EFFICIENT PARALLEL FREQUENCY MINING BASED ON A NOVEL TOP-DOWN PARTITIONING SCHEME FOR TRANSACTIONAL DATA

A THESIS

SUBMITTED TO THE DEPARTMENT OF COMPUTER ENGINEERING

AND THE INSTITUTE OF ENGINEERING AND SCIENCE

OF BILKENT UNIVERSITY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

MASTER OF SCIENCE

By

Eray Özkural

January, 2002

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Prof. Cevdet Aykanat(Advisor)

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Assist. Prof. Attila Gürsoy

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Assist. Prof. Uğur Güdükbay

Approved for the Institute of Engineering and Science:

Mehmet Baray
Director of the Institute

ABSTRACT

EFFICIENT PARALLEL FREQUENCY MINING BASED ON A NOVEL TOP-DOWN PARTITIONING SCHEME FOR TRANSACTIONAL DATA

Eray Özkural
M.S. in Computer Engineering
Supervisor: Prof. Cevdet Aykanat
January, 2002

In recent years, large quantities of data have been amassed with advances in data acquisition capabilities. Automated detection of useful information is required for vast data obtained from scientific and business domains. Data Mining is the application of efficient algorithmic solutions on a variety of immense data for such knowledge discovery.

Frequency mining discovers all frequent patterns in a transaction or relational database and it comprises the core of several data mining algorithms such as association rule mining and sequence mining. Frequent pattern discovery has become a challenge for parallel programming since it is a highly complex operation on huge datasets demanding efficient and scalable algorithms.

In this thesis, we propose a new family of parallel frequency mining algorithms. We introduce a novel transaction set partitioning scheme that can be used to divide the frequency mining task in a top-down fashion. The method operates on the graph of frequent patterns with length two (G_{F_2}) from which a graph partitioning by vertex separator (GPVS) is mapped to a two-way partitioning on the transaction set. The two parts obtained can be mined independently and therefore can be utilized for concurrency. In order for this property to hold, there is an amount of replication dictated by the separator in G_{F_2} which is minimized by the GPVS algorithm. A k-way partitioning is derived from recursive application of 2-way partitioning scheme which is used in the design of a generic parallel frequency mining algorithm. First we compute G_{F_2} in parallel, succeeding that we designate a k-way partitioning of the database for k processors with a parallel

recursive procedure. The database is redistributed such that each processor is assigned one part. Subsequent mining proceeds simultaneously and independently at each processor with a given serial mining algorithm. A complete implementation in which we employ FP-Growth as the sequential algorithm has been achieved. The performance study of the algorithm on a Beowulf system demonstrates favorable performance for synthetic databases. For hard instances of the problem, we have gained approximately twice the speedup of a state-of-the-art algorithm.

We also present a correction and optimization to FP-Growth algorithm.

Keywords: Parallel Data Mining, Frequency Mining.

ÖZET

YENİ BİR İŞLEM VERİSİ PARÇALAMA ŞEMASI TABANLI ETKİN PARALEL FREKANS TARAMA

Eray Özkural
Bilgisayar Mühendisliği, Yüksek Lisans
Tez Yöneticisi: Prof. Cevdet Aykanat
Ocak, 2002

Son yıllarda, gelişen veri toplama yetenekleriyle birlikte büyük miktarlarda veri toplanmıştır. Bilimsel ve iş alanlarından elde edilen çok geniş veriler için yararlı bilgilerin otomatik olarak bulunmasi gerekmektedir. Veri tarama bu tarz bilgi keşfi için etkin algoritmik çözümlerin değişik büyük veriler üzerinde uygulanmasıdır.

Frekans tarama bir işlem ya da ilişkisel veri tabanındaki bütün sık desenleri keşfeder ve ilişki kuralı tarama ve dizi kurali tarama gibi bir çok veri tarama algoritmalarının özünü oluşturur. Sık desen keşfi paralel programlama için dev veriler üzerinde karmaşık bir işlem olması itibariyle önemli bir problem haline gelmiştir.

Bu tezde, yeni bir sınıf paralel frekans tarama algoritması öneriyoruz. Frekans tarama işini tepeden aşağı bölmek için kullanılabilecek bir işlem verisi parçalama şeması takdim ediyoruz. Yöntemimiz iki uzunluğundaki sık desenlerin çizgesi (G_{F_2}) üzerinde çalışmakta olup, bu çizgenin köşe ayracıyla parçalanması (GPVS) işlem kümesi üzerinde iki-yollu bir parçalamaya eşlenmektedir. Elde edilen iki parça bağımsız olarak taranabilir ve bu sayede eş zamanlılık için kullanilabilir. Bu özelliğin tutması için G_{F_2}'deki ayraç tarafından belirlenen ve GPVS tarafından minimize edilen bir yineleme bulunmaktadır. Genel bir paralel frekans taramasi algoritmasında kullanılan bir k-yollu parçalama iki-yollu parçalama şemasından türetilmektedir. İlk olarak G_{F_2}'yi paralel olarak hesaplarız ve ertesinde veri tabanının k-yollu bir parçalaması k işlemci için paralel kendini çağıran bir yöntemle belirlenir. Veri tabanı her işlemciye bir parça düşecek şekilde yeniden dağıtılır. İzleyen tarama her işlemcide verilen seri bir tarama algoritmasıyla eş

zamanlı biçimde devam eder. FP-Growth'u seri algoritma olarak kullandiğımız tam bir program gerçekleştirilmiştir. Bir Beowulf sistemi üzerinde yapılan performans çalışması sentetik veritabanları için iyi hızlanma kaydettiğimizi göstermektedir. Problemin zor örneklerinde gelişmiş bir algoritmanın yaklaşık iki katı hızlanma kazanmış bulunmaktayız.

Ayrıca FP-Growth için bir düzeltme ve hızlandırma sunuyoruz.

Anahtar sözcükler: Paralel Veri Tarama, Frekans Tarama.

To My Family and Friends,

Acknowledgements

I am grateful to my advisor Cevdet Aykanat for his guidance and motivation throughout our research. Theoretical and algorithmic portions of this thesis owe much to his insightful comments and our long discussions. Working with him has been fruitful and joyous.

I would like to thank Bora Uçar for the original idea that led to the development of this thesis. His valuable suggestions and co-operation were essential to our findings. I appreciate Cevdet Aykanat, Bora Uçar and Uğur Güdükbay for taking their time to review the draft copy.

I would like to also thank to colleagues and my friends for their moral and intellectual support during my studies. I could not have endured without Arda, Atacan, Barla, Bülent, Engin, Mehmet, Mercan, Mustafa, Selim, Sengör, Murat, Umut and others whose names I have not written. Many thanks especially to my old friend Atacan. I feel much priviliged for having the opportunity to talk to you of those subjects that so few can.

I would like to thank to my family whose persistent support and understanding were the most vital ingredients in my studies.

Contents

1 Introduction **1**

1.1 Outline 2

1.2 Problem Statement 2

2 Background **6**

2.1 Data Mining and Knowledge Discovery 7

2.2 Frequency Mining 11

2.2.1 Association Rules 12

2.2.2 Search Space 13

2.2.3 Mining Algorithms 15

2.3 Parallel Frequency Mining 22

2.3.1 Overview of Parallel Mining Algorithms 24

2.3.2 Apriori Based Parallel Algorithms 24

2.3.3 Parallel algorithms based on Eclat and Clique 25

2.3.4 Other Studies and Remarks 27

2.4 Graph Partitioning 28

2.4.1 Application Domains 28

2.4.2 Graph Partitioning Methods 29

2.4.3 Problem Description 30

3 Transaction Set Partitioning **32**

3.1 Objective 32

3.2 Transaction Set Partitioning 33

3.3 Two-way Partitioning of Transaction Database 35

3.4 k-way Partitioning of Transaction Database 38

4 A Parallel Algorithm for Frequency Mining **42**

4.1 Overview 42

4.2 Computation of F and G_{F_2} 43

4.3 Partitioning 46

4.3.1 Using GPVS to find a partition 46

4.3.2 Load Balancing 47

4.3.3 Redistribution of Transaction Set 49

4.3.4 Computing Vertex Induced Subgraph 49

4.4 Optimizations 51

4.4.1 Using An F_2 Matrix of Rank $|F|$ 51

4.4.2 Redistributing Transaction Set In A Single Pass 51

4.4.3 Distributed Graph for G_{F_2} and Local Pruning 53

4.4.4 Compact Structures and Buffering for Communication . . 54

4.5 Concurrent Mining of Partitions 54

4.5.1 An Improved Version of FP-Growth 55

4.5.2 A Correction To FP-Growth Algorithm 55

4.5.3 Eliminating Conditional Pattern Base Construction 56

5 Implementation **60**

5.1 System Hardware and Software 60

5.2 Code 61

5.2.1 Communication Routines 62

5.2.2 Data Structures 62

5.3 Initial Distribution 62

5.4 Computing F and G_{F_2} 63

5.5 Partitioning 63

5.6 FP-Growth Implementation 64

6 Performance Study **66**

6.1 Data 66

6.2 Running Time 67

6.3 Speedup 75

6.4 Load Balancing 75

6.5 Interpretation 75

6.6 Comparison with Parallel Eclat 85

7 Conclusions **87**

A Detailed Performance Results **100**

B Proof and Algorithm **105**

List of Figures

2.1 Search space of frequency mining problem 14

2.2 Classification of sequential association rule mining algorithms. . . 16

2.3 An FP-Tree Structure. 20

3.1 G_{F_2} graph of transaction set in Table 1.1 33

3.2 A synthetic data set with 1000 transactions and 1000 items containing 32 patterns 34

3.3 G_{F_2} graph of dataset in Table 1.2 with a support threshold of 4 . 36

3.4 GPVS of G_{F_2} graph in Figure 3.3. Dashed lines enclose parts A,B and separator S. 37

3.5 G_{F_2} graph of dataset in Figure 3.2 with a support threshold of 5% 39

3.6 GPVS of G_{F_2} graph in Figure 3.5. Dashed lines enclose parts A,B and separator S. 39

3.7 Two-way partitioning of transaction set in Figure 3.2. 41

5.1 C++ code to sort transactions in a unique decreasing order 65

6.1 Running time for support threshold 0.75% 69

6.2 Running time for support threshold 0.45% 70

6.3 Running time for support threshold 0.40% 71

6.4 Running time for support threshold 0.35% 72

6.5 Running time for support threshold 0.30% 73

6.6 Running time for support threshold 0.25% 74

6.7 Speedup for support threshold 0.75% 76

6.8 Speedup for support threshold 0.45% 77

6.9 Speedup for support threshold 0.40% 78

6.10 Speedup for support threshold 0.35% 79

6.11 Speedup for support threshold 0.30% 80

6.12 Speedup for support threshold 0.25% 81

6.13 Speedup vs. support on 16 processors. 82

6.14 Comparative performance of three load estimate functions for support threshold of 0.25% 83

A.1 Running time for T10.I6.800K and T10.I6.1600K 101

A.2 Running time for T10.I4.1024K and T15.I4.367K 102

A.3 Speedup for T10.I6.800K and T10.I6.1600K 103

A.4 Speedup for T10.I4.1024K and T15.I4.367K 104

List of Tables

1.1 A Transaction Set T with $I = \{a, b, c, d, e, f, g\}$ 3

1.2 A Transaction Set T with $I = \{a, b, c, d, e, f\}$ 4

2.1 Sequential association rule mining algorithms. 15

2.2 A sample Transaction Set 19

3.1 $\Pi_{TS}(T) = (T_1, T_2)$ of transaction set in Table 1.2 38

4.1 A Transaction Set T with $I = \{a, b, c, d, e, f, g, h, i\}$ 56

6.1 Dataset parameters 67

6.2 Synthetic data sets 67

List of Algorithms

1 APRIORI(T, ϵ) . . . 18
2 MAKE-FP-TREE(DB, ϵ) . . . 22
3 INSERT-TRIE$([p|P], T)$ . . . 22
4 FP-GROWTH$(Tree, \alpha)$ . . . 23
5 PAR-FREQ$(T_i, \epsilon,$ MINE-FREQ$)$ . . . 43
6 COMPUTE-F-$G_{F_2}(T_i, \epsilon)$ . . . 44
7 COUNT-1-ITEMS(T_i, ϵ) . . . 44
8 COUNT-2-ITEMS(T_i, ϵ, F) . . . 45
9 K-WAY-PARTITION$(T_i, \epsilon, G_{F_2}, Processors)$ . . . 47
10 2-WAY-PARTITION$(T_i, \epsilon, F, G_{F_2}, Processors)$ . . . 47
11 PARTITION-PROCESSORS(P_1, P_2) . . . 49
12 REDISTRIBUTE-DB$(P_1, P_2, Processors_1, Processors_2)$ . . . 50
13 CYCLE(d, P) . . . 50
14 VERTEX-INDUCED-SUBGRAPH(G, A) . . . 51
15 K-WAY-PARTITION$^*(T_{local}, \epsilon, G_{F_2}, Processors)$ . . . 52
16 REDISTRIBUTE-DB$^*(T_{local}, Parts)$ . . . 53
17 FP-GROWTH$^*(Tree, \alpha)$ . . . 56
18 CONS-CONDITIONAL-FP-TREE$(Tree, s)$ . . . 57
19 COUNT-PREFIX-PATH$(node, count)$ . . . 58
20 GET-PATTERN$(node)$ . . . 58
21 INSERT-PATTERN$(Tree, pattern)$ . . . 59
22 PAR-FP-GROWTH(T_i, ϵ) . . . 60
23 FP-MINE(T_i, ϵ, G_{F_2}) . . . 60
24 COUNT-2-ITEMS$^*(T_i, \epsilon, F)$ . . . 106

Chapter 1

Introduction

Frequency mining is the discovery of all frequent patterns in a transaction or relational database. Frequent pattern discovery comprises the core of several data mining algorithms such as association rule mining and sequence mining [35], and it dominates the running time of these algorithms. It has been shown to have an exponential worst case running time in the number of items[1], therefore much research has been devoted to increasing the efficiency of this task.

