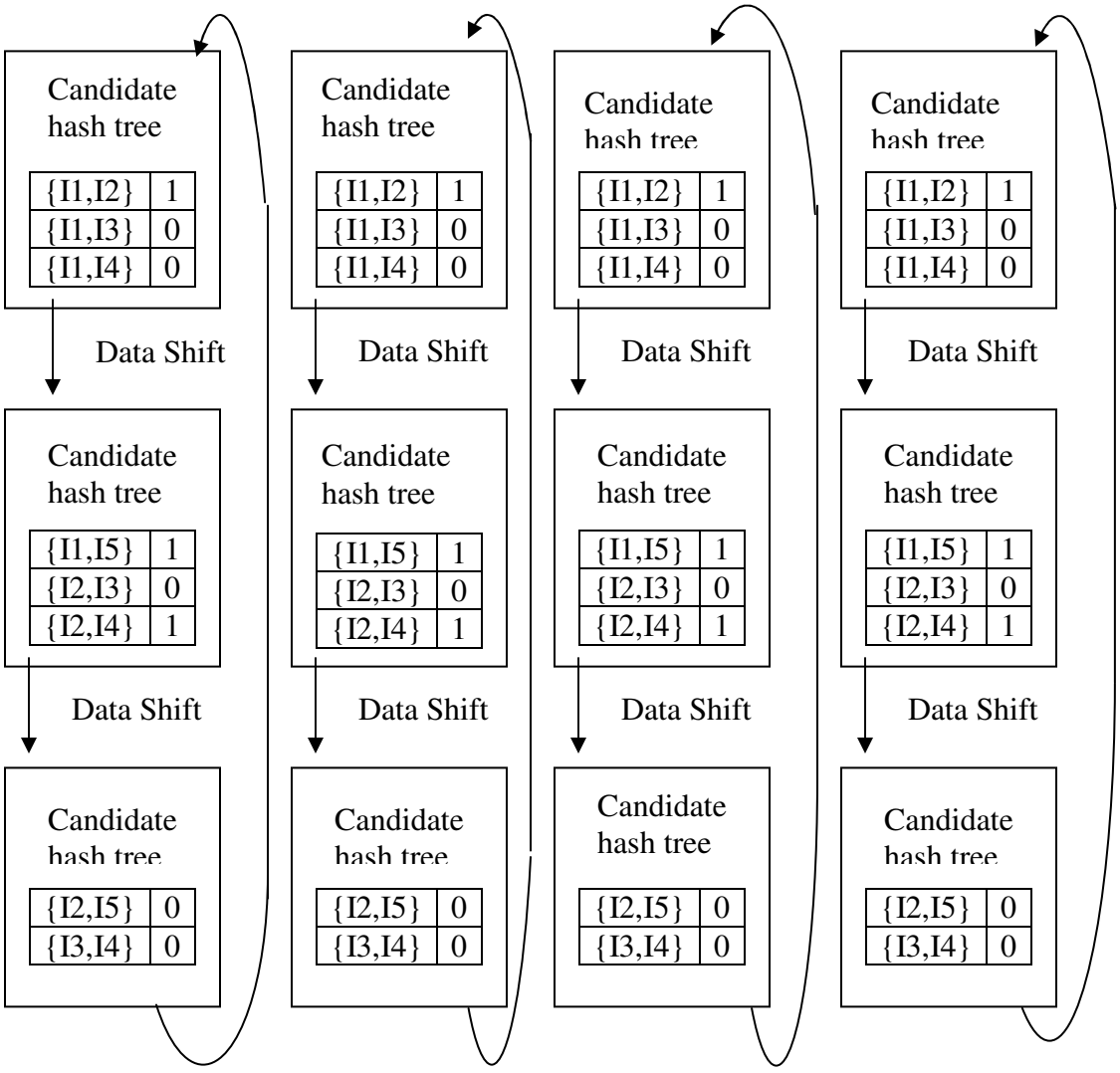


Figure 2.2.14 Data Movement Along Columns for HD (Han, Karypis & Kumar, 2000).

Step 2: Reduction Operation Along the Rows

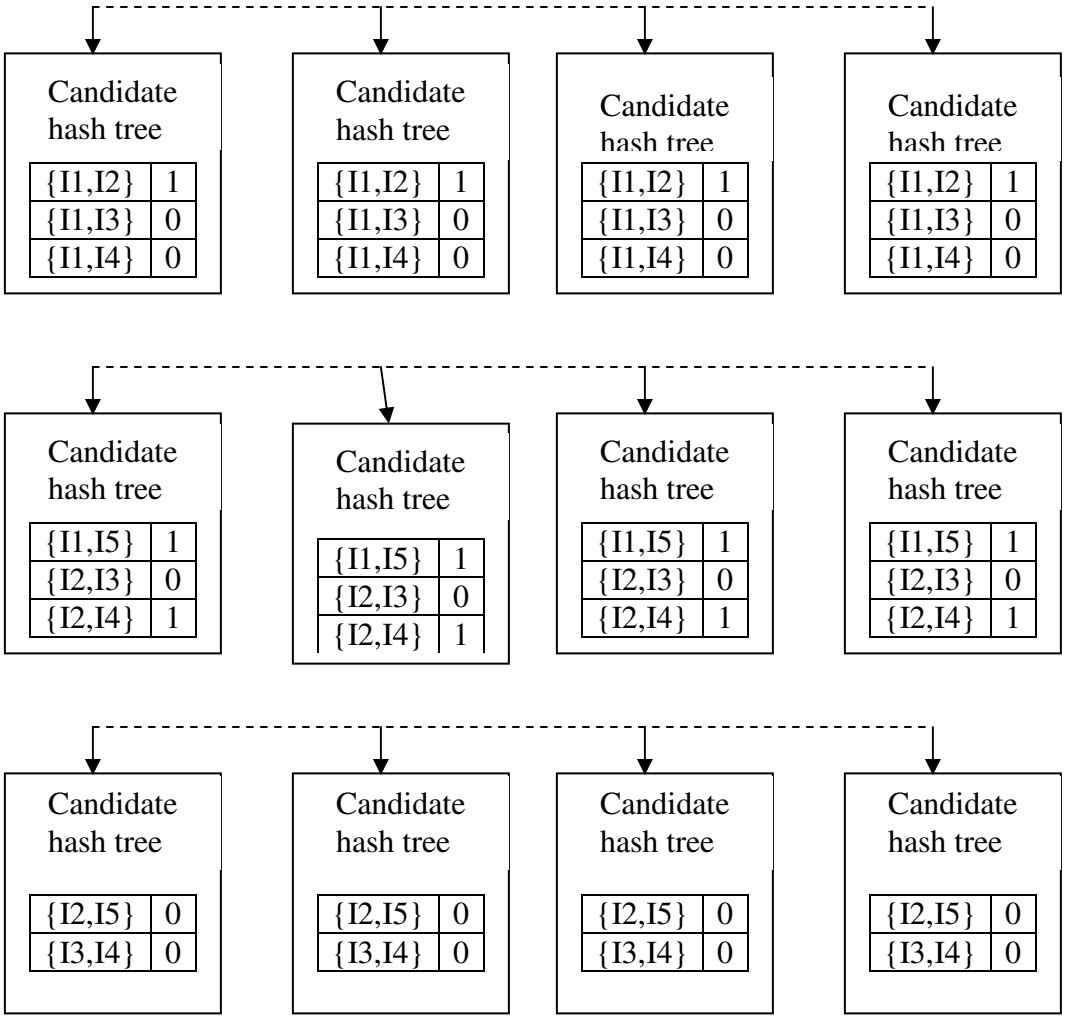


Figure 2.2.15 Reduction Operation Along Rows for HD (Han, Karypis & Kumar, 2000).

Step 3: All-to-All Broadcast Along Columns

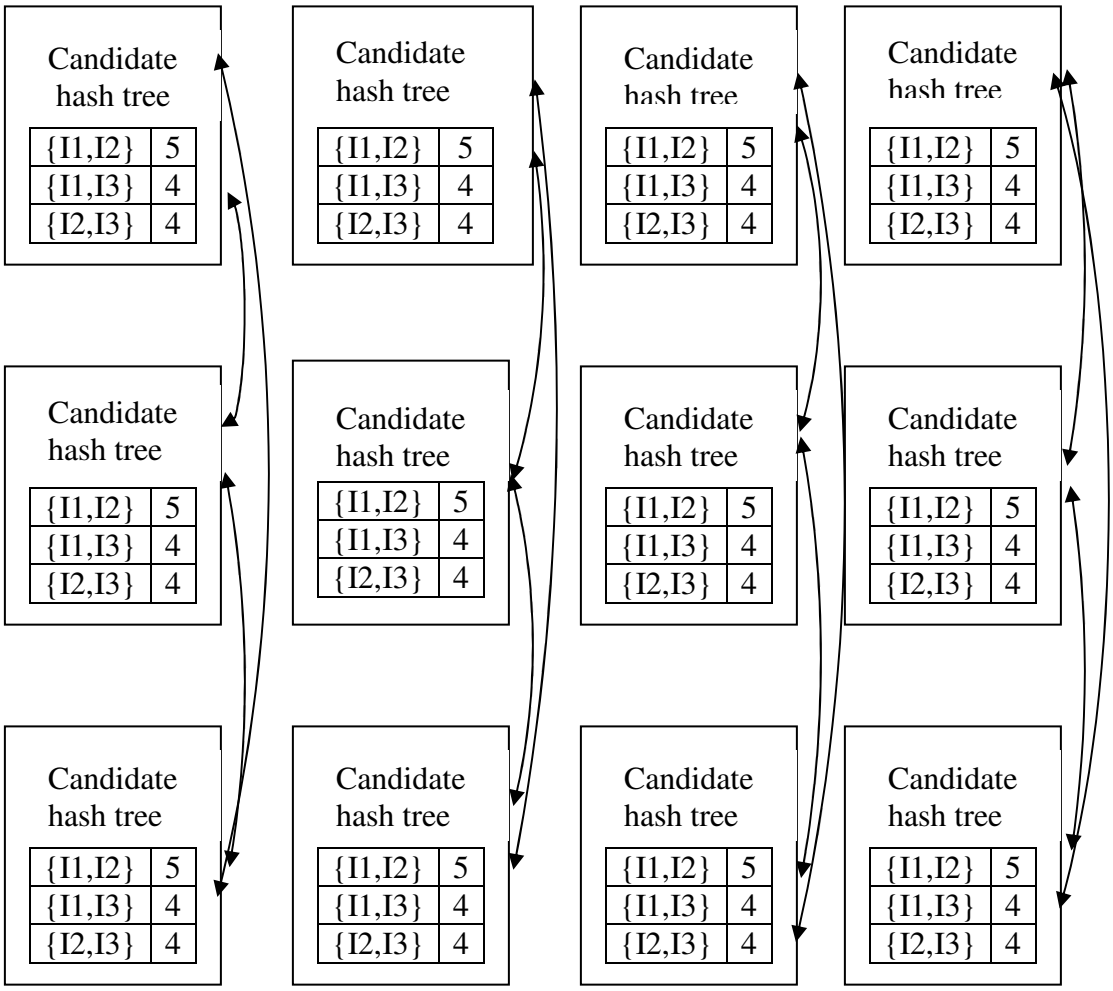


Figure 2.2.16 Hybrid Distribution (HD) (Han, Karypis & Kumar, 2000).

Table 2.2.7 Sample Database for HD Algorithm

TID	**List of Items**
T100	I1, I2, I5
T200	I2, I4
T300	I2, I3
T400	I1, I2, I4
T500	I1, I3
T600	I2, I3
T700	I1, I3
T800	I1, I2, I3, I5
T900	I1, I2, I3
T1000	I2, I3
T1100	I1, I2
T1200	I2, I4

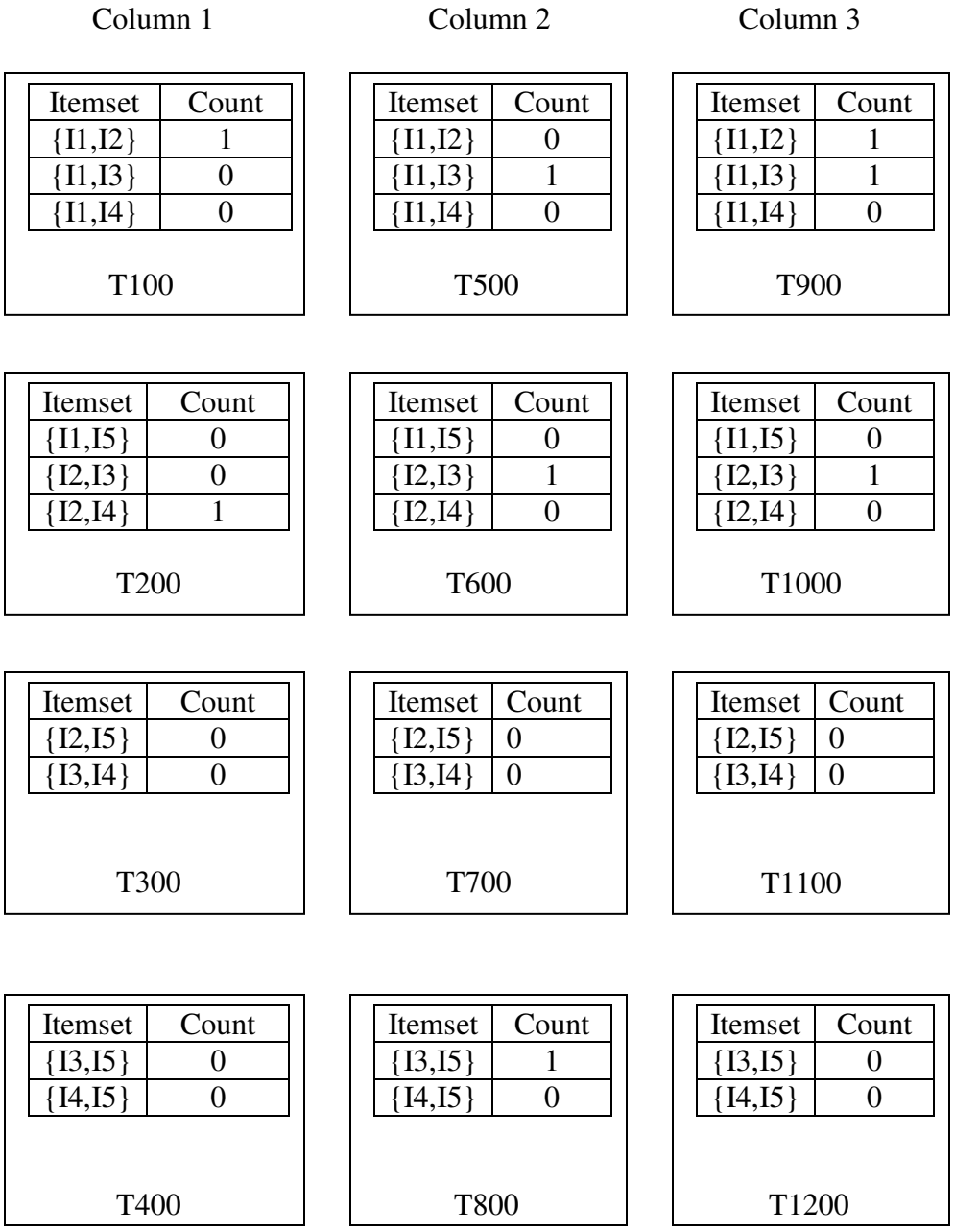


Figure 2.2.17 Initial Count for HD

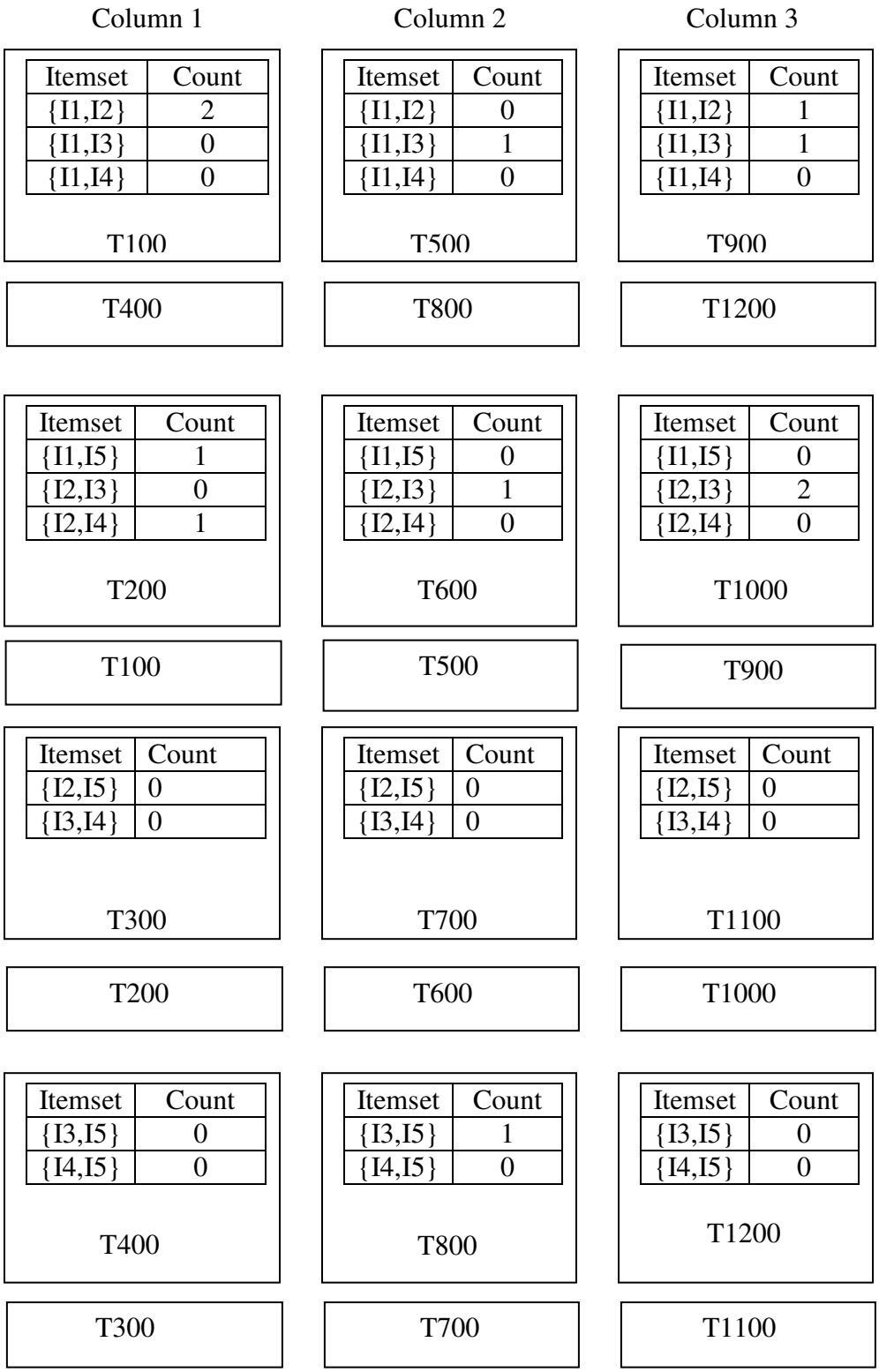


Figure 2.2.18 Data Movement Along Columns for HD (1)

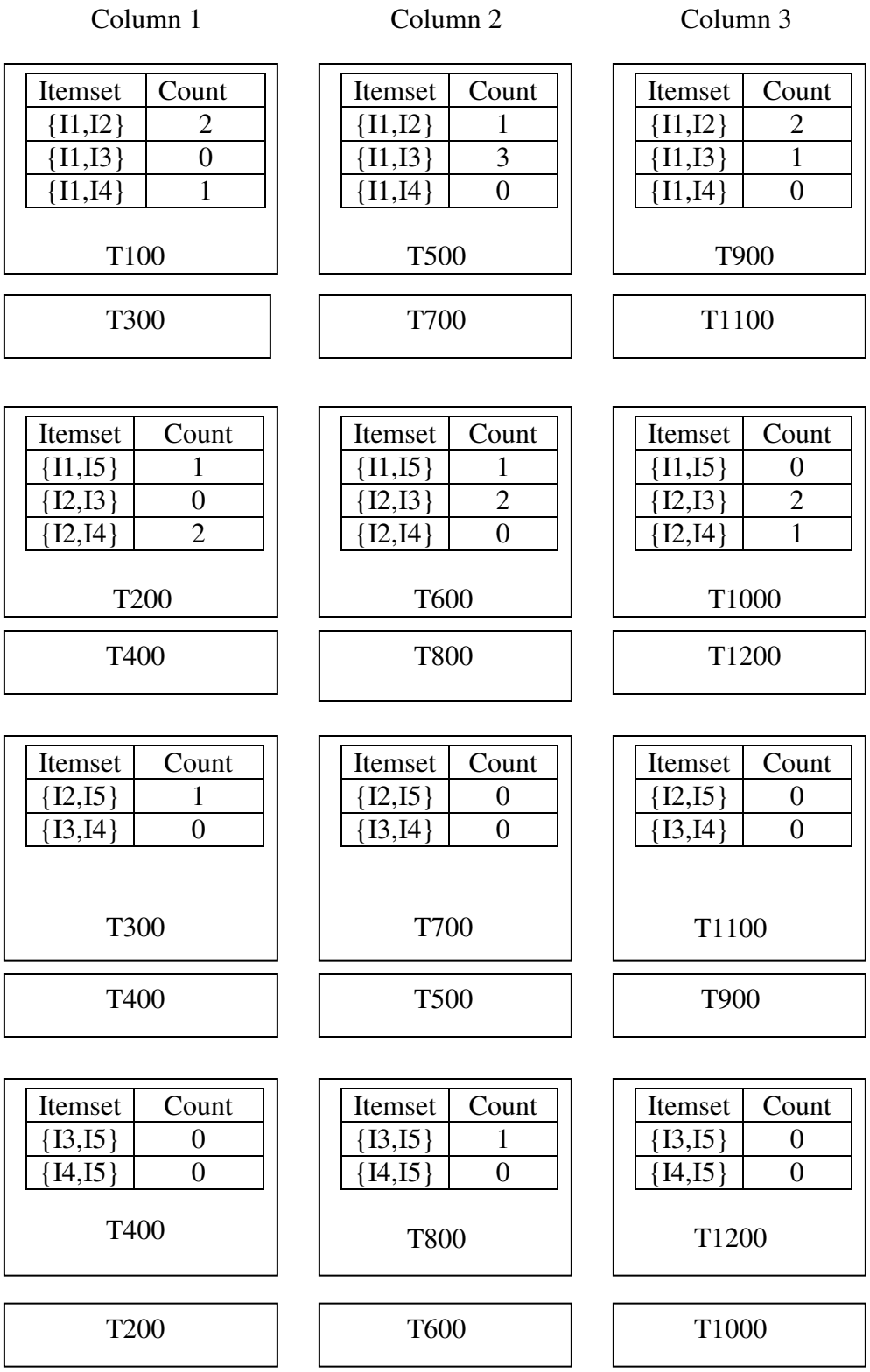


Figure 2.2.19 Data Movement Along Columns for HD (2)

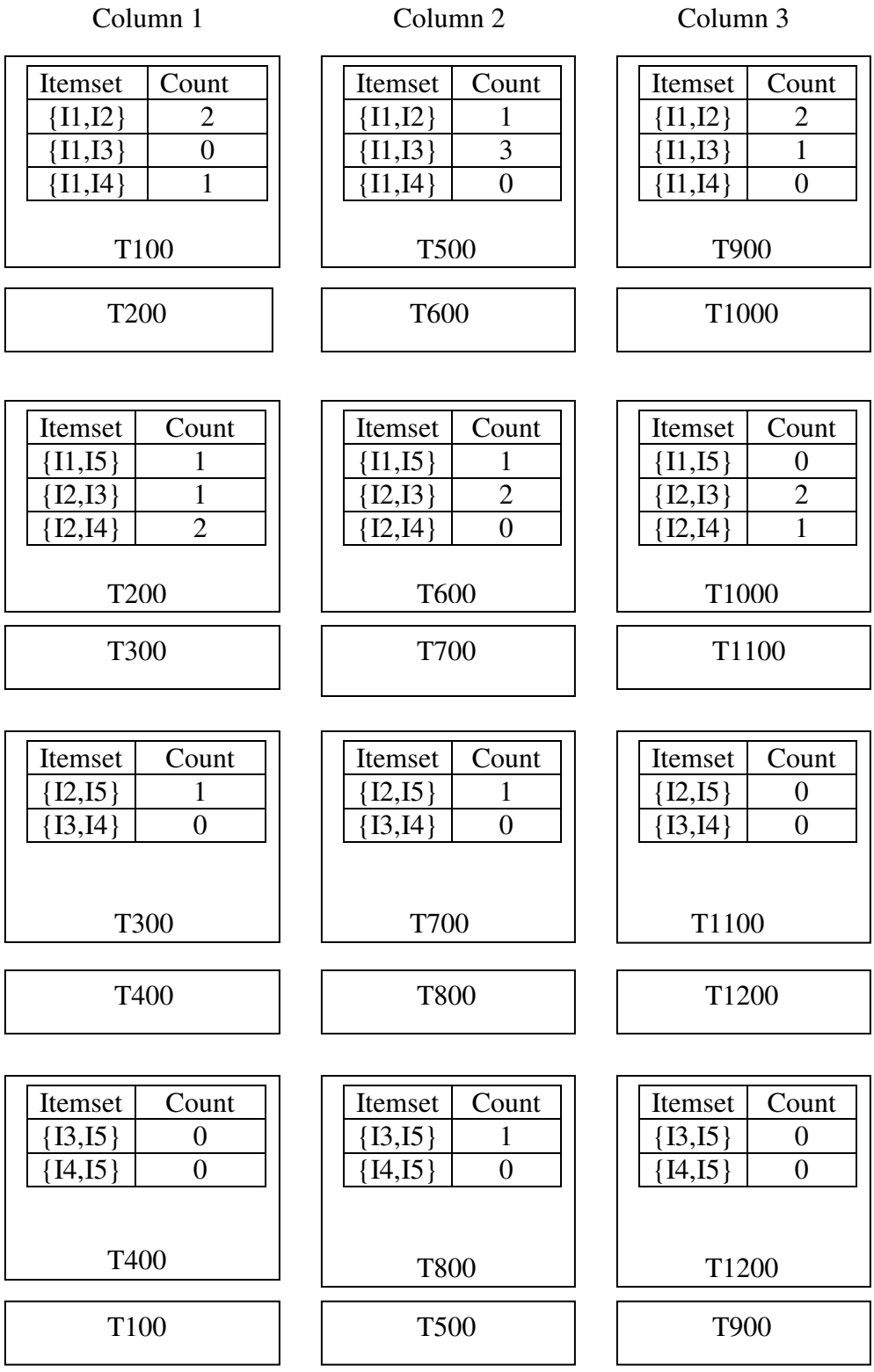


Figure 2.2.20 Data Movement Along Columns for HD (3)

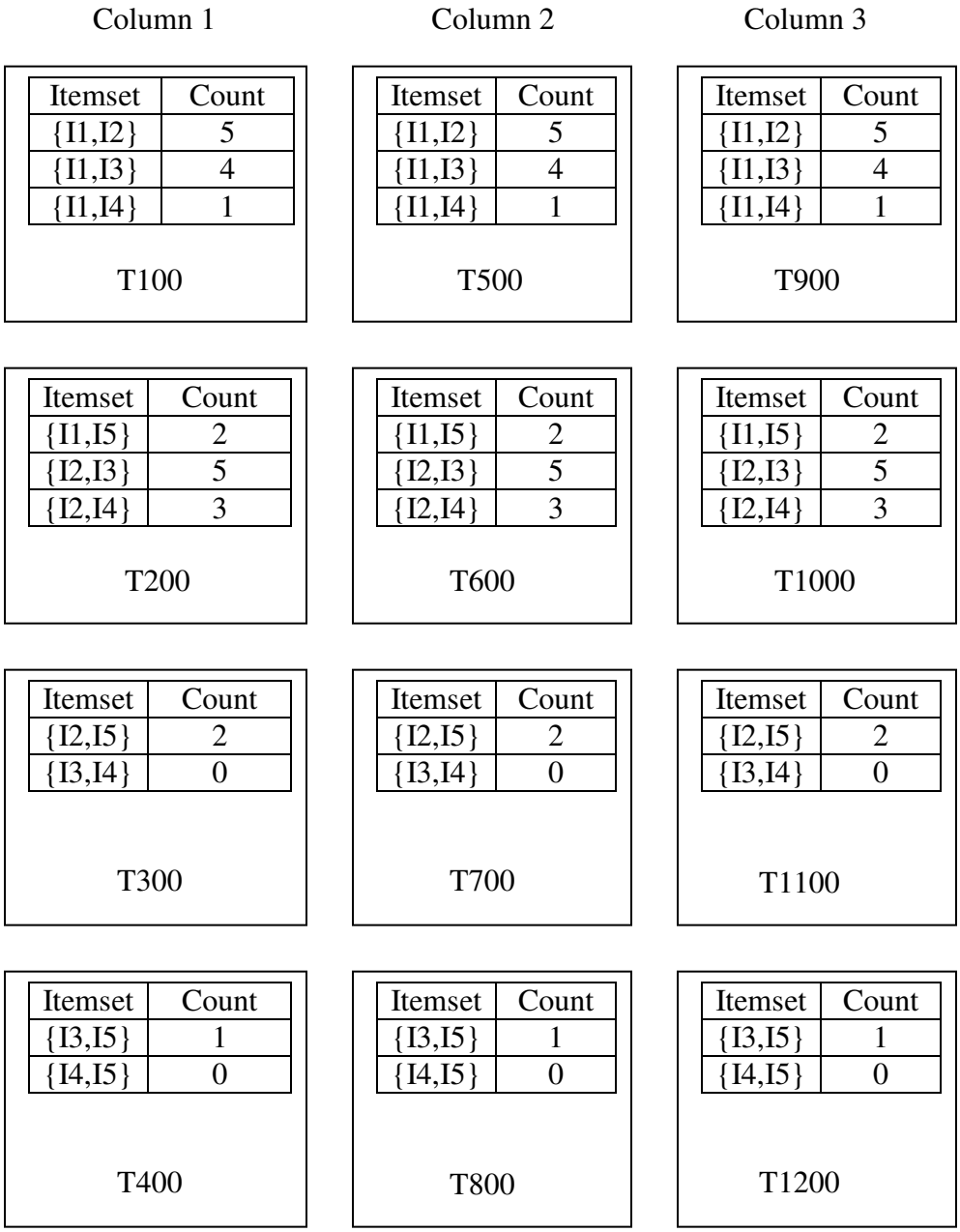


Figure 2.2.21 Use of CD to Broadcast Local Counts (HD)

2.2.7.5 Comparison of Algorithms

The Data Distribution (DD) algorithm scales poorly and has a high communication cost. DD exploits the aggregate memory of the multiprocessor better than Count Distribution (CD). In addition it makes fewer passes in the case of datasets with large transaction and frequent itemset lengths. There are also idle processors due to the communication scheme. If the communication buffer of any receiving processor is full and the outgoing communication buffers are full, then the send operation is blocked. During the last several passes of the algorithm, there is only a small number of items in the candidate set. However, each processor in the DD still sends the locally stored data to all other processors. There is not a corresponding decrease in communication with decrease in computation.

CD reduces the communication overhead of DD significantly since it only broadcasts the candidate itemsets. CD does not parallelize the computation of building the candidate hash tree and is a bottleneck with large number of processors. CD scales linearly with the number of transactions.