Since both the data size and the computational costs are large, parallel algorithms have been studied extensively. Scalable data mining can only be provided by a parallel system in the face of massive I/O and computation. Frequency mining has become a challenge for parallel programming since it is a highly complex operation on huge datasets requiring efficient and scalable algorithms.

In this thesis, we build upon the work of Agrawal [5], Zaki [82], and Jiawei Han [35] investigating a new family of algorithms using a novel top-down data partitioning scheme to achieve parallelism. We present complete implementation of a proposed algorithm and demonstrate its parallel performance.

[1] In the more common database terminology, the number of attributes.

1.1 Outline

The organization of this thesis progresses from the theoretical to the more practical. Chapter 2 surveys the frequency mining problem as a prominent task in data mining. Chapter 3 consists of our theoretical observations from which we construct the parallel frequency mining algorithm of Chapter 4. Succeeding that, Chapter 5 conveys the implementation details of our system. An extensive performance study of our system is presented in Chapter 6, before Chapter 7 in which we summarize our findings. Readers who are acquainted with parallel data mining and frequency mining may skip Chapter 2.

In the following section, we present the frequency mining problem in a formal manner.

1.2 Problem Statement

Frequency mining involves a transaction database T which consists of a set of transactions each of which are drawn from a set I of items. The mining algorithm finds all patterns that occur with a frequency satisfying a given absolute support threshold ϵ.[2]

Definition 1. *A **transaction set** is a collection of transactions drawn from a set of items I.*

For a transaction set T and a set of items I,

$$T = \{X | X \subseteq I\} \tag{1.1}$$

Equation 1.1 gives us the formal definition of a transaction database.[3] In practice, the number of items $|I|$ is in the order of magnitude of 10^3 and more. The number of transactions is much larger, at least 10^5.

[2]Support thresholds in all definitions are absolute values rather than relative.
[3]The collection can contain multiple instances of the same transaction.

In Table 1.1 an example transaction set with an item set $I = \{a, b, c, d, e, f, g\}$ is exhibited. Each row in the table below the legend corresponds to a transaction $X \in T$, and each member of a transaction is marked with $\times$. In this sample transaction set there are $|I| = 7$ items and $|T| = 5$ transactions. Table 1.2 illustrates another transaction set with $|I| = 6$ and $|T| = 9$.

Transaction	a	b	c	d	e	f	g
$t_1 = \{b, c, e, g\}$		$\times$	$\times$		$\times$		$\times$
$t_2 = \{a, d, f, g\}$	$\times$			$\times$		$\times$	$\times$
$t_3 = \{a, b, d, e, g\}$	$\times$	$\times$		$\times$	$\times$		$\times$
$t_4 = \{b, c, e, f\}$		$\times$	$\times$		$\times$	$\times$	
$t_5 = \{a, d, e, g\}$	$\times$			$\times$	$\times$		$\times$

Table 1.1: A Transaction Set T with $I = \{a, b, c, d, e, f, g\}$

Definition 2. *The occurrence function $o : 2^I \times I \mapsto \{0, 1\}$ detects whether an item is present in a given transaction.*

$$o(X, i) = \begin{cases} 1 & \text{if } i \subseteq X, \\ 0 & otherwise \end{cases} \tag{1.2}$$

We have defined a utility function for computing frequency of items. For transaction $t_1 = \{a, c\}$ in Table 1.1, $o(t_1, b) = 0$ while $o(t_1, c) = 1$.

Definition 3. *The frequency function $f : 2^{2^I} \times I \mapsto \mathbb{N}$ computes the number of times a given item $i \in I$ occurs in the transaction set T.*

$$f(T, i) = \sum_{X \in T} o(X, i) \tag{1.3}$$

For transaction set T in Table 1.1, $f(T, c) = 2$ since c occurs only 2 times.[4]

Definition 4. *A **pattern** is $X \subseteq I$, a subset of I. The set of all patterns in a transaction set T is $\mathcal{P}(T) = 2^I$, which is the power set of I.*

[4]In this and following definitions and lemmas, please assume that all sets are multi-sets in which duplicates are permitted.

Transaction	a	b	c	d	e	f
$t_1 = \{a, b, c, f\}$	×	×	×			×
$t_2 = \{b, c\}$		×	×			
$t_3 = \{a, d, e, f\}$	×			×	×	×
$t_4 = \{b, d, e, f\}$		×		×	×	×
$t_5 = \{a, c, d, e\}$	×		×	×	×	
$t_6 = \{a, b, d, e\}$	×	×		×	×	
$t_7 = \{b, c, e, f\}$		×	×		×	×
$t_8 = \{a, b, c, d, e\}$	×	×	×	×	×	
$t_9 = \{b, c, f\}$		×	×			×

Table 1.2: A Transaction Set T with $I = \{a, b, c, d, e, f\}$

The frequency function is extended to $f : 2^{2^I} \times 2^I \mapsto \mathbb{N}$ for computing the frequency of a given *pattern*. The frequency of a given pattern X is the number of items it has occured in the transaction set. We define an extended occurrence function for computing the frequency of a pattern.

$$o(X, Y) = \begin{cases} 1 & \text{if } Y \subseteq X, \\ 0 & otherwise \end{cases} \tag{1.4}$$

$$f(T, Y) = \sum_{X \in T} o(X, Y) \tag{1.5}$$

where X is a transaction as in Equation 1.2, and Y is a pattern. For a pattern $\{b, d\}$ in transaction set T of Table 1.2, $f(T, \{b, d\}) = 3$.

Definition 5. *Frequency mining is the discovery of all* ***frequent patterns*** *in a transaction set with a frequency of support threshold ϵ and more. The set of all frequent patterns $\mathcal{F}$ in a given transaction set T is defined as:*

$$\mathcal{F}(T, \epsilon) = \{X | X \in 2^I \wedge f(T, X) \geq \epsilon\} \tag{1.6}$$

The set of all frequent patterns in transaction set T of Table 1.1 for a support threshold of 3 is $\mathcal{F}(T, 3) = \{\{a, d\}, \{b, e\}, \{a, d\}, \{g, a\}, \{g, d\}, \{a, g, d\}, \{g, e\}\}$

There are definitions of frequency mining more suitably recognized as subsets of the total mining problem such as discovery of maximal frequent patterns [79] and closed frequent patterns [77].

In the algorithms we will describe two additional sets require special consideration. F is the set of frequent items, and F_2 is the set of frequent patterns with length 2. More formally stated,

$$F = \{x \in I \mid f(T, X) \geq \epsilon\} \tag{1.7}$$

$$F_2 = \{X \in \mathcal{F} \mid |X| = 2\} \tag{1.8}$$

We also state a significant property of frequency mining which was introduced in [7]. We provide a proof in Appendix B.

Lemma 1. ***(Downward Closure)*** *If* $X \in \mathcal{F}(T, \epsilon)$ *then* $\forall Y \subset X, Y \in \mathcal{F}(T, \epsilon)$.

The rest of this thesis is devoted to computation of $\mathcal{F}$. Computing $\mathcal{F}$ for large T and I has enormous time and space requirements. Many serial and parallel algorithms have been designed to tackle this problem. In the next chapter we will survey previous such work.

Chapter 2

Background

In recent years, large quantities of data have been amassed with advances in data acquisition capabilities [16]. The domain of this data ranges from retail transactions [4], world wide web [20] and telecommunications[72] to astrophysics [61], stock-market [39], biological databases [19, 62], weather, geological, environmental [24] and several others. There are virtually no limits to the kind of data that can be collected, yet without the means to analyze them they are of little value. It has been long conjectured that this enormous landscape of data contains within interesting new facts about these domains. However, comprehending such data is beyond the capacity of human processing [59]. Automated methods are required, hence Data Mining has emerged combining the works in Artificial Intelligence and Database Systems. Due to the scale and complexity of knowledge discovery and data mining, traditional analysis and algorithms cannot be maintained. Data Mining carries out Machine Learning in these vast databases for deeper understanding of what lies beneath. The field thus has proliferated gathering attention from many researchers from various fields as we need new algorithms and approaches for accomplishing this task.

2.1 Data Mining and Knowledge Discovery

Data Mining is an important step in Knowledge Discovery in Databases (KDD) Process which consists of selection, pre-processing, transformation, data mining, interpretation/evaluation [25]. Data Mining may be described as "a process of nontrivial extraction of implicit, previously unknown and potentially useful information from data in databases" [16]. Fayyad et al. define data mining as "a step in the KDD process that consists of applying data analysis and discovery algorithms that, under acceptable computational efficiency limitations, produce a particular enumeration of patterns (or models) over the data" [27]. KDD itself is referred to as overall process of discovering useful knowledge from data [26]. However, it might not be accurate to view data mining as a toolkit without a specific aim on its own. Indeed, data mining, as implied by the definition of Fayyad [26], is the most algorithmic and technical of the steps in KDD. However, it is not without deliberate design of the qualitative aspects of input and output that we should conduct research in data mining. For instance, data clustering strives to discover conceptual structure implicit in data. The I/O and processing of such a task depends strictly on cognitive aspects of concept acquisition and interpretation of such data by humans as well as the sort of data being worked on. The boundary between KDD and data mining is not clear cut. Therefore the definition we maintain for data mining is: "Application of efficient algorithmic solutions on a variety of immense data for knowledge discovery". That is distinct from other steps in KDD, in that it is comprised fully of *autonomous* processes, and that it solves the problems of KDD with *algorithmic* means operating on large databases; in many cases with novel algorithms.[1] In that respect, computing the variance of a numerical attribute may be part of KDD[2], but it is not necessarily data mining. Likewise, the algorithmic details and efficiency considerations are vital in data mining research whereas KDD is interested in the *outcome* and *applicability* of those algorithms.

KDD and data mining present new challenges to researchers; Fayyad, Shapiro

[1] However, interactivity is not excluded in data mining. See for instance the work on online association rule mining [41].

[2] In preprocessing for instance.

and Smyth [28, 26] enumerate massive databases and high dimensionality, interaction and comprehensibility, overfitting and assessing statistical significance, missing data, data in flux, integration and multi-source multi-type data as challenges of KDD. According to Chen et al. [16], the requirements and challenges of data mining are handling of different types of data, efficiency and scalability, usefulness of results, presentation and interactivity, distributed data sources, and privacy / security issues.

Among common data mining functions are summarization, clustering, classification, regression, deviation detection and dependency modelling [28]:

Summarization Discovery of a compact and meaningful description for a subset of data has found several applications. This operation usually finds patterns in the data and derives higher level knowledge from them. Among these are association rule mining which finds rules that correlate the presence of one set of items with that of another set of items[9] and sequence mining which finds interesting sequential relations in event data. For instance, association rule mining can discover which items are sold together in a store. This knowledge can be used to shelve or advertise products closer to one another [9]. Application of association rule mining includes discovery of web content and usage rules [20], phrase association rules from text [38], reducing telecommunication order failures and detecting redundant medical tests [10], recurrent images in medical image databases [76], co-citation in scientific papers [55], automatic classification of e-mail messages [43]. There are variants of association rule mining such as implication rules which can help find rules like "heads of households do not have personal care limitations" from U.S Census data [14]. We will revisit association rules once again since it is interesting to us as the chief application of frequency mining. Sequence rule mining is an extension of association mining which considers the timestamps of transaction events [8]. Applied to a retail transaction database, sequence mining can find which items a customer is likely to purchase in the near future [9]. Sequence mining has important applications such as telecommunications alarm management [56], stock market prediction [52, 39], and predicting plan failures [80]. Other interesting branches of

work that summarizes data is data cubes and Online Analytical Processing (OLAP) [37, 2] which consider systems that are interactive and can provide automated reports on the data. Closely related is work on high-dimensional multivariate data visualization [73, 15].

Clustering is a prominent problem in data mining which aims identifying conceptual distinctions in large high dimensional volumes of data [3, 45, 46, 63, 31]. A program scans the whole dataset, possibly in multiple passes, and classifies the data points into clusters of points with no prior training. Data clustering is indeed equivalent to the unsupervised conceptual learning in Machine Learning, and has been extensively studied before [70]. Typically, a data clustering algorithm tries to maximize intra-cluster similarity while minimizing inter-cluster similarity. The clusters found aid us in characterization of data. For instance a clustering of text documents may discover documents with similar subjects. Applications of clustering include characterization of different customer groups from purchasing patterns, categorization of documents on the World Wide Web, grouping of genes with similar functionality and grouping of spatial locations prone to earth quakes from seismological data [46]. Other applications being worked on include automatic categorization of objects in the sky [25].

Classification corresponds to supervised learning while clustering corresponds to unsupervised learning. In classification algorithms, the categories or classes of objects are given to the system and the system learns to classify new objects according to this *training set*. The classifier system develops an internal model of concepts using features in the training set [16]. The model's performance is tested upon a *test set* which the trained model classifies. Most classification algorithms are based on decision trees [57, 67, 11, 44, 64]. Decision tree is a flowchart like tree in which internal nodes are tests and leaf nodes are classes. A decision tree is constructed and refined in the training phase. For a specific datum, tests are conducted starting from the root node, a path of tests lead to the class of object [44]. Applications of classification includes credit approval, product marketing and medical diagnosis [44], classification of trends in financial markets[27]. For instance

in credit approval one could categorize applicants' credit ratings as good or poor based on their income and debt [27, 9]. Classification also has several successful scientific applications such as categorization of objects in the sky and finding volcanos on Venus [25].

Regression is similar to classification, in that it learns a *value* from a training set, the value here being a real-valued prediction variable instead of a class[3]. It also discovers functional relationships between variables [28]. Alternatively known as predictive modelling [17] regression has applications such as prognostic models in medicince [1], modelling customer retention in banks [23], predicting the duration of an automobile trip [36] and estimating when a stock will change the direction of its slope [75].

Dependency Modelling finds significant dependencies between models . The dependency information is conveyed at two levels: structural and quantitative. The applications of dependency mining range from probabilistic medical expert systems to information retrieval and human genome. An interesting study that concerns structure is subgraph discovery [22, 50]. Often structural data is not taken into account in mining, however substructure discovery can lead to interesting knowledge such as discovering similar substructures in molecular biology data [19]. In [21], substructure discovery with an approach taking advantage of frequency mining as in [50] is used to find frequent substructures in chemical compounds.

Deviation Detection is concerned with the detection of outliers by finding those data points that differ significantly from the majority of data points [9]. Medical diagnosis and credit card fraud detection [9] are among the applications of this method.

The scope of data mining is not confined to those we have described. There are several variants of those subfields we have outlined as well as many hybrid or combined methods, for instance one may cluster frequent item sets found when mining association rules [34].

[3]Which is a set rather than a number.

Efficiency and scalability of algorithms is a prime requirement in data mining research [16]. Today's experimental databases of mere gigabytes will exceed terabytes in the future. The immense magnitude of scientific data such as in space sciences made available by NASA [61] will require great milestones in both the content and methods of Knowledge Discovery research.[4] In particular, the sort of scalability is available to only parallel computing systems which can store and process enormous volumes of data. Therefore we believe that there is a need for novel parallel algorithms.

We refer the reader to Fayyad et al. [27, 28, 26, 71] for an overview of KDD and data mining. For an overview from the database perspective, see Chen's survey [16]. In another introductory article Mannila discusses the field and open problems [55]. For association rule mining, an excellent survey is due to Zaki [78] which covers both sequential and parallel algorithms and discusses open problems in parallel association mining. Hipp et al. [42] study sequential algorithms and benchmark some of them.

In following sections we expound on previous work in literature related to the subject of this thesis. Frequency mining is a core operation in several data mining algorithms for determining the relevant patterns in data. It is the subject of the next section in which we explain the design of serial frequency mining algorithms. In the following section, we deem the parallel algorithms for solving the same problem.

2.2 Frequency Mining

Frequency mining problem presented in Section 1.2 comprises the core of a myriad of data mining algorithms [35]. Many mining algorithms append a phase to frequency mining to extract useful knowledge from the frequent patterns, for instance in association rule [4], sequence [8] mining and their derivatives: correlation [13], dependence rule [68], episode [56] mining.

[4] Only NASA Earth Observing System will deliver close to a terabyte of remote sensor data per day[9].

2.2.1 Association Rules

Most noteworthy of those is the association rule mining which has spawned the entire field of data mining with Agrawal's seminal paper [4]. In this paper the interesting idea was to take advantage of a theoretical observation for an algorithm that could compute what was introduced as association rules in market-basket data with relatively few database scans. It effectively extended a method in a similar vein to machine learning for databases that would not fit into main memory. Association rule mining presented an automated means to discover useful knowledge from a large database with a novel algorithm. This kind of approach has been the recurring theme of data mining since then.

An association rule is an expression $X \Rightarrow Y$ where X and Y are frequent patterns and $X \cap Y = \emptyset$. Intuitively, it means that whenever a transaction t_i contains X it is likely to also contain Y. The *support* of the rule is $f(X \cup Y)$ the frequency of $X \cup Y$, whereas the confidence of the rule is $f(X \cup Y)/f(X)$ [4]. The confidence of the rule may also be understood as the conditional probability $p(Y \subseteq T | X \subseteq T)$ [42]. In the prototypical application of data mining on market-basket data, an association rule $\{x_1, x_2\} \Rightarrow \{x_3\}$ means "A customer who purchases x_1 and x_2 is likely to purchase x_3.". Discovery of association rules had immediate practical value and with a simple yet effective algorithm it had much impact.

In order to study the efficiency of the algorithm, we need a better understanding of the problem's nature. The problem is first decomposed into two subproblems [4]:

1. Computing all frequent item sets $\mathcal{F}$ with a given support threshold ϵ.
2. Discovering rules $X \Rightarrow Y$ that satisfy a given confidence with disjoint sets X and Y chosen from $\mathcal{F}$.

The second phase is computationally easier to solve, therefore almost all algorithms have focused on efficiently computing phase 1, which is the subject of

this thesis. The goal of phase 1 is to discover frequent itemsets among $2^{|I|}$ subsets of I determining the frequencies of patterns from the complete database.[5] We will explain some of the leading frequency mining algorithms in more detail. Before doing so however, we give an alternate view of the problem for better understanding of the similarities and differences between algorithms.

2.2.2 Search Space

It is profitable to view the frequency mining problem as a *search problem* [65]. The state space consists of the following. The set of all states is 2^I, the powerset of I. The initial state is $\{\}$. The set of actions is adding an item to a set or removing an item from a set. The goal of the search is to discover all members of $\mathcal{F}$.[6] Figure 2.1 depicts the search space of item set $\{1, 2, 3, 4\}$ which is also named the *lattice* of the problem[42]. The dashed line separates the set of frequent item sets from infrequent ones in the lower region for an imaginary transaction set. As evidenced by Lemma 1, there is such a border for all transaction sets and ϵ [42].

Besides being intuitive, this view of the problem allows us to reason about the operation and efficiency of an algorithm.[7] Any algorithm will have to determine all $X \in \mathcal{F}$ and the actions we are giving for an item set demonstrate the application of Lemma 1. However, it may also deceive us in that it gives the illusion of a traditional search problem while it is not. One cannot simply deploy a traditional uninformed search algorithm to discover $\mathcal{F}$. A general algorithm cannot complete in a viable running time without taking into account the most important distinction of this problem: the frequency of each itemset can be computed only by considering the whole transaction database. As typical of search problems, the state space does not fit into memory. Neither does the transaction set fit into memory. Therefore we follow a *generate and test* approach in the design of our algorithms and we should additionally contrive means to efficiently compute the frequency function f.

[5] In this thesis, we do not consider sampling approaches which give approximate results.

[6] Which is rather unconventional for a traditional search problem.

[7] Another view of the problem is presented in [32] in which the relation to hypergraph transversal problem and learning theory is examined.

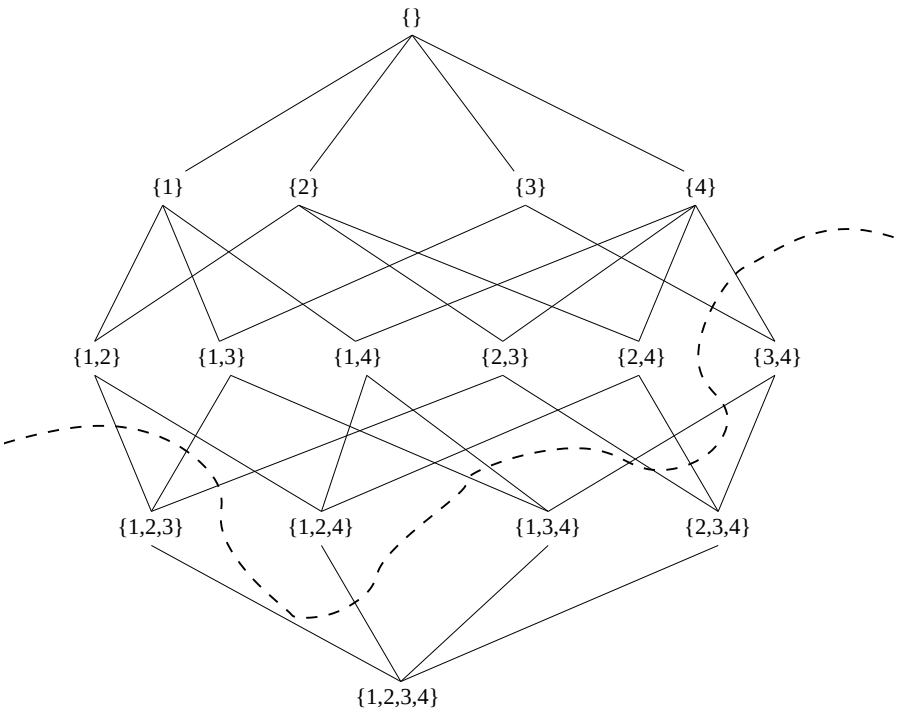

Figure 2.1: Search space of frequency mining problem

Frequency mining problem can be solved by an algorithm that traverses the state space and enumerates all $X \in \mathcal{F}$ efficiently. Nevertheless, it turns out that efficient computation of $\mathcal{F}$ presents challenges for algorithm design. Let us review some of the forthcoming algorithms which systematically enumerate all frequent item sets.

2.2.3 Mining Algorithms

Algorithm	Search	Structure	Scans	Layout
Apriori	BFS	Hash Tree	k	Horizontal
DHP	BFS	Hash Tree	k	Horizontal
Partition	BFS	Array	2	Vertical
SEAR	BFS	Trie	k	Horizontal
Spear	BFS	Trie	2	Horizontal
DIC	BFS	Trie	$\leq k$	Horizontal
Eclat	DFS	Array	≥ 3	Vertical
Clique	DFS&BFS	Array	≥ 3	Vertical
Fp-Growth	Multi-Constraint	FP-Tree	2	Horizontal

Table 2.1: Sequential association rule mining algorithms.

With so many algorithms available, a classification is mandatory. In [78] sequential mining algorithms are classified according to their database layout, data structure, search strategy, enumeration, optimizations and number of database scans while [42] classifies them according to search strategy and frequency computation. We take Zaki's classification scheme without enumeration and optimizations. By enumeration Zaki means whether all or maximal item sets are output. In this section, we consider only those algorithms that determine all item sets.

In Table 2.1 and Figure 2.2 we present a classification of sequential algorithms based on [78] and [42]. We extend Zaki's classification by considering the type of frequency computation and we exclude optimizations. We also add the FP-Growth algorithm which is not available in Zaki's classification. The properties of algorithms are sorted in order of decreasing distinctive significance.