IDD solves the communication problem of DD by using a ring-based all-to-all broadcast network that eliminates contention. It also uses a bit map that eliminates the redundant work of DD. It uses bin-packing to achieve equal distribution of the candidate itemsets. As more processors are added however, it becomes more difficult to balance the work due to the fact that there is a decrease in the number of candidates. The hash tree is smaller for a smaller number of candidates and requires less computation work per transaction. The HD inherits all the good features of the IDD while reducing the amount of data movement. However, it uses a hash tree and requires the movement of data

among processors in the same column as a result there is a cost associated with hash tree construction and traversal as well as data movement (Han, Karypis, & Kumar, 2000).

2.2.8 Lattice Theory

A binary relation $\leq$ on a set L is a partial order if it is transitive, reflexive and antisymmetric. A lattice consists of the pair $(L, \leq)$ in which $\leq$ is a partial order on L, and every subset $\{a, b\}$ consisting of two elements has a least upper bound (LUB) and a greatest lower bound (GLB). A lattice is a mathematical structure with two binary operators, which are Join and Meet.

Join

An element c is an upper bound of elements a and b of L, if $a \leq c$ and $b \leq c$. The least upper bound or join of elements a and b is c if c is an upper bound of a and b and, for any y such that $a \leq y$ and also $b \leq y$, $c \leq y$. The join of a and b LUB ($\{a, b\}$) is denoted by $a \vee b$ (Ganter & Willie, 1999; Street, & Wallis, 1982; Zaki, 2000).

Meet

An element c is a lower bound of elements a and b of L, if $c \leq a$ and $c \leq b$. The greatest lower bound or meet of elements a and b is c if c is a lower bound of a and b and, for any y such that $y \leq a$ and also $y \leq b$, $y \leq c$. The meet of a and b GLB ($\{a, b\}$) is denoted by $a \wedge b$ (Ganter & Willie, 1999; Street & Wallis, 1982; Zaki, 2000).

Properties of Lattices

The meet, join, unique maximum and unique minimum element are always defined. Given a lattice $(L, \leq)$, a non-empty subset S of the set L is called a sub-lattice if $a \vee b \in S$ and $a \wedge b \in S$ whenever $a \in S$ and $b \in S$.

1. Idempotent Properties: $a \vee a = a$ and $a \wedge a = a$

2. Commutative Properties $a \vee b = b \vee a$ and $a \wedge b = b \wedge a$

3. Associative Properties $a \vee (b \vee c) = (a \vee b) \vee c$ and $a \wedge (b \wedge c) = (a \wedge b) \wedge c$

4. Absorption Properties: $a \wedge (b \vee c) = a$ and $a \vee (b \wedge c) = a$

5. A lattice L is said to be distributive if for all a, b, c $\in L$,

$a \wedge (b \vee c) = (a \wedge b) \vee (a \wedge c)$ and $a \vee (b \wedge c) = (a \vee b) \wedge (a \vee c)$ (Street & Wallis, 1982; Zaki, 2000).

Zaki (2000) used a lattice-theoretic approach to identify the frequent itemsets in the transaction database. In this approach the power set lattice on the set of database items are decomposed into sublattices that can be processed independently to find the frequent itemsets. The prefix-based approach to decompose the lattice with a bottom-up search strategy for the enumeration of the frequent itemsets will be presented in this section (Zaki, 2000; Zaki, 2000c).

Definition 1

Let X, Y and Z be elements of an ordered set A.

Let the sign < denote set inclusion. If $X < Y$ and there is no Z such that

$X < Z < Y$ then Y covers X, written as $X \sqsubset Y$ (Zaki, 2000).

Definition 2

Let X, Y be elements of an ordered set A. If $X \vee Y$ exists for all X, $Y \in A$ then A is referred to as a join semilattice. If $X \wedge Y$ exists for all X, $Y \in A$ then A is referred to as a meet semilattice. If A is both a join and a meet semilattice then it is referred to as a lattice i.e., if $X \vee Y$ and $X \wedge Y$ exist for all pairs of elements X, $Y \in A$.

If $\vee B$ and $\wedge B$ exist for all subsets $B \subseteq A$ then A is a complete lattice. If $X, Y \in A$, $X \vee Y \in R$ and $X \wedge Y \in R$ and $R \subset A$ then R is referred to as a sublattice of A (Zaki, 2000).

Lemma 2.2.1

All subsets of a frequent itemsets are frequent (Zaki, 2000).

Proof

In this representation the set of all frequent itemsets forms a meet semilattice since it is closed under the meet operation. If A and B are frequent itemsets then $A \cap B$ is also frequent. It is important to note that $A \cup B$ is not necessarily frequent.

Corollary

All supersets of an infrequent itemset are infrequent (Zaki, 2000).

If an itemset I does not satisfy the minimum support it is not frequent. If an item A is added to itemset I, the resulting itemset ($I \cup A$) cannot occur more frequently than I, therefore $I \cup A$ is not frequent either.

Lemma 2.2.2

All frequent itemsets are subsets of the maximal frequent itemsets. The search for frequent itemsets can be implemented using a procedure that quickly identifies the maximal frequent itemsets (Zaki, 2000).

Definition 3

Let $\perp$be the bottom element of the lattice L**.** If $\perp \sqsubset S$ and $S \in L$ then S is called an atom. The set of atoms of L are denoted by $A(L)$ (Zaki, 2000).

Definition 4

A lattice L is called a Boolean lattice if

1. It is Distributive

2. It has $\top$(top) and $\perp$(bottom) elements

3. Each member S of the lattice has a complement

Each database item S is an atom with a tid-list $L(S)$. $L(S)$ represents a list of all the transaction identifiers in which the atom was found (Zaki, 2000).

Lemma 2.2.3

For a finite Boolean lattice L, with $X \in L$,

$X = \vee \{Y \in A(L) \mid Y \leq X\}$

The join of a subset of the set of atoms can be used to generate the elements of a Boolean lattice. The join operation corresponds to union in the powerset $P(I)$ which is a Boolean lattice (Zaki, 2000).

Lemma 2.2.4

For any $X \in P(I)$, let $F = \{Y \in A(P(I)) \mid Y \leq X\}$

Then $X = \cup_{Y \in F} Y$, and $\sigma(X) = |\cap_{Y \in F} L(Y)|$

The join of some atoms of a lattice can be used to generate any itemset. The intersection of the tid-lists of the atoms can be used to compute the support of an itemset (Zaki, 2000).

Lemma 2.2.5

$F = \{Y \in A(P(I)) \mid Y \leq X\}$

For any $X \in P(I)$, let $X = \cup_{Y \in F} Y$. Then

$\sigma(X) = | \cap_{Y \in F} L(Y)|$

The intersection of the tid-lists of elements in F will give the support of an itemset that is the union of a set of items in F. The intersection of any two (k-1) length subsets can be used to generate the support of any k-itemsets (Zaki, 2000).

Lemma 2.2.6

Let R and S be two itemsets, with $R \subseteq S$. Then

$L(R) \supseteq L(S)$

Proof

This follows from the definition of support.

If R is a subset of S, then the cardinality of the tid-list of S must be less than or equal to the cardinality of the tid-list of R. The cardinality of the tid-list of the subset is greater than the cardinality of the superset. The counting and intersection operations are faster as you travel up the lattice due to the decrease in cardinality of the tid-lists (Zaki, 2000).

Definition 5

Let A be a set.

An equivalence relation on A is a binary relation $\equiv$ such that for all X, Y, $Z \in A$, the relation is:

1. Reflexive: $X \equiv X$
2. Symmetric: $X \equiv Y$ implies $Y \equiv X$.
3. Transitive: $X \equiv Y$ and $Y \equiv Z$, implies $X \equiv Z$.

The equivalence relation partitions the set A into disjoint subsets called equivalence classes. The equivalence class $X \in A$ is given as $[X] = \{Y \in A \mid X \equiv Y\}$

Define a function

P: $P(I) \times N \mapsto P(I)$,

Where $p(X, k) = X[1{:}k]$, the k length prefix of X. Define an equivalence relation θ_k on the lattice $P(I)$ as follows:

$\forall X, Y \in P(I), X \equiv \theta_k Y \Leftrightarrow p(X, k) = p(Y, k)$.

Two itemsets are equivalent if they share a common k-length prefix. We refer to θ_K as a prefix-based equivalence relation and the set of equivalent itemsets as a class (Zaki, 2000).

Lemma 2.2.7

Each equivalence class $[T]\ \theta_k$ induced by the equivalence relation θ_k is a sublattice of $P(I)$.

Proof

Let R and S be any two elements in the class $[T]$. This implies that they both share a common prefix T. $R \vee S = R \cup S \supseteq T$ implies that $R \vee S \in [T]$, and

$R \wedge S = R \cap S \supseteq T$ implies that $R \wedge S \in [T]$. Therefore $[T]\ \theta_k$ is a sublattice of $P(I)$ (Zaki, 2000).

The database of transactions shown in Table 2.2.3 is reproduced in Table 2.2.8 and its vertical organization in Table 2.2.9. The lattice of itemsets is shown in Figure 2.2.22.

Table 2.2.8 Transaction Database

TID	**List of Items**
T100	I1, I2, I5
T200	I2, I4
T300	I2, I3
T400	I1, I2, I4
T500	I1, I3
T600	I2, I3
T700	I1, I3
T800	I1, I2, I3, I5
T900	I1, I2, I3

Table 2.2.9 Vertical View of Transaction Database

I1	**I2**	**I3**	**I4**	**I5**
T100	T100	T300	T200	T100
T400	T200	T500	T400	T800
T500	T300	T600		
T700	T400	T700		
T800	T600	T800		
T900	T800	T900		
	T900			

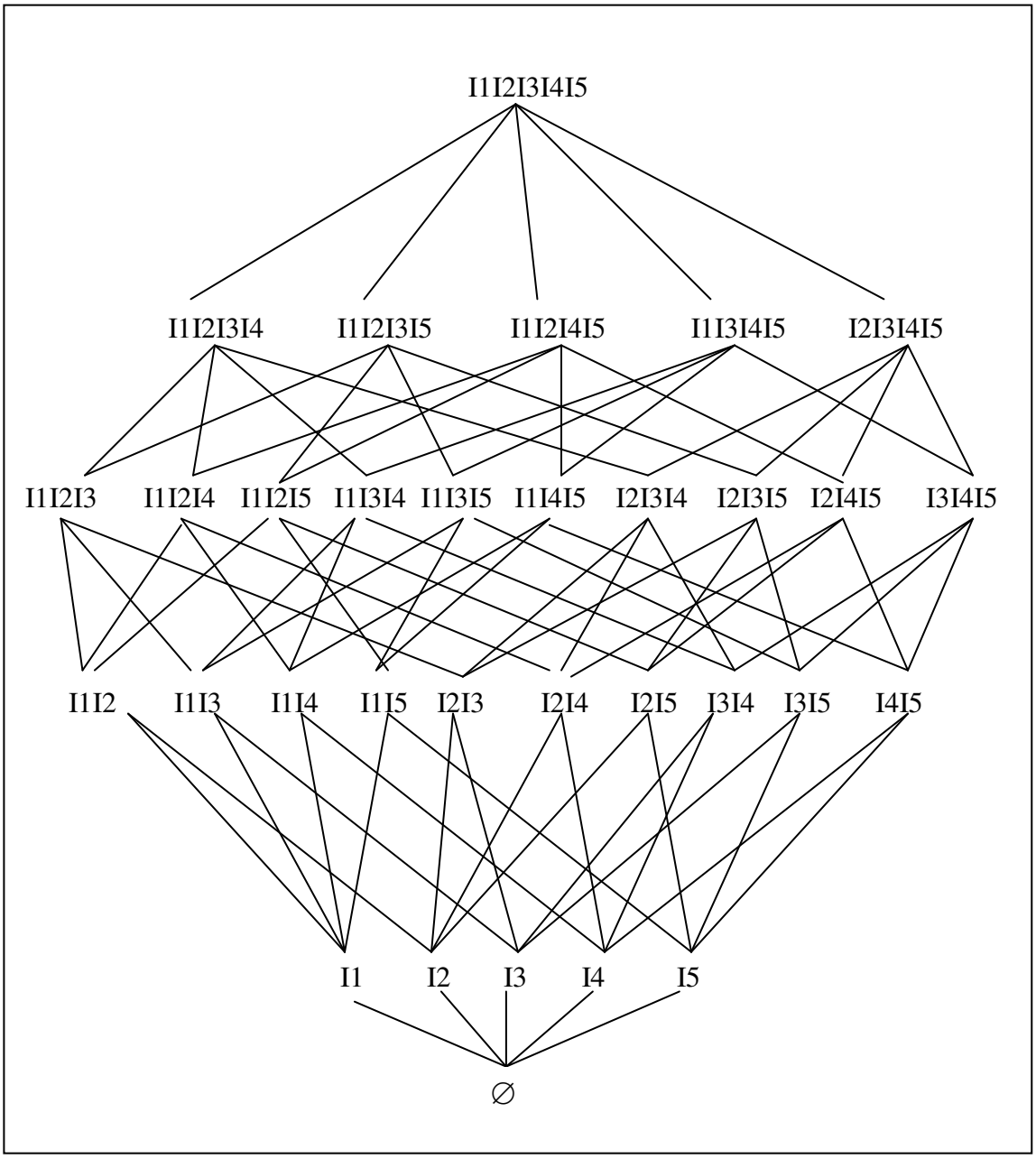


Figure 2.2.22 Lattice of Itemsets

2.2.8.1 Serial Prefix-Based Method with Bottom-Up Search Algorithm

The following example is an illustration of how the prefix-based with bottom-up search algorithm works using the database in Table 2.2.8. The pseudo code for the bottom-up search algorithm is shown in Figure 2.2.23. The decomposition of the lattice using a prefix-based approach for classes generated by θ_1 is shown below (Zaki, 2000).

Lattice Decomposition: Prefix-Based Classes

The algorithm is based on the assumption that L_k is lexicographically partitioned into equivalence classes based on their common k-1 prefix (Zaki, 2000). The equivalence class $x \in L_{k-2}$ is given as :

$S_x = [x] = \{b \in L_{k-1} \quad x[1: k-2] = b[1: k-2]\}$

Using this function we partition the itemsets into classes using θ_1 as shown below:

[I1] = {{I1, I2}, {I1, I3}, {I1, I4}, {I1, I5}}

[I2] = {{I2, I3}, {I2, I4}, {I2, I5}}

[I3] = {{I3, I4}, {I3, I5}}

[I4] = {{I4, I5}}

Each class will be processed independently to identify the frequent itemsets.

These classes will be processed using the Bottom-Up search method to discover the frequent itemsets

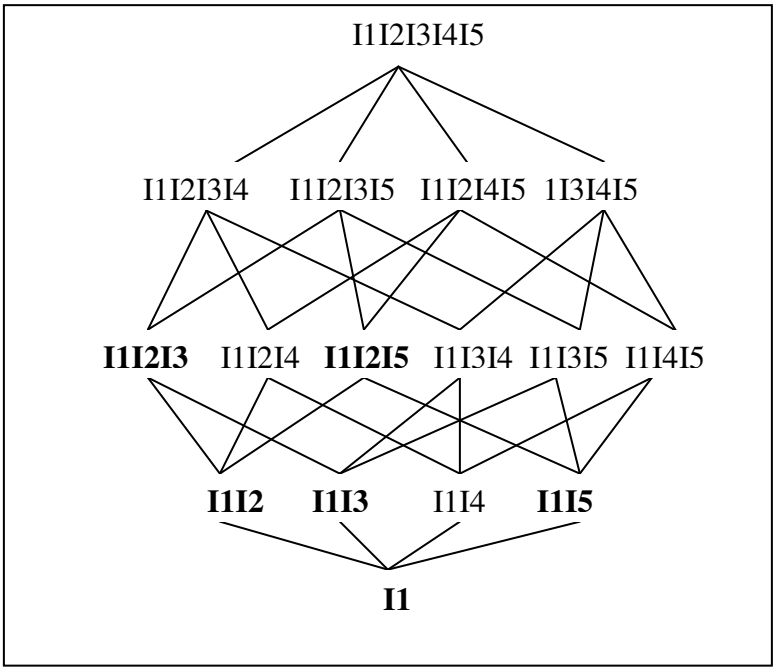
In the pseudo code in Figure 2.2.23 $\mathcal{L}$(R) represents the tid-list of item R.

Frequent 2-itemsets = {{I1, I2}, {I1, I3}, {I1, I5}, {I2, I3}, {I2, I4}, {I2, I5}}

The lattice generated by class I1 is shown in Figure 2.2.24. The intersection of the tid-lists for class I1 is shown in Figure 2.2.25. In Figure 2.2.24 the frequent itemsets are shown in bold. It can be seen that {I1, I2}, {I1, I3}, {I1, I5}, {I1, I2, I3} and {I1, I2, I5} are shown in bold since these are frequent itemsets.

```
Bottom-Up(S) // Set of atoms
for all atoms A_i ∈ S do
T_i = ∅ // Frequent itemsets
for all atoms A_j ∈ S with j > i do
R = A_i ∪ A_j
ℒ(R) = ℒ(A_i) ∩ ℒ(A_j) //tid-list of item R
if σ(R) ≥ min_sup then
  T_i = T_i ∪ {R}; F_|R| = F_|R| ∪ {R}; // Frequent k-itemsets
end
end
Delete S; //reclaim memory
for all T_i ≠ ∅ do Bottom-Up(T_i)
```

Figure 2.2.23 Pseudo Code for Bottom-Up Search (Zaki, 2000)



Frequent itemsets are shown in bold

Figure 2.2.24 Lattice Generated by Class I1

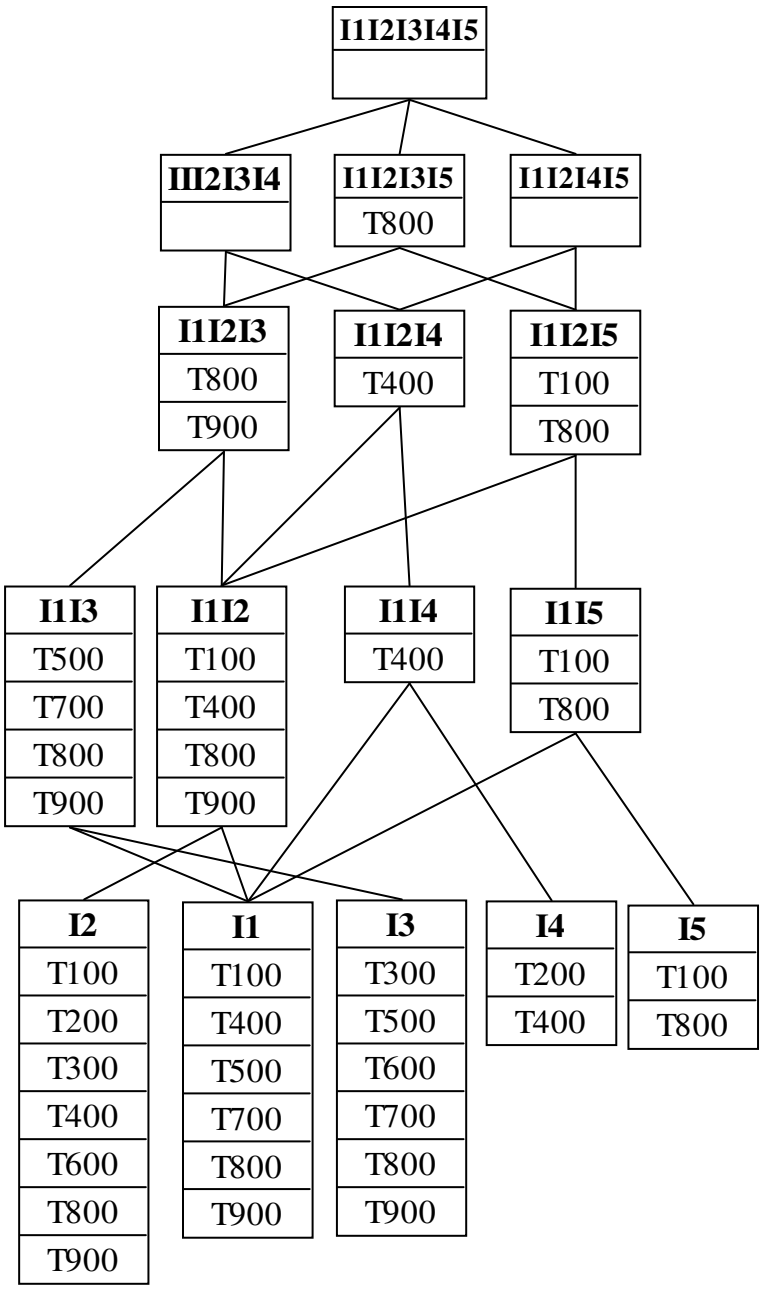


Figure 2.2.25 Intersection of Itemsets in Class I1

Figure 2.2.25 shows the intersection of the tid-lists for class I1. The tid-list for each item consists of all the transactions in which the item was found. When we intersect two tid-lists the resulting tid-list contains the transactions that are common to the two intersecting tid-lists. In Figure 2.2.25 the intersection of I3 and I1 gives a new tid-list for {I1, I3} consisting of transactions T500, T700, T800 and T900. The intersection of the tid-lists for I1 and I4 gives a new tid-list with only transaction T400 since it is the only transaction common to both tid-lists.

The bottom-up search algorithm shown in Figure 2.2.23 will be used to process class I1 as shown below. The algorithm will be called recursively until all the frequent itemsets are generated. It is first called with the following values:

R is the item

S is the set of atoms of the class I1

Support is the minimum support for the frequent itemsets

T stores the frequent itemsets

$F_{|k|}$ is the frequent k-itemsets

$\mathcal{L}(R)$ stores the tid-list of item R

A_i is atom i

σ is the support count

S = {{I1, I2}, {I1, I3}, {I1, I4}, {I1, I5}}

support = 2

The bottom-up search algorithm is called with S

Bottom-Up(S)

$i = 1, A_1$ = {I1, I2}

$T_1 = \varnothing$

$j = 2, A_2 = \{I1, I3\}$

$R = A_1 \cup A_2$

$= \{I1, I2\} \cup \{I1, I3\}$

$= \{I1, I2, I3\}$

We form the union of the first two atoms of the class to get {I1, I2, I3}. We them determine the count of {I1, I2, I3} by intersecting the tid-lists of the first two atoms of the class as shown in the following operations.

$\mathcal{L}(\{I1, I2, I3\}) = \mathcal{L}(\{I1, I2\}) \cap \mathcal{L}(\{I1, I3\})$

$= \{T100, T400, T800, T900\} \cap \{T500, T700, T800, T900\}$

$= \{T800, T900\}$

There are two transactions that contain these three items ({I1, I2, I3}) together and the support count is 2. This is represented as:

$\sigma(\{I1, I2, I3\}) = 2$

The support for {I1, I2, I3} is 2 which makes it frequent. We add it to the set of frequent itemsets

$T_1 = \varnothing \cup \{I1, I2, I3\}$

$= \{I1, I2, I3\}$

We also add I1I2I3 to the set of frequent 3-itemsets

$F_{|3|} = F_{|3|} \cup \{R\}$

$= \varnothing \cup \{I1, I2, I3\}$

$= \{I1, I2, I3\}$

We repeat the process for the next atom

$j = j + 1 = 2 + 1 = 3;$

$A_3 = \{I1, I4\}$

$R = A_1 \cup A_3$

$= \{I1, I2\} \cup \{I1, I4\}$

$= \{I1, I2, I4\}$

$\mathcal{L}(\{I1, I2, I4\}) = \mathcal{L}(\{I1, I2\}) \cap \mathcal{L}(\{I1, I2\})$

$= \{T100, T400, T800, T900\} \cap \{T200, T400\}$

$= \{T400\}$

There is one transaction that contains these three items ({I1, I2, I4}) together and the support count is 1. This is represented as:

$\sigma(\{I1, I2, I4\}) = 1$

We drop {I1, I2, I4} since it is not frequent

We go on to the next atom of class I1

$j = j + 1 = 3 + 1 = 4;$

$A_4 = \{I1, I5\}$

$R = A_1 \cup A_4$

$= \{I1, I2\} \cup \{I1, I5\}$

$= \{I1, I2, I5\}$

$\mathcal{L}(\{I1, I2, I5\}) = \mathcal{L}(\{I1, I2\}) \cap \mathcal{L}(\{I1, I5\})$

$= \{T100, T400, T800, T900\} \cap \{T100, T800\}$

$= \{T100, T800\}$

There are two transactions that contain these three items ({I1, I2, I5}) together and the support count is 2. This is represented as:

$\sigma(\{I1, I2, I5\}) = 2$

The support for {I1, I2, I5} is 2 which makes it frequent. We add it to the set of frequent itemsets

$T_1 = T_1 \cup \{I1, I2, I5\}$

$= \{\{I1, I2, I3\}, \{I1, I2, I5\}\}$

We also add I1I2I5 to the set of frequent 3-itemsets

$F_{|3|} = F_{|3|} \cup \{R\}$

$= \{I1, I2, I3\} \cup \{I1, I2, I5\}$

$= \{\{I1, I2, I3\}, \{I1, I2, I5\}\}$

We proceed to process the next atom

$i = 2, A_2 = \{I1, I3\}$

$T_2 = \varnothing$

$j = 3, A_3 = \{I1, I4\}$

$R = A_2 \cup A_3$

$= \{I1, I3\} \cup \{I1, I4\}$

$= \{I1, I3, I4\}$

$\mathcal{L}(\{I1, I3, I4\}) = \mathcal{L}(\{I1, I3\}) \cap \mathcal{L}(\{I1, I4\})$

$= \{T500, T700, T800, T900\} \cap \{T400\} = \varnothing$

The support count for {I1, I3, I4} is 0 . This is represented as:

$\sigma(\{I1, I3, I4\}) = 0$

{I1, I3, I4} is not frequent so it is not added to the set of frequent itemsets.

$T_2 = \varnothing$

$j = j + 1 = 3 + 1 = 4; A_4 = \{I1, I5\}$

$R = A_2 \cup A_4$

$= \{I1, I3\} \cup \{I1, I5\} = \{I1, I3, I5\}$

$\mathcal{L}(\{I1, I3, I5\}) = \mathcal{L}(\{I1, I3\}) \cap \mathcal{L}(\{I1, I5\})$

$= \{T500, T700, T800, T900\} \cap \{T800\}$

$= \{T800\}$

There is one transaction that contains these three items ({I1, I3, I5}) together and the support count is 1. This is represented as:

$\sigma(\{I1, I3, I5\}) = 1$

{I1, I3, I5} is not frequent so it is not added to the set of frequent itemsets.

$i = 3, A_3 = \{I1, I4\}$

$j = 4, A_4 = \{I1, I5\}$

$R = A_3 \cup A_4$

$= \{I1, I4\} \cup \{I1, I5\} = \{I1, I4, I5\}$

$\mathcal{L}(\{I1, I4, I5\}) = \mathcal{L}(\{I1, I4\}) \cap \mathcal{L}(\{I1, I5\})$

$= \{T400\} \cap \{T100, T800\} = \varnothing$

The support count for {I1, I4, I5} is 0 . This is represented as:

$\sigma(\{I1, I4, I5\}) = 0$

$T_3 = \varnothing$

We call the bottom-up algorithm again with T_1

$T_1 = \{\{I1, I2, I3\}, \{I1, I2, I5\}\}$

Bottom-up (T_1)

$i = 1, A_1 = \{I1, I2, I3\}$

$T_1 = \varnothing$

$j = 2, A_2 = \{I1, I2, I5\}$

$R = A_1 \cup A_2$

$= \{I1, I2, I3\} \cup \{I1, I2, I5\} = \{I1, I2, I3, I5\}$

$\mathcal{L}(\{I1, I2, I3, I5\}) = \mathcal{L}(\{I1, I2, I3\}) \cap \mathcal{L}(\{I1, I2, I5\})$

$= \{T800, T900\} \cap \{T100, T800\}$

$= \{T800\}$

There is one transaction that contains these four items ({I1, I2, I3, I5}) together and the support count is 1. This is represented as:

$\sigma(\{I1, I2, I3, I5\}) = 1$

{I1, I2, I3, I5} is not frequent so it is not added to the set of frequent itemsets. Since there are no more elements in the set the algorithm terminates. The output is the set of frequent 3-itemsets.

$F_{|3|} = \{\{I1, I2, I3\}, \{I1, I2, I5\}\}$

The frequent 3-itemsets generated by Class I1 are {{I1, I2, I3}, {I1, I2, I5}}.

The lattice generated by class I2 is shown in Figure 2.2.26. The intersection of the tid-lists is shown in Figure 2.2.27.

The computation of the itemsets generated by class I2 is as follows.

$S = \{\{I2, I3\}, \{I2, I4\}, \{I2, I5\}\}$

support = 2

The bottom-up search algorithm is called with S

Bottom-Up(S)

$i = 1, A_1 = \{I2, I3\}$

$T_1 = \varnothing$

$j = 2, A_2 = \{I2, I4\}$

$R = A_1 \cup A_2$

$= \{I2, I3\} \cup \{I2, I4\} = \{I2, I3, I4\}$

$\mathcal{L}(\{I2, I3, I4\}) = \mathcal{L}(\{I2, I3\}) \cap \mathcal{L}(\{I2, I4\})$

$= \{T300, T800, T900\} \cap \{T200, T400\} = \varnothing$

The support count for {I2, I3, I4} is 0 . This is represented as:

$\sigma(\{I2, I3, I4\}) = 0$

{I2, I3, I4} is not frequent so it is not added to the set of frequent itemsets. We proceed to the next atom.

$j = j + 1 = 2 + 1 = 3$

$A_3 = \{I2, I5\}$

$R = A_1 \cup A_3$

$= \{I2, I3\} \cup \{I2, I5\}$

$= \{I2, I3, I5\}$

$\mathcal{L}(\{I2, I3, I5\}) = \mathcal{L}(\{I2, I3\}) \cap \mathcal{L}(\{I2, I5\})$

$= \{T300, T800, T900\} \cap \{T100, T800\}$

$= \{T800\}$

There is one transaction that contains these three items ({I2, I3, I5}) together and the support count is 1. This is represented as:

$\sigma(\{I2, I3, I5\}) = 1$

{I2, I3, I5} is not frequent so it is not added to the set of frequent itemsets. We proceed to the next atom.

$i = 2, A_2 = \{I2, I4\}$

$T_2 = \varnothing$

$j = 3, A_3 = \{I2, I5\}$

$R = A_2 \cup A_3 = \{I2, I4\} \cup \{I2, I5\} = \{I2, I4, I5\}$

$\mathcal{L}(\{I2, I4, I5\}) = \mathcal{L}(\{I2, I4\}) \cap \mathcal{L}(\{I2, I5\})$

$= \{T200, T400\} \cap \{T100, T800\} = \varnothing$

The support count for {I2, I4, I5} is 0 . This is represented as:

$\sigma(\{I2, I4, I5\}) = 0$

$T_2 = \varnothing$

There are no frequent itemsets generated by class I2

Since there are no more elements in the set the algorithm terminates. The output is the empty set.

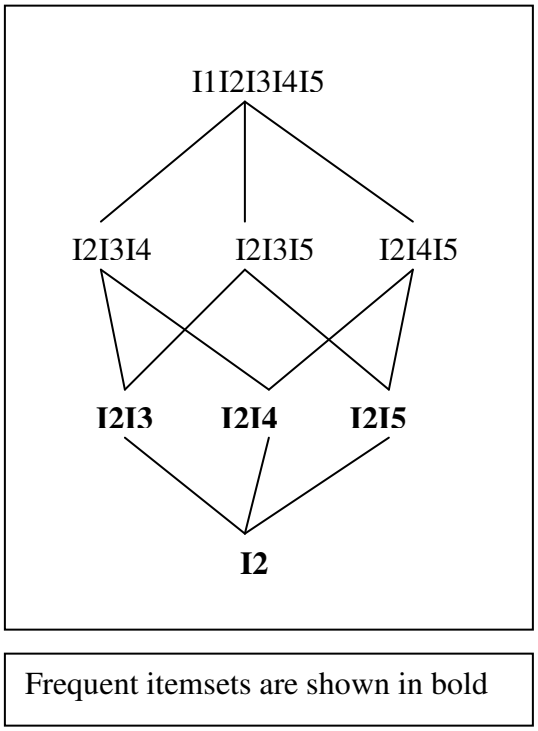


Figure 2.2.26 Lattice Generated by Class I2

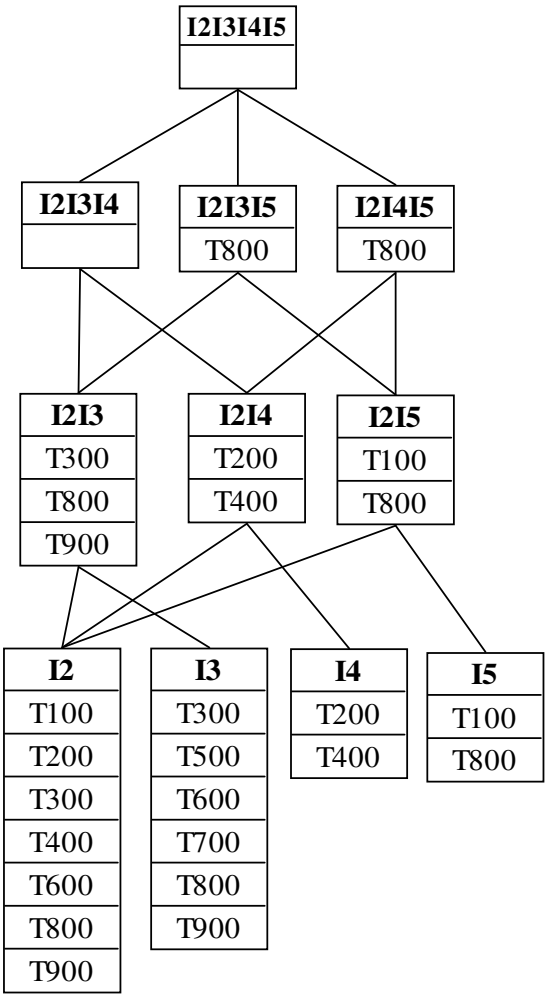


Figure 2.2.27 Intersection of Itemsets in Class I2

The lattice for class I3 is shown in Figure 2.2.28. The intersection of the tid-lists for class I3 is shown in Figure 2.2.29. The computation of the frequent itemsets using a bottom-up approach is as follows:

$S = \{\{I3, I4\}, \{I3, I5\}\}$

Support = 2

The bottom-up search algorithm is called with S

Bottom-Up(S)

$i = 1, A_1 = \{I3, I4\}$

$T_1 = \varnothing$

$j = 2, A_2 = \{I3, I5\}$

$R = A_1 \cup A_2$

$= \{I3, I4\} \cup \{I3, I5\} = \{I3, I4, I5\}$

$\mathcal{L}(\{I3, I4, I5\}) = \mathcal{L}(\{I3, I4\}) \cap \mathcal{L}(\{I3, I5\})$

$= \{\varnothing\} \cap \{T800\} = \varnothing$

The support count for {I3, I4, I5} is 0 . This is represented as:

$\sigma(\{I3, I4, I5\}) = 0$

There are no frequent itemsets in class I3

The lattice generated by class I4 is shown in Figure 2.2.30. There is no intersection diagram since there is only one atom. In general a class with only one atom can be eliminated since it cannot generate candidates.

$\sigma(I4I5) = 0$

There are no frequent itemsets in class I4.

It should be noted that in the serial approach all classes are processed independently by a single processor.

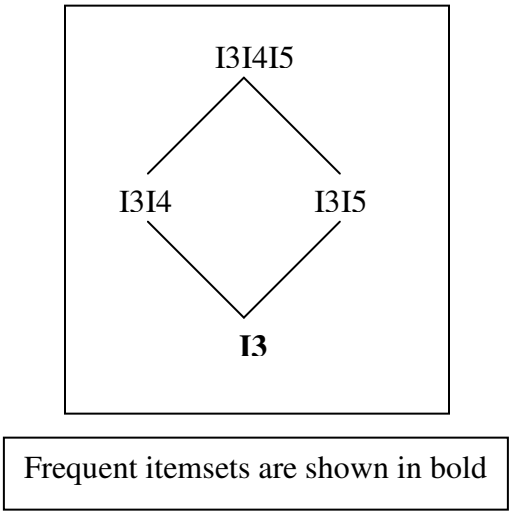


Figure 2.2.28 Lattice Generated by Class I3

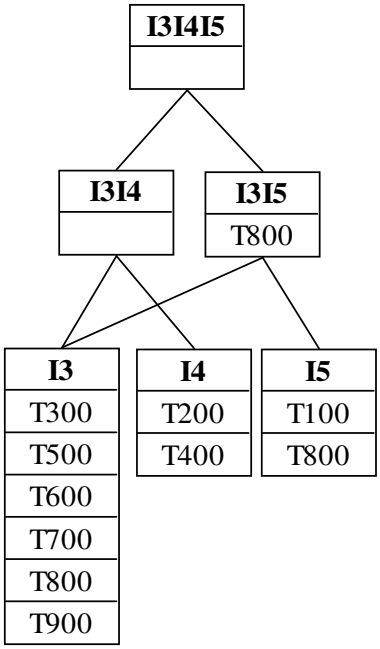


Figure 2.2.29 Intersection of Itemsets in Class I3

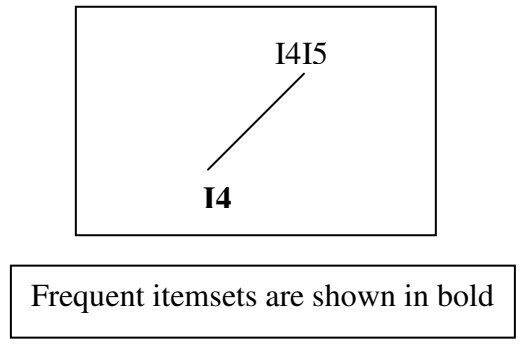


Figure 2.2.30 Lattice Generated by Class I4

2.2.8.2 Parallel Prefix-Based Method with Bottom-Up Search Algorithm.

In the parallel implementation we assume that there are two processors. The following is an illustration of the parallel prefix-based with bottom-up search algorithm. In the example the diagrams for the intersection of the itemsets are omitted as they are identical to those used in the serial algorithm in Figure 2.2.24 to Figure 2.2.30. The pseudo code for the algorithm is shown in Figure 2.2.31.

The sorted tid-lists are shown in Table 2.2.10. Table 2.2.11 shows the assignment of the tid-lists to the processors. The goal is to assign an equal number of items to each processor. The length of each tid-list is shown in brackets in Table 2.2.11.

Table 2.2.10 Tid-Lists Sorted on Number of Transactions

Tid List	**No of Transactions**
I2	7
I1	6
I3	6
I4	2
I5	2

Table 2.2.11 Assignment of Tid-Lists to Processors

Processor (P_0)	**Processor (P_1)**
I2(7)	I1(6)
I4(2)	I3(6)
I5(2)	

```
Begin
   /* Initialize Phase */
   F2 = (Set of Frequent 2-itemsets}
   Generate Independent Classes from F2 using
     Prefix-based Partitioning
   Schedule Classes among the processors P
   Scan local database partition
   Transmit relevant tid-lists to other processors
    Receive tid-lists from other processors

  /* Asynchronous Phase */
   for each assigned Class, C2
      Compute Frequent Itemsets: Bottom-Up(C2)

   /* Final Reduction Phase */
   Aggregate Results and Output Association
End
```

Figure 2.2.31 Pseudo Code for Parallel Prefix-Based Algorithm (Zaki, 2000c).

Class Schedule

The classes are assigned to the processors based on the size of each class. The goal is to assign the classes equally among the processors using the class size. The size of a class is computed using $\binom{s}{2}$, where s is the number of atoms in the class.

Table 2.2.12 shows the classes sorted on size. The allocation of the classes to processors is shown in Table 2.2.13. Table 2.2.14 shows the assignment of tid-lists to processors after receival of additional tid-lists needed to compute the frequent itemsets. Figure 2.2.32 shows the allocation of tid-lists and frequent 2-itemsets to the processors. Figure 2.2.33 shows the assignment of classes to processors. Figure 2.2.34 shows the assignment of tid-lists to processors after exchange of additional tid-lists.

Table 2.2.12 Classes Sorted on Size

Class	**Size**
I1	6
I2	3
I3	2
I4	1

Table 2.2.13 Assignment of Classes to Processors

Processor (P_0)	**Processor (P_1)**
C^1 (6)	C_2 (3)
	C_3 (2)
	C_4 (1)

Table 2.2.14 Assignment of Tid-Lists to Processors After Exchange of Tid-Lists

Processor (P_0)	**Processor (P_1)**
I2(7)	I1(6)
I4(2)	I3(6)
I5(2)	I2
I3	I4
	I5

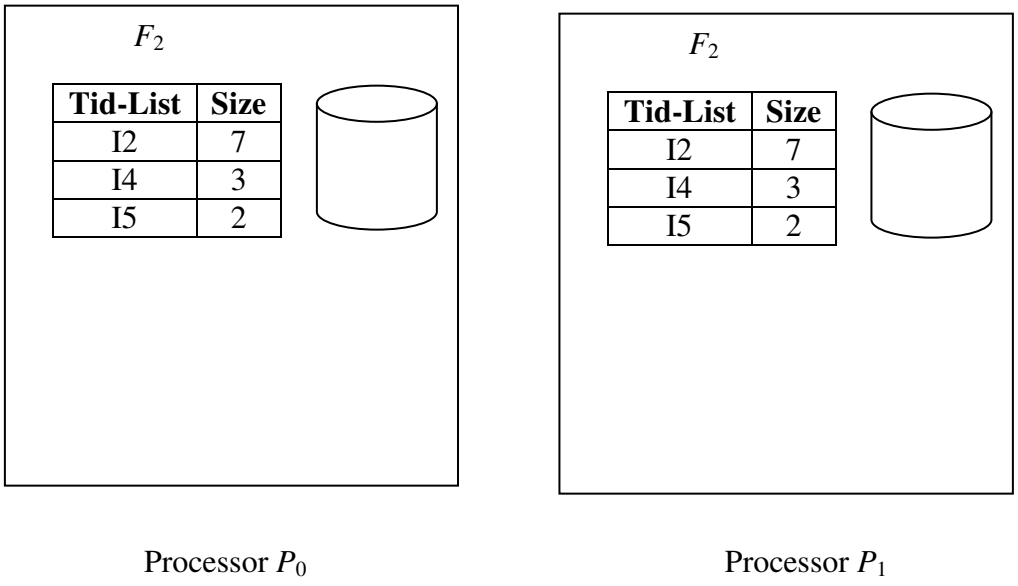


Figure 2.2.32 Assignment of Tid-Lists to Processors

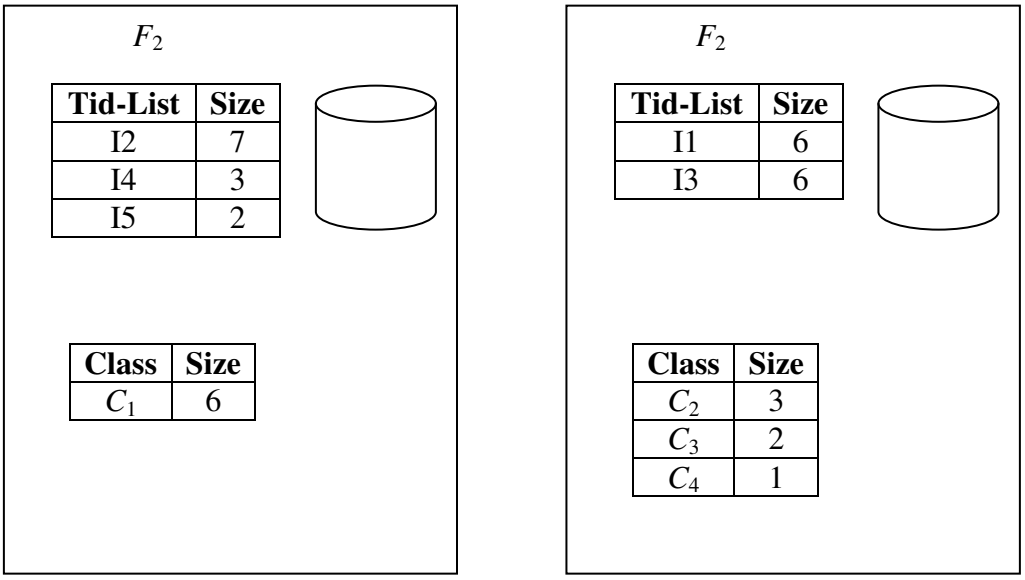


Figure 2.2.33 Assignment of Classes to Processors

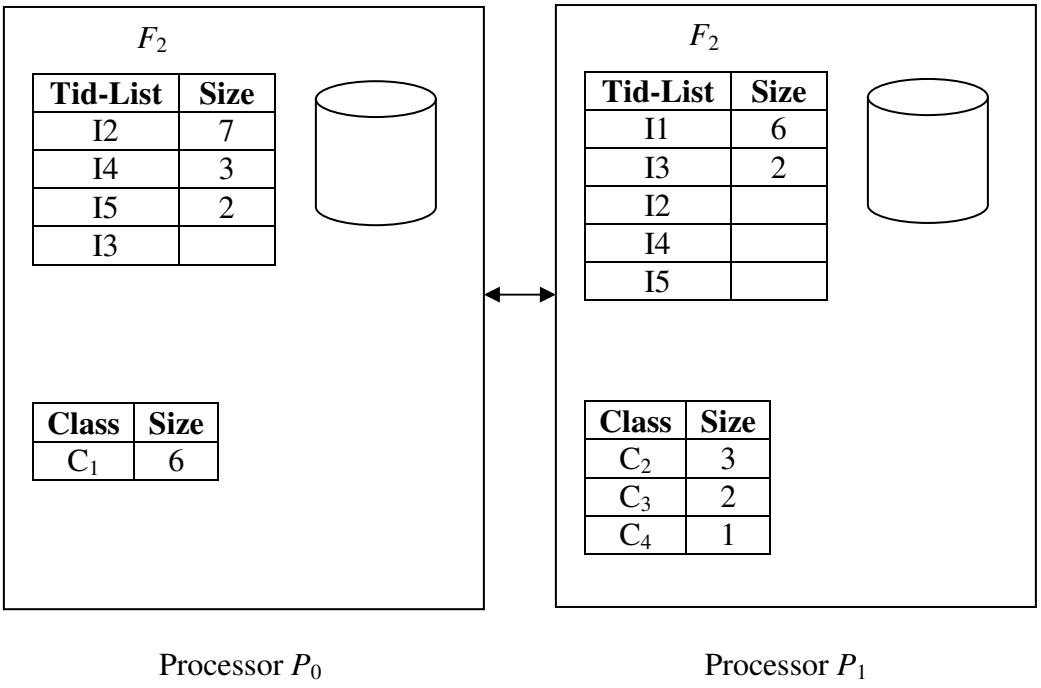


Figure 2.2.34 Assignment of Tid-Lists to Processors After Exchange of Tid-Lists

In Figure 2.2.34 processor P_0 will use class C_1 to compute the frequent itemsets. It uses the bottom-up search to identify the frequent itemsets. Classes C_2, C_3 and C_4 will be processed by processor P_1. They will be used to compute the frequent itemsets associated with each. As indicated above class C_4 has a single atom and will be eliminated since it cannot generate additional candidates. The actual processing is not shown as it is identical to the results shown for the serial approach except that the computations will now be carried out in parallel.

2.3 Dynamic Distributed Rule Mining (DDRM)

The DDRM system is designed to run in a network of PCs environment. This implies that there may be differences in the capabilities of these PCs such as the clock speed of the CPU, capacity and transfer rate of disk drives, and the size of main memory. The efficiency of the algorithm can be improved on if for example the larger classes are assigned to the faster PCs on the network. In a heterogeneous environment the high performance nodes will be identified and the larger classes assigned to them based on their ability to process these classes faster than the other nodes.

According to Tamara and Kitsuregawa (1999) in a PC network it is possible that the PCs found in the environment may vary in their capabilities. It is possible that when the network was implemented initially all the machines may have shared the same specifications. However, over time the homogeneous environment may change into a heterogeneous one with a wide range of PCs with different processing capabilities. The performance of parallel algorithms that were implemented in a PC cluster environment can be improved upon if the larger tasks are assigned to the high performance PCs in the

cluster. This will improve on both the computation and communication time in the overall solution to the problem.

In DDRM the high performance PCs will be identified. These PCs will be assigned the larger classes. In addition the controller will be located on one of the high performance PCs. The system will also include a class migration approach that is activated after all the classes have been assigned. At this point a class may be subdivided and redistributed among the idle processors. This approach is similar to the candidate migration and transaction migration strategies developed for a heterogeneous PC cluster environment by Tamara and Kitsuregawa (1999).

DDRM uses a dynamic load balancing approach. It partitions the lattice into several sublattices. These sublattices are then collected by each processor for processing. If there is more lattice to be processed it will be selected by the next available processor. This is in contrast to the previous approaches where the data is partitioned and assigned statically to all processors participating in the processing. It uses a distributed memory system and is similar to Apriori based algorithm in this respect such as CD and IDD. It will result in better utilization of the available processors. DDRM sends only the class atoms to processors for processing. The processors then use the atoms to generate the frequent itemsets associated with the lattice. This reduces the communications overhead significantly when compared to other algorithms such as CD.

2.4 The Contribution This Study Makes to Data Mining

Itemsets form a large lattice with the set of all items at the top and the empty set at the bottom (Brin, Motwani, Ullman, & Tsur, 1997). The main limitation of previous parallel algorithms such as Count Distribution (CD), Data Distribution (DD), Intelligent Data

Distribution (IDD) and Hybrid Distribution (HD) is that they make repeated passes over the database partitions. In addition there is an exchange of counts of candidates or data partitions assigned to processors. The communications overhead is also high due to the need to exchange information on a regular basis. It is also necessary to synchronize the operations taking place on the processors. These activities are completely or significantly reduced in the case of communications overhead in the DDRM algorithm. In this new algorithm the sublattices are assigned to processors dynamically by the controller and frequent itemsets returned to the controller.

Zaki, et al. (1997) used a static scheduling approach to assign the partitions to the processors. Static scheduling was also used in CD, HD, IDD, and DD for the assignment of itemsets to the processors for processing. DDRM uses a dynamic scheduling approach to assign work to the processors participating in the computations for the generation of the frequent itemsets. The communications overhead associated with IDD is high. This has been significantly reduced by eliminating the need to exchange data among the processors on a regular basis.

The system uses a controller process to distribute the classes among the processors. The controller is also responsible for the receival and processing of all results generated by the processors. The algorithm is implemented in six steps as follows:

1. The database is partitioned among *n* processors for the computation of the tid-lists. The database is partitioned among the processors using the number of items in each transaction to determine the next processor to be assigned a transaction. The next transaction is always assigned the processor with the lowest count of items for all transactions assigned to it so far.

2. The processors use the assigned transactions to generate the local tid-list, which is then transmitted to the controller.
3. The controller uses all the tid-lists received from all the processors to create a single tid-list for the database.
4. The tid-list is partitioned among the *n* processors using the length of the tid-list for each item to determine the processor to be assigned the next tid-list. A count of the total length of all tid-lists assigned to each processor is used to determine which processor will be assigned the next tid-list. The next-tid-list is always assigned to the processor with the lowest count.
5. The itemsets are partitioned into classes that the first *n* classes allocated to the n processors. The controller uses the partition algorithm described in Figure 3.2.4 below to partition the itemsets into classes and assigns the first n classes to the n processors. The remaining classes are assigned dynamically to the next available processors. The controller sends only the atoms for the lattice associated with each class to the processor. The processors use all the classes to compute the frequent itemsets. The processors use the intersection of the tid-lists for the itemsets to determine the frequent itemsets. Each processor sends the frequent itemsets for the class to the controller. The dynamic exchange of classes and itemsets between the controller and the processors takes place. The controller receives the frequent itemsets from each processor and sends a class to the next available processor for processing.
6. The controller generates the rules from the frequent itemsets. The controller computes the set of rules using the frequent itemsets generated above.

The improvements made by the proposed algorithms are as follows:

1. Reduction in communications among processors: Dynamic Distributed Rule Mining (DDRM) will significantly reduce the communications bottleneck among the processors. There is no need for processors to exchange data since each sublattice can be processed independently. The exchange is between the controller and each processor and involves the exchange of atoms of the sublattice and the frequent itemsets found in each class.
2. Improved load balancing: The classes generated by the DDRM algorithm are all stored at the controller and assigned to each processor as soon as each processor becomes idle. This is a significant improvement over the static assignment of the classes. In a static approach if it is discovered early that some classes assigned to a given processor have no frequent itemsets then these classes will not be processed any further. This may result in the processor becoming idle while the other processors may have excess work that could be assigned to this idle processor. However, due to the static assignment of classes it will not be possible to take some of the classes from the busy processors for assignment to the idle processor. The use of dynamic load balancing will significantly improve on the efficiency of the computations and use of the processors.
3. No synchronization: DDRM uses a lattice theoretic approach which partitions the itemsets into sublattices that can be assigned to each processor and processed independently. Processors only communicate with the controller to collect classes for processing and to return any frequent itemsets found in the assigned class.

2.5 Summary

This chapter discussed data mining with emphasis on the mining of association rules. We presented several approaches to the mining of association rules that are based on the use of parallel architectures. A discussion on lattice theory and its application to the mining of association rules was also presented. The chapter also discussed the proposed DDRM algorithm.

Chapter 3

Methodology

3.1 Lattice Theoretic Approach

According to Brin, et al. (1997) itemsets form a large lattice with the set of all items at the top and the empty itemset at the bottom. Zaki (2000) proposed a lattice theoretic approach to decompose the original search space into smaller pieces, which can be processed independently. The most efficient known way to parallelize finding large itemsets involved dividing the database among the processors and to have each processor count all the itemsets for its own local data. In this approach the issues related to load balancing and synchronization are critical (Brin, et al., 1997).

3.1.1 Lattice Theory

Let A be the set of distinct attributes I1, I2, …, I5. We can represent any subset of A as a sequence that is sorted according to lexicographic order of attribute names. A subset of the sequence {I1, I2, I3} is {I1, I2} and is identified as {I1, I2}. It is also the same as {I2, I1}. A one-to-one mapping exists between the set of all sequences and the power set (2^A). The set of all sequences can be identified with 2^A. The power set (2^A) is a Boolean lattice where $\varnothing$ and A are the bottom and top respectively. We denote the order in 2^A with $\leq$ which coincides with set inclusion, $b \leq c$ reads b is a subset of c. A [I] is the i-rank attribute in A. Ranks are counted starting from 1. The cardinal of a subset s is denoted by

$|s|$. A subset with cardinal k is referred to as a *k-itemset*. For example, if A = {I1, I2, I3}, then |{I1, I2}| ≤ |{I1, I2, I3}| (Adamo, 2001).

The power set lattice (2^A) of the set of items {I1, I2, I3, I4, I5} is shown in Figure 2.2.22. In this representation the set of all frequent itemsets forms a meet semilattice since it is closed under the meet operation. If A and B are frequent itemsets then $A \cap B$ is also frequent. It is important to note that $A \cup B$ is not necessarily frequent.

If there were enough memory all the frequent itemsets could be enumerated by traversing the power set lattice and using intersections to obtain itemset supports. Zaki (2000) has shown that the power set lattice can be subdivided into a number of sublattices that can be processed independently. He used an equivalence relation to partition the lattice into disjoint subsets called equivalence classes. The lattice theoretic approach will be used to partition the itemsets into independent sublattices to be assigned to processors.

Dynamic Distributed Rule Mining (DDRM) Algorithm

The system was implemented using C/C++ language and generated all the rules satisfying the required minimum support and confidence indicated by the user. It was implemented using an Ethernet LAN consisting of 7 workstations and one server. The configuration of each workstation on the network was an AMD Athlon XP 2800+ with 512 Mbytes of memory. The processors were interconnected via a 10/100 Mbps switch. The switch used 100BASE-T (Fast Ethernet) technology, which provides greater bandwidth. The message passing interface (MPI) was used for communications. The implementation of MPI was the windows message passing interface (WMPI) for 8 workstations from Critical Software Ltd.

3.2.1 Message Passing Interface (MPI)

The MPI model consists of P processors each with local memory, connected over a communication network. The cost for a processor to access its own memory is cheaper than for it to communicate with another processor. MPI facilitates communications among a set of processors that have only local memory through the mode of sending and receiving of messages. MPI is a standardized, portable, and widely available message-passing system that is robust and efficient. To use MPI, the program is written in C/C++ and the MPI library included in the program. The processes communicate with each other by calling the appropriate routine in the MPI library which implements the communications between processors. The following table shows some of the MPI functions used by DDRM system.

MPI Command	**Description**
MPI_Init()	Initialize MPI (no MPI function calls before that)
MPI_Comm_size()	Get total number of processors
MPI_Comm_rank()	Get process ID
MPI_Finalize()	Terminate MPI (no MPI function calls after)
MPI_Send()	Send a message
MPI_Recv()	Receive a message

The pseudo code for the Dynamic Distributed Rule Mining (DDRM) algorithm is shown in Figure 3.2.1. The preprocessing stage used to determine the capabilities of the processors is not shown in the pseudo code. The system uses a controller process to distribute the classes among the processors. The controller is also responsible for the receival and processing of all results generated by the processors. The algorithm is implemented in six steps as follows:

In step 1 the database is partitioned among *n* processors using the number of items in each transaction to determine the next processor to be assigned a transaction. The next

transaction is always assigned to the processor with the lowest count of items for all transactions assigned to it so far. The processors use the assigned transactions to generate the local tid-list, which is then transmitted to the controller. The controller uses all the tid-lists received from all the processors to create a single tid-list for the database.

In step 2 the tid-list is partitioned among the processors and used to compute F_2.

The tid-list is partitioned among the n processors using the length of the tid-list for each item to determine the processor to be assigned the next tid-list. A count of the total length of all tid-lists assigned to each processor is used to determine which processor will be assigned the next tid-list. The next tid-list is always assigned to the processor with the lowest count.

In step 3 the itemsets are partitioned into classes and the first n classes allocated to the n processors. The controller uses the partition algorithm described in Figure 3.2.4 to partition the itemsets into classes and assigns the first n classes to the n processors. The remaining classes are assigned dynamically to the next available processor. The controller sends only the atoms for the lattice associated with each class to the processor.

In step 4 the processors use all the classes to compute the frequent itemsets

The processors use the intersection of the tid-lists for the itemsets to determine the frequent itemsets. Each processor sends the frequent itemsets for the class to the controller.

In step 5 the dynamic exchange of classes and itemsets between the controller and the processors takes place. The controller receives the frequent itemsets from each processor and sends a class to the next available processor for processing. After all classes have

been assigned the processors may subdivide a class and its subclass migrated to an idle processor.

In step 6 the controller generates the rules from the frequent itemsets. The controller computes the set of rules using the frequent itemsets generated above.

The DDRM system was implemented using C/C++ language. It generated all the rules satisfying the required minimum support and confidence indicated by the user. The performance of the algorithm was compared with the prefix-based with bottom-up search algorithm as proposed by Zaki (2000). The specific steps followed were:

1 The DDRM algorithm was implemented using C/C++

2 The algorithm was executed with a varying number of processors.

3 The system captured the execution time for the algorithm for a set of transactions

4 The speedup of the system was measured by keeping the number of classes constant while increasing the number of processors.

5 The Scaleup of the system was measured by increasing the number of classes and processors.

The five steps above were repeated for the prefix based algorithm and a comparison made with the DDRM.

Input: Database of transactions and minimum support
Output: Frequent itemsets

Step 1
- i) The algorithm partitions the database among the N processors
- ii) Each processor uses the partition assigned to it to generate the tid-list
- iii) Each processor sends the tid-list to the controller
- iv) Controller creates a database of all tid-lists

Step2

//F_2 = (Set of Frequent 2-itemsets)
- i) Partition tid-list among processors
- ii) Each processor converts tid-list to generate F_2
- iii) Each processor sends F_2 to controller
- iv) Controller creates a database of F_2
- v) Each processor gets a copy of F_2

Step 3
- i) Controller uses partition algorithm to generate classes
- ii) Controller sends first N classes to processors

Step 4
- i) Each processor P_i computes the frequent itemsets in the class assigned to it
- ii) Each processor P_i sends frequent itemsets for class to controller

Step 5
- i) Controller receives frequent itemsets from each processor
- ii) Controller sends next class to available processor

This step is repeated until all classes have been processed
$j = N + 1$
Repeat
  Receive result from processor P_k
  Send class j to processor P_k
  $j = j + 1$
  Update frequent itemset list
Until $j >$ no of classes
Balance()
Step 6
Generate rules
End

Figure 3.2.1 Dynamic Distributed Rule Mining Algorithm

Balancing Algorithm

In order to make use of the processors more efficiently and to minimize the need for communication during the last phase where there is a need to redistribute the load the system will need to keep statistics on the performance of each processor to assist in the decision making process. The system will have an initial phase before the start of the computation of frequent itemsets. This phase will be used to identify the relative speed of each processor. This will be accomplished by creating a special class that will be assigned to all the processors in the computations. Each processor will measure the time taken to identify all frequent itemsets in the class and to send this information back to the controller. The controller will then store this information in a table sorted in ascending order based on the time taken by each processor.

The system will also capture the time taken to send and receive the data for each processor. This time will be the difference between the returned time and the sum of the send time and duration. This piece of information can be used to assist in determining whether or not there will be any gain in subdividing a class that was previously assigned to another processor.

When a processor becomes idle we will seek to redistribute the load initially assigned to the busy processors by subdividing the class with the longest remaining time to process and assign one of the sublattices to the idle processor. The remaining time to complete will not be precise as there is no way to accurately determine the time to process a class. However, we can do an estimation of the total time required to process a class based on statistics captured previously on a class of known size and the associated time required to send the class to the processor and to receive the results.

Assume the following information was collected on the test class

Size = S

Time to process = T

Transmission time = TX (This is the time to send and receive a class and its frequent itemsets to and from the controller)

The information on the test class will be collected in the preprocessing stage that will be used to determine the capabilities of all the processors participating in the data mining system.

Let us say that the size of the class currently assigned to a processor is X, elapsed time since assignment is TE.

Based on the statistics collected previously the time required (TR) to process this class is approximately given by

$TR = X/S*T$

Remaining time to complete $TL = TR - TE$

Approximate time required for transmission $TT = X/S*TX$

Let $\Delta T = TL - TT$

We will split the class and redistribute the load if $\Delta T > 0$

This is repeated for all processors. The processor with the largest ΔT will be selected for partitioning of the class assigned to it and a part sent to the idle processor.

The following example illustrates the steps of the DDRM algorithm using the sample data shown in Table 2.2.8. In step 1 shown in Figure 3.2.2 the transaction is partitioned among the two processors and the tid-lists generated. The tid-list from each processor is then aggregated by the controller to form a single tid-list.