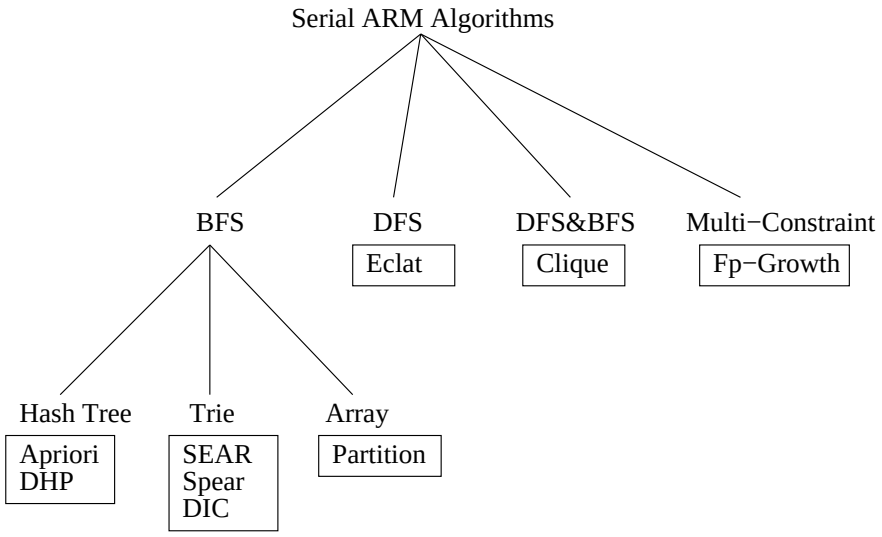

Figure 2.2: Classification of sequential association rule mining algorithms.

Following are the properties in our taxonomy.

Search: The search strategy used. In uninformed search, two well known strategies are Breadth First Search (BFS) and Depth First Search (DFS) [65]. BFS is the most common strategy employed by frequency mining problems as it may be said to be tracking the frequency information more manageable. In more recent algorithms other search strategies have been adapted.

Structure: The data structure used for storing candidate patterns and frequency information. The efficiency of a frequency mining algorithm is much dependent on the data structure used. The data structures typically facilitate a fast set (of sets) implementation with a way to determine counts of patterns in an efficient manner.

Scans: The number of database scans the algorithm must perform. k denotes the length of maximum length frequent pattern in transaction set. When there are long frequent patterns algorithms that require k scans may not be feasible since the databases are very large.

Layout: The database layout assumed by the algorithm. Two common formats

are horizontal in which each transaction's transaction's *tid* is stored alongside items of the transaction and vertical in which a list of all *tid*s is stored for each item [78]. Algorithms that use horizontal layout use complex intermediate structures to count occurrences while in vertical layout they store only *tidlist* arrays and compute their intersections [78].

We will next summarize the APRIORI algorithm which is responsible for much of the interest in association rule and frequency mining. Subsequently, we will investigate algorithms which use summary structures for representing relevant information and then we will focus on the leading frequency mining algorithm FP-GROWTH [35] which uses compressed structures.

The reader is referred to [78] and [42] for more information on sequential frequency mining algorithms.

2.2.3.1 Apriori Algorithm

ARIORI (Algorithm 1) [7] employs BFS and uses a hash tree structure to count candidate item sets efficiently. The algorithm generates candidate item sets (patterns) of length k from $k-1$ length item sets. Then, the patterns which have an infrequent sub pattern are pruned. According to Lemma 1, the generated candidate set contains all frequent k length item sets. Following that, the whole transaction database is scanned to determine frequent item sets among the candidates [78]. For determining frequent items in a fast manner, the algorithm uses a hash tree to store candidate itemsets.[8]

2.2.3.2 Compact Structures

Compact data structures have been used for efficient storage and query/update of candidate item sets in frequency mining algorithms. As illustrated in Table 2.1, SEAR [60], SPEAR [60], and DIC[14] use tries[9] while FP-GROWTH [35] uses

[8] A hash tree has item sets at the leaves and hash tables at internal nodes [78].
[9] Also known as prefix trees.

Algorithm 1 APRIORI(T, ϵ)

```
1: L_1 ← {large 1-itemsets}
2: k ← 2
3: while L_{k-1} ≠ ∅ do
4:    C_k ← GENERATE(L_k − 1)
5:    for all transactions t ∈ T do
6:       C_t ← SUBSET(C_k, t)
7:       for all candidates c ∈ C_t do
8:          count[c] ← count[c] + 1
9:       end for
10:   end for
11:   L_k ← {c ∈ C_k | count[c] ≥ ε}
12:   k ← k + 1
13: end while
14: return ⋃_k L_k
```

FP-Tree which is an enhanced trie structure.

Using concise structures can reduce both running time and size requirements of an algorithm. Tries are well known structures that are widely used for storing strings and have decent query/update performance. The algorithms mentioned exploit this property of the data structure for better performance. Tries are also efficient in storage. A large number of strings can be stored in this dictionary type which would not otherwise fit into main memory. For frequency mining algorithms both properties are critical as our goal is to achieve efficient and scalable algorithms. In particular, the scalability of these structures are unmatched [35] as they allow an algorithm to track the frequency information of the candidate patterns for very large databases. The FP-Tree structure in FP-GROWTH allows the algorithm to maintain *all* frequency information in the main memory obtained from two database passes. Using the FP-Tree structure has also resulted in novel search strategies.

A notable work on compact structures is [74] in which a binary-trie based summary structure for representing transaction sets is proposed. The trie is further compressed using Patricia tries. Although significant savings in storage and improvements in query time are reported, the effectiveness of the scheme in a frequency mining algorithm remains to be seen.

In this thesis, the FP-Growth algorithm is utilized as the sequential mining algorithm of choice. A closer analysis of FP-Growth is in order.

2.2.3.3 FP-Growth Algorithm

The FP-Growth algorithm uses the frequent pattern tree (FP-Tree) structure. FP-Tree is an improved trie structure such that each itemset is stored as a string in the trie along with its frequency. At each node of the trie, *item*, *count* and *next* fields are stored. The *item*s of the path from the root of the trie to a node constitute the item set stored at the node and the *count* is the frequency of this item set. The node link *next* is a pointer to the next node with the same *item* in the FP-Tree. Field *parent* holds a pointer to the parent node, *null* for root. Additionally, we maintain a header table which stores heads of node links accessing the linked list that spans all same items. FP-Tree stores only frequent items. At the root of the trie is a *null* item, and strings are inserted in the trie by sorting item sets in a unique[10] decreasing frequency order [35].

Table 2.2 shows a sample transaction set and frequent items in descending frequency order. Figure 2.3 illustrates the FP-Tree of sample transaction set in Table 2.2. As shown in [35], FP-Tree carries complete information required for frequency mining and in a compact manner; the height of the tree is bounded by maximal number of frequent items in a transaction. Make-FP-Tree (Algorithm 2) constructs an FP-Tree from a given transaction set T and support threshold ϵ as described.

Transaction	Ordered Frequent Items
$t_1 = \{f, a, c, d, g, i, m, p\}$	$\{f, c, a, m, p\}$
$t_2 = \{a, b, c, f, l, m, o\}$	$\{f, c, a, b, m\}$
$t_3 = \{b, f, h, j, o\}$	$\{f, b\}$
$t_4 = \{b, c, k, s, p\}$	$\{c, b, p\}$
$t_5 = \{a, f, c, e, l, p, m, n\}$	$\{f, c, a, m, p\}$

Table 2.2: A sample Transaction Set

[10] All strings must be inserted in the same order; the order of items with the same frequency must be the same.

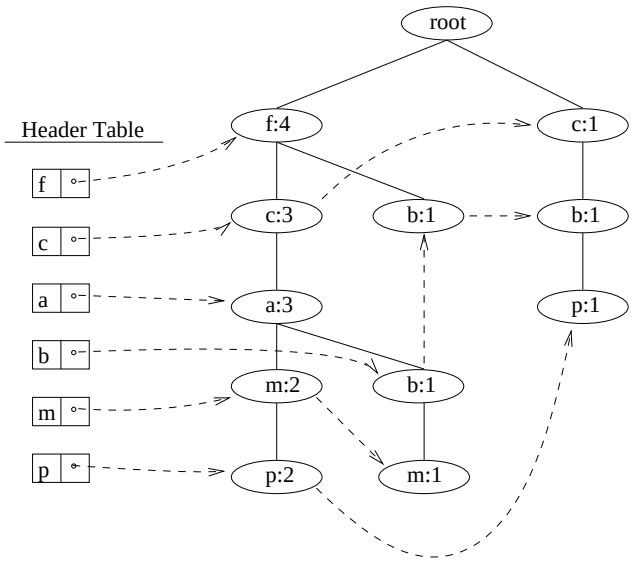

Figure 2.3: An FP-Tree Structure.

In Algorithm 4 we describe FP-Growth which has innovative features such as:

1. Novel search strategy
2. Effective use of a summary structure
3. Two database passes

FP-Growth turns the frequency k length pattern mining problem into "a sequence of k frequent 1-item set mining problems via a set of conditional pattern bases" [35]. It is claimed that with FP-Growth there is "no need to generate any combinations of candidate sets in the entire mining process"[11]. With an FP-Tree *Tree* given as input the algorithm generates all frequent patterns. There are two points in the algorithm that should be explained: the single path case and conditional pattern bases. If an FP-Tree has only a single path, then an optimization is to consider all combinations of items in the path.[12] Otherwise,

[11] This is not an accurate picture as we will examine.
[12] Single path case is the basis of recursion in FP-Growth.

the algorithm constructs for each item a_i in the header table a *conditional pattern base* and an FP-Tree $Tree_\beta$ based on this structure for recursive frequency mining. Conditional pattern base is simply a compact representation of a derivative database in which only a_i and its prefix paths in the original $Tree$ occur. Consider path $< f : 4, c : 3, a : 3, m : 2, p : 2 >$ in $Tree$. For mining patterns including m in this path, we need to consider only the prefix path of m since the nodes after m will be mined elsewhere.[13] Considering the prefix path $< f : 4, c : 3, a : 3 >$ any pattern including m can have frequency equal to the frequency of m, therefore we may adjust the frequencies in the prefix path as $< f : 2, c : 2, a : 2 >$ which is called a *transformed prefix path* [35]. The set of transformed prefix paths of a_i forms a small database of patterns which co-occur with a_i and thus contains complete information required for mining patterns including a_i. Therefore, recursively mining conditional pattern bases for all a_i in $Tree$ is equivalent to mining $Tree$.[14] $Tree_\beta$ is simply the FP-Tree of the conditional pattern base.

FP-Growth is indeed remarkable with its unique divide and conquer approach. Nevertheless, it does generate candidates contrary to the title of "...Without Candidate Generation" [35]. The conditional pattern base is clearly a set of candidates among which only some of them turn out to be frequent. The main innovation however remains intact: FP-Growth takes advantage of a tailored data structure to solve the frequency mining problem with a divide-and-conquer method and with demonstrated efficiency and scalability. Besides, the conditional pattern base is guaranteed to be smaller than the original tree, which is a desirable property. An important distinction of this algorithm is that, when examined within our taxonomy of algorithms, it employs a unique search strategy. When the item sets tested are considered, it is seen that this algorithm is neither DFS nor BFS. The classification for FP-Growth in Figure 3 of [42] may be misleading. As Hipp later writes in the same paper, "FP-Growth does not follow the nodes of the tree ..., but directly descends to *some part* of the itemsets in the search space". In fact, the part is so well defined that it would be unjust to classify FP-Growth as conducting a DFS. It does not even start with item sets of small length and proceed to longer item sets. Rather, it considers a

[13] In this case only p.

[14] Which is equivalent to mining the complete DB.

set of patterns at the same time by taking advantage of the data structure. This search strategy may be called *Multi-Constraint*, however it is hard to classify FP-Growth in the context of traditional uninformed search algorithms.

Algorithm 2 Make-FP-Tree(DB, ϵ)

1: Compute F and $f(x)$ where $x \in F$
2: Sort F in frequency decreasing order as L
3: Create root of an FP-Tree T with label "null"
4: **for all** transaction $t_i \in T$ **do**
5: Sort frequent items in t_i according to L. Let sorted list be $[p|P]$ where p is the head of the list and P the rest.
6: Insert-Trie($[p|P]$)
7: **end for**

Algorithm 3 Insert-Trie($[p|P], T$)

1: **if** T has a child N such that $item[N] = item[p]$ **then**
2: $count[N] \leftarrow count[N] + 1$
3: **else**
4: Create new node N with $count = 1$, $parent$ linked to T and node-link linked to nodes with the same item via $next$
5: **end if**
6: **if** $P \neq \emptyset$ **then**
7: Insert-Trie(P, N)
8: **end if**

2.3 Parallel Frequency Mining

As the transaction sets are large in both the number of items and transactions, frequency mining algorithms have a strict requirement for scalability. High performance computing has become an essential element of data mining as very large data is becoming available in both scientific and business applications. As the sensor data and simulation results accumulate, scientists need better means to analyze them for discovering new knowledge [53, 25].

We must depend on parallel systems to analyze the massive volumes of data in frequency mining problem [5, 78]. Zaki points out to the challenges for obtaining good performance: communication minimization, load balancing, suitable data

Algorithm 4 FP-GROWTH($Tree, \alpha$)

```
1: if Tree contains a single path P then
2:   for all combination β of the nodes in path P do
3:     generate pattern β ∪ α with support minimum support of nodes in β
4:   end for
5: else
6:   for all a_i in header table of Tree do
7:     generate pattern β ← a_i ∪ α with support = support[a_i]
8:     construct β's conditional pattern base and then β's conditional FP-Tree
       Tree_β
9:     if Tree_β ≠ ∅ then
10:      FP-GROWTH(Tree_β, β)
11:    end if
12:  end for
13: end if
```

representation and decomposition and disk I/O minimization. In addition to the requirements of a typical parallel algorithm, a parallel mining algorithm must consider parallelism in disk operations. Zaki identifies three design dimensions: parallel architecture, type of parallelism and load balancing strategy. Following is a brief discussion of design options:

Architecture: Although programming for SMP systems is easier, only $NUMA$ systems can liberate the algorithm from the bus bottleneck. This thesis is concerned exclusively with shared-nothing type of architectures in which compute nodes with local CPU, RAM and disk communicate over an interconnection network.

Type of Parallelism: Task and data parallelism are the forthcoming paradigms in parallel algorithm design. In frequency mining problem, data parallelism corresponds to distributing the data among p processors and discovering frequent patterns collectively. Task parallelism is the division of work, with a suitable definition of atomic task, among the processors. Each processor has access to all data or the part of data required for completion of the task which can be accomplished by selective replication or explicit communication of local portions [78]. Hybrid strategies are also possible.

Load Balancing Strategy: Static load balancing uses a heuristic cost function to partition data/tasks among processor and does no further movement, while dynamic load balancing seeks to migrate load off heavily loaded processors.

2.3.1 Overview of Parallel Mining Algorithms

For designing a new algorithm, it is necessary to understand the design of the algorithms proposed previously. We will embark on the significant contributions to parallel frequency mining. The algorithms all start with equal disjoint portions of the transaction set at each processor. We will consider APRIORI based parallel algorithms and Zaki's parallel algorithms PAR-ECLAT and PAR-CLIQUE which are arguably the most advanced parallel mining algorithms proposed.

Upon the success of APRIORI several parallelizations of it were made. These algorithms logically work in the same way as Apriori does, that is with a BFS, frequency counting and candidate pruning, however using differing parallel methods.

2.3.2 Apriori Based Parallel Algorithms

In [5], the designers of APRIORI suggest 3 parallelizations of it. COUNT-DISTRIBUTION minimizes communication and DATA-DISTRIBUTION tries to make use of collective system memory while CANDIDATE-DISTRIBUTION reduces communication costs by taking into account the task-data dependencies and redistributing data. Each algorithm parallelizes the iteration which is comprised of a concurrent computation phase and a collective communication phase.[15]

In COUNT-DISTRIBUTION, each processor computes all C_k at the beginning of the iteration and makes a local passes determining local counts. Then, the global counts are computed with a global sum-reduction to all processors. Each

[15]Except in the largely asynchronous phase of CANDIDATE-DISTRIBUTION.

processor computes all L_k from global counts.

The objective of DATA-DISTRIBUTION is to exploit total system memory better. Each processor generates $|C_k|/p$ of candidates. The algorithm is communication-happy as each processor must scan the entire database to determine counts of the candidate sets it owns. As the authors indicate, this algorithm requires machines with very fast communication.

CANDIDATE-DISTRIBUTION is the most sophisticated of three algorithms as it partitions both data and candidate sets permitting independent mining of parts. This was designed due to the fact that *no* load balancing is done in Count and Data Distribution, a processor has to wait for all other processors at each iteration's synchronization step. Up to an intermediate iteration l either of the previous algorithms is used. At iteration l, the algorithm partitions the item set into p parts such that each processor can compute global counts of assigned item sets independently while attaining load balance. At the end of the iteration the database is redistributed according to item set partitioning. The partitioning algorithm considers a lexicographical ordering of L_k and L_{k-1}. The item sets X in L_{k-1} with the same $k-1$ length prefixes as item sets Y in L_k are sufficient to compute the candidates and results of Y [6]. The load balanced partition of item sets is achieved by distributing the connected components in a weighted dependency graph which represents candidate generation dependencies among $k-1$ length prefixes of L_k. After iteration l, each processor proceeds independently only using pruning information from other processors as it becomes available [78].

Among three algorithms COUNT-DISTRIBUTION performs best, in a rather unexpected way since CANDIDATE-DISTRIBUTION is the most advanced algorithm.

2.3.3 Parallel algorithms based on Eclat and Clique

Parallel versions of ECLAT and CLIQUE have achieved tremendous success. Zaki employs two improved item set partitioning schemes for task parallelism in the

design of these algorithms [78].

Equivalence class clustering uses the same idea as the partitioning CANDIDATE-DISTRIBUTION described in the previous section. Here we shall dwell on this scheme with an example from [6]. L_k in this scheme have their item sets represented as *lexicographically ordered strings*. Let $L_3 = \{ABC, ABD, ABE, ACD, ACE, BCD, BCE, BDE, CDE\}$, $L_4 = \{ABCD, ABCDE, ABDE, ACDE, BCDE\}$, $L_5 = \{ABCDE\}$. Consider a part in L_3 $\alpha = \{ABC, ABD, ABE\}$ with the common prefix AB. Computation of candidates $ABCD, ABCDE, ABDE, ABCDE$ with the same prefix depend only on items in α. Depending on this property, each set of items with the same $k-1$ length prefix in L_k is identified as a cluster. In the example of [78], let $L_2 = \{12, 13, 14, 15, 16, 17, 18, 23, 25, 27, 28, 34, 35, 36, 45, 46, 56, 58, 68, 78\}$. One of the clusters, with the prefix 2 would be $\alpha_2 = \{23, 25, 27, 28\}$.

Maximal uniform hypergraph clique clustering obtains a more accurate partitioning by making use of a graph theoretical observation. Let us interpret L_k as a k-uniform hypergraph in which vertices are items and edges are item sets. In this hypergraph the set of maximal cliques C contains all maximal frequent itemsets [78]. In other words, C gives us a good estimate of maximal frequent itemsets, containing all maximal frequent item sets and infrequent ones and $|C|$ gives us an upper bound on the number of maximal frequent patterns. Clusters are derived in the same way as in equivalence clustering, for each unique $k-1$ length prefix in L_k. In the example L_2, the cluster for prefix 2 is identified as maximal cliques $\{235, 258, 278\}$.

The clustering schemes obtain k clusters where $k > p$. The k clusters must be assigned to processors so as to maintain load balance and minimal communication. For this purpose, each cluster's load must be weighed. A cluster α_i is given weight $\binom{|\alpha_i|}{2}$ which estimates the computational load of frequency mining within the cluster. The clusters are binned to processors with a greedy heuristic.

It is assumed that L_2 has been computed and tidlists are partitioned in a preprocessing step. Parallel algorithms in Zaki's work are comprised of three phases:

1. Item set clustering and scheduling of clusters among processors
2. Redistribution of vertical database according to schedule
3. Independent computation of frequent patterns

In all algorithms L_2 is used for partitioning so that redistribution can be made as soon as possible. Independent mining is performed by either a BFS or hybrid DFS/BFS search strategy.

Zaki shows as the advantages of his algorithms its distribution of data, decoupling of the processors in the beginning, vertical database layout and fast intersections avoiding structure overhead. In the experiments, it is seen that the more advanced maximal clique clustering does in fact improve upon equivalence class clustering by providing more exact load balancing information. The most important contribution of the work in question is the novel item set partitioning scheme.

2.3.4 Other Studies and Remarks

In an interesting benchmark study [12], it is noted that the results confirmed on artificial sets do not carry over to real-world data sets. In particular it is said that the choice of algorithm does not seem to matter for a feasible number of rules. Evaluating performance on real world applications is certainly valuable for any data mining algorithm and we should expect the availability of large real-world data sets for evaluating the effectiveness of parallel mining algorithms.

A tight upper bound on the number of maximal candidate patterns given L_k is presented in [29]. The bound is derived from a combinatorial result by Kruskal and Katona. It is also shown experimentally that the estimates are fairly accurate in mining of artificial data sets. It is suggested that the results may be used in optimization of mining algorithms. In the author's opinion, this theoretical work may be useful for improving load balance in parallel mining algorithms.

In [58], a parallel implementation of FP-Growth is presented. The authors report favorable speedups on a distributed shared-memory SGI Origin machine. Note that Zaki's experiments have been conducted on a shared-nothing DEC Alpha cluster with DEC Memory Channel which is also a custom fast network.

Zaki's survey of association rule mining algorithms not only classifies sequential and parallel mining algorithms according to their design choices but also gives a list of open problems in parallel frequency mining: high dimensionality, large size, data location, data skew, rule discovery, parallel system software, and generalizations of rules [78]. In [54], Maniatty and Zaki analyze the hardware and software requirements of parallel data mining, especially databases, file systems and parallel I/O techniques.

2.4 Graph Partitioning

Graph partitioning is a cardinal problem that has extensive applications in various areas. The problem is concerned with splitting a graph into k parts which are approximately equivalent in size. The partition is separated by a set of either *edges* or *vertices*. There are many parameters and criterion for the quality of the partition. For instance, a specific application may require that a two way partition is separated by a minimum number of edges. As Liu [51] notes, the fundamental importance of the problem is due to its strong connection to the divide-and-conquer paradigm. Many search problems with detailed structure and global goals lend themselves to graceful modeling with the partitioning problem such as task scheduling, VLSI design, and scientific computing.

2.4.1 Application Domains

Not surprisingly, the first problems that were recognized to be effectively modeled by graph partitioning come from the realm of computers and engineering. Kernighan and Lin describe the problems of placement of electronic components,

and optimizing paging properties of programs as immediate examples [49]. Since then, graph partitioning has proven to be a versatile tool in VLSI design and many software problems. Following are examples to common applications of graph partitioning.

- The problem of task scheduling for parallel computing suits beautifully to graph partitioning problem. Since one would wish to reduce the expensive communication between tasks, she can model the tasks as vertices and the communication volumes as edges of an ordinary graph. A partition of the graph into the number of processors which minimize the total communication volume would then give an exact solution to her needs.

- The solution of sparse symmetric matrices, for instance in linear programming, is best described as a graph partitioning problem in which the n vertices of the graph correspond to the n columns of the matrix and the edges represent the non-zero elements of the matrix [40]. Although there are algorithms which operate directly on matrices, the graph partitioning based algorithms have been shown to be more powerful [33].[16]

2.4.2 Graph Partitioning Methods

Because of its spectacular generality, several methods to solve the partitioning problem have been developed for more than three decades. Since the problem is known to be NP-complete, any solution has to be based on heuristics. Currently, there are various heuristic methods which give partitions of high quality within reasonable time-space bounds [33].

One of the first breakthroughs in graph partitioning problem is due to Kernighan and Lin [49] in their now classical paper. To date, heuristic procedures usually employ a form of their approach. In that paper, they rule a number of false heuristics out.[17]

[16]Note that such a matrix and a graph are numerically identical structures, however the graph theoretic approach yields more elegant and effective results.

[17]They give a measure of how big the search space is, they state that for a 32x32 matrix the

We should first distinguish between two families of algorithms:

Single-Level Algorithms These algorithms operate on the graph as a whole. That is they do not derive an equivalent structure instead of the graph (or matrix). The terms in their formulation usually correspond to sets or elements of vertices (edges) of the graph. They work by finding an initial partition and then improving it according to a heuristic.

Multi-Level Algorithms Multilevel algorithms first reduce the size of the graph by collapsing vertices (and edges as accordingly), and then partition this condense representation of the fine graph. The next step is to project the partition of the coarse graph to the original one. The more effective of these algorithms refine the partition at each level while projecting using a heuristic. All three steps mentioned can be performed in differing manners, thus making the overall algorithms distinct. A survey and evaluation of multilevel methods is available in [48].

The reader is referred to [48, 40] for a survey of partitioning algorithms.

2.4.3 Problem Description

Definition 6. ***K-way Graph Partitioning Problem:*** *Let a graph* $G = (V, E)$ *where* $|V| = n$ *be partitioned into subsets* $V_1, V_2, \ldots, V_k$ *such that for all* $1 \leq i \leq j \leq k$, $V_i \cap V_j = \emptyset$ *and* $|V_i| \cong n/k$ *and* $\bigcup_i V_i = V$ *where edge cut* $E_c = \{(u,v)|(u,v) \in E \wedge \exists i,j(i \neq j \wedge u \in V_i \wedge v \in V_j)\}$ *and* $|E_c|$ *is minimal. The partitioning is denoted as* $\Pi_{GPES}(G) = V_1, V_2, V_3, \ldots, V_k$.

The formal definition also introduces E_c the edge cut. Indeed, we can define graph partitioning by edge separator. In other words, removal of the edge separator E_c partitions the connected graph G into k roughly equal connected components.

chance of a random trial to hit the optimal peak is about 10^{-7}.

As previously stated, partitioning of a graph by edge separator is distinct from partitioning of a graph by vertex separator.