P_0

Transaction	Size
T100	3
T400	3
T600	2
T800	4
Total	12

P_1

Transaction	Size
T200	2
T300	2
T500	2
T700	2
T900	3
Total	11

I1	I2	I3	I4	I5
T100	T100	T600	T400	T100
T400	T400	T800		T800
T800	T600			
	T800			

I1	I2	I3	I4	I5
T500	T200	T300	T200	
T700	T300	T500		
T900	T900	T700		
		T900		

I1	I2	I3	I4	I5
T100	T100	T300	T200	T100
T400	T200	T500	T400	T800
T500	T300	T600		
T700	T400	T700		
T800	T600	T800		
T900	T800	T900		
	T900			

Figure 3.2.2 Step 1 of DDRM: Generation of Tid-Lists

P_0

Itemset	Size
I2	7
I4	2
I5	2

Transaction			
1	I2		I5
2	I2	I4	
3	I2		
4	I2	I4	
5			
6	I2		
7			
8	I2		I5
9	I2		

P_1

Itemset	Size
I1	6
I3	6

Transaction		
1	I1	
2		
3		I3
4	I1	
5	I1	I3
6		I3
7	I1	I3
8	I1	I3
9	I1	I3

Items	1	2	3	4	5
1	0	4	4	1	2
2	0	0	4	2	2
3	0	0	0	0	1
4	0	0	0	0	0
5	0	0	0	0	0

F_2 = {{I1, I2}, {I1, I3}, (I1, I5},{I2, I3}, {I2, I4},{I2, I5}}

Figure 3.2.3 Step 2 of DDRM: Generation of F_2

Step 2 in Figure 3.2.3 shows the generation of the frequent 2-itemsets. The itemsets are partitioned among the processors, inverted into a horizontal format and passed back to the controller where they are then used to generate the set of all frequent 2-itemsets. It should be noted that in this approach candidate itemsets that are not included in at least 1 transaction will not be generated.

3.2.2 Lattice Partition

The pseudo code for the partition algorithm is shown in Figure 3.2.4. The algorithm uses the equivalence relation to partition the lattice for distribution among the processors (Adamo, 2001). In the algorithm the term cas refers to an itemset and a *k*-cas is the same as a *k*-itemset.

$C(s)$ = the class in $\theta(k|s)$ with $k \neq s$.

A is the set of itemsets. A = {I1, I2, I3, I4, I5}

$A[k]$ is the item in A with rank k, the leftmost element in A has a rank of 1. For example, $A[1]$ = I1, $A[3]$ = I3.

$\theta(k|s)$ is the set of *k-cass* in 2^A that have prefix s.

$|\theta(k|s)|$ is the count of *k-cass* in 2^A that have prefix s. It is a filter for the set of classes. The finer filter is obtained when $k = |s|$.

```
void split (int k)
  for (h = k – 1; h >= 0; h--) {
                    SetofCass cas (A, k, h) = set of all h-cass that can be
                    formed with attributes in
                    Pre (k-1, A) sorted according to
                    Lexicographic order;

    for all s in cas (A, k, h) {
      Generate the two sets:
        ∪k + 1 ≤ j ≤ |A| {s.A[k].A[j]} and ∪k + 1 ≤ j ≤ |A| {s.A[j]}
    }
     }
}
```

Figure 3.2.4 Procedure to Partition Lattice (Adamo, 2001)

Theorem 3.2.1

Let $\theta(k|s) = \{C \mid C$ is in $\theta(k)$ and $|s| \leq k$ and for all cass u in C, u has prefix $s\}$ and let r denote the rank in A of the last attribute in s (r is 0 when $s = \varnothing$). The size of $\theta(k|s)$ is $|\theta(k|s)| = C(|A| - r, k - |s|)$.

Proof

$|\theta(k|s)|$ is the count of k-cass in 2^A that have prefix s. In those cass, s is a fixed cas whose tail can be any (k - $|s|$)-length combination that can be formed with the last $|A|$ - r attributes in A. $\theta(k|s)$ works as a filter for the set of classes $\theta(k)$ (Adamo, 2001).

Cas (A, 2, 0) = {$\varnothing$}

To split the itemsets into 4 classes we proceed as follows:

$M = 4 = 2^k$

$K = 2$

And the four classes are as shown

Generate the two sets:

$$\cup_{k+1 \leq j \leq |A|} \{s.A[k].A[j]\} \text{ and } \cup_{k+1 \leq j \leq |A|} \{s.A[j]\}$$

The set of 1-itemset that can be formed with attributes in pre(k-1, A) is represented as

Cas (A, 2, 1) = {I1}

s = {I1}

The two sets generated from I1 are {{I1, I2, I3}, {I1, I2, I4}, {I1, I2, I5}} and

{{I1, I3}, {I1, I4}, {I1, I5}}

The set of 0-itemset that can be formed with attributes in pre(k-1, A) is represented as

Cas (A, 2, 1) = {$\varnothing$}

$s = \{\varnothing\}$

The two sets generated from ∅ are {{I2, I3}, {I2, I4}, {I2, I5}} and {I1, I4, I5}

Therefore the four classes are:

{{I1, I2, I3}, {I1, I2, I4}, {I1, I2, I5}}, {{I1, I3}, {I1, I4}, {I1, I5}}, {{I2, I3}, {I2, I4}, {I2, I5}}, {{I1, I4, I5}}

We can partition the itemsets into four sublattices as shown in the Table 3.2.5.

Lattice/ Class #	**Atoms**
1	I1I2I3 I1I2I4 I1I2I5
2	I1I3 I1I4 I1I5
3	I2I3 I2I4 I2I5
4	I3 I4 I5

Figure 3.2.5 Sublattices of Itemsets

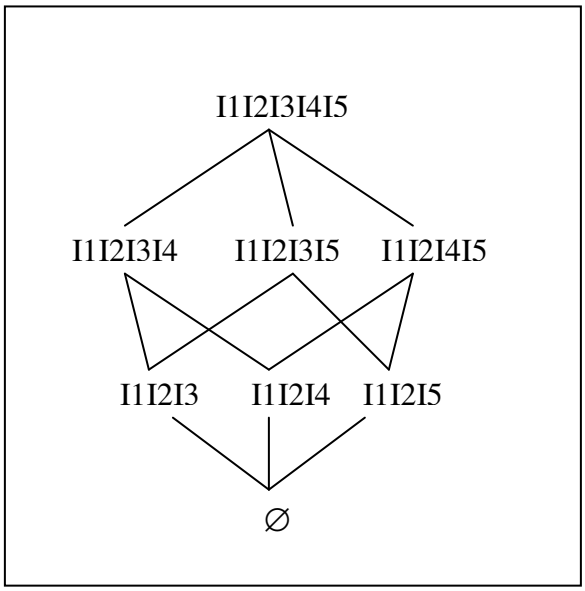


Figure 3.2.6 Lattice for Class 1

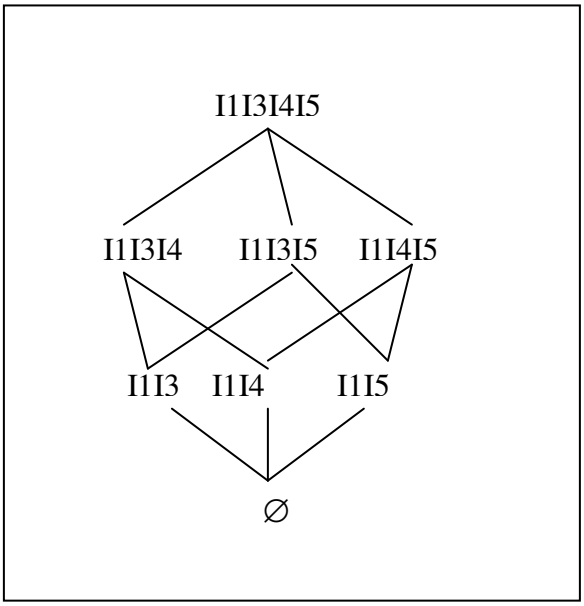


Figure 3.2.7 Lattice for Class 2

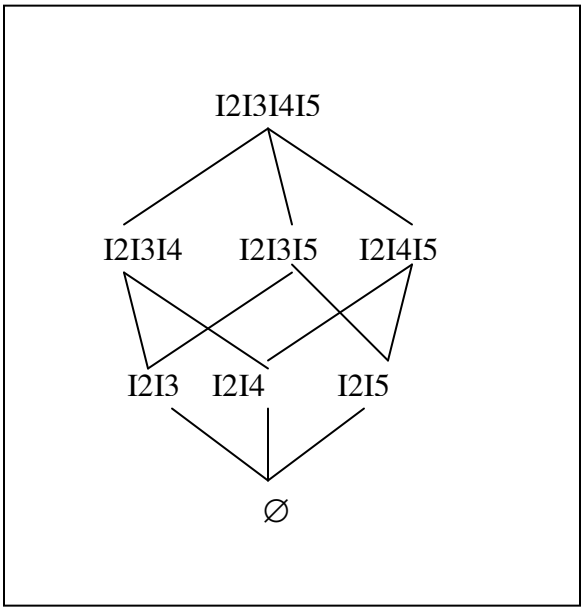


Figure 3.2.8 Lattice for Class 3

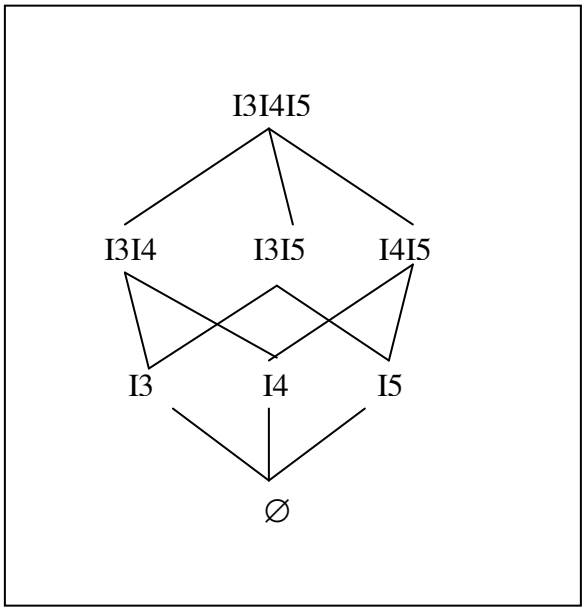
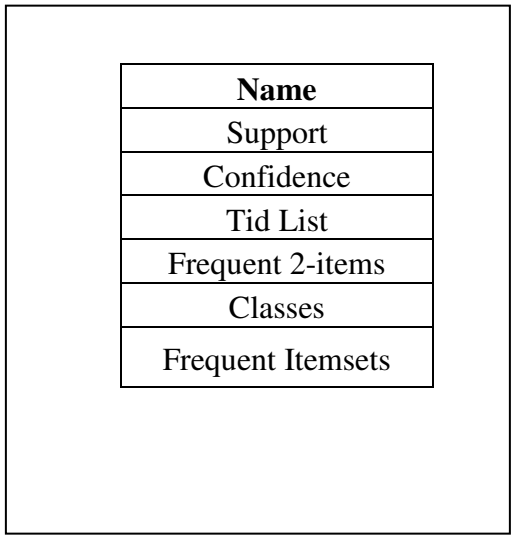


Figure 3.2.9 Lattice for Class 4

In DDRM we use an estimate of the time to process a class based on the number of intersections to determine all frequent itemsets in the class. An estimate of this cost is shown in the Table 3.2.3. This estimate can be seen from the intersection diagram for each class.

Step 3 in Figure 3.2.10 shows the allocation of two of the four classes generated by the partition algorithm.

Table 3.2.1 Typical Information for Controller



Controller

Table 3.2.2 Number of Class Intersections

Class	Number of Intersections
1	10
2	6
3	6
4	3

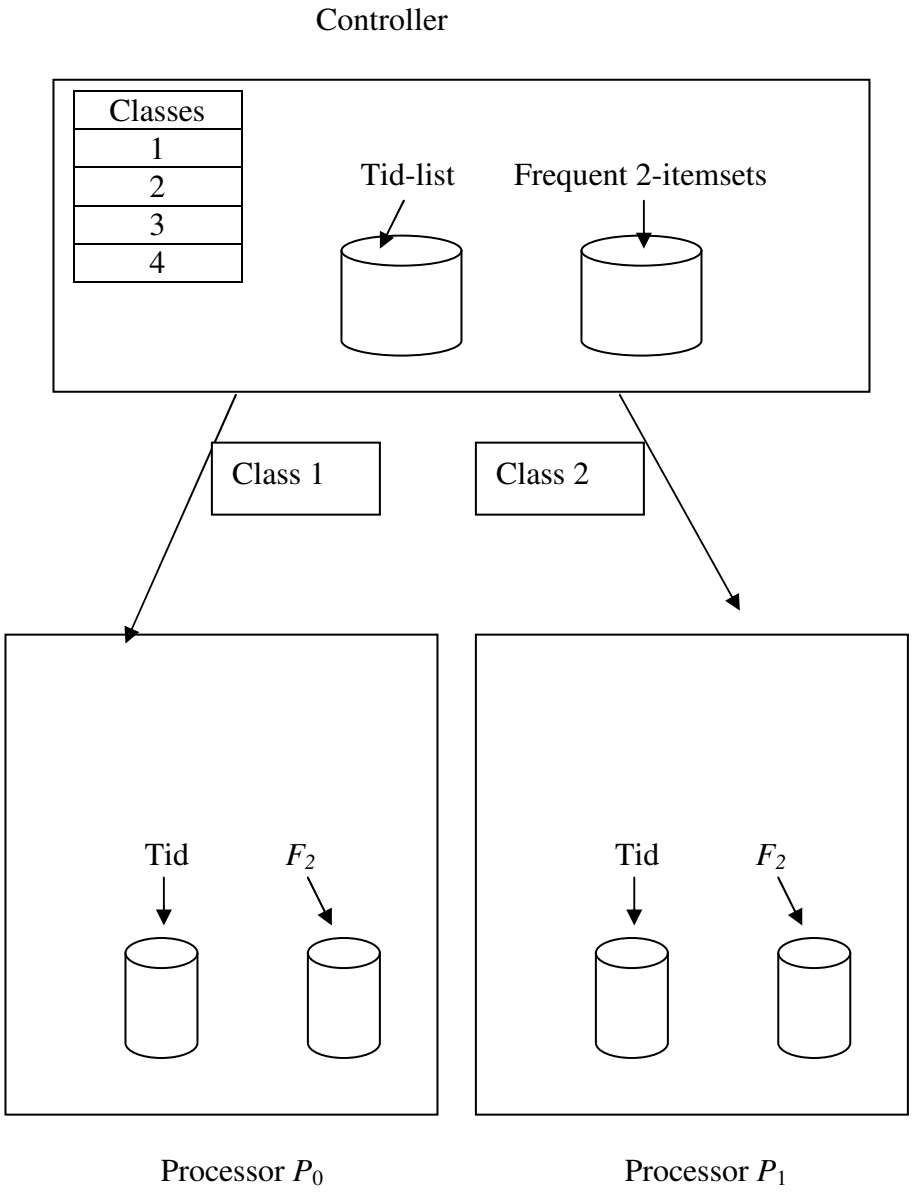


Figure 3.2.10 Step 3 of DDRM: Allocation of Classes

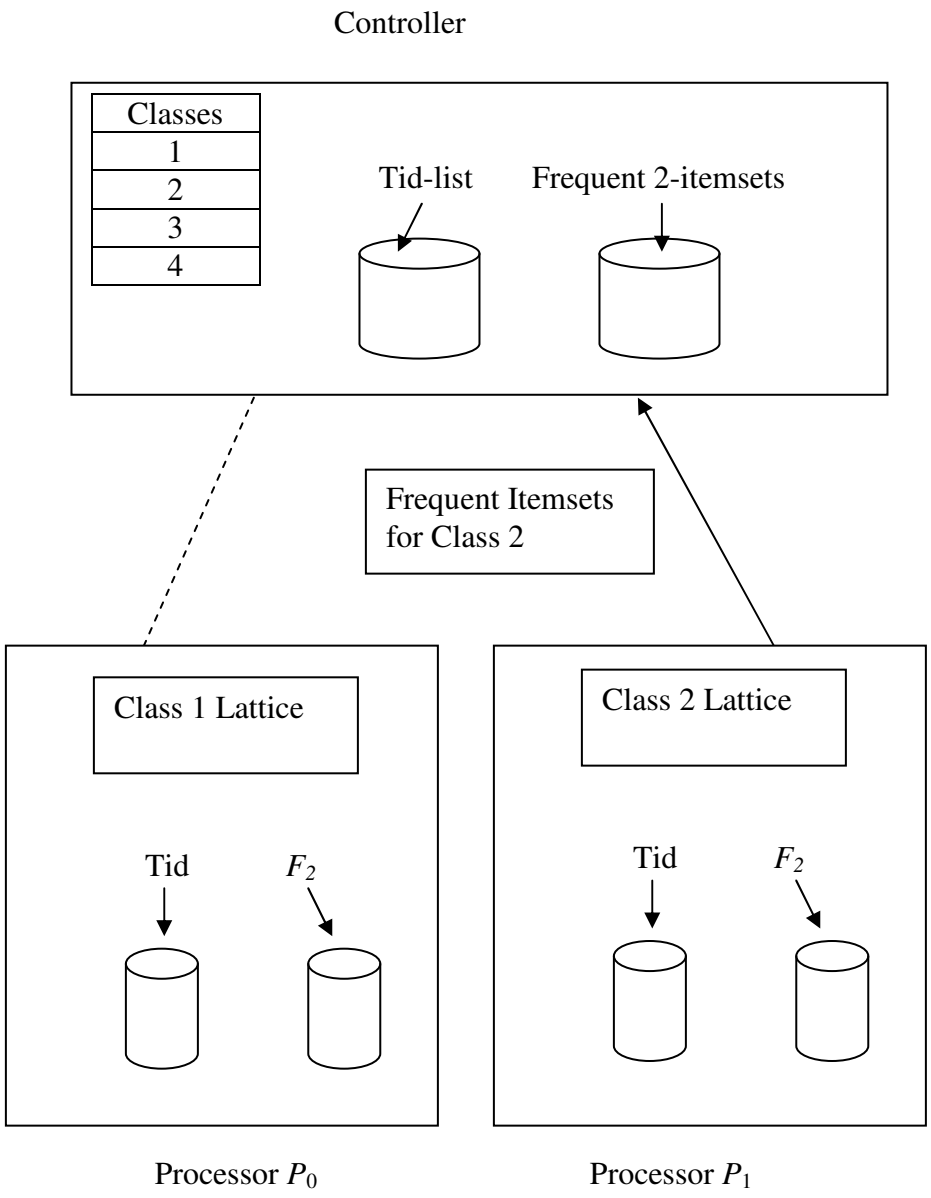


Figure 3.2.11 Step 4 of DDRM: Processing of Classes

Figure 3.2.11 and Figure 3.2.12 represent steps 4 and 5 respectively of the DDRM algorithm. These two figures show the dynamic allocation of the remaining classes to the processors. We omit the diagram that shows the assignment of class 4. The intersection diagrams for the four classes together with the lattice generated by each class are shown in Figure 3.2.13 to Figure 3.2.20.

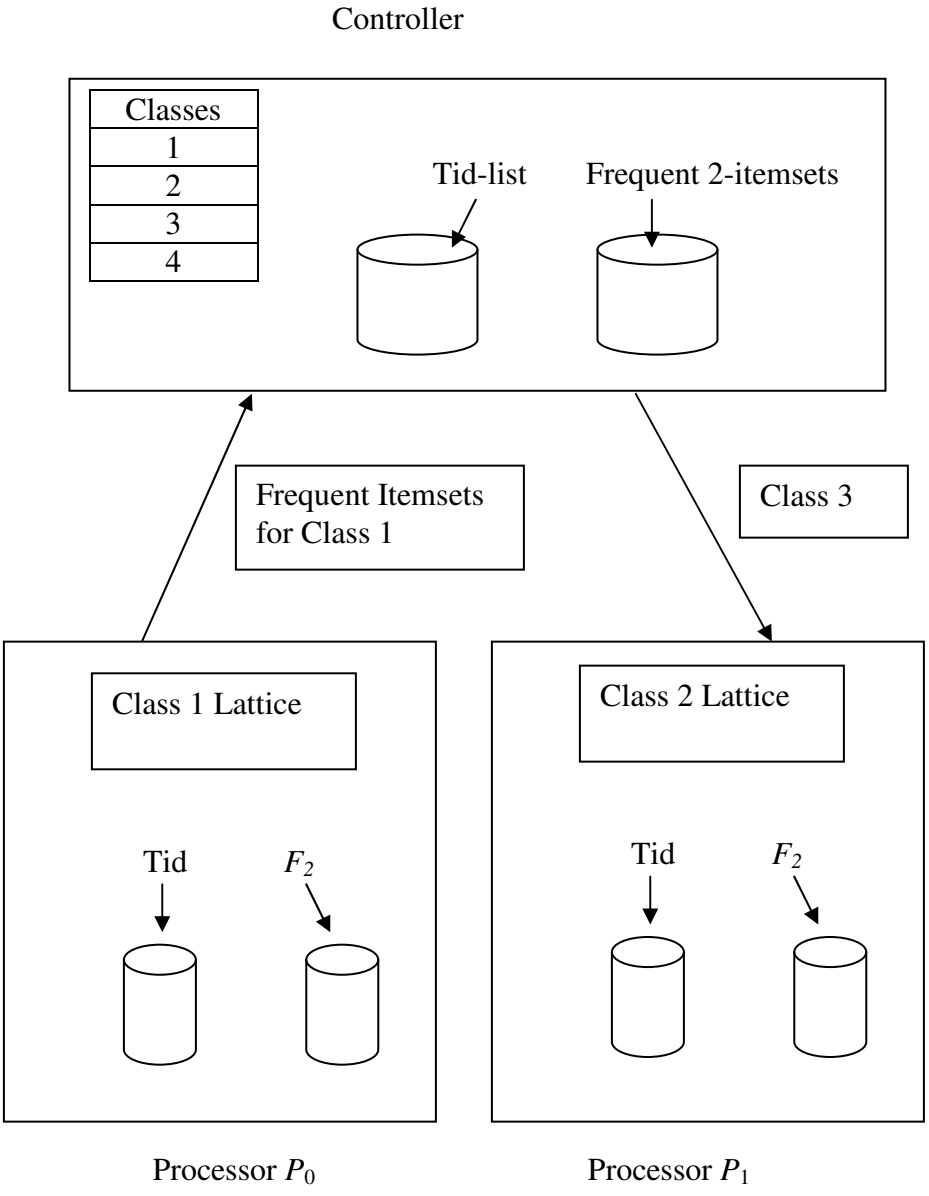


Figure 3.2.12 Step 5 of DDRM: Processing of Classes

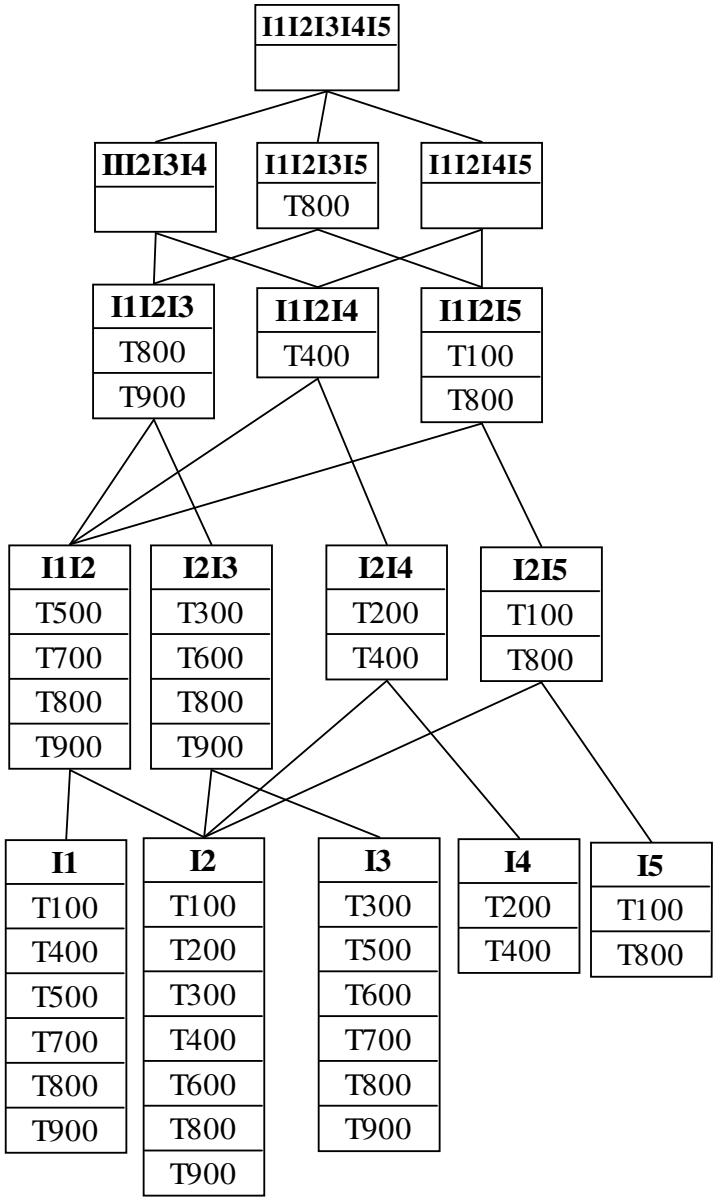


Figure 3.2.13 Intersection of Itemsets in Class 1

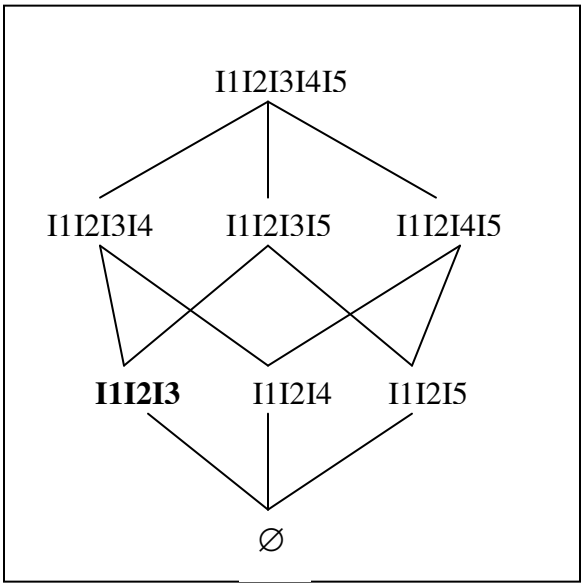


Figure 3.2.14 Lattice Generated by Class 1

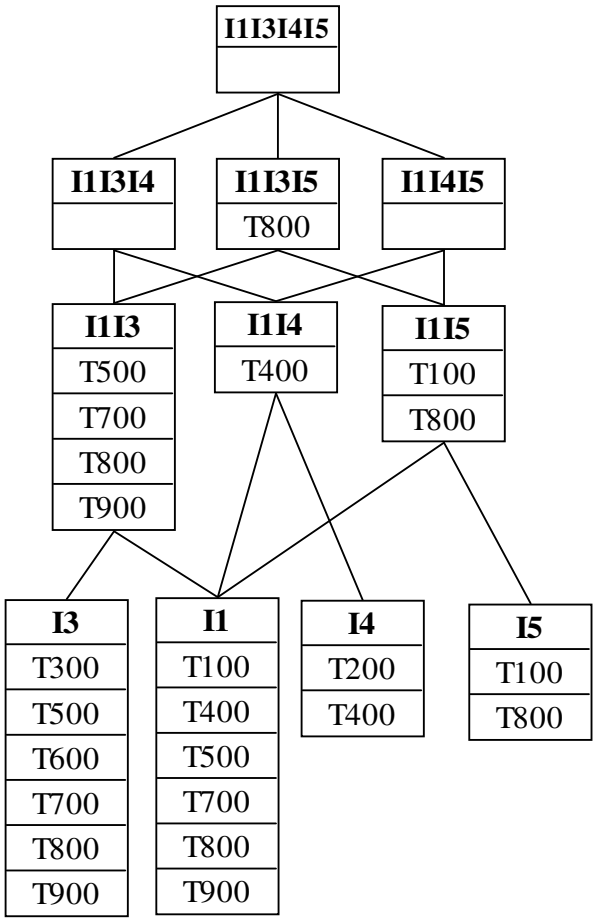
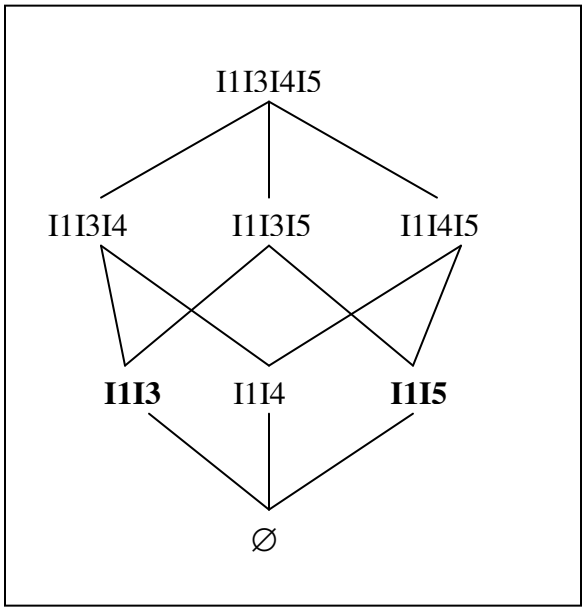


Figure 3.2.15 Intersection of Itemsets in Class 2



Frequent itemsets are shown in bold

Figure 3.2.16 Lattice Generated by Class 2

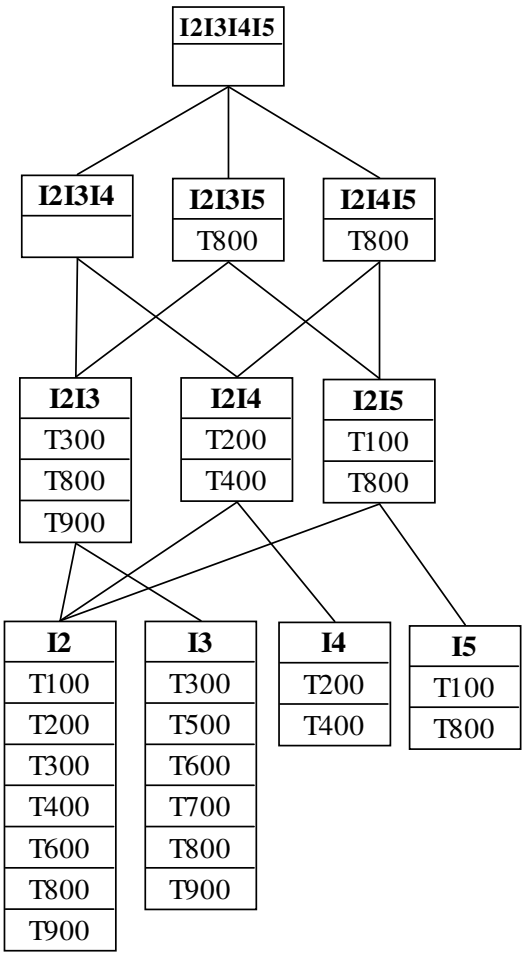
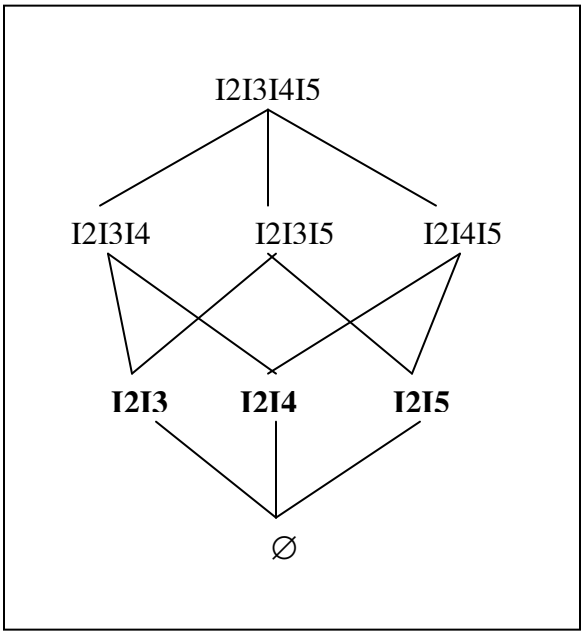


Figure 3.2.17 Intersection of Itemsets in Class 3



Frequent itemsets are shown in bold

Figure 3.2.18 Lattice Generated by Class 3

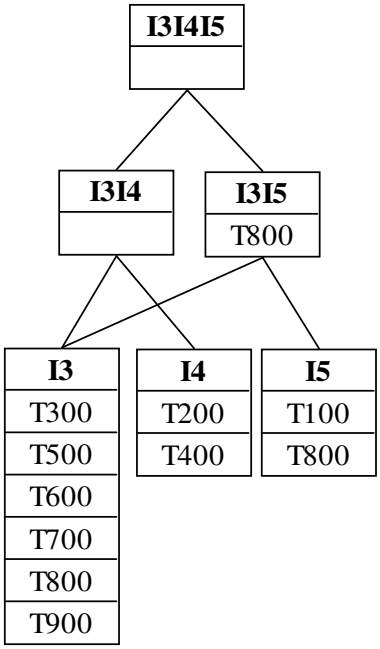


Figure 3.2.19 Intersection of Itemsets in Class 4

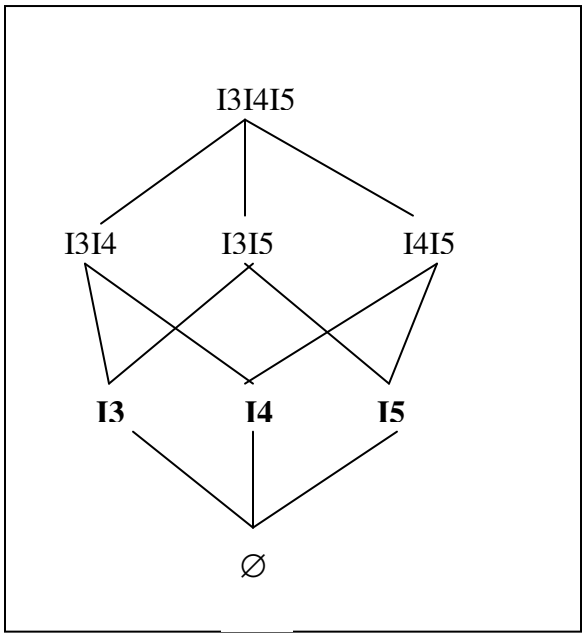


Figure 3.2.20 Lattice Generated by Class 4

It can be seen from Figure 3.2.14 that the frequent 3-itemsets are {I1, I2, I3} and {I1, I2, I5}. There are no frequent 3-itemsets for the remaining classes *C*2, *C*3 and *C*4.