Definition 7. ***K-way Graph Partitioning by Vertex Separator Problem:*** *The k-way partition of a connected graph by vertex separator is defined as follows. Let a connected graph* $G = (V, E)$ *where* $|V| = n$ *be partitioned into subsets* $V_1, V_2, \ldots, V_k$ *and* V_s *such that for all* $1 \leq i \leq j \leq k$, $V_i \cap V_j = \emptyset$ *and* $|V_i| \cong \frac{n-|V_s|}{k}$ *and* $\bigcup_i V_i = V - V_s$ *and* $\forall u, v(u \in V_i \wedge v \in V_j) \rightarrow (u, v) \notin E$ $|V_s|$ *is minimal. The partitioning is denoted as* $\Pi_{GPVS}(G) = (\{V_1, V_2, V_3, \ldots, V_k\}, S)$.

We will denote V_s with S in the rest of the thesis and in 2-way graph partitioning problem by vertex separator, which is also known as graph bisection by vertex separator problem, V_1 and V_2 will be denoted with A and B respectively. This special case will be denoted as $\Pi_{GPVS}(G) = (\{A, B\}, S)$.

An immediate extension to these problems is the addition of vertex and edge weights. We will denote vertex weights by $w(x)$ and edge weights by $w(u, v)$. In general, if there are no weights in the graph these quantities may be safely assumed to be 1 for all existent vertices and edges. Note that addition of the weight notion changes the problem description in that:

- In the edge separator problem we minimize $\sum_{(u,v)\in E_c} w(u, v)$ and consider the total vertex weight of partitions for the balance constraint.
- In the vertex separator problem we minimize $\sum_{u\in S} w(u)$ and consider the total vertex weight of partitions for the balance constraint.

Chapter 3

Transaction Set Partitioning

In this chapter, we describe our theoretical contributions which will be developed into a parallel algorithm in Chapter 4. We introduce a graph based partitioning scheme that can be used to divide the frequency mining task in a top-down fashion. The method used operates on the G_{F_2} graph from which a graph partitioning by vertex separator (GPVS) is mapped to a two-way partitioning on the transaction set. The two parts obtained can be mined independently and therefore can be utilized for concurrency. In order for this property to hold, there is an amount of replication dictated by the separator in G_{F_2} which is minimized by the graph partitioning algorithm.

In the following sections, we first present the objective of transaction set partitioning. Then, we expound on the theoretical content of two-way transaction set partitioning. The last section extends 2-way partitioning to k-way partitioning.

3.1 Objective

The objective of transaction set partitioning is to divide a transaction set such that each part can be mined independently while not inflating the data prohibitively. Once such a partitioning is obtained, an algorithm such as [82] can

be designed which consists of a redistribution phase and a following local mining phase as described in Section 2.3.

In general, a parallel algorithm can be said to exploit data parallelism or task parallelism. Our method investigates a partitioning on the data since the databases involved are usually large, i.e. on the order of 100MB and more. Therefore, communication becomes an important obstacle to the scalability of parallel algorithm as size increases. An approach which replicates a big portion of the database would not yield a practical algorithm, especially on clusters without custom network hardware.[1]

We show that a graph partitioning by vertex separator is sufficient to designate such a partition on the transaction set. Our work assumes that the G_{F_2} graph is sparse, since graph partitioning may not be feasible on graphs with large connectivity.

3.2 Transaction Set Partitioning

Definition 8. *$G_{F_2} = (F, F_2)$ is an undirected graph in which each node $u \in F$ is a frequent item and each edge $\{u, v\} \in F_2$ is a frequent pattern.*

The G_{F_2} graph of sample transaction set of Table 1.1 is illustrated in Figure 3.1.

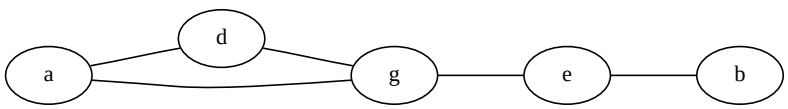

Figure 3.1: G_{F_2} graph of transaction set in Table 1.1

This graph is relatively easy to compute with respect to the complexity of the whole mining task, and it tends to efficient parallel algorithms. G_{F_2} contains

[1]Such as the Beowulf class supercomputer on which we have made our experimental studies.

valuable information which can be used to predict certain properties of complete frequency mining. The maximal cliques in G_{F_2} give us the potentially maximal patterns, which in turn can be used to achieve task parallelism.

Our method, on the other hand, does not require finding maximal cliques. Instead, we perform a graph partitioning on G_{F_2} which allows us to define independent parts on the transaction set. The partitioning identifies lack of cliques among two sets of items rather than enumerating all cliques.

In Figure 3.2, a synthetic data set generated with the procedure described in [7] is plotted. The parameters are 1000 transactions, 1000 items, average transaction length of 8, 32 patterns, and an average pattern length of 3.

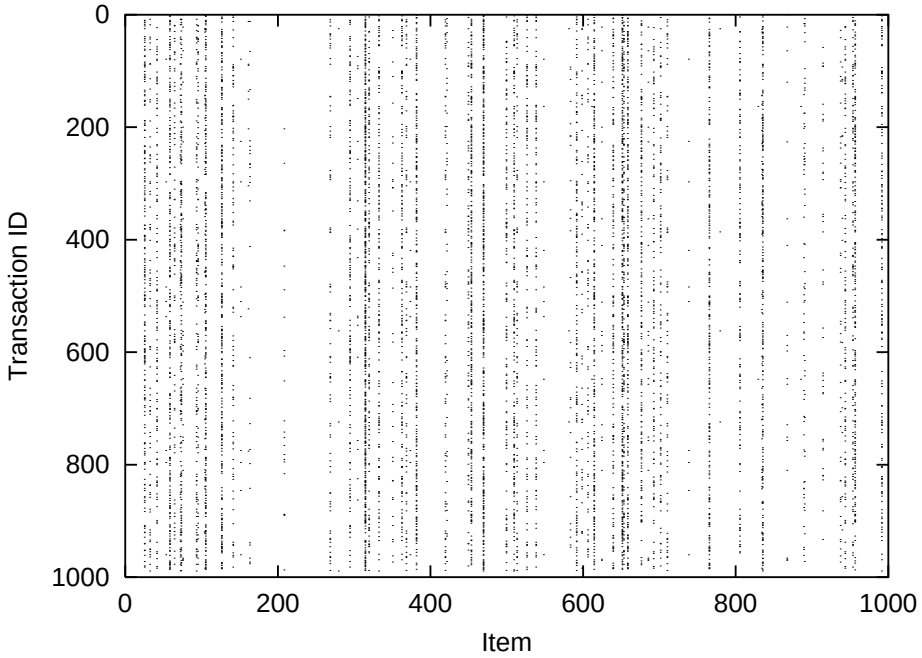

Figure 3.2: A synthetic data set with 1000 transactions and 1000 items containing 32 patterns

3.3 Two-way Partitioning of Transaction Database

Definition 9. *A transaction database projected from T over a set of items A is* $\pi_A(T) = \{X \cap A \mid X \in T\}$.

Definition 10. *We determine two-way transaction set partitioning* $\Pi_{TS}(T) = (T_1, T_2)$ *from* $\Pi_{GPVS}(G_{F_2}) = (\{A, B\}, S)$ *where,*

$$T_1 = \pi_{A \cup S}(T) \tag{3.1}$$

$$T_2 = \pi_{B \cup S}(T) \tag{3.2}$$

The partitioning is achieved by projecting each transaction in transaction set T into two parts. Intuitively, we project the intersection of transactions with $A \cup S$ into T_1 since there can be patterns contained in only A, or contained in A and S. Likewise for $B \cup S$ and T_2. The database has been divided into two, replicating those items in S to facilitate independent mining of parts. In the following text, we show that mining two partition databases result in complete frequency mining of the original transaction set T.

Figure 3.3 depicts the G_{F_2} graph of transaction set in Table 1.2. Π_{GPVS} of this graph is drawn in Figure 3.4 and the transaction set partition Π_{TS} corresponding to this GPVS is illustrated in Table 3.1.

Lemma 2. *If there is a frequent pattern P in T, then there is a corresponding clique in G_{F_2} with vertices labeled as items in P.*

Proof. Due to the downward closure property, a pattern P can be frequent if and only if all subsets of the pattern are frequent patterns. A frequent pattern $P \subseteq F$ contains $\binom{|P|}{2}$ subsets (sub patterns) with cardinality 2. Then, $\forall u, v \in P, \{u, v\}$ is a frequent pattern. By definition of G_{F_2}, each frequent pattern with cardinality 2 is an edge in G_{F_2}; hence $\forall u, v \in P, (u, v) \in F_2$. If there is an edge between every two vertices among a subset of vertices in a graph, it is called a clique K. Therefore, if there is a frequent pattern P, then there is a clique in G_{F_2} whose vertices are P. □

Lemma 3. *There is no clique in G_{F_2} with vertices in both A and B of GPVS (A, B, S) of G_{F_2}.*

Proof. Assume that there is a clique with a vertex u in A and a vertex v in B. This contradicts the fact that there are no edges between parts A and B in GPVS, thus there can be no such clique. □

Lemma 4. *There is no frequent pattern with items in both A and B of GPVS (A, B, S) of G_{F_2}.*

Proof. Since there can be no clique in G_{F_2} that contains items in both A and B, there can be no such frequent pattern. □

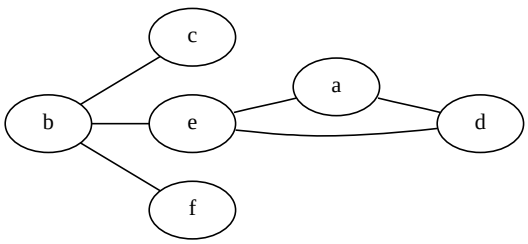

Figure 3.3: G_{F_2} graph of dataset in Table 1.2 with a support threshold of 4

Lemma 5. *A frequent pattern can be a subset of either A, B, $A \cup S$, $B \cup S$ or S.*

Proof. A clique K in G_{F_2} can have vertices in

1. Within either of A, B and S in GPVS of G_{F_2} since there can be arbitrary edges within parts and the separator S.
2. In $A \cup S$ and $B \cup S$ since there can be arbitrary edges between a part and the separator S.
3. There can be no clique with edges in both A and B.

Therefore, there can be frequent patterns drawn from A, B, $A \cup S$, $B \cup S$ or S. □

Lemma 6. *Independent discovery of frequent patterns in parts T_1 and T_2 result in discovery of all frequent patterns in T.*

Proof. $\forall$ frequent pattern P in T,

1. If $P \subset A$ or $P \subset A \cup S$, then $P \in$ pattern set of T_1.
2. If $P \subset B$ or $P \subset B \cup S$, then $P \in$ pattern set of T_2.
3. If $P \subset S$, then $P \in$ pattern set of T_1 and $P \in$ pattern set of T_2.

Therefore, every pattern is found in either part or both parts. □

In Figure 3.5 is the G_{F_2} of transaction set in Figure 3.2 with $\epsilon = 0.05\%$ (of number of transactions). GPVS of the same graph is given in Figure 3.6. The two-way transaction set partitioning corresponding to the vertex separator is illustrated in Figure 3.7.

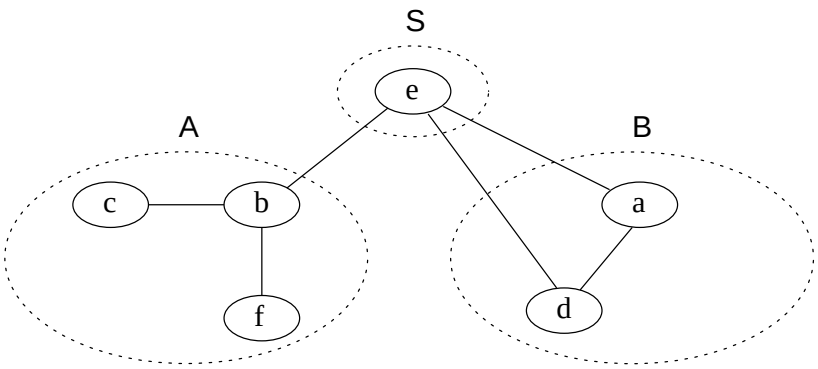

Figure 3.4: GPVS of G_{F_2} graph in Figure 3.3. Dashed lines enclose parts A,B and separator S.

Data replication in partition $\Pi_{TS} = (T_1, T_2)$ on T is determined by vertex separator S. By definition of two-way partitioning, for every transaction $X \in T$, $X \cap S$ is projected in both T_1 and T_2.

We shall now show the amount of data replication.

Lemma 7. *The amount of data replication is* $\sum_{\forall u \in S} f(u)$.