The following data was collected, analyzed and compared with similar data for the parallel prefix-based and partition algorithms:

1. Total execution time for different number of classes and different number of processors
2. Speedup
3. Scaleup
4. Number of Transactions
5. Wait Time
6. Turnaround Time
7. CPU Utilization
8. Communication Time

Description of C/C++ Functions Used in DDRM Implementation

We show the data structures in DDRM in Appendix A. These are used to store the itemset, tidlist, and other important information needed by the system. Appendix B through to Appendix K show some of the main functions used in the implementation. Appendix B shows the function used to identify the set of all n-itemset that can be formed from pre(k –1) of an itemset. This function is used to generate the set of classes. The partition function used to create and store the set of all n-itemset in pre(k-1) of an itemset is shown in Appendix C. The n-itemsets are stored in a vector that is then used to generate the classes. It uses the setOfNCas function to generate all the n-itemsets. Appendix D shows the function used to generate all the classes used as sublattices of the

search space for the generation of the rules. It uses the vector generated by the DDRM partition function. In Appendix E the function used to generate two classes for each n-itemset in pre(k-1) is shown. This function is used by Generate All Classes function. The functions used to broadcast and receive the vector of tidlists are shown in Appendix F and Appendix G respectively. These functions are used by the system to broadcast and receive the tidlist vector to stations in the cluster. Appendix H shows the function used to send a class to a processor while Appendix I shows the function used to receive a class from a processor. The functions used to send and receive the frequent itemsets are shown in Appendix J and Appendix K respectively.

3.3 Comparison of Prefix-Based and DDRM Algorithms

The prefix-based approach uses an equivalence relation to partition the itemsets into classes, which can be processed independently to identify the frequent itemsets (Zaki, et al., 1997; Zaki, 2000).

The major strengths of the prefix-based approach are:

1. The utilization of the aggregate memory of all the parallel system by partitioning the candidate itemset among the processors.
2. The repartitioning of the database so that each processor can compute the frequent itemsets independently. This eliminates the need to communicate the frequent itemsets at the end of each iteration.
3. The use of a vertical database layout that clusters the transactions containing an itemset into tid-lists. This layout reduces the number of scans of the database. Computes frequent itemsets by intersection of the tid-lists. Eliminates the

overheads associated with the building and searching of complex hash tree data structures.

4. The avoidance of the overhead of generating all the subsets of a transaction and checking them against the candidate hash tree during support counting.
5. The use of the equivalence class recursively to cluster related itemsets during each iteration.

The above characteristics are also shared by DDRM since it seeks to improve on the prefix-based approach. The prefix-based algorithm uses a static approach to load balancing, which requires prior knowledge of the execution times for each class. The unavailability of information on the computation requirements makes it difficult to utilize the processors efficiently. A major goal of load balancing is utilization of all processors in order to improve the throughput. Performance measures such as throughput and completion time are directly affected by processor utilization.

A simple heuristic strategy to achieve higher utilization of a system is to avoid having idle processors as much as possible. In DDRM classes are assigned to the next available processor by the controller. Assignment of classes, computation of frequent itemsets and return of the results to the controller are carried out asynchronously.

The prefix-based approach uses the class to partition the load among the processors. This partitioning is final as there is no subsequent operation to address imbalances that may result during the computations. As a result one processor may be heavily loaded while there are idle processors available to assist in the computations. In DDRM, which uses a dynamic load balancing approach, the work is assigned to a processor as soon as it becomes idle. Dynamic load balancing will incur additional cost due to the movement of

work but is beneficial especially when there is a large work imbalance and the load changes with time.

DDRM uses a control monitor to assign classes to processors dynamically as a processor becomes idle. This is an improvement over the prefix-based approach, which assigns all the classes to the available processors statically. The communications cost associated with DDRM is low since most of the data required to process a class are already stored at the processor. In this approach a processor can only be idle if there are no more available classes at the controller to be processed. High processor utilization is an indication of high throughput. The controller can be viewed as a queue where all classes exit to be assigned to the next available processor. A comparison of Count Distribution, Prefix-Based and DDRM is shown in Table 3.2.3.

Table 3.2.3 Comparison of DDRM and Prefix-Based

Characteristics	**CD**	**Prefix-Based**	**DDRM**
Equivalence	No	Yes	Yes
Intersection	No	Yes	Yes
Communications	Itemset count	Class, Count	Class, Count
Static Scheduling	Yes	Yes	No
Dynamic Scheduling	No	No	Yes
Dynamic Load Balancing	No	No	Yes
Counts Itemsets Independently	No	Yes	Yes
Synchronization	Yes	No	No
Assign Work to Idle Processors	No	No	Yes
Hash Table	Yes	No	No

3.3.1 **Static Approach**

It is difficult for a static approach to accurately reflect the behavior of the classes assigned to each processor based on computation time required by each class. In a dynamic approach it is possible to partition the class assigned to a processor and migrate some of these partitions to idle processors. This will improve the throughput of the system. The partitioning of a task and subsequent migration of some of the resulting subtasks will result in increased communications overhead. The system must balance the overhead communication cost against the resulting improvement in the throughput.

According to Jacob and Lee (1999) in cases where the communication or computation characteristics change with time the shrinking of the task may improve on the throughput of the system. For example, the computation of count of *k*-itemsets is computationally more demanding for small *k* than for larger *k*. This means that for small *k* the computation phase would dominate the communication phase while for large *k* the communication phase would dominate and therefore a smaller number of processors would be used.

A static approach which uses a simple strategy of partitioning the problem initially as finely as possible may be an indication that support for further partitioning of the task is unnecessary. Using the finest partition may result in a high overhead associated with the assignment of tasks to the processors. The algorithm complexity and time to assign tasks to processors could be reduced with coarse grained partitioning of the problem (Jacob & Lee, 1999).

A disadvantage of the static approach is that if there are no frequent itemsets in the classes assigned to a processor while there are many frequent itemsets in the classes

assigned to the other processor then the turnaround time will be very high. This is due to the fact that one processor will be kept busy processing the classes assigned to it while the other processor will be idle most of the time. To illustrate this condition we use the database of transactions shown in Table 3.3.1 with vertical layout shown in Figure 3.3.2. There are 5 items in the database. We will partition the itemsets into four classes. Class 1 which is the largest class is assigned to processor 0 and the remaining classes assigned to processor 1.

After the first set of intersections it is clear that there are no frequent itemsets in class 1 as shown in Figure 3.3.1. The processing of class 1 will end at this point and since there is no additional class assigned to processor 0 it will remain idle. On the other hand the remaining classes assigned to processor 1 all contain several frequent itemsets, which means that processor 1 will spend a significant amount of time processing each class, while processor 0 remains idle. The intersection diagrams for the classes assigned to processor 0 are shown in Figure 3.3.2 and Figure 3.3.3. In a dynamic approach the classes are assigned to processors as soon as they are available. In addition whenever there is an idle processor a large class can be subdivided and a subclass migrated to the idle processor.

Table 3.3.1 Transaction Database

TID	**List of Items**
100	I1, I2
200	I1, I5
300	I2, I3,I5
400	I2, I3, I4, I5
500	I1, I3
600	I1, I4
700	I1, I2
800	I4, I5
900	I2, I3, I4
1000	I2, I3, I4, I5
1100	I2, I4, I5

Table 3.3.2 Vertical View of Transaction Database

I1	**I2**	**I3**	**I4**	**I5**
100	100	300	400	200
200	300	400	600	300
500	400	500	800	400
600	700	900	900	800
700	900	1000	1000	1000
800	1000		1100	1100
	1100			

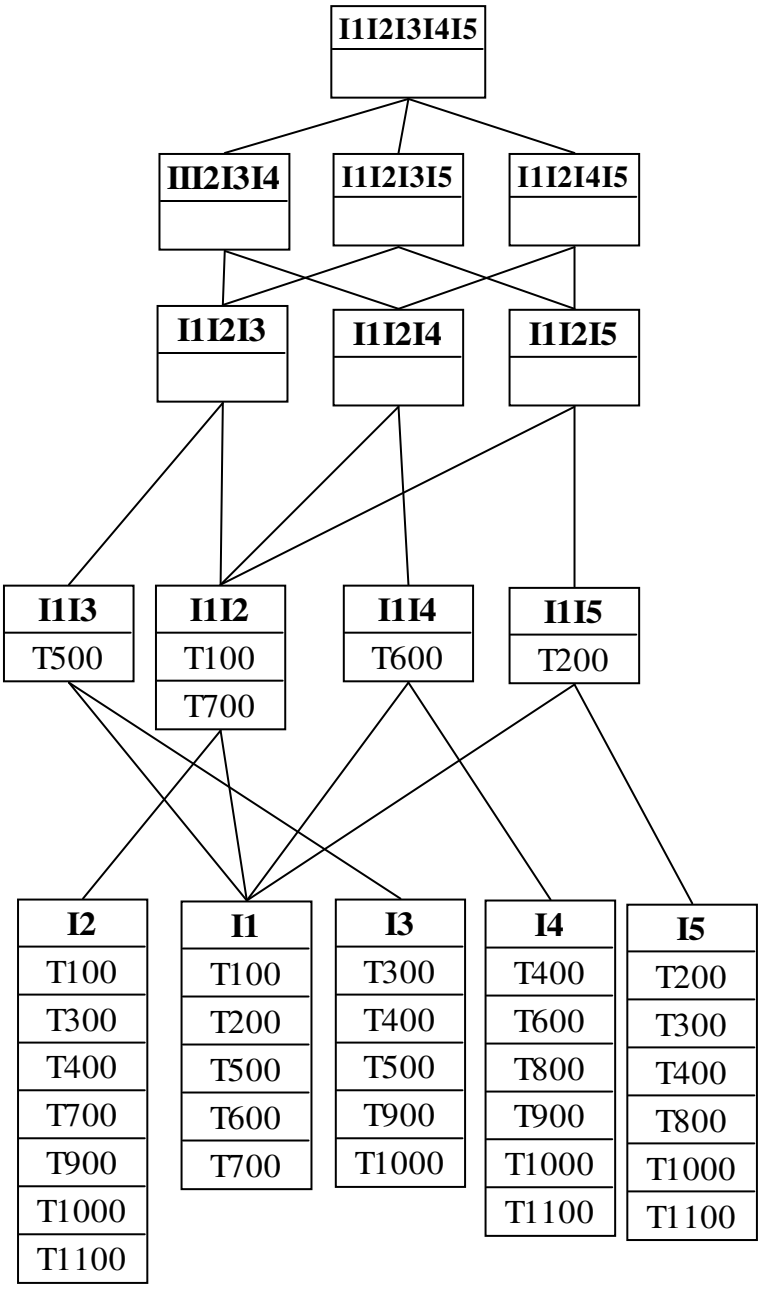


Figure 3.3.1 Intersection of Itemsets in Class I1

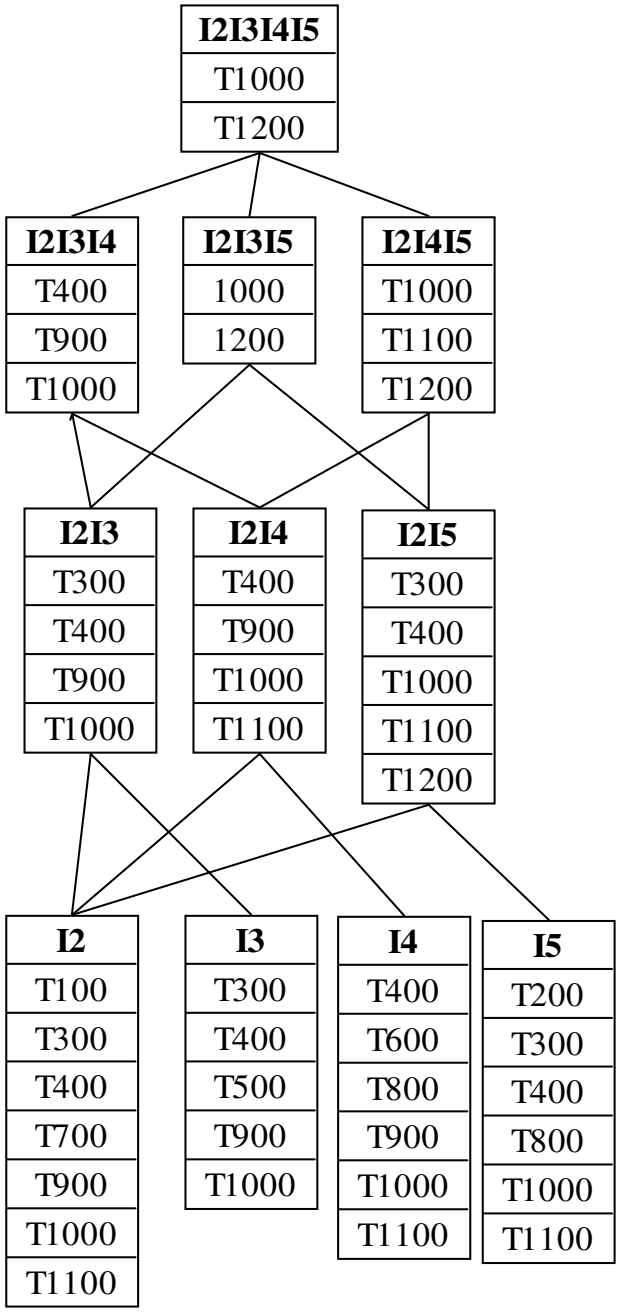


Figure 3.3.2 Intersection of Itemsets in Class I2

In Figure 3.3.2 the intersection of the tid-lists all give rise to new tid-lists that are also frequent. This class will require significantly more computation time than class 1. Similarly class 3, which is shown in Figure 3.3.3, will also give rise to tid-lists that are frequent from the intersection of the intermediate tid-lists.

This example illustrates the potential benefit that can be obtained in a dynamic approach. In a dynamic approach the work associated with classes 2 and 3 would be partitioned between the two processors instead of being assigned to one processor while there is another available idle processor.

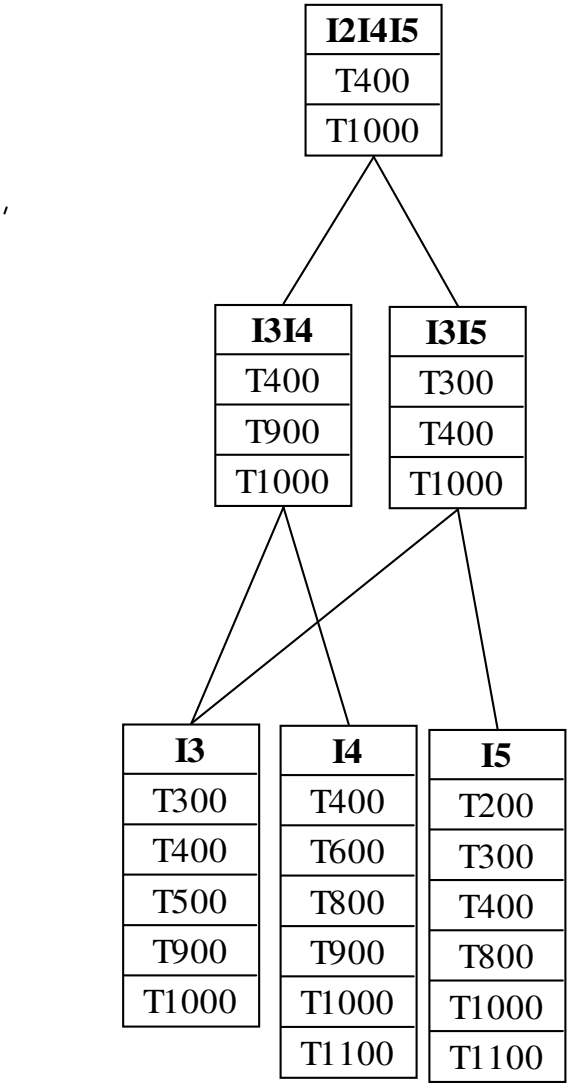


Figure 3.3.3 Intersection of Itemsets in Class I3

Lattice Partition and Data Independence

A binary relation ≤ on a set L is a partial order if it is transitive, reflexive and antisymmetric. A lattice consists of the pair $(L, \leq)$ in which ≤ is a partial order on L, and every subset $\{a, b\}$ consisting of two elements has a least upper bound (LUB) and a greatest lower bound (GLB). A lattice is a mathematical structure with two binary operators, which are Join and Meet.

Properties of Lattices

The meet, join, unique maximum and unique minimum element are always defined.

Given a lattice $(L, \leq)$, a non-empty subset S of the set L is called a sub-lattice if $a \vee b \in S$ and $a \wedge b \in S$ whenever $a \in S$ and $b \in S$.

The lattice formed from the atoms {a, b, c, d) is the powerset shown below:

L = {a, b, c, d, ab, ac, ad, bc, bd, cd, abc, abd, bcd, abcd}

This set is a lattice since the meet and join of all elements are defined.

We use θ_k to partition the lattice as shown in the table below:

θ_k	**Set1**	**Set2**	**Set3**	**Set4**
1	{**a**, **a**b, **a**c, **a**d, **a**bc, **a**bd, **a**bcd}	{**b**, **b**c, **b**d, **b**cd}	{**c**, cd}	{**d**}
2	{**ab**, **ab**c, **ab**d, **ab**cd}	{**bc**, **bc**d}	{**cd**}	
3	{**abc**, **abc**d}	{**bcd**}		
4	{**abcd**}			

The elements of each set are equivalent since they share a common prefix.

We have highlighted the prefix in each set that is common to all elements of the set for k = 1 to 4.

For example, for a prefix length of k = 2 we see that the equivalent elements of Set1 are **ab**, **ab**c, **ab**d, and **ab**cd. It can also be seen that the join and meet are defined for these

elements that make up the sublattice corresponding to Set1. Set1 is therefore a lattice since the join and the meet are defined. In addition Set1 is a subset of *L*.

For example, **Join** : **ab** $\vee$ **ab**c = **ab**c and **Meet: ab** $\wedge$ **ab**c = **ab**.

We can therefore use the equivalence function to partition the lattice into sublattices that can be processed independently of each other.

We can use the property of a lattice to partition our data set into sublattices, which we can then process independently. We represent the set of itemsets as a lattice that we then partition into sublattices using the equivalence operation shown above. These sublattices are then assigned independently to processors for processing and identification of the frequent itemsets.

3.4 Summary

In this chapter we discussed the specific procedures employed, the use of lattice theory, parallel data mining systems, and their applications to the mining of association rules. This chapter also presented a detailed description of the DDRM algorithm and an illustration of how it works. We also presented a comparison of the DDRM and prefix-based algorithms.

Chapter 4

Results

In this chapter, we will discuss parallel processing and two key parameters used to measure the efficiency of parallel algorithms. We also present the results obtained from the implementation of the Dynamic Distributed Rule Mining (DDRM) algorithm. It is a lattice-based algorithm that partitions the lattice into sublattices to be assigned to processors for processing and identification of frequent itemsets. It generates the frequent itemsets by partitioning the itemsets into sublattices that are assigned to the processors based on their availability.

4.1 Parallel Algorithms

In parallel systems the scheduling of work is an attempt to distribute the work equally among the processors. In static scheduling the load is balanced before run time and requires an estimation of the run time for each task. Knowledge of the characteristics of the problem is generally used to inform the estimation process. A good dynamic scheduling algorithm can schedule the load at run time and seeks to balance and overlap computation and communication. The goal of this algorithm is to reduce communication while at the same time increasing the extent of concurrency.

Ideally, there should be no limit to the number of processors and increasing the number of processors should produce a corresponding increase in the power of the system. Placing an unbounded number of processors close to global memory will retard

the processing speed. The performance will also be negatively affected the farther the processors are from memory (DeWitt, & Gray, 1992). These limitations that are associated with the use of global memory are eliminated by DDRM since it uses a shared-nothing architecture. In parallel processing we add more processors to a system in order to execute a job faster or to execute a larger job in approximately the same amount of time.

Speedup is defined as holding the job size constant while reducing the execution time. We refer to holding the execution time constant while increasing the size of the job as scaleup. In general the type of problem and the characteristics of the physical platform on which the job is executing determine the shape of the curve associated with speedup and scaleup.

Systems that are bus-based such as symmetric multiprocessors (SMP) are not scalable to larger configurations. The bandwidth bottleneck associated with memory and bus prevents the number of central processing units (CPUs) from increasing beyond a fairly small number. The design of DDRM is different from the SMP since DDRM uses a shared-nothing architecture. Therefore, DDRM does not suffer from the bandwidth bottleneck associated with the memory and the bus in the SMP architecture. Linear speedup can be accomplished on shared-nothing architectures. Shared-nothing architecture facilitates the addition of CPUs in order to reduce the time taken for a set of operations. This architecture also supports linear scaleup. The rapid improvement in performance of single CPU systems makes it possible to build more powerful systems using single CPU systems.

4.2 Performance Parameters and Benchmark

Benchmarking has played an important role in the development and research of data mining. The design of the benchmarking is to facilitate an analytical comparison of DDRM with other methods of parallel mining of association rules. The benchmarking for DDRM consists of the following:

1. A set of algorithms for the parallel mining of association rules. These algorithms are DDRM, Partition and the Prefix-based.
2. A set of performance curves. These curves will be generated from the algorithms included in the tests. We will measure the execution time, scaleup, and speedup.
3. The datasets consist of the census data and data obtained from Knowledge Discovery and Data Mining (KDD) Cup Competition in Association for Computing Machinery (ACM) Special Interest Group on Knowledge Discovery and data Mining (SIGKDD) conference.

An important area of parallel systems performance measure is scalability. The hardware and software must be able to grow in response to the changing demands on the system. This change in demand can occur in areas such as number of transactions, number of users, complexity of applications and the need for improved execution time. The goal in designing a parallel system is to facilitate its development along any of these dimensions in a flexible and productive manner.

Ideally there should be no inherent upper limit that would prevent the satisfaction of future requirements. In constructing a good parallel benchmark that takes scalability into account there are a number of metrics to be investigated. It should be executed over a

range of problem size and machine size. We add more processors to a system to execute a given job faster or a larger job in the same amount of time.

The shape of the speedup curve depends on the problem size since larger problems will be better able to utilize larger number of CPUs more efficiently than will small problems. In scaleup the gap between the achieved output versus the ideal decreases as a percentage of total runtime. The values obtained for speedup and scaleup will depend on the type of problem as well as the physical platform on which the job is executing. In a distributed memory architecture messages are used to move data around over the interconnection network.

Benchmarking can be used for comparing systems. It can assist in determining which of a set of competing products can do the job faster for a given problem size. The construction of a good parallel benchmarking must include a number of techniques for scalability including speedup and scaleup. The speedup curve obtained is subject to Amdahl's law. This law states that if the problem size is held constant, eventually the sequential component of the problem dominates and a point is reached at which adding more nodes no longer improves performance.

In scaleup we increase both the problem size and the machine size together, a larger problem executing on a larger machine. In this case tripling both the problem and the machine size should result in no change in executing time. In the complexity measure the machine size is held constant while measuring the performance on a set of problems of varying sizes.

4.3 Dynamic Distributed Rule Mining (DDRM) Algorithm

The DDRM algorithm was developed and implemented using C/C++ as the programming language. We implemented the DDRM algorithm on an Ethernet LAN consisting of 7 workstations and one server. Each workstation on the network is an AMD Athlon XP 2800+ with 512 Mbytes of memory. The processors are interconnected via a 10/100 Mbps switch. The switch used 100BASE-T (Fast Ethernet) technology, which provided greater bandwidth and improved the client/server response time. For communications we used the message passing interface (MPI). We used the windows message passing interface (WMPI) for 8 workstations from Critical Software Ltd to implement our algorithm. The support count used was 8% with a confidence of 50%. The experiment was conducted by partitioning the database among the processors.

Datasets

We used the 1987 census data from the Statistical Institute of Jamaica to generate the data used in this experiment. The size of the database was 26 Mbytes with 1.1 million records. We selected parish, race, religion, type of school/university attended, and examination passed as categories to be investigated. Each of these categories was further subdivided into specific items, with each item being assigned an integer value used to represent it in the data file. Each record has 63 attributes. Jamaica is divided into fourteen parishes and each parish is assigned an integer code. The codes assigned to the parishes of St. Catherine and St Ann are 14 and 6 respectively. The sample input file with the codes assigned to these fields is shown in Appendix L

We also used data from the KDD Cup 2000 dataset. This data set was based on e-commerce data obtained from a small dot-com company called Gazelle.com. The size of

the data was approximately 4.5 Mbytes with 3072 transactions. It included customer information such as gender, occupation, age, marital status, estimated income and home market value.

4.4 Experimental Results

In the speedup experiment the data was partitioned equally among the processors for each run. For example, for two processors the data was partitioned into two parts and each part assigned to a processor. In Table 4.1.1 we show an integer at the end of each file name to indicate the assignment of files to the processors. The file c:\\data3_26\\Mypopdata3 would be assigned to processor 2 while the file c:\\data3_26\\Mypopdata1 would be assigned to processor 0. An example of the output file obtained is shown in Appendix M.

In the scaleup experiment, the data was partitioned into eight parts and a processor added for each part processed. In this approach, we keep adding a processor to do the additional work required for each partition added.

In the experiment on the number of transactions processed, the number of processors was fixed at 7 while growing the size of the database. The database size was varied from 9.8 MB to 26.3 MB as shown in Table 4.1.2.

Table 4.1.1 Description of Data Files

FILE NAME	DESCRIPTION
c:\\data2_26\\Mypopdata2	Data partitioned into two files
c:\\data3_26\\Mypopdata3	Data partitioned into three files
c:\\data4_26\\Mypopdata4	Data partitioned into four files
c:\\data5_26\\Mypopdata5	Data partitioned into five files
c:\\data6_26\\Mypopdata6	Data partitioned into six files
c:\\data7_26\\Mypopdata7	Data partitioned into seven files

Table 4.1.2 Size of Transactions Data Files

FILE NAME	SIZE (MB)
Db_1	9.8
Db_2	13.1
Db_3	16.4
Db_4	19.7
Db_5	23.0
Db_6	26.3

Response Time

In Figure 4.2.1 we show a plot of the response time on the vertical axis and the number of processors on the horizontal axis for the DDRM algorithm. The results obtained for this experiment is shown in Table 4.1.3. The database was partitioned based on the number of processors. The search space was partitioned into 32 classes, which were assigned dynamically to the processors participating in the cluster. It can be seen that as we increase the number of processors the response time decreases as well. The results obtained for the Prefix-based and Partition algorithms are shown in Figure 4.2.2 and Figure 4.2.3 respectively. In Figure 4.2.4 we plot the response times for all three algorithms. It can be seen that the performance of DDRM is better than both Partition and the Prefix-based. Figure 4.2.4 shows that for the same number of processors DDRM is able to process the classes in a shorter time than both the Prefix-based and Partition algorithms. This improvement in the response time is due to the fact that the dynamic assignment of the classes to idle processors results in a more efficient use of the processors than the static assignment of classes to processors used by the other two algorithms. This improvement in the efficiency of the usage of the processors is also demonstrated in Figure 4.2.7 where the Prefix-based and Partition algorithms take a longer time than DDRM to process each of the databases shown.

Speedup

Table 4.1.4 shows the results obtained for the speedup experiment. The speedup experiment was conducted to determine how DDRM performs as the number of processors is increased with the number of transactions remaining constant. The speedup

obtained on a fixed size database and varying number of processors and partitions is shown in Figure 4.2.5. The speedup obtained for DDRM is better than that obtained for the Prefix-based and Partition algorithms.

Scaleup

In the scaleup experiment, the number of partitions to be processed was incremented with a corresponding increase in the number of CPUs. Table 4.1.5 and Figure 4.2.6 show the results obtained for the scaleup experiment. Ideally, the time to generate the rules should remain constant, since an additional CPU is assigned to each additional partition to be processed. However, the time to process each class will vary due to the fact that not all classes will necessarily generate rules. In general, classes with rules require more time to process than classes without rules. In addition, classes with a high concentration of frequent itemsets will take longer to process than classes with a low concentration of frequent itemsets. For example, a class may be discarded after the first set of intersection of attributes if the resulting itemset is not frequent. However, if it is frequent then processing will continue until the resulting itemset is not frequent or all rules have been identified and generated.

Number Of Transactions

In this experiment, the number of processors was fixed at 7 CPUs while growing the size of the database. The database size was varied from 4.8 MB to 26.3 MB. Table 4.1.6 and Figure 4.2.7 show the results obtained for the number of transactions processed. It can be seen from Figure 4.2.7 that as the size of the database increases, there is an increase in the time to process the classes for all three algorithms. However, DDRM takes less time to process these transactions as it uses the CPUs more efficiently.