Proof. The frequency function f gives us how many times a given item occurs. Summation of $f(x)$ over a set of items yields how many transaction data exist in the projection of transaction set with respect to a given item set. Since S exists in both parts of the bipartition, the amount of data replication is the number of transaction data projected over S. □

Lemma 8. *GPVS of* G_{F_2} *with item frequencies as vertex weights minimizes the amount of replication.*

Proof. Graph bisection by vertex separator (of a weighted graph) will minimize the total weight of the separator as its objective. The graph partitioning minimizes data replication since the total weight of the separator is equal to amount of data replication.[2] □

a	b	c	d	e	f
	×	×			×
	×	×			
				×	×
	×			×	×
		×		×	
	×			×	
	×	×		×	×
	×	×		×	
	×	×			×

a	b	c	d	e	f
×					
×			×	×	
			×	×	
×			×	×	
×			×	×	
				×	
×			×	×	

Table 3.1: $\Pi_{TS}(T) = (T_1, T_2)$ of transaction set in Table 1.2

3.4 k-way Partitioning of Transaction Database

In this section, we describe means to extend 2-way partitioning to a k-way partitioning. We show that by plain recursion, it is possible to obtain as many parts

[2]Using an unweighted graph will not minimize data replication, but will result in substantial reduction in any event.

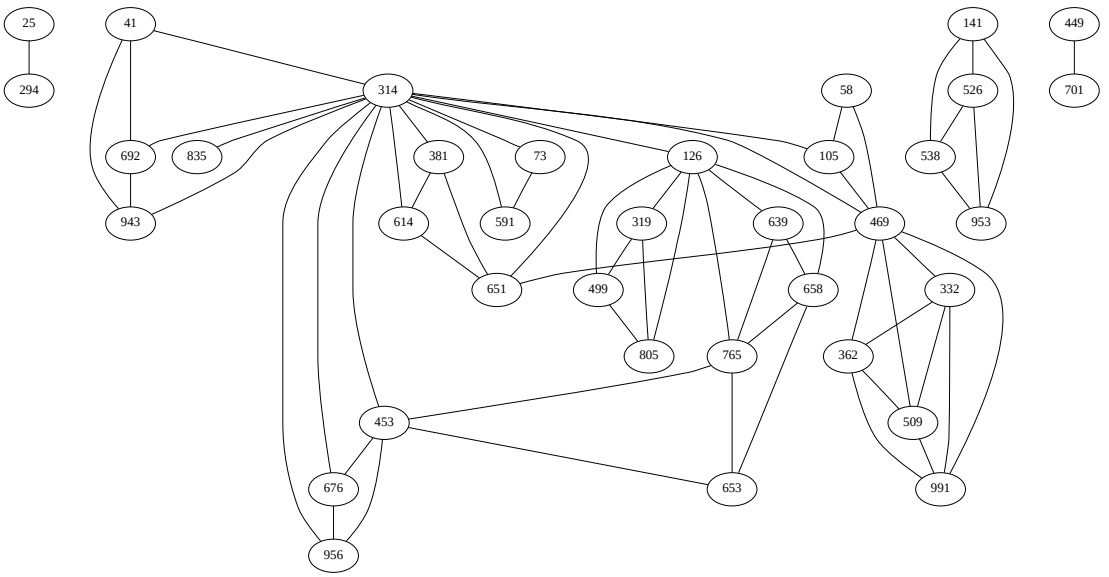

Figure 3.5: G_{F_2} graph of dataset in Figure 3.2 with a support threshold of 5%

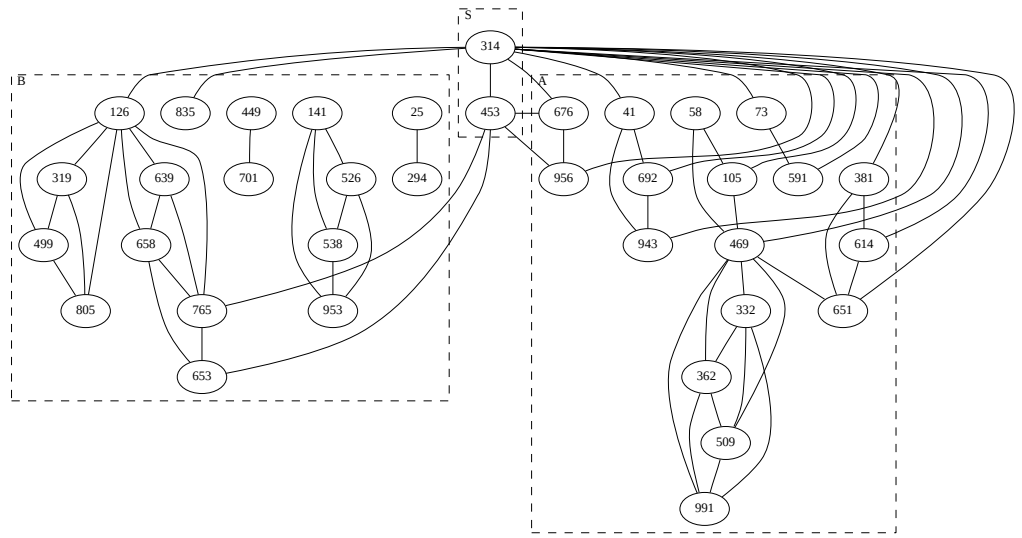

Figure 3.6: GPVS of G_{F_2} graph in Figure 3.5. Dashed lines enclose parts A,B and separator S.

as necessary.

Two-way transaction partitioning can be applied recursively to divide the two projected databases. Since the resulting projected databases of transaction set partitioning are transaction sets themselves, we can apply the same method to divide them further. It is a choice of algorithm to decide how many levels of recursion is needed to actually obtain such a k-way partition.

In order to partition the derived datasets, one must obtain the G_{F_2} of the two parts. This can be accomplished by simply running the same algorithm for the output transaction set, however this can be costly. There is no need to recompute F and G_{F_2} since they are already known.

Lemma 9. *F of a part $\pi_A(T)$ in $\Pi_{TS}(T)$ is A and G_{F_2} of $\pi_A(T)$ is subgraph of G_{F_2} of T induced by vertex set A.*

Proof. By definition of $\pi_A(T)$ and $\mathcal{F}$, frequent patterns in $\pi_A(T)$ part are identical to those patterns in $\{X \in \mathcal{F}(T) | X \subseteq A\}$. G_{F_2} graph's vertices are frequent items and its edges are frequent patterns with length 2, therefore the vertex induced subgraph of G_{F_2} by A is identical to G_{F_2} of $\pi_A(T)$. □

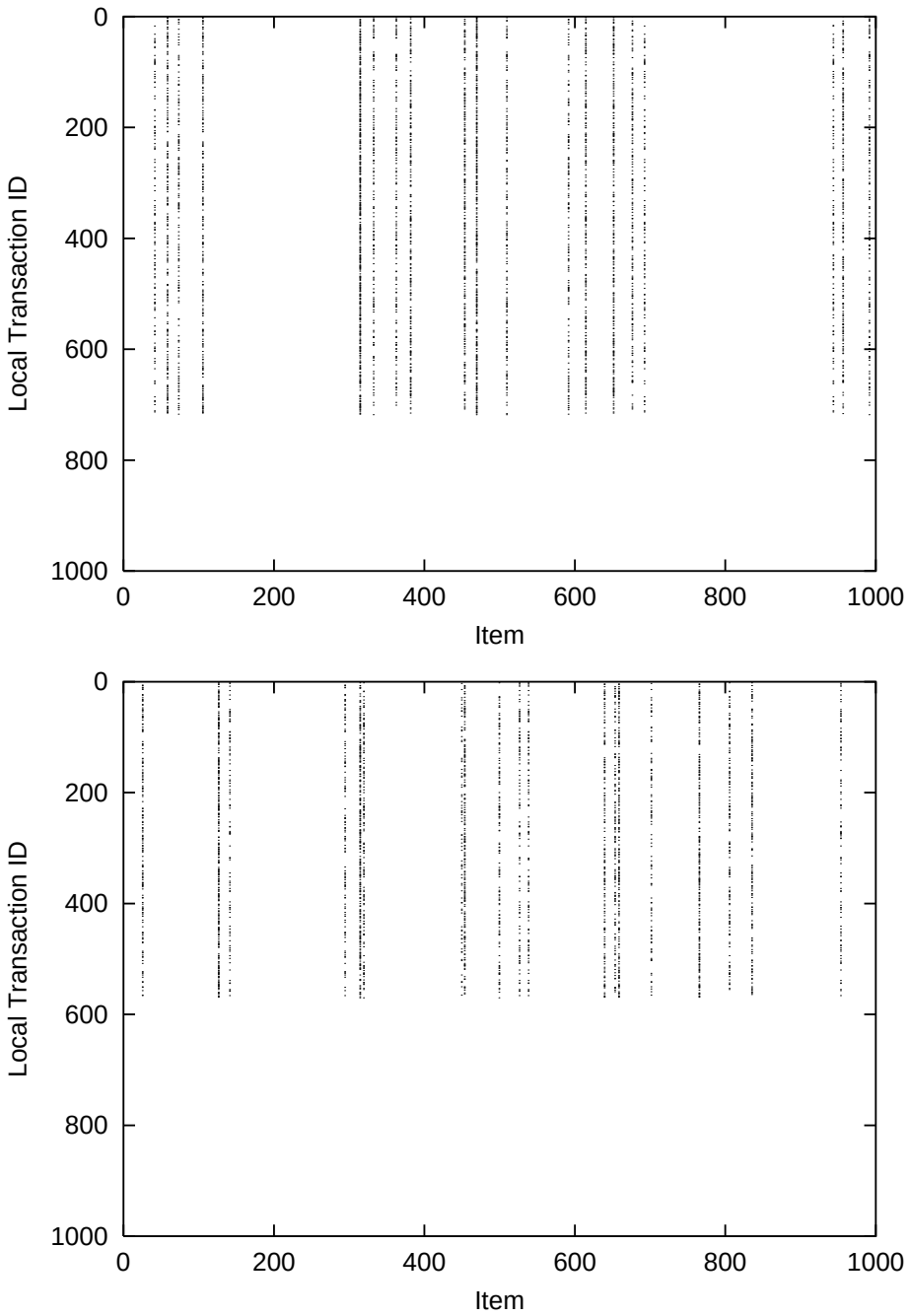

Figure 3.7: Two-way partitioning of transaction set in Figure 3.2.

Chapter 4

A Parallel Algorithm for Frequency Mining

In this chapter we present a parallel algorithm based on our theoretical observations of Chapter 3. Par-Freq algorithm computes a k-way partitioning in parallel and redistributes transaction database. Subsequently, mining proceeds at each processor with a local mining algorithm. The algorithm is independent of the serial frequency mining algorithm, although we employ a particular serial algorithm, namely FP-Growth, for use with Par-Freq.

In the following section, we give an overview of the algorithm. The remaining sections deal with each important step of the algorithm which are the computation of the G_{F_2} graph, k-way partitioning, optimizations and the serial algorithm used.

4.1 Overview

In the parallel algorithms of this thesis, p is the number of processors and *pid* is the running processor's identifier.

The algorithm takes as input a local transaction set T_i at processor i, a support threshold ϵ, and a serial mining procedure Mine-Freq. We assume that the

transaction database has been partitioned transaction-wise prior to execution of the mining algorithm. Each local transaction set T_i is disjoint and contains approximately $|T|/p$ transactions from the global transaction set T. However, we do not assume any specific form of partitioning; the partitioning may be random. It is sufficient that the local database sizes are approximately equal. Support threshold ϵ is the absolute threshold as in Equation 1.6. MINE-FREQ is any serial frequency mining algorithm that computes all frequent patterns, and their frequencies from a given transaction set.

PAR-FREQ computes both the set of all frequent patterns $\mathcal{F}$ and the frequency function f for the global transaction set T. Algorithm 5 conveys the top level algorithm. First F and G_{F_2} are computed which is the prerequisite for partitioning algorithm. k-way partitioning procedure KWAY-PARTITION computes a p-way partition recursively. The partitioning algorithm generates a local transaction set T_{part} at each processor which may be mined independently. In the last step of the algorithm, the serial mining algorithm MINE-FREQ is executed on T_{part} with support threshold ϵ. We additionally give G_{F_2} which the serial algorithm may utilize.

Algorithm 5 PAR-FREQ$(T_i, \epsilon,$ MINE-FREQ$)$

1: $G_{F_2} \leftarrow$ COMPUTE-F-$G_{F_2}(T_i, \epsilon)$
2: $T_{part} \leftarrow$ KWAY-PARTITION$(T_i, \epsilon, G_{F_2}, Processors)$
3: MINE-FREQ$(T_{part}, \epsilon, G_{F_2})$

4.2 Computation of F and G_{F_2}

We require the computation of F and G_{F_2} graph prior to partitioning. We compute F and G_{F_2} in parallel, using a simple algorithm similar to early parallel mining algorithms. To compute G_{F_2} in parallel, we first determine F in Algorithm 6.