Support

We conducted experiments on three sets of data where we vary the support from 4% to 10% for all three algorithms. The tables for the results obtained for the three sets of data are shown in Table 4.1.7, Table 4.1.8, and Table 4.1.9. The results of these experiments are shown in Figure 4.2.9, Figure 4.2.10 and Figure 4.2.11. It can be seen from these figures that as we decrease the minimum support from 10% to 4%, there is a corresponding increase in the execution time of all three algorithms. This is in keeping with our expectations, since a decrease in minimum support will lead to an increase in the number of frequent itemsets that would satisfy this requirement. In addition an increase in the number of frequent itemsets will lead to an increase in the processing time required to identify the relevant rules. It can also be observed that DDRM is able to process these itemsets in a shorter time than Partition and Prefix-based algorithms.

Transaction Width

Table 4.1.10 and Figure 4.2.12 show the results of our experiment to determine the impact of varying the transaction width, on the processing time. The number of attributes was varied from 10 to 50 for the five databases that were used in this experiment. It can be seen from Figure 4.2.12 that as the transaction size is increased that there is a corresponding increase in the processing time. DDRM is able to process these transactions in a shorter time than the Partition and Prefix-based algorithms.

Table 4.1.3 Execution Time

CPUs	**2**	**3**	**4**	**5**	**6**	**7**
	Seconds	Seconds	Seconds	Seconds	Seconds	Seconds
Partition	4910	3321	3064	3925	3780	2782
Prefix	4814	3018	3502	3128	3182	2910
DDRM	3871	2478	2435	2506	2415	2179

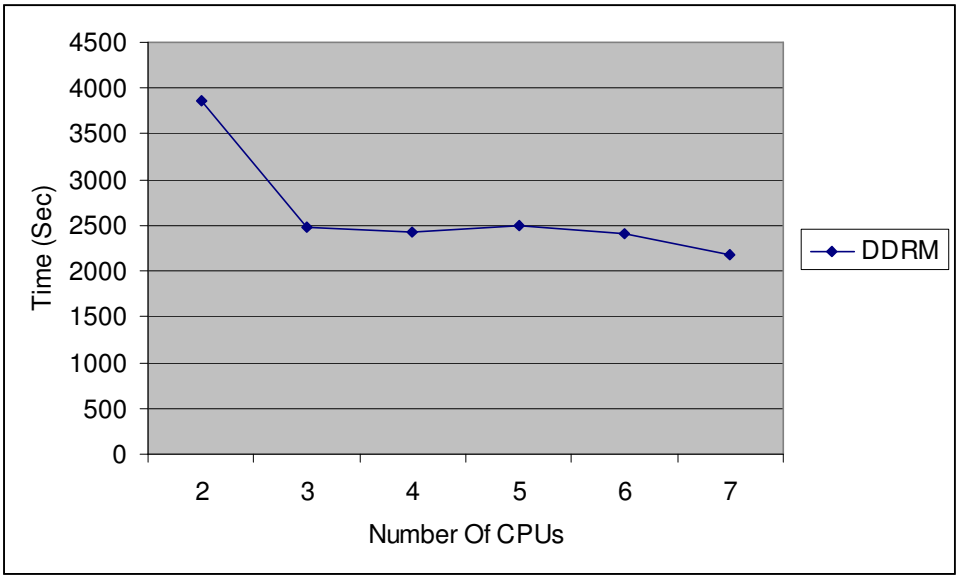


Figure 4.2.1 Execution Time For DDRM

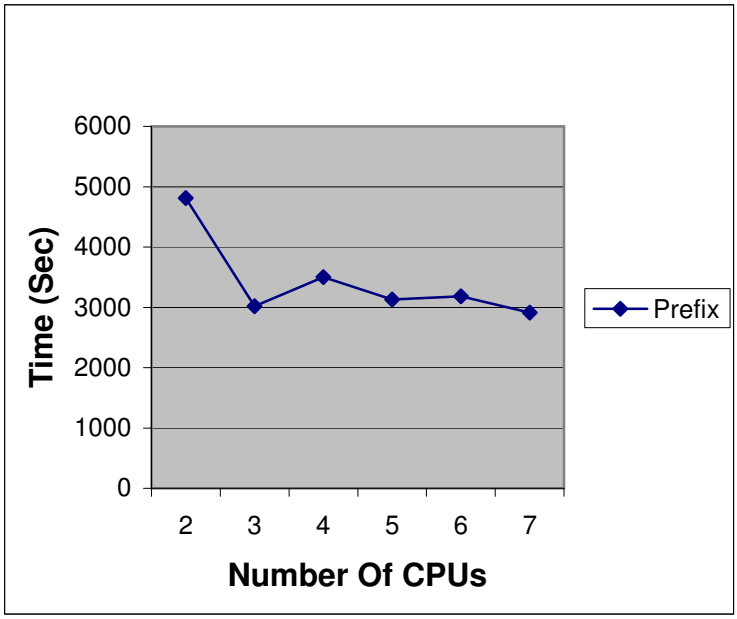


Figure 4.2.2 Execution Time for Prefix

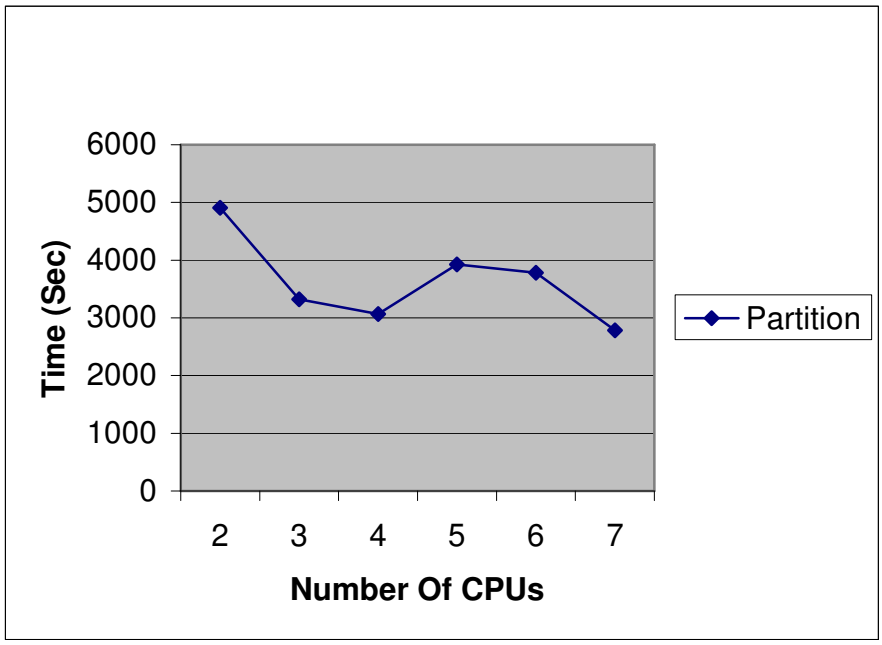


Figure 4.2.3 Execution Time for Partition

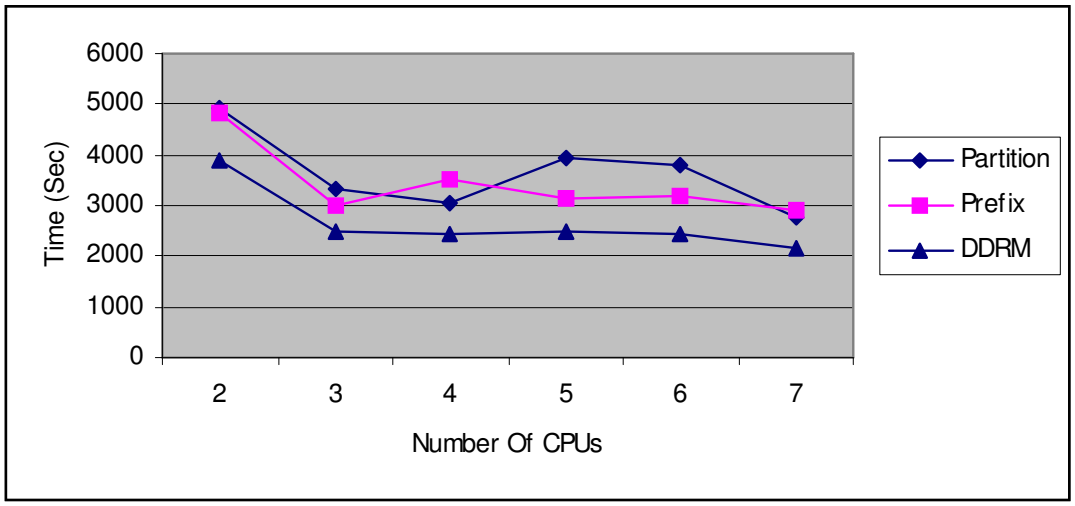


Figure 4.2.4 Execution Time for DDRM, Partition, and Prefix

Table 4.1.4 Speedup

CPUs	2	3	4	5	6
Partition	1.5	1.6	1.3	1.3	1.8
Prefix	1.6	1.4	1.6	1.5	1.7
DDRM	2.0	2.0	2.0	2.0	2.3

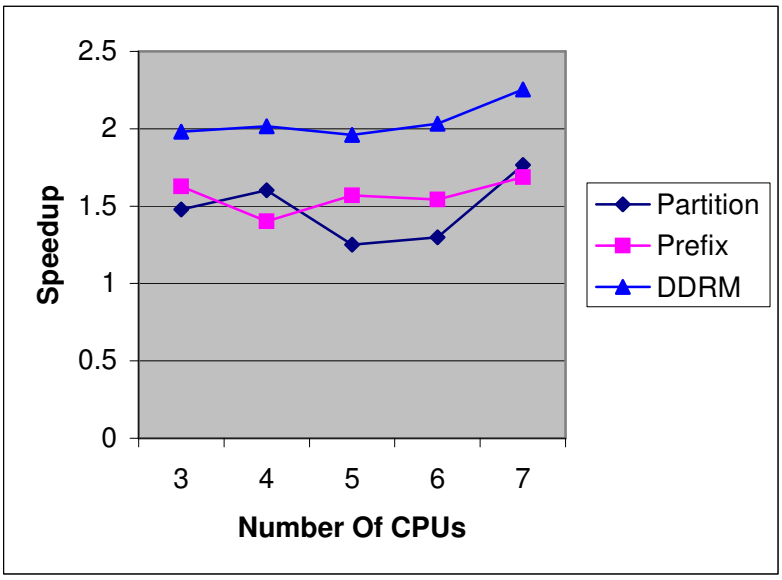


Figure 4.2.5 Speedup for DDRM, Partition, and Prefix-based

Table 4.1.5 Scaleup

CPUs	**7**	**6**	**5**	**4**	**3**	**2**
	Seconds	Seconds	Seconds	Seconds	Seconds	Seconds
Partition	5542	5236	4571	4182	4028	5114
Prefix	4736	3903	3544	2921	2867	4315
DDRM	3077	2896	2100	1454	1906	2966

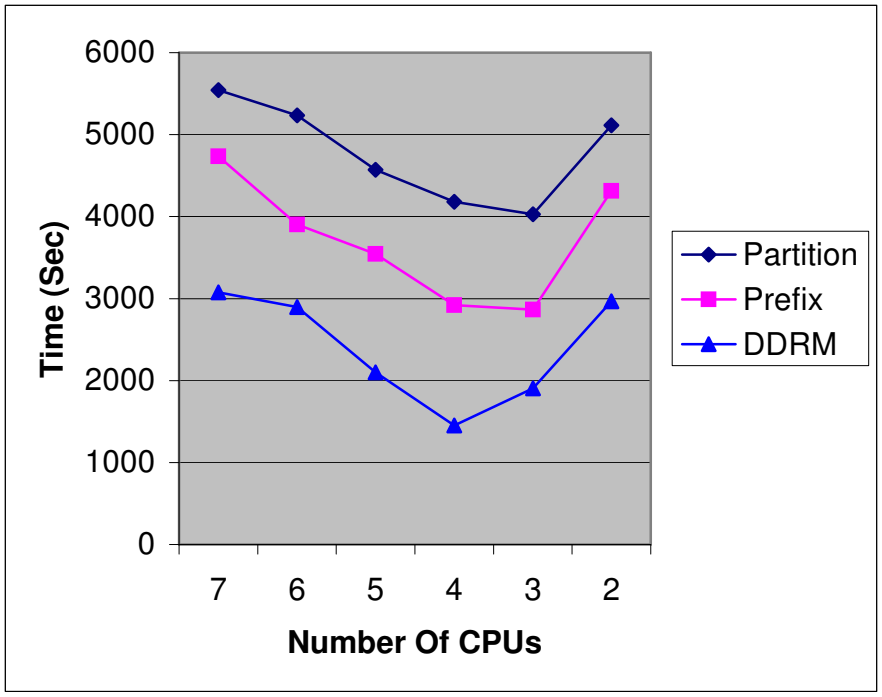


Figure 4.2.6 Scaleup for DDRM, Partition, and Prefix-based

Table 4.1.6 Databases

Database	DDRM	Prefix	Partition
	Seconds	Seconds	Seconds
DB_1	2525	2697	3092
DB_2	2472	3747	2882
DB_3	3020	3566	4269
DB_4	4242	4683	4645
DB_5	5097	5621	5076
DB_6	5249	5709	5722

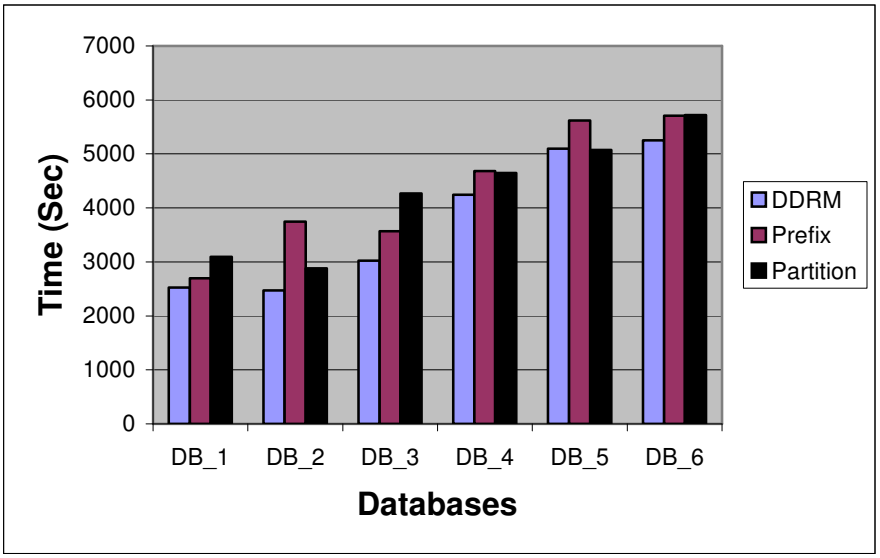


Figure 4.2.7 Number of Transactions

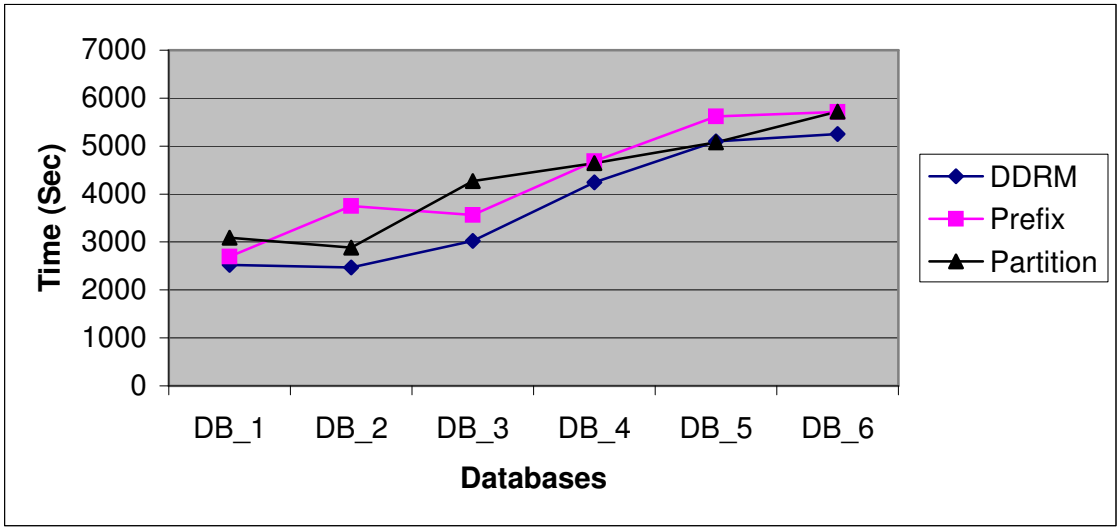


Figure 4.2.8 Number of Transactions

Table 4.1.7 Supports (Census)

Support	**10%**	**8%**	**6%**	**4%**
	Seconds	Seconds	Seconds	Seconds
Partition	2033	3136	5108	6488
Prefix	1468	2551	4337	6361
DDRM	742	2372	3142	5600

Table 4.1.8 Supports (KDD)

Support	**10%**	**8%**	**6%**	**4%**
	Seconds	Seconds	Seconds	Seconds
Partition	735	2082	3671	5223
Prefix	731	2116	3470	5385
DDRM	565	2100	2813	5173

Table 4.1.9 Supports (KDDWIDE)

Support	**10%**	**8%**	**6%**	**4%**
	Seconds	Seconds	Seconds	Seconds
Partition	4400	7275	8817	13218
Prefix	4303	6800	8995	13536
DDRM	2065	3577	4643	6828

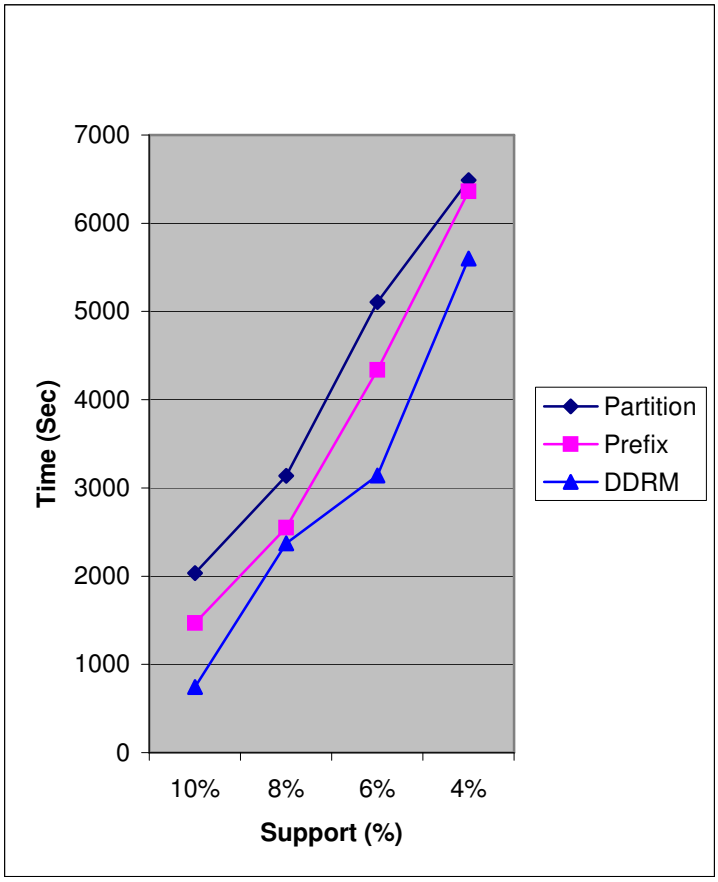


Figure 4.2.9 Support for Census

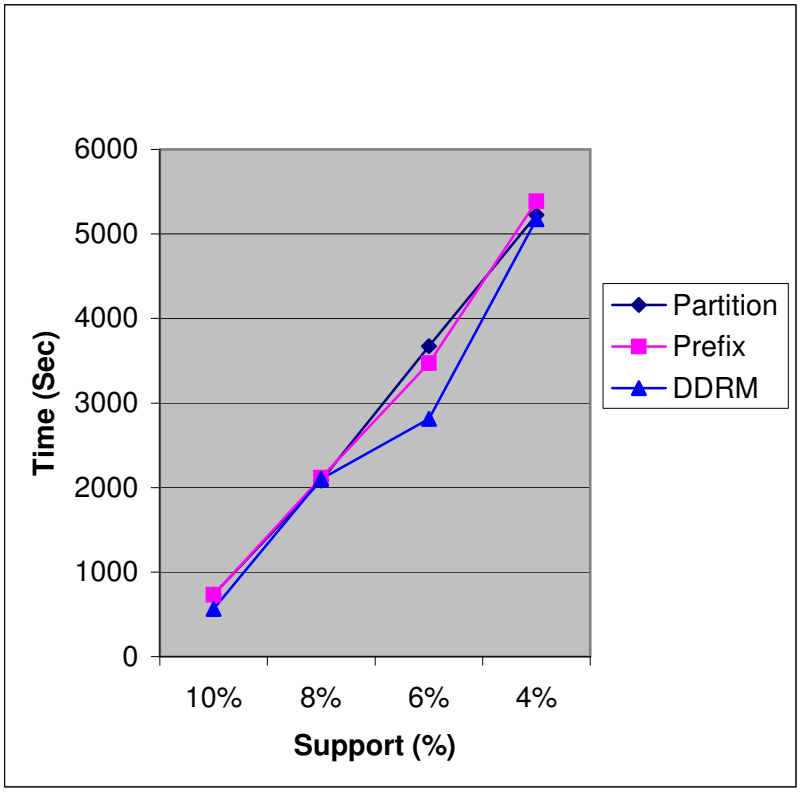


Figure 4.2.10 Support for KDD

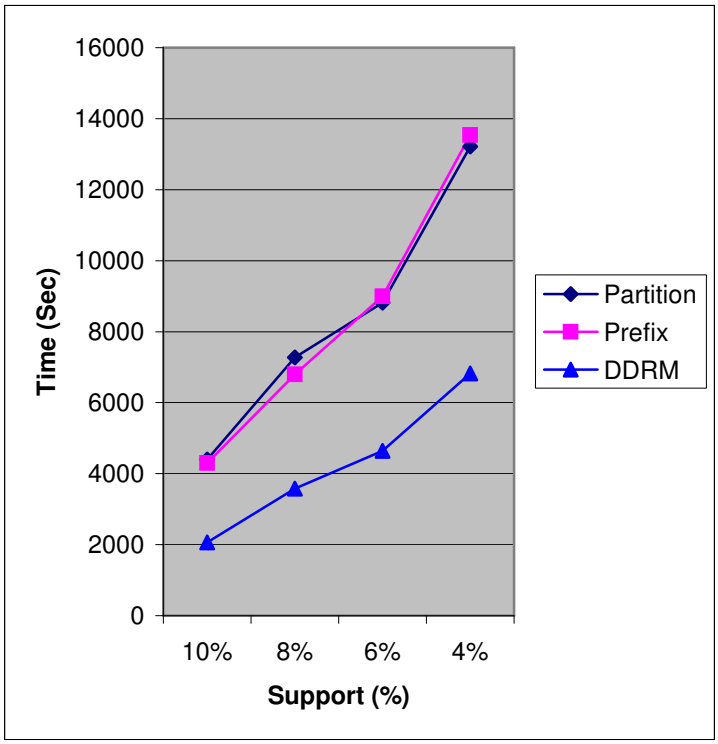


Figure 4.2.11 Support for KDDWIDE

Table 4.1.10 Transaction Width

Transactions	**KDD10**	**KDD20**	**KDD30**	**KDD40**	**KDD50**
	Seconds	Seconds	Seconds	Seconds	Seconds
Partition	33	1006	2916	7112	7109
Prefix	59	744	2944	7133	7133
DDRM	14	509	1685	3512	3477

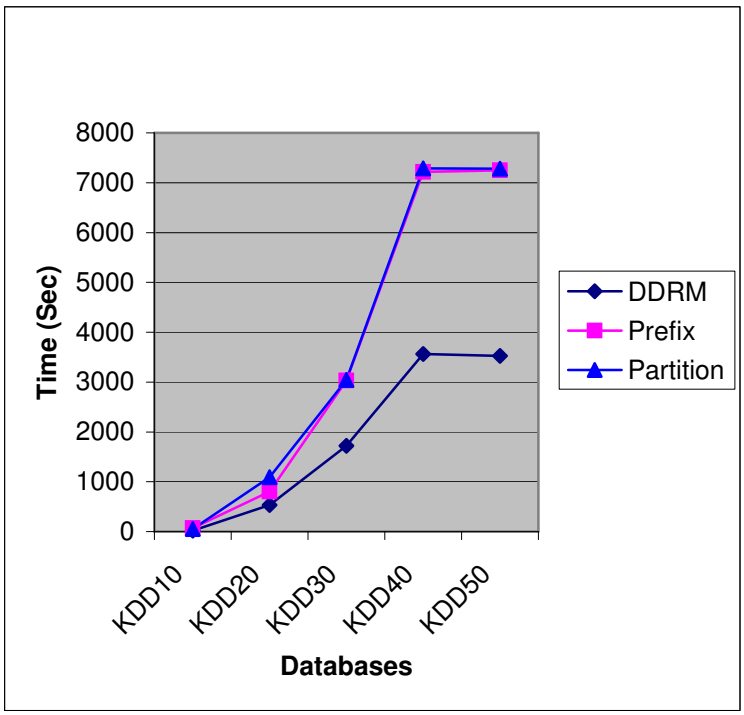


Figure 4.2.12 Transaction Width

Wait Time

We plot the wait time for all three algorithms in Figure 4.2.13 and Figure 4.2.14 for KDD20 and KDD50 respectively. The wait time associated with DDRM is shown to be less than the wait times for both the Prefix and Partition algorithms for the two datasets. Table 4.1.11 and Table 4.1.12 show the data obtained from the wait time experiments.

Communication Time

The communication time associated with the three algorithms is shown in Figure 4.2.15. The corresponding data for the result is shown in Table 4.1.13. The communication time was measured for the KDD10, KDD20, KDD30, KDD40 and KDD50 datasets. DDRM communication cost is less than that associated with Partition and Prefix algorithms.

Turnaround Time

In Figure 4.2.16 we plot the turnaround time for the three algorithms. It can be seen that the turnaround times for Prefix and Partition algorithms are greater than that for the DDRM algorithm. The turnaround time data is shown in Table 4.1.14.

CPU Cycles

Due to the dynamic assignment of classes the DDRM is better able to utilize the idle processors than the Prefix and partition algorithms. This can be seen form Figure 4.1.18 as the CPU cycles utilized by DDRM are greater than the cycles for Partition and Prefix algorithms. The corresponding data for the CPU cycles is shown in Table 4.1.16.

CPU Utilization

The utilization of the CPU for the three algorithms is shown in Figure 4.2.17. The data obtained from the experiment and used to plot the graph is shown in Table 4.1.15. This

experiment was conducted using six processors. An analysis of the graph shown in Figure 4.2.17 will show that the CPU utilization for DDRM is better than that shown for the Prefix and Partition algorithms.

CPU Usage

We used Task Manager to capture the CPU usage of each station for DDRM, Partition and the Prefix-based algorithms. We label the stations used as Station 1, Station 2 and Station 4. For the Prefix and Partition algorithms we captured two screens for each algorithm during execution. The screens were captured on each station in the cluster. In order to distinguish each screen we label each figure with the extension S1 for screen 1 and S2 for screen 2. The screens for the Prefix algorithm are shown in Figure 4.2.19, Figure 4.2.20, Figure 4.2.24, Figure 4.2.25 and Figure 4.2.29. In this experiment Station 2 was the first to finish the classes assigned to it and became idle after 13 of the 32 classes were processed. This situation is captured in the screen shown in Figure 4.2.24. Station 1 later completed all the tasks assigned to it and the screen capture is shown in Figure 4.2.20. At this time a total of 24 of the 32 classes have been processed with the remaining classes currently being processed by Station 4, where they were assigned at the start of the processing. At this time there are two idle processors available, however, Station 4 cannot share the classes currently assigned to it with any of these processors.

The screens for the Partition algorithm are shown in Figure 4.2.21, Figure 4.2.22, Figure 4.2.26, Figure 4.2.27, and Figure 4.2.30. Figure 4.2.26 shows that Station 2 was the first station to complete the classes assigned to it. This was followed by Station 1 as shown in Figure 4.2.22. At this point Station 4 is occupied with the classes assigned to it and is not able to share these with any of the idle stations.

Figure 4.2.23, Figure 4.2.28 and Figure 4.2.31 show the efficient utilization of the CPUs by DDRM. It can be seen from these three screen shots that Station 1, Station 2, and Station 3 are all kept busy for the duration of the computations and is able to avoid having idle processors that cannot be assigned available tasks, as is the case with the Prefix and Partition algorithms.

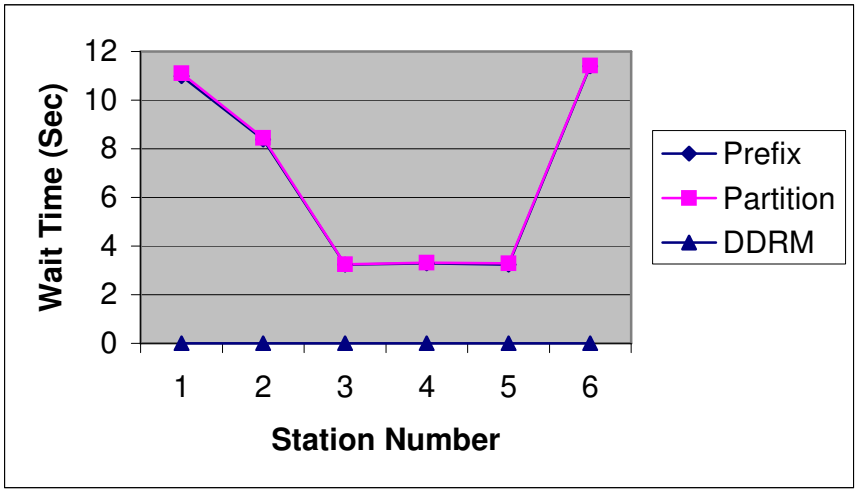


Figure 4.2.13 Wait Time for KDD20

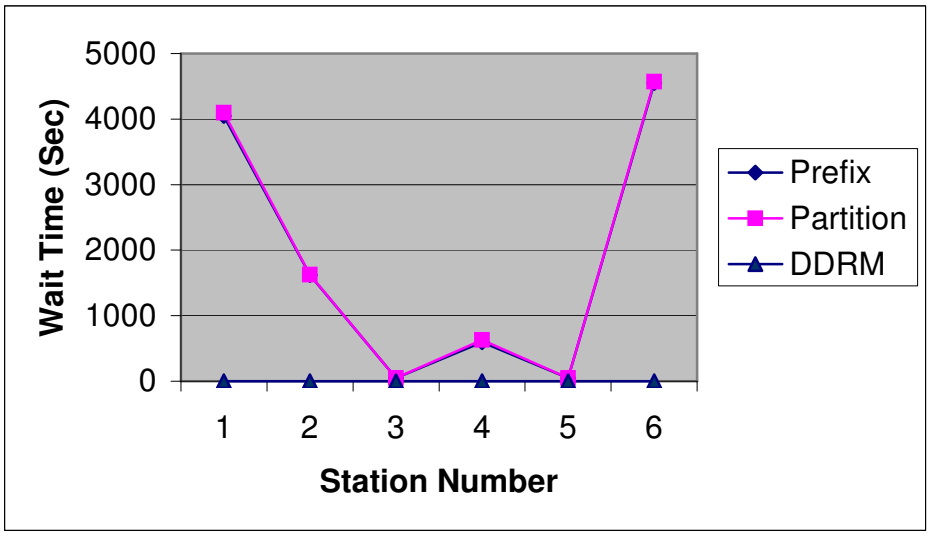


Figure 4.2.14 Wait Time for KDD50

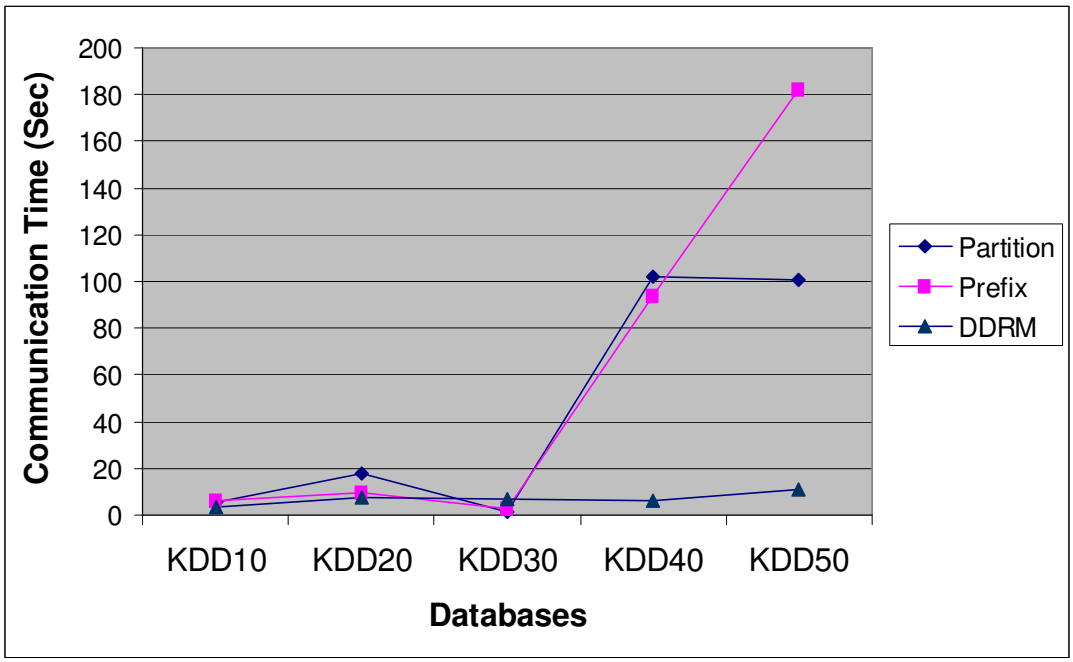


Figure 4.2.15 Communication Time

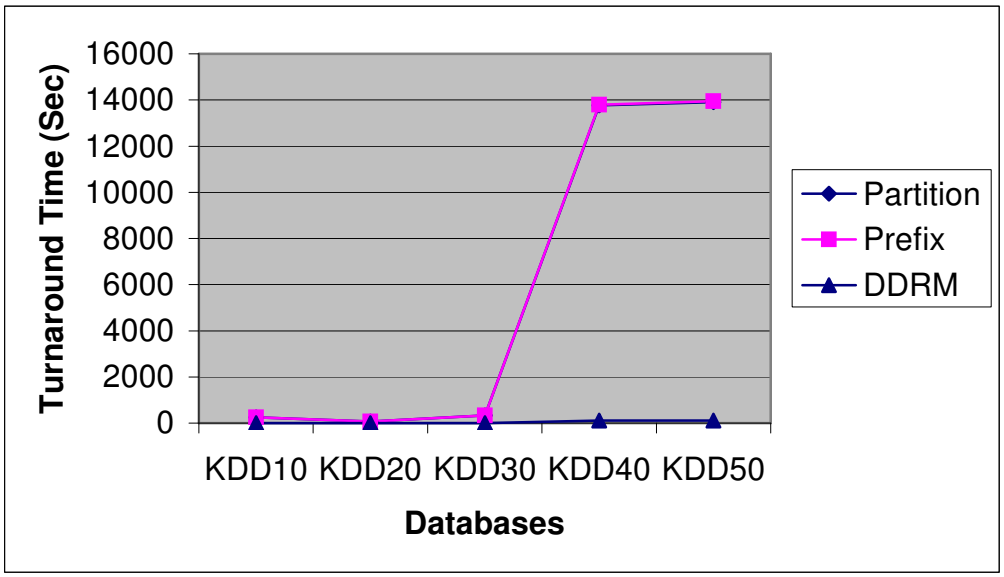


Figure 4.2.16 Turnaround Time

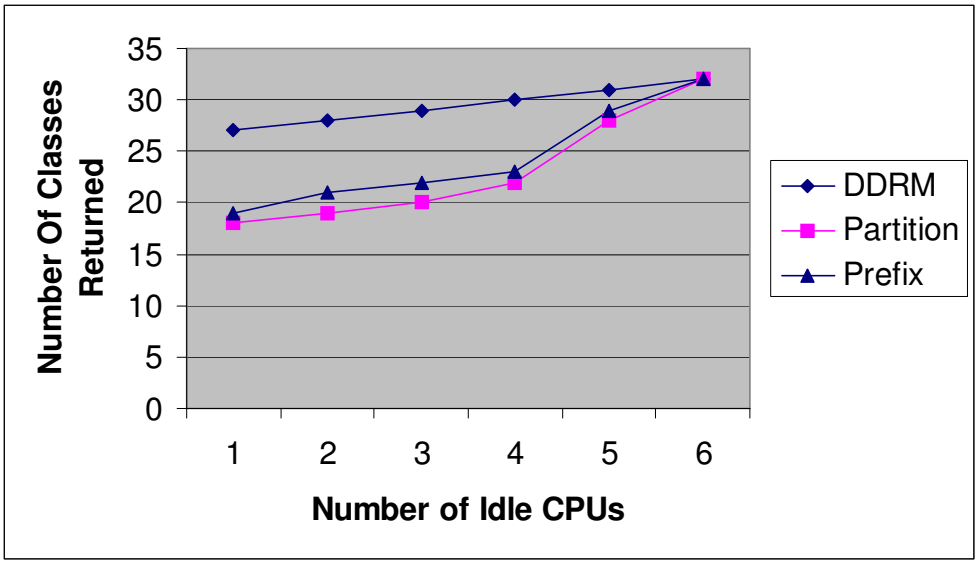


Figure 4.2.17 CPU Utilization

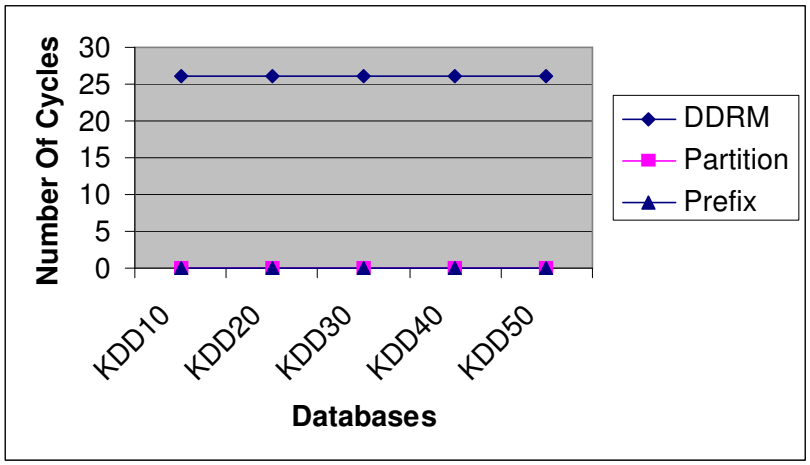


Figure 4.2.18 CPU Cycles

Table 4.1.11 Wait Time KDD20

Station	Prefix	Partition	DDRM
	Seconds	Seconds	Seconds
1	10.98	11.11	0
2	8.37	8.45	0
3	3.23	3.25	0
4	3.29	3.31	0
5	3.23	3.28	0
6	11.39	11.42	0

Table 4.1.12 Wait Time KDD50

Station	Prefix	Partition	DDRM
	Seconds	Seconds	Seconds
1	4048	4094	0
2	1619	1623	0
3	50.	50	0
4	597	631	0
5	48	50	0
6	4549	4569	0

Table 4.1.13 Communication Time

Databases	**Partition**	**Prefix**	**DDRM**
	Seconds	Seconds	Seconds
KDD10	5.3	5.9	3.5
KDD20	17.7	9.8	7.4
KDD30	1.6	2.8	7.1
KDD40	102.1	93.8	6.0
KDD50	100.3	182.3	11.1

Table 4.1.14 Turnaround Time

Databases	**Partition**	**Prefix**	**DDRM**
	Seconds	Seconds	Seconds
KDD10	251	251	1
KDD20	65	64	1
KDD30	338	338	4
KDD40	13771	13789	105
KDD50	13903	13954	106

Table 4.1.15 CPU Utilization (KDD10)

	DDRM	Partition	Prefix
Number of Idle CPUS	**Number of Classes Returned**	**Number of Classes Returned**	**Number of Classes Returned**
1	27	18	19
2	28	19	21
3	29	20	22
4	30	22	23
5	31	28	29
6	32	32	32
7	0	0	0

Table 4.1.16 CPU Cycles

Databases	DDRM	Prefix	Partition
KDD10	26	0	0
KDD20	26	0	0
KDD30	26	0	0
KDD40	26	0	0
KDD50	26	0	0

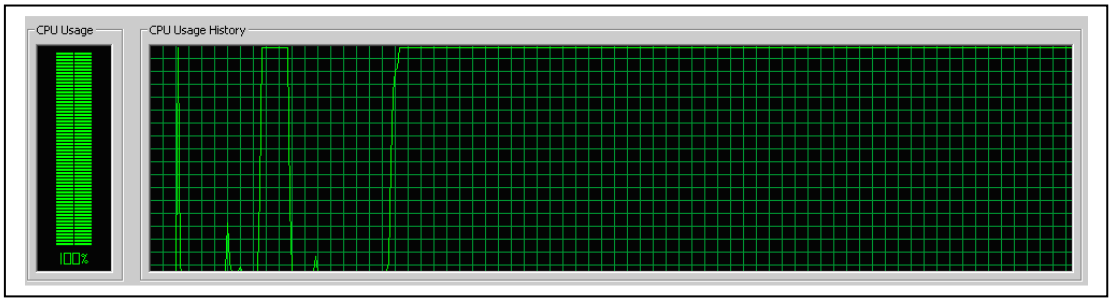


Figure 4.2.19 CPU Utilization by Prefix _S1 (Station 1)

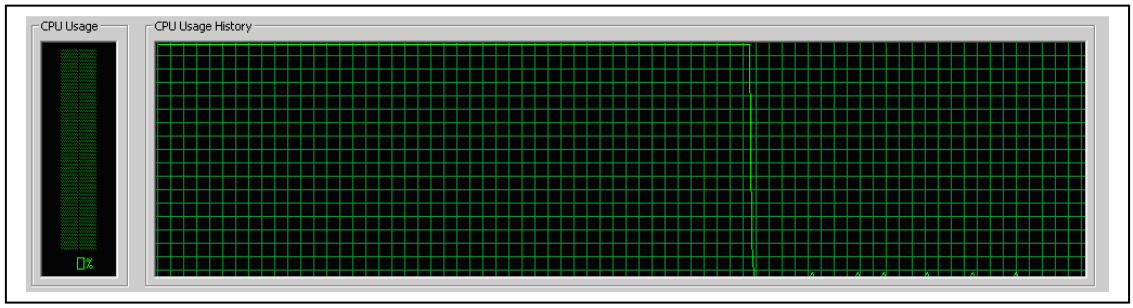


Figure 4.2.20 CPU Utilization by Prefix_S2 (Station 1)

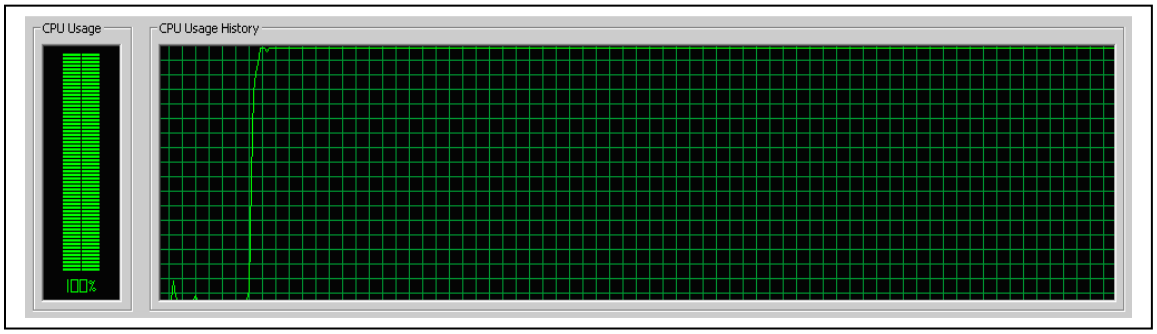


Figure 4.2.21 CPU Utilization by Partition_S1 (Station 1)

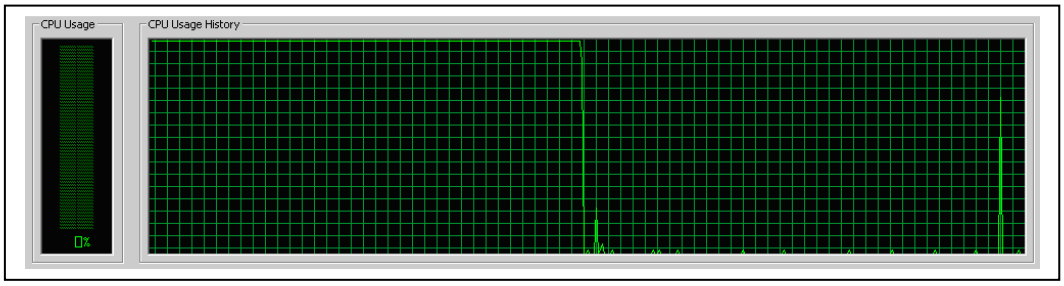


Figure 4.2.22 CPU Utilization by Partition_S2 (Station 1)

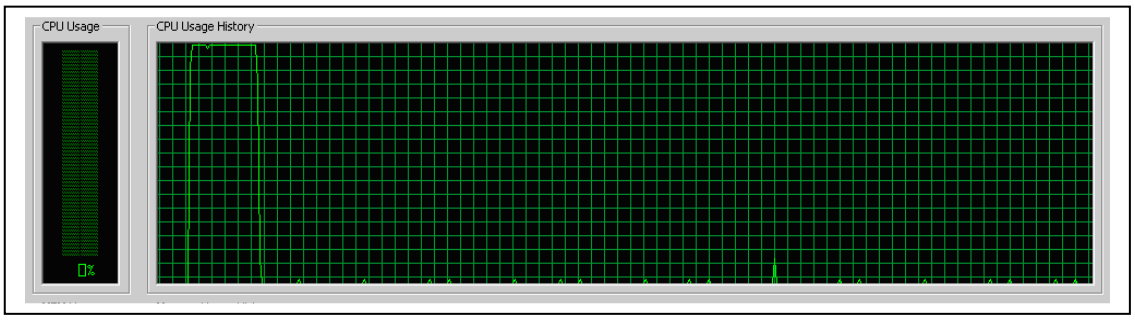


Figure 4.2.23 CPU Utilization by DDRM (Station 1)

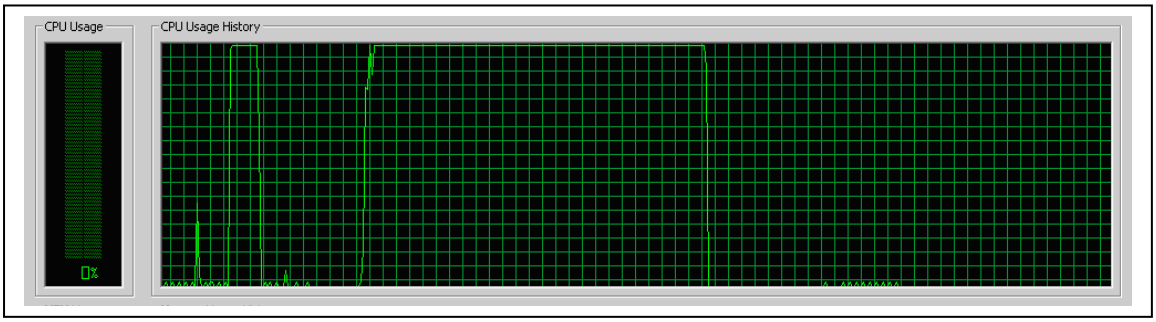


Figure 4.2.24 CPU Utilization by Prefix_S1 (Station 2)

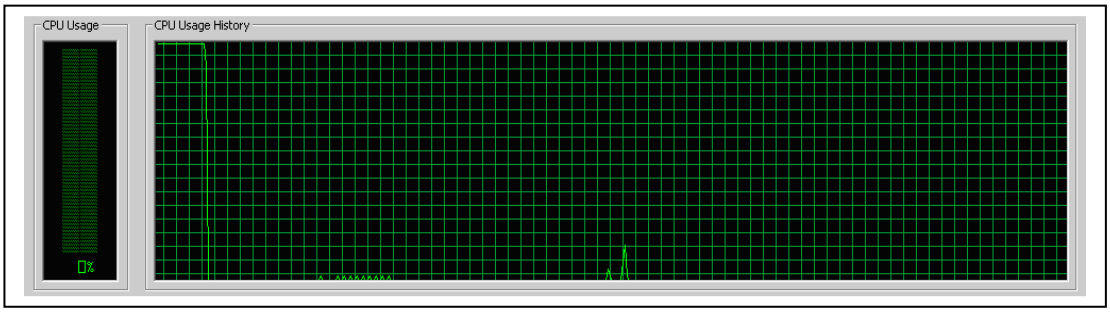


Figure 4.2.25 CPU Utilization by Prefix_S2 (Station 2)

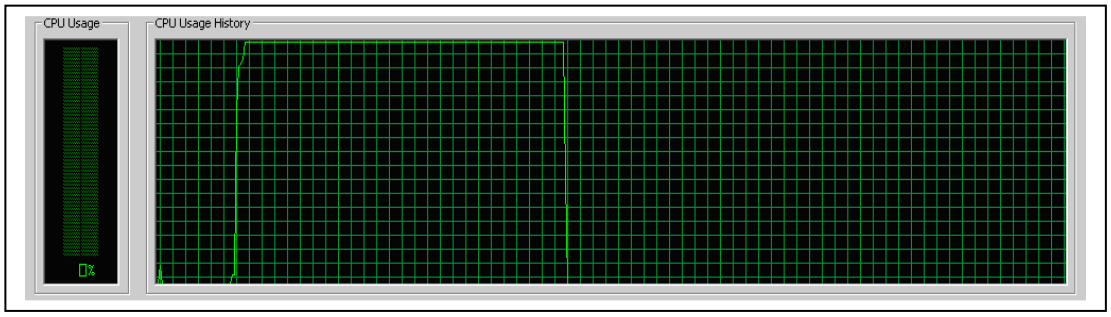


Figure 4.2.26 CPU Utilization by Partition_S1 (Station 2)

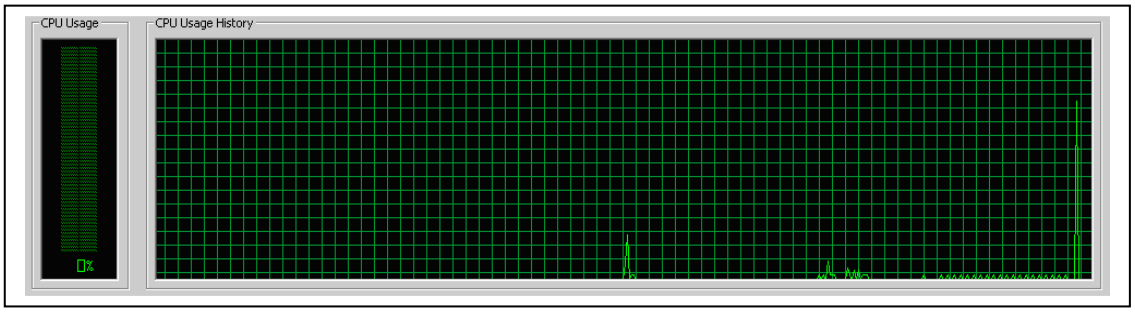


Figure 4.2.27 CPU Utilization by Partition_S2 (Station 2)

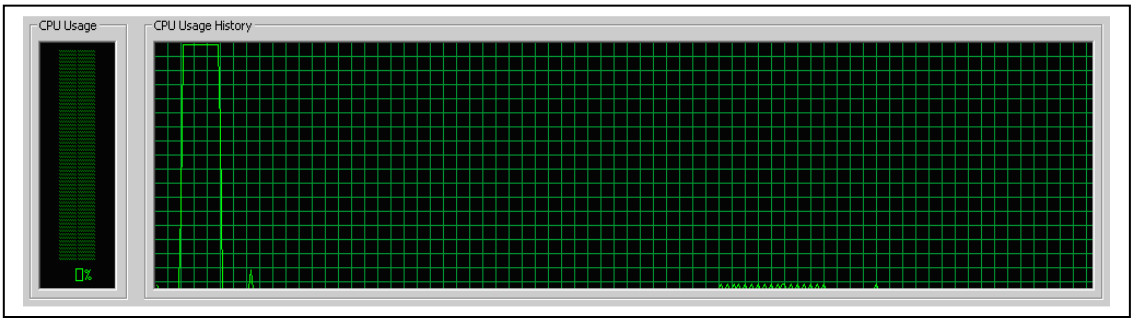


Figure 4.2.28 CPU Utilization by DDRM (Station 2)

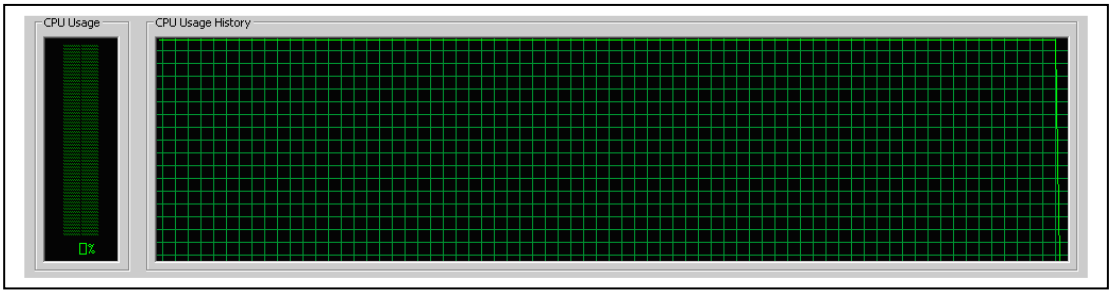


Figure 4.2.29 CPU Utilization by Prefix (Station 4)

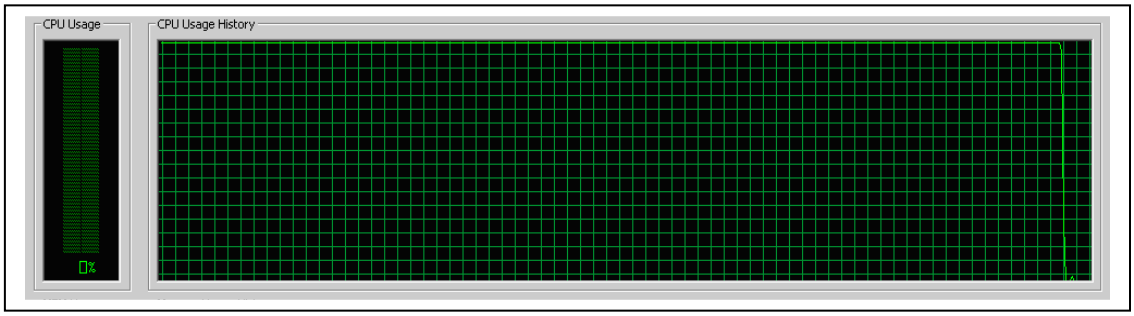


Figure 4.2.30 CPU Utilization by Partition (Station 4)

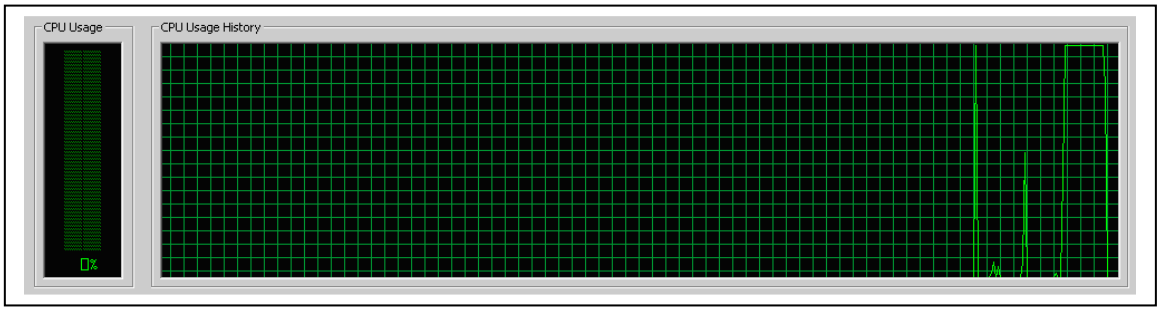


Figure 4.2.31 CPU Utilization by DDRM (Station 4)

4.5 Comparison of DDRM and Prefix-Based Algorithms

The DDRM algorithm executes faster than the prefix-based algorithm. This is due to the fact that DDRM is able to optimise the use of the available CPUs. The ability to keep all CPUs busy as long as there is work to be done is an improvement of DDRM over the Prefix-based algorithm. The throughput associated with DDRM is high since it is able to keep track of all idle processors in the cluster so that the available classes to be processed can be assigned to these idle processors.

The use of memory by DDRM is based on the principle of sharing the available work among the processors using the finest granularity possible. The finest granularity is based on the class. By storing and assigning one class at a time the memory utilization of all the available memory on the processors is an improvement over the Prefix-based approach.

In DDRM available tasks are assigned to processors as soon as they are available. Once a processor has completed its assigned task a new task is taken from the task heap and assigned to this processor. This approach guarantees that no processor will be idle while there are additional classes available for processing. In this approach classes are only assigned one at a time and a class is only assigned to an idle processor.

A major challenge in parallel processing is to balance the distribution of work across processors. It is challenging for a static scheduling algorithm to produce good load balancing, especially in an environment where there is no prior knowledge of the execution characteristics of the data. The best approach to load balancing is to use a dynamic scheduling algorithm. This is an efficient approach when the time needed to send and receive each class is low relative to the processing time. DDRM is able to balance the load using a dynamic scheduling algorithm that assigns the next task to the

first available processor. In this approach there will never be an idle processor while there is additional work that is not being processed.

The wait time associated with the Prefix is large when compared to the wait time for DDRM. This is due to the efficient utilization of the processors by DDRM. Classes are assigned dynamically to stations as they become idle.

In the DDRM algorithm the turnaround time is small when compared to the Prefix and Partition algorithms. Since classes assigned to processors in these algorithms must wait to be processed by the station assigned to it there is an increase in the turnaround time due to the long wait at each station.

CPU utilization by DDRM is better than that obtained for the Prefix and Partition algorithms. Since idle CPUs cannot be utilized by the Prefix and Partition algorithms, they are likely to suffer from processors not being fully utilized.

DDRM algorithm is able to use CPU cycles as they become available in the cluster. The Prefix and Partition algorithms cannot use these CPU cycles associated with the available idle processors.

Load and Task Balancing

DDRM balances the load across the stations in the cluster better than the Prefix and Partition algorithms. The DDRM algorithm is able to assign classes to processors as soon as they become idle. In this approach the classes are equally distributed across all the stations in the cluster. The Prefix and Partition algorithms suffer from poor load balancing, since they are incapable of reassigning classes from busy stations to stations that are idle.

All stations performed the same tasks on independent data sets, therefore no task balancing was conducted in this research.

4.6 Summary

In this chapter we presented and discussed the results of our experiments. We implemented the DDRM algorithm that used a lattice theoretic approach to partition the frequent itemset search space into independent search spaces. We found that the DDRM algorithm showed good speedup and the response time was significantly improved with the addition of each processor while keeping the work to be done fixed. The algorithm also shows improvement in the response time, wait time, turnaround time, CPU utilization and cycle time, when compared to the Prefix-based and Partition, which are static scheduling algorithms.

Chapter 5

Conclusions, Implications, Recommendations, and Summary

5.1 Conclusions

The primary goal of this research and dissertation was to develop and implement a parallel algorithm for the mining of association rules. The Dynamic Distributed Rule Mining (DDRM) algorithm used a lattice to represent the search space for the generation of the frequent itemsets. The algorithm was implemented using C/C++ as the programming language. The DDRM algorithm was implemented on an Ethernet LAN consisting of 7 workstations and one server. For communications the message passing interface (MPI) was used. The windows message passing interface (WMPI) for 8 workstations from Critical Software Ltd was used as the MPI interface.

The improvements made by the algorithm are as follows:

1. Improved load balancing: The classes generated by the DDRM algorithm are assigned dynamically to the processors as they become available. This approach was found to be more efficient than the static approach.
2. No synchronization: DDRM used a lattice theoretic approach, which partitioned the itemsets into sublattices that were assigned to each processor to be processedindependently. Processors only communicate with the controller to collect classes for processing and to return any frequent itemsets found in the assigned classes.

3. Reduction in communications among processors: There was a significant reduction in the communication cost associated with the processing of each class. This is due to the fact that the only communications cost incurred for each class is the assignment of the class and the subsequent return of the results for the class.
4. Better CPU utilization: There was a significant improvement in the CPU utilization by DDRM when compared to the Prefix and Partition algorithms.
5. Improved wait time and turnaround time: The wait time and turnaround time obtained for the DDRM algorithm showed improvement over the Partition and Prefix algorithms.

5.2 Implications

Business organizations have recognized the importance of information-driven marketing processes and the competitive advantages that they offer. These processes allow marketers to develop and implement customized marketing programs and strategies. These organizations are turning to data mining technology to facilitate the process of extracting valuable information from large databases. The extraction of previously unknown information from large databases can be used to generate new marketing strategies. Because of the very large size of the databases used to store the transactions used in the mining of association rules, parallel algorithms are required to process these transactions. The DDRM algorithm used a lattice to represent the search space for the generation of frequent itemsets. The DDRM algorithm is important for the following reasons:

1. It provides improved communications among the processors
2. It reduces the execution time for the classes

3. It reduces the cost associated with mining association rules
4. It is scalable.
5. It provides increased processor efficiency
6. It utilizes memory well
7. Better CPU utilization
8. Improved turnaround time and wait time.

5.3 Recommendations

In this dissertation, we presented the theory, description, inference and implementation of the Dynamic Distributed Rule Mining (DDRM) algorithm that is based on a lattice theoretic approach.

The approach used in this research can be used in organizations with multiple sites where the databases are stored. This is an attractive approach to these organizations since it eliminates the need to have all these databases in one location. The cost savings associated with this approach will make it attractive to these organizations. In addition processors at different sites can participate in the computations.

The DDRM algorithm does not require any special architecture for its implementation. It is designed to operate on an existing LAN as a cluster where the PCs can be added to the cluster and used to participate in the computations of the classes. The database of transactions can also be distributed over the network. This flexibility of the algorithm will result in significant savings to the organization as it uses the resources that are already available within the organization. This reduction in cost is due to the fact that there is no need for specialized architecture and makes the algorithm attractive to an organization that currently operates a network with databases distributed over it.

DDRM uses a dynamic load balancing approach to assign classes to the processors. Since classes are assigned to processors only after they become available, the algorithm avoids and completely eliminates the possibility of assigning more than one class to a processor while there are idle processors. This approach contributes to the improvement of DDRM as compared to the Prefix-based approach in the generation of the rules. In all cases DDRM is able to improve on the computation time associated with the Prefix-based and Partition algorithms.

The high processor utilization of DDRM impacts positively on throughput and response time. DDRM improved on the throughput and response time due to the dynamic assignment of classes to the processors. The processor efficiency of DDRM is an improvement over the Prefix-based and Partition algorithms due to the reduction in the number of idle processors.

In the lattice theoretic approach the rules will be generated faster since the processors can do the computations independently of each other. In the DDRM all idle processors will be fully utilized during the computation of rules resulting in a faster time to generate the rules. In a static approach if it is discovered early that there are no frequent itemsets in the classes assigned to a processor these classes will not be processed any further. The processor will now be idle, but there is no mechanism in place to move some of the classes from the busy processors to the idle processor. The ability of DDRM to dynamically assign classes to idle processors makes it more efficient than the static algorithms.

Areas or topics for further research include:

1. Mechanism for interfacing with a database management system

2. Automation of the assignment of codes to attributes used
3. Impact of the interconnection network topology

5.4 Summary

In this paper, we highlighted the need for parallel solutions to data mining problems. Parallel algorithms are required for the mining of association rules to improve on the execution time. We also presented our goal to develop and implement a parallel algorithm for the mining of association rules using a lattice theoretic approach and utilizing dynamic scheduling for the assignment of tasks. In Chapter 2 we reviewed data mining, with emphasis on mining of association rules. We also presented several approaches to the mining of association rules based on parallel architectures. We also presented a discussion on lattice theory. In Chapter 3 we presented a detailed investigation of the principles of data mining, parallel data mining, and lattice theory. We also presented some examples to demonstrate these principles.

In addition we also proposed the Dynamic Distributed Rule Mining (DDRM) algorithm, which is a parallel algorithm for data mining that is based on lattice theory and uses dynamic scheduling to assign tasks to the processors. A detailed description of DDRM and how it works was also presented.

In Chapter 4, we presented and discussed the results of our experiments. We found that the DDRM algorithm showed good speedup and the execution time was significantly improved with the addition of each processor while keeping the work to be done fixed. The algorithm also showed improvement in the execution time when compared to the Prefix-based, static scheduling algorithm. In Chapter 5, we presented our conclusions and further research directions.

Appendixes

The following appendixes contain listings of the source code for the functions used in the DDRM and Prefix based algorithms.

Appendix A

Data Structures Used in Implementation

```
//Definition of Structures

typedef std::set< int > item_set;
typedef std::vector<item_set>setItemsetVect;
typedef std::vector<setItemsetVect>classVect;//stores the cset of classes for all h-casses

#include <algorithm>
struct itint
{
 bool operator()(const int n1, const int n2)const
 {return(n1 < n2);}
};
typedef std::vector< int > trans_attribs; //stores list of attributse satisfying min support;
std::ostream_iterator< int > output( cout, " " );

//Defines structure to store interestingness measure as percentage of transactions
typedef struct
{
    int support;
    int confidence;
} intrstMeas;

typedef struct
{
    setItemsetVect vectF1;     // tid lists
    trans_attribs  F1_items;   // frequent itemsets
} info_for_F1;

//This vector stores the information on the set of frequent items used to generate rules
typedef struct
{
    setItemsetVect vectItemSets; //frequent itemsets
    setItemsetVect vectTIDs;       //tid lists for each frequent itemset
} freq_items;
```

Appendix B

Function to Create Set of N-Itemsets

```
setItemsetVect setOfNCas(item_set theSet, int k_count)
{
      std::ostream_iterator< int > output( cout, " " );
      item_set subKCas;
      setItemsetVect setOfKItemSets;
      item_set::iterator i, j, start, k;
      int sizeOfSet;
      int count = 0;
      int numSubMemb = k_count;
      int subItemSize ;

      sizeOfSet = theSet.size();

      for(i = theSet.begin(); i != theSet.end();  i++)
        {
            item_set stemSet;
       for(j = i, count = 1; count < k_count; count++,j++)
            stemSet.insert(*j);
       for(start = i, count = 1; count < k_count; start++)
        count++;
            for(k = start; k != theSet.end();  k++)
          {
            item_set iSet = CreateStem(stemSet);
            iSet.insert(*k);
            setOfKItemSets.push_back(iSet);
          }
        }
      return setOfKItemSets;
}
```

Appendix C

DDRM Partition Function

```
// Function to compute the set of prefix classes for the given thetaval
classVect ddrmPartition(item_set freqAttrib,   int numPartitions)
   {
        classVect allHVectors; //Stores a vector of vectors with all ha cass
        setItemsetVect setOfNcas;  //stores the set of itemsets that can be formed from
                                   //prefix  K-itemset
        item_set::iterator start,  k;
       item_set theSet;
       int hValue;
       int count = 0,
           k_count;
       int pause;
      allHVectors.clear();
      k_count = Get_KPower_Of2(numPartitions);
      if(k_count > freqAttrib.size())
        {
           printf("\nThe size of the frequent attributes = %d\n",freqAttrib.size());
           printf("\Hit any key to continue .....\n");
           scanf("%d", &pause);
           return allHVectors;
        }

    for(hValue = k_count-1; hValue >= 0; hValue--)
     {
        setOfNcas = setOfNCas(freqAttrib, hValue, k_count);
        allHVectors.push_back(setOfNcas);
     }

    return allHVectors;
  }
```

Appendix D

Generate All Classes Function

```
//Function to generate all classes and save these in a class vector
classVect Generate_All_Classes(classVect allHVectors, classVect allClassVecs,
item_set freqAttrib, int kcount)
  {
   int hVectSize;
   int index = 0,nextS;
   item_set tempSet;
   classVect twoClasses;
   setItemsetVect vecSet;
   setItemsetVect tempVec1,tempVec2;
   hVectSize = (int)allHVectors.size();
   for(index = 0; index < hVectSize; index++)
    {
     vecSet = allHVectors[index];
      nextS = (int)vecSet.size();
     if((int)vecSet.size() > 0)
       for(nextS = 0; nextS < (int)vecSet.size(); nextS++)
         {
           tempSet = vecSet[nextS];
           twoClasses = Gen_Two_Classes(tempSet, freqAttrib , kcount);
           tempVec1 = twoClasses[0];
           tempVec2 = twoClasses[1];
           allClassVecs.push_back(tempVec1);
           allClassVecs.push_back(tempVec2);
           tempSet.clear();
         }
      else
       {
         twoClasses = Gen_Two_Classes(tempSet, freqAttrib , kcount);
         tempVec1 = twoClasses[0];
         tempVec2 = twoClasses[1];
         allClassVecs.push_back(tempVec1);
         allClassVecs.push_back(tempVec2);
         tempSet.clear();
       }
    }
  return allClassVecs;
 }
```

Appendix E

Generate Two Classes Function

```
//Function to generate atoms for each class
//This function generates the two atoms from pre(k-1) which are the singleton
(shorter)
//and union (longer 1). These are stored in the vector tempVec1 which is returned to
calling
//program

classVect Gen_Two_Classes(item_set set, item_set attribs, int kVal)
  {
    item_set iSet,tempSet,stem1,stem2;
    item_set::iterator iter,iter2;
    setItemsetVect tempVec1,tempVec2;
    classVect resultVec;
    int index;
    if(set.size() > 0)
      {
        stem2 =  CopySet(set);
        stem1  =  CopySet(set);
      }
    for(iter = attribs.begin(), index = 0; index < kVal-1; iter++)
      index++;
   stem1.insert(*iter);
   iter++;
  for(iter2 = iter; iter2 != attribs.end(); iter2++)
    {
   item_set  tempSet =  CopySet(stem1);
      item_set iSet  =  CopySet(stem2);
   tempSet.insert(*iter2);
   iSet.insert(*iter2);
      tempVec2.push_back(iSet);
      tempVec1.push_back(tempSet);
   }
 resultVec.push_back(tempVec1);
 resultVec.push_back(tempVec2);
 return resultVec; // return the two atoms for the class
}
```

Appendix F

Broadcast TID Vector Function (1 of 2)

```
// Function to broadcast a setTidsetVect to all processes
//Variable length lists
void Bcast_TidsetVect2(setItemsetVect theTidVect, int procNum,int sArray[])
{
    int index = 0,
    index1 = 0,
    size = 0,
    count,
    nElem,
    arrSize;
    int totAtribs = 0,
    setSize,
    numTidSets;
    int pause;
    item_set::iterator i, enditer;
    sendSizeAtrb[2];
    item_set tempSet;
    numTidSets = (int)theTidVect.size(); //Number of itemsets in vector
    sendSizeAtrb[0] = 0;
    sendSizeAtrb[1] = 0;
    if(numTidSets == 0)
     {
      printf("\n** The tid vector is empty.....\n");
      return;
     }
   sendSizeAtrb[1] = numTidSets;  //Number of itemsets in the vector
   MPI_Bcast(sendSizeAtrb,
                2,
                MPI_INT,
                procNum,
                MPI_COMM_WORLD);
```

Appendix F

Broadcast TID Vector Function (2 of 2)

```
 for( index1 = 0; index1 < numTidSets; index1++)
  {
   tempSet = theTidVect[index1];
   totAtribs = 0;
   setSize = (int)tempSet.size();
   nElem = setSize;
   count = nElem/100000;
   arrSize = 100000;
   sendSizeAtrb[0] = setSize;
   sendSizeAtrb[1] = numTidSets;  //Number of itemsets in the vector
   MPI_Bcast(sendSizeAtrb,
             2,
             MPI_INT,
             procNum,
             MPI_COMM_WORLD);
  arrSize = 100000;
  for(index = 0; index < count; index++)
    {
     Convert_Itemset_To_Arr2(tempSet, sArray, index);
     MPI_Bcast(sArray,
       arrSize,
       MPI_INT,
       procNum,
       MPI_COMM_WORLD);
    }
  count = nElem % 100000;
  if(count > 0)
    {
     Convert_Itemset_To_Arr2(tempSet, sArray, -1);
     MPI_Bcast(sArray,
       count,
       MPI_INT,
       procNum,
       MPI_COMM_WORLD);
    }
  }
}
```

Appendix G

Receive Broadcast of TID Vector Function (1 of 2)

```
// Function to Receive broadcast of a  setTidsetVect by  processe 0
//Variable length lists
setItemsetVect Rcv_Bcast_TidsetVect2(int procNum,int rArray[])
{
   #define TAG           100
   #define TRACE 0
   #define TRIANGARRASIZE 1953
   int index = 0,
   index1 = 0,
   count,
   nElem,
   size = 0;
   int j;
   int totTids = 0,
   arrSize = 0,
   setSize,
   numSets;
   int val;
   int   pause;
   setItemsetVect theSetVect;
   item_set::iterator i, enditer;
   int recvSizeAtrb[2],  //0 size (Value of -1 indicates end of list from proc , 1 atribute
number
    sendSizeAtrb[2];
   recvSizeAtrb[0] = 0;
   recvSizeAtrb[1] = 0;
   theSetVect.clear();
   MPI_Bcast(recvSizeAtrb,
             2,
             MPI_INT,
             procNum,
             MPI_COMM_WORLD);

   numSets = recvSizeAtrb[1];
   totTids = recvSizeAtrb[0];
```

Appendix G

Receive Broadcast of TID Vector Function (2 of 2)

```
    for(index1 = 0; index1 < numSets; index1++)
     {
      MPI_Bcast(recvSizeAtrb, 2, MPI_INT, procNum,
                MPI_COMM_WORLD);
     setSize = recvSizeAtrb[0];
     nElem = setSize;
     count = nElem/100000;
     arrSize = 100000;
     item_set tempSet;
     for(index = 0; index < count; index++)
       {
         MPI_Bcast(rArray,
                   arrSize,
                   MPI_INT,
                   procNum,
                   MPI_COMM_WORLD);
        for(j = 0; j < arrSize; j++)
          {
            val = rArray[j];
            tempSet.insert(val);
          }
       }
     count = nElem % 100000;
     if(count > 0)
      {
        arrSize = count;
        MPI_Bcast(rArray,
                  arrSize,
                  MPI_INT,
                  procNum,
                  MPI_COMM_WORLD);
       for(j = 0; j < arrSize; j++)
        {
          val = rArray[j];
          tempSet.insert(val);
        }
      }
    theSetVect.push_back(tempSet);
   }
  return theSetVect;
}
```

Appendix H

Send Class Function (1 of 2)

```
// Function to send a class to a process
void Send_class(setItemsetVect theClass, int procInfo[],int sArray[])
{

  int index = 0,
     size = 0;
  int procNum;
  int totAtoms = 0,
     atomSize;
  item_set::iterator i, enditer;
  int sendSizeAtrb[3];  //0 size (Value of -1 indicates end of list from proc ,
                  // 1 atribute number, 2 class Number
  item_set tempSet;
  size = (int)theClass.size(); //Number of atoms in class
  procNum = procInfo[0];
  sendSizeAtrb[0] = 0;
  sendSizeAtrb[1] = 0;
  sendSizeAtrb[2] = 0;

/*********************************************************
 *** check to see if this is to signal end of class *******
 *** transmission                              *******
 *********************************************************
 */
  if(size == 0)
    {
     sendSizeAtrb[0] = -1; //Number of elements in each atom
     sendSizeAtrb[1] = -1;  //Number of atoms in the class
     sendSizeAtrb[2] = -1; // the class number
     MPI_Send(sendSizeAtrb,
              3,
              MPI_INT,
              procNum,
              TAG,
              MPI_COMM_WORLD);
    return;
   }
```

Appendix H

Send Class Function (2 of 2)

```
   for( index = 0; index < size; index++)
   {
    tempSet = theClass[index];
    atomSize = (int)tempSet.size();
    for(i = tempSet.begin(); i != tempSet.end(); i++)
     {
      sArray[totAtoms] = *i;
      totAtoms++;
     }
   }
  sendSizeAtrb[0] = atomSize; //Number of elements in each atom
  sendSizeAtrb[1] = size;  //Number of atoms in the class
  sendSizeAtrb[2] = procInfo[1]; // the class number
  MPI_Send(sendSizeAtrb,
                3,
                MPI_INT,
                procNum,
                TAG,
                MPI_COMM_WORLD);

 MPI_Send(sArray,
        totAtoms,
        MPI_INT,
        procNum,
        TAG,
        MPI_COMM_WORLD);
}
```

Appendix I

Receive Class Function (1 of 2)

```
// Function to receive a class to a process
//setItemsetVect Recv_class(int procInfo[])
setItemsetVect Recv_class(int procInfo[],int rArray[])
{
  int atomSize; //Number of attributes in each atom
  int totalAtoms; //Total atoms in class
  MPI_Status status;
  #define TAG           100
  #define TRIANGARRASIZE 1953
  int index = 0,
     index2 = 0,
     atrib= 0,
     size = 0;
  int rnElem = 0;
  int procNum;
  setItemsetVect theClass;
  int recvSizeAtrb[3],  //0 size (Value of -1 indicates end of list from proc,
                  // 1 atribute number, 3 class Number
     sendSizeAtrb[2];
  recvSizeAtrb[0] = 0;
  recvSizeAtrb[1] = 0;
  theClass.clear();
   /* Receive a message from a process:              */
  procNum = procInfo[0]; // proc num
  MPI_Recv(recvSizeAtrb,
        3,
        MPI_INT,
        procNum,
        TAG,
        MPI_COMM_WORLD,
         &status);
```

Appendix I

Receive Class Function (2 of 2)

```
/*************************************************************
 *** check to see if this is to signal end of class *******
 *** transmission                                  *******
 *************************************************************
 */
  if(recvSizeAtrb[0] == -1)
    return theClass; // return the empty class
  atomSize = recvSizeAtrb[0];
  totalAtoms = recvSizeAtrb[1];
  procInfo[1] = recvSizeAtrb[2];
  rnElem = atomSize * totalAtoms;

 MPI_Recv(rArray,
          rnElem,
          MPI_INT,
          0,
          TAG,
          MPI_COMM_WORLD,
          &status);
  for(index = 0; index < totalAtoms; index++)
   {
     item_set iSet;
     for(index2 = 0; index2< atomSize; index2++)
      {
        iSet.insert(rArray[atrib]);
        atrib++;
      }
    theClass.push_back(iSet);
  }
 return theClass;
}
```

Appendix J

Send Frequent Itemsets Function (1 of 2)

```
// Function to send frequent items of class to a process
void Send_FreqItems(setItemsetVect freqItems, int procNum,int classNum, int
sArray[])
{

   int index = 0,
        size = 0;
   int j;
   int totItems = 0,
      itemSize;
   item_set::iterator i, enditer;
   int sendSizeAtrb[3];  //0 size (Value of -1 indicates end of list from proc ,
                                        1 attribute number

   item_set tempSet;
   size = (int)freqItems.size(); //Number of atoms in class
   sendSizeAtrb[0] = 0;
   sendSizeAtrb[1] = 0;
   sendSizeAtrb[2] = classNum; // class number
   if(sArray[0] == -1)
     {
      sendSizeAtrb[0] = -1; //Number of elements in each itemset
      sendSizeAtrb[1] = size;  //Number of itemsets that are in vector
      MPI_Send(sendSizeAtrb,
               3,
               MPI_INT,
               procNum,
               TAG,
               MPI_COMM_WORLD);
      return;
     } // size is 0 no freuent items in class
```

Appendix J

Send Frequent Itemsets Function (2 of 2)

```
   for( index = 0; index < size; index++)
     {
      tempSet = freqItems[index];
      itemSize = (int)tempSet.size();
      for(i = tempSet.begin(); i != tempSet.end(); i++)
        {
         sArray[totItems] = *i;
         totItems++;
        }
     }
   sendSizeAtrb[0] = itemSize; //Number of elements in each atom
   sendSizeAtrb[1] = size;  //Number of atoms in the class
   MPI_Send(sendSizeAtrb,
                3,
                MPI_INT,
                procNum,
                TAG,
                MPI_COMM_WORLD);
   MPI_Send(sArray,
         totItems,
         MPI_INT,
         procNum,
         TAG,
         MPI_COMM_WORLD);
  sendSizeAtrb[0] = -1;
}
```

Appendix K

Receive Frequent Itemsets Function (1 of 2)

```
// Function to receive frequent itemsets of a class from a process
setItemsetVect Recv_Freqitems(int procInfo[],int rArray[])
{
  int source;
  int itemSize; //Number of attributes in each atom
  int totalItems; //Total atoms in class
  MPI_Status status;
  #define TAG           100
  #define TRIANGARRASIZE 1953
  int index = 0,
     index2 = 0,
     atrib= 0,
     size = 0;
  int rnElem = 0;
  int pause;
  setItemsetVect freqItemsVec;
  int recvSizeAtrb[3];  //0 size (Value of -1 indicates end of list from proc ,
                              1 attribute number
  recvSizeAtrb[0] = 0;
  recvSizeAtrb[1] = 0;
  recvSizeAtrb[2] = 0;

  freqItemsVec.clear();

   /* Receive a message from a process:                    */
  MPI_Recv(recvSizeAtrb,
        3,
        MPI_INT,
        MPI_ANY_SOURCE,
        TAG,
        MPI_COMM_WORLD,
         &status); //MPI_STATUS_IGNORE);
 source = status.MPI_SOURCE;
 procInfo[0] = source;
 procInfo[1] = recvSizeAtrb[2];// class number
 procInfo[2] = recvSizeAtrb[2];
 itemSize = recvSizeAtrb[0];
 totalItems = recvSizeAtrb[1];
```

Appendix K

Receive Frequent Itemsets Function (2 of 2)

```
/* Changed to send 0 in procInfo[2] to indicate no rules */
  if(itemSize < 0)
    {
      procInfo[2] = 0;
      return freqItemsVec; //there are no frequent items in the class
    }
  rnElem = itemSize * totalItems;
  printf("\n** rnElem to receive = %d\n", rnElem);
  MPI_Recv(rArray,
           rnElem,
           MPI_INT,
           source,
           TAG,
           MPI_COMM_WORLD,
           &status);

  for(index = 0; index < totalItems; index++)
   {
         item_set iSet;
     for(index2 = 0; index2 < itemSize; index2++)
      {
        iSet.insert(rArray[atrib]);
        atrib++;
      }
    freqItemsVec.push_back(iSet);
  }
 return freqItemsVec;
}
```

Appendix L

Sample Data File

```
TID,Parish,Race,Religion,SchUnivAtd,SchUniv,ExamPassed
1,13,15,26,47,49,63
9,13,,,,,
17,13,15,26,44,50,57
25,13,15,26,43,49,56
33,13,15,26,47,50,63
41,13,,,,,
49,13,15,26,47,49,63
57,13,15,24,47,49,63
65,13,15,26,47,54,63
73,13,15,24,47,54,63
81,13,15,24,42,48,55
89,13,15,26,47,49,63
97,13,15,41,42,48,55
105,13,15,24,47,54,63
113,13,15,23,42,48,55
121,13,15,32,42,48,55
129,13,15,24,43,49,56
137,13,15,26,43,49,56
145,13,15,31,43,49,56
153,13,15,26,44,50,57
161,13,15,26,47,49,63
169,13,15,26,47,49,63
177,13,,,,,
185,13,15,29,47,49,63
193,13,15,23,43,49,56
201,13,15,26,47,54,63
209,13,15,41,,49,
217,13,15,24,47,49,63
225,13,15,41,47,54,63
233,13,15,23,44,50,57
241,13,15,23,43,49,56
249,13,,,,,
481,13,15,23,42,48,55
489,13,15,23,47,50,63
497,13,15,24,43,49,56
```

Appendix M

Sample Output for DDRM (1 of 5)

```
STATISTICAL DATA FOR DDRM
Total number of transactions = 1116759
Support for this run is = 8
Support Count for this run is = 89340
Confidence for this run is = 50
Number of Classes = 8
Number of CPUs = 4
Number of Processes = 4

**** List of frequent attributes :
2 13 14 15 20 24 26 41 43 47 49 50 54 56 63

Itemset for F2 **** contains:
2 15
2 47
2 49
2 63
14 15
14 47
14 63
15 24
15 26
15 41
15 43
15 47
15 49
15 50
15 54
15 56
15 63
20 47
20 63
26 47
26 49
26 63
41 47
41 49
41 63
43 49
```

Appendix M

Sample Output for DDRM (2 of 5)

```
43 49
43 56
47 49
47 50
47 54
47 63
49 56
49 63
50 63
54 63


The following is the List of Classes:
Class 0 containsthe following atoms:
2 13 14 15
2 13 14 20
2 13 14 24
2 13 14 26
2 13 14 41
2 13 14 43
2 13 14 47
2 13 14 49
2 13 14 50
2 13 14 54
2 13 14 56
2 13 14 63
Class 1 contains the following atoms:
2 13 15
2 13 20
2 13 24
2 13 26
2 13 41
2 13 43
2 13 47
```

Appendix M

Sample Output for DDRM (3 of 5)

2 13 49
2 13 50
2 13 54
2 13 56
2 13 63
Class 2 contains the following atoms:
2 14 15
2 14 20
2 14 24
2 14 26
2 14 41
2 14 43
2 14 47
2 14 49
2 14 50
2 14 54
2 14 56
2 14 63
Class 3 contains the following atoms:
2 15
2 20
2 24
2 26
2 41
2 43
2 47
2 49
2 50
2 54
2 56
2 63
Class 4 contains the following atoms:
13 14 15
13 14 20
13 14 24
13 14 26
13 14 41
13 14 43

Appendix M

Sample Output for DDRM (4 of 5)

13 14 47
13 14 49
13 14 50
13 14 54
13 14 56
13 14 63
Class 5 contains the following atoms:
13 15
13 20
13 24
13 26
13 41
13 43
13 47
13 49
13 50
13 54
13 56
13 63
Class 6 contains the following atoms:
14 15
14 20
14 24
14 26
14 41
14 43
14 47
14 49
14 50
14 54
14 56
14 63
Class 7 contains the following atoms:
15
20
24
26
41
43
47

Appendix M

Sample Output for DDRM (5 of 5)

```
49
50
54
56
63
Rule number 1
47    ===========>     2 63
 63    ===========>     2 47
 2 47    ===========>     63
 2 63    ===========>     47


 Rule number 2
47    ===========>     63
 63    ===========>     47
 Start time for computation is = 1114196968
End time for computation is = 1114199081
Total time for computation is = 2113
Total time to process database is = 614
Total time to Compute F2 is = 557
Total time to send all classes is = 826
Total time to receive all classes is = 723
Total time to process all classes is = 826
Total time to generate all rules is = 903


 Arrival Time of First set of Classes:

Class Number 1 arrived at 103 from Processor 2
Class Number 0 arrived at 103 from Processor 1
Class Number 2 arrived at 104 from Processor 3
Class Number 4 arrived at 173 from Processor 1
Class Number 5 arrived at 173 from Processor 3
Class Number 6 arrived at 368 from Processor 1
Class Number 3 arrived at 371 from Processor 2
Class Number 7 arrived at 826 from Processor 3
```

Reference List

Adamo, J. (2001). *Data Mining for Association Rules and Sequential Patterns Sequential and Parallel Algorithms*. New York, NY: Springer-Verlag New York, Inc.

Agrawal, R., Gehrke, J., Gunopulos, D., & Raghaven, P. (1998). Automatic subspace clustering of high dimensional data for data mining applications. *Proceedings of ACM SIGMOD International Conference on Management of Data*, 94-105.

Agrawal, R., Imielinski, T., & Swami, A. (1993). Mining association rules between sets of items in large databases. *Proceedings of ACM SIGMOD International Conference on Management of Data*, 207-216.

Agrawal, R., & Shafer, J. C. (1996). Parallel mining of association rules. *IEEE Transaction on Knowledge and Data Engineering,* 8 (6), 962-969.

Agrawal, R., & Srikant, R. (1994). Fast algorithms for mining association rules. *Proceedings of the 20th International Conference on Very Large Databases*, 487-499. San Francisco, CA: Morgan Kaufman Publishers.

Bayardo Jr., R. J., Agrawal, R., & Gunopulos, D. (1999). Constraint-based rule mining in large, dense databases. *Proceedings of the 15th International Conference on Data Engineering*, 188-197.

Brin, S., Motwani, R., Ullman, J. D., & Tsur, S. (1997). Dynamic itemset counting and implication rules for market basket data. *Proceedings of ACM SIGMOD International Conference on Management of Data*, 255-264.

Carter, C. L., & Hamilton, H. J. (1998). Efficient attribute-oriented generalization for knowledge discovery from large databases. *IEEE Transaction on Knowledge and Data Engineering,* 10 (2), 193-208.

Cheung, D. W., Han, J., Ng, V. T., Fu, A. W., & Fu, Y. (1996). A fast distributed algorithm for mining association rules. *Proceedings of the 4th International Conference on Parallel and Distributed Information Systems*, 31-43, Los Altamos, CA: IEEE Computer Society Press.

Cheung, D. W., Hu, K., & Xia, S. (1998). Asynchronous parallel algorithms for mining association rules on shared-memory multi-processors. *Proceedings of 10th ACM Symposium on Parallel Algorithms and Architectures*, 279-288, New York, NY: ACM Press.

Cheung, D., & Xiao, Y. (1998). Effect of data skewness in parallel mining of association rules. *Proceedings of Pacific-Asia Conference on Knowledge Discovery and Data*

Mining, Lecture Notes in Computer Science, vol. 1394, 48-60, New York, NY: Springer-Verlag.

Chow, R. & Johnson, T. (1998). *Distributed Operating Systems and Distributed Algorithms*. Berkeley, CA: Addison-Wesley.

DeWitt, D. & Gray, J. (1992). Parallel database systems: The future of high performance database systems. *Communications of ACM*, 35 (6), 85-98.

Ganter, B., & Willie, R. (1999). *Formal Concept Analysis Mathematical Foundations*. New York, NY: Springer.

Han, J., & Kamber, M. (2001). *Data Mining Concepts and Techniques*. San Francisco, CA: Morgan Kaufman.

Han, E., Karypis, G., & Kumar, V. (2000). Scalable parallel data mining for association rules. *IEEE Transaction on Knowledge and Data Engineering*, 12(3), 337-352.

Hand, D., Mannila, H., & Smyth, P. (2001). *Principles of Data Mining*. Cambridge, MA: MIT Press.

Jacob, M., & Lee, S. (1999). Task spreading and shrinking on multiprocessor systems and networks of workstations. *IEEE Transaction on Parallel and Distributed Systems*, 10 (10), 1082-1101.

Jian, L., Yingjun, L., Xiaoxing, M., Min, C., Xianping, T., Guanqun, Z., & JianzHong, L. (2000). A hierarchical framework for parallel seismic applications. *Communications of ACM*, 43 (10), 55-59.

Megiddo, N., & Srikant, R. (1998). Discovering predictive association rules. *Proceedings of the 4th International Conference on Knowledge Discovery and Data Mining* (*KDD*-98), 274-278.

Park, J. S., Chen, M., & Yu, P. S. (1995). An effective hash-based algorithm for mining association rules. *Proceedings of ACM SIGMOD International Conference on Management of Data*, 175-186.

Parthasarathy, S., Zaki, M. J., & Li, W. (1998). Memory placement techniques for parallel association mining. *Proceedings of the 4th International Conference on Knowledge Discovery and Data Mining* (*KDD*-98), 304-308.

Shintani, T., & Kitsuregawa, M. (1998). Parallel mining algorithms for generalized association rules with classification hierarchy. *Proceedings of ACM SIGMOD International Conference on Management of Data*, 25-36.

Shintani, T., & Kitsuregawa, M. (1996). Hash based parallel algorithms for mining association rules. *Proceedings of 4th International Conference on Parallel and Distributed Information Systems*, 19-30, Los Altamos, CA: IEEE Computer Society Press.

Srikant, R., Vu, Q., & Agrawal, R. (1997). Mining association rules with item constraints. *Proceedings of the 3rd International Conference on Knowledge Discovery and Data Mining*, 67-73.

Street, A. P., & Wallis, W. D. (1982). *Combinatorics: A First Course*. Manitoba, Canada: The Charles Babbage Research Centre.

Tamura, M., & Kitsuregawa, M. (1999). Dynamic load balancing for parallel association rule mining on heterogeneous PC cluster system. *Proceedings of the 25th International Conference on Very large Databases*, 162-179.

Wu, M. (1997). On runtime parallel scheduling for processor load balancing. *IEEE Transaction on Parallel and Distributed Systems*, l 6 (2), 640-656.

Zaki, M. J. (2000). Scalable algorithms for association mining. *IEEE Transactions on Knowledge and Data Engineering*, 12(3), 372-390.

Zaki, M. J. (2000b). Parallel and distributed data mining: An introduction. In Zaki, M. J. & Ho, C. (Eds.), *Large-Scale Parallel Data Mining* (pp. 1-23). New York, NY: Springer-Verlag.

Zaki, M. J. (2000c). Hierarchical parallel algorithms for association mining. In Kargupta, H & Chan P. (Eds.), *Advances in Distributed and Parallel Knowledge Discovery* (pp. 339-336). Cambridge, MA: MIT Press.

Zaki, M. J., Parthasarathy, S., & Li, W. (1997). A localized algorithm for parallel association mining. *Proceedings of 9th Annual ACM Symposium on Parallel Algorithms and Architectures* (*SPAA*), 321-330.

Zaki, M. J., Parthasarathy, S., Ogihara, M., & Li, W. (1997). New algorithms for fast discovery of association rules. *Proceedings of 3rd International Conference on Knowledge Discovery and Data Mining*, 283-286.