Algorithm 7 computes F in parallel. We use an array C_l to hold the local counts. We count concurrently how many times each item occurs in T_i, in lines

Algorithm 6 COMPUTE-F-$G_{F_2}(T_i, \epsilon)$

$F \leftarrow$ COUNT-1-ITEMS(T_i, ϵ)
$G_{F_2} \leftarrow$ COUNT-2-ITEMS(T_i, ϵ, F)
return G_{F_2}

Algorithm 7 COUNT-1-ITEMS(T_i, ϵ)

1: $\triangleright$ C_l is an array to hold local counts
2: zero C_l
3: $F \leftarrow \emptyset$
4: **for all** $X \in T_i$ **do**
5: **for all** $u \in X$ **do**
6: $C_l[u] \leftarrow C_l[u] + 1$
7: **end for**
8: **end for**
9: $\triangleright$ At this stage, $C_l[i] = f(T_i, i)$
10: Reduce sum C_l to C at all processors $\triangleright$ C is the global count array
11: **for** $u \leftarrow 0$ to $|I| - 1$ **do**
12: **if** $C[u] \geq \epsilon$ **then**
13: $F \leftarrow F \cup \{u\}$
14: **end if**
15: **end for**
16: **return** F

4–8. After counting we have obtained the local frequency function for 1-items; $C_l[i] = f(T_i, i)$. Then we reduce the sums of local counts to each processor in order to compute the global counts for 1-items. Between lines 11-15 we perform the same operation at all processors to compute the set of items F which contains all items with a support of at least ϵ.

Algorithm 8 COUNT-2-ITEMS(T_i, ϵ, F)

```
 1: ▷ A is an upper triangular matrix to hold local counts
 2: A ← MAKE-UT-MATRIX(|I|)
 3: A ← 0
 4: for all X ∈ T_i do
 5:    for all {u, v} ⊂ X ∩ F do
 6:       if u > v then
 7:          swap u and v
 8:       end if
 9:       a_uv ← a_uv + 1
10:    end for
11: end for
12: C ← MAKE-UT-MATRIX(|I|)
13: Reduce sum A to C at processor 0    ▷ C is the global UT count matrix
14: if pid = 0 then
15:    V(G_F2) ← F
16:    for u ← 0 to |I| − 1 do
17:       for v ← u + 1 to |I| − 1 do
18:          if c_uv ≥ ε then
19:             E(G_F2) ← E(G_F2) ∪ (u, v)
20:             w(u, v) ← c_uv
21:          end if
22:       end for
23:    end for
24: end if
25: return G_F2
```

In Algorithm 8, we apply the method of Algorithm 7 for computing the frequency of item sets with length 2. Instead of an array we use an upper triangular matrix A to hold all 2-combinations of item set I. In this matrix, a_{ij} stores the local frequency function's value $f(T_i, \{i, j\})$. For each transaction X, we consider its intersection with F, $X \cap F$, since non-frequent items cannot form frequent patterns of length 2. We increment the counts of all 2-length subsets of $X \cap F$. Following that, we reduce adding the local counts in A to a global count matrix C

at root processor. From elements of C frequent 2 length item sets are determined and G_{F_2} graph is constructed.

4.3 Partitioning

Actual partitioning is achieved in Algorithm 9. This algorithm employs the k-way partitioning scheme outlined in Chapter 3 to compute p partitions. The algorithm first computes a 2-way partitioning. It divides the current set of processors in two parts using the transaction set partition. Consequently, the whole database is redistributed such that assigned processor sets obtain the parts determined in 2-way transaction set partitioning. We then compute which of the two partitions the running processor lies in.

In the recursive step, we call K-WAY-PARTITION recursively until each processor has been assigned a partition. Algorithm checks if it has reached the basis, whether it's processor group has only a single processor. Otherwise, it proceeds with recursion. Since only the root processor (the processor with least numbered *pid*) has G_{F_2} first we broadcast G_{F_2} to all processors in group $Processors$. Then, we project items from given F and G_{F_2} as described in Chapter 3 computing the vertex induced subgraph of G_{F_2} by vertex set P_i. We call K-WAY-PARTITION recursively at this stage on the present local transaction set partition T_{part} with the same absolute support threshold ϵ, its item set P_i comprised of the items assigned to this part, the vertex induced subgraph G'_{F_2} of the graph G_{F_2} induced by P_i, and the new processor group $Processors_i$.

4.3.1 Using GPVS to find a partition

In Algorithm 10, we determine the replicating partition over the items that will be used to redistribute transaction set. In lines 1-3 we compute a two-way GPVS of G_{F_2} graph.[1] In this algorithm, we use a serial GPVS algorithm however for better

[1] Also called a bisection by vertex separator

Algorithm 9 K-WAY-PARTITION$(T_i, \epsilon, G_{F_2}, Processors)$

1: $(P_1, P_2) \leftarrow$ 2-WAY-PARTITION(G_{F_2})
2: $(Processors_1, Processors_2) \leftarrow$ PARTITION-PROCESSORS$(P_1, P_2, Processors)$
3: $T_{part} \leftarrow$ REDISTRIBUTE-DB$(P_1, P_2, Processors_1, Processors_2)$
4: Determine i such that $pid \in Processor_i$
5: **if** $|Processors_i| = 1$ **then**
6: **return** T_{part}
7: **else**
8: Broadcast G_{F_2} from $min(Processors)$ to $Processors$
9: $G'_{F_2} \leftarrow$ VERTEX-INDUCED-SUBGRAPH(G_{F_2}, P_i)
10: **return** K-WAY-PARTITION$(T_{part}, \epsilon, G'_{F_2}, Processors_i)$
11: **end if**

scalability a parallel GPVS algorithm might be preferable. We do not specify which GPVS algorithm is used. There are several efficient serial algorithms that could be used in this step such as [48]. For using a parallel GPVS algorithm, the graph structure should also be distributed. In the algorithms we give, the graph is stored only in the root processor of every processor group.

After the partition is computed, the item sets P_1 and P_2 of two parts T_1 and T_2 respectively are determined by set union as described in Chapter 3.

Algorithm 10 2-WAY-PARTITION$(T_i, \epsilon, F, G_{F_2}, Processors)$

1: **if** $pid = min(Processors)$ **then**
2: $(A, B, S) \leftarrow GPVS(G_{F_2})$
3: **end if**
4: Broadcast (A, B, S) from $min(Processors)$ to $Processors$
5: $P_1 \leftarrow A \cup S$
6: $P_2 \leftarrow B \cup S$
7: **return** $(P1, P2)$

4.3.2 Load Balancing

The algorithm estimates computational load of each partition with respect to frequency mining for achieving load balance. Estimating the load is non-trivial, since we cannot know in advance how many patterns are present in the data. However, we can reason about the potential number of nodes in the search space

that the algorithm will need to traverse. Although every algorithm follows a different strategy for determining frequent patterns, a measure of the portion of the search space containing potentially frequent patterns gives us a good estimate as in [5, 82]. In this algorithm however, computing the maximal cliques in G_{F_2} graph like in [82] will incur additional and undesirable overhead since we are already performing $GPVS$ which is an expensive operation. Therefore we use a simpler function for load estimation.

For $load(X)$ function in Algorithm 10, we can use the datum size within the given item set in a manner resembling to [5].

$$load_1(X) = \Sigma_{u \in X} f(u) \tag{4.1}$$

Nevertheless, this approach is simplistic. Equation 4.1 is not irrelevant, however it does not take into account the actual complexity of the task. A better approximation which is inexpensive can be found in again [82]. The equation

$$load_2(X) = \Sigma_{u \in X} \binom{d(u)}{2} \tag{4.2}$$

is based on Zaki's itemset clustering where $d(u)$ is the degree of vertex u in G_{F_2} graph. This estimate approximates the number of potential frequent patterns of length 3 by calculating how many 2 combinations of patterns of length 2 exist. In practice, it may be preferable to employ a more empirical load estimate function. We have designed such a function for FP-Growth:

$$load_3(X) = max_{u \in X} f(u).\frac{\Sigma_{u \in X} d(u) log(d(u))}{|X|} \tag{4.3}$$

Equation 4.3 models an algorithm that has superlinear running time complexity in the number of patterns of length 2 an item participates in, assumed constant for each transaction. While $max_{u \in X} f(u)$ is an estimate of number of transactions in the part (actually lower bound), the second term is the average of $d(u)log(d(u))$ for items in X. This gives us a superlinear, but subquadratic function of the degree of an item in G_{F_2} for estimating the running time. Equation 4.3 is purely hypothetical, however it elicits the best load balancing performance we have observed. It is in fact a combination of certain aspects of the previous two load estimate functions.

Algorithm 11 PARTITION-PROCESSORS(P_1, P_2)

1: **if** $pid = min(Processors)$ **then**
2: $load_1 \leftarrow load(P_1)$
3: $load_2 \leftarrow load(P_2)$
4: **end if**
5: Broadcast $load_1$ and $load_2$ from $min(Processors)$ to $Processors$
6: $load_{total} \leftarrow load_1 + load_2$
7: $procs_1 \leftarrow |Processors|.load_1/load_{total}$
8: $procs_2 \leftarrow |Processors|.load_2/load_{total}$
9: **if** $procs_1 < procs_2$ **then**
10: swap $procs_1 \leftrightarrow procs_2$
11: **end if**
12: $Processors_1 \leftarrow \lfloor procs_1 \rfloor$ processors from $Processors$
13: $Processors_2 \leftarrow Processors - Processors_1$
14: **return** $(Processors_1, Processors_2)$

4.3.3 Redistribution of Transaction Set

Algorithm 12 rearranges the local transaction sets T_i such that processor group $Processors_1$ stores $\pi_{P_1}(T_i)$ and $Processors_2$ stores $\pi_{P_2}(T_i)$. d_1 and d_2 keep track of which processor will be targeted for sending the next scanned transaction in part 1 and 2 respectively. $D(i, j)$ is an array of message buffers that hold transaction sets to send from processor i to processor j. The local transaction set is scanned once to split the transactions to the two parts. For each transaction in local transaction set $X \in T_i$, we split X into intersections with P_1 and P_2 and add it to the send buffer with the destination processor attaining cyclic distribution of transactions per group. Having constructed the message buffers in this manner, we execute a collective all-to-all personalized communication for exchanging messages in $D(i, j)$. After this step, all processors construct their local part T_{part} from the message buffers and REDISTRIBUTE-DB returns T_{part}.

4.3.4 Computing Vertex Induced Subgraph

Computation of a vertex induced subgraph has a straightforward algorithm. Algorithm 14 traverses all adjacency lists of A, and adds those edges whose both

Algorithm 12 REDISTRIBUTE-DB($P_1, P_2, Processors_1, Processors_2$)

```
1: d_1 ← min(Processors_1)
2: d_2 ← min(Processors_2)
3: for all X ∈ T_i do
4:    X_1 ← X ∩ P_1
5:    if |X_1| > 2 then
6:       D(pid, d_1) ← D(pid, d_1) ∪ X_1
7:       d_1 ← CYCLE(d_1, Processors_1)
8:    end if
9:    X_2 ← X ∩ P_2
10:   if |X_2| > 2 then
11:      D(pid, d_2) ← D(pid, d_2) ∪ X_1
12:      d_2 ← CYCLE(d_2, Processors_2)
13:   end if
14: end for
15: AAPC D(i, j) holds messages from processor i to processor j
16: T_part ← ∅
17: for all i ∈ Processors do
18:    T_part ← T_part ∪ D(i, pid)
19: end for
20: return T_part
```

Algorithm 13 CYCLE(d, P)

```
1: d ← d + 1
2: if d > max(P) then
3:    d ← min(P)
4: end if
```

vertices are in A to newly constructed graph G'.

Algorithm 14 VERTEX-INDUCED-SUBGRAPH(G, A)

```
1: G' ← MAKE-GRAPH()
2: V(G') ← A
3: for all u ∈ A do
4:    for all v ∈ Adj(G, u) do
5:       E(G') ← E(G') ∪ (u, v)
6:    end for
7: end for
8: return G'
```

4.4 Optimizations

4.4.1 Using An F_2 Matrix of Rank $|F|$

Although the matrix sizes can be optimized to a rank of $|F|$ rather than $|I|$, the given algorithm does not incorporate it for clarity. We present such an optimization in Algorithm 24. Using the optimized version may be desirable since the original version will require $O(|I|^2)$ storage and communication bandwidth instead of $O(|F|^2)$. The optimized version uses two arrays to map indices between the count matrices and real item numbers.

4.4.2 Redistributing Transaction Set In A Single Pass

K-WAY-PARTITION (Algorithm 9) redistributes the transaction set every time a 2-way partition is computed. Nevertheless, it is possible to compute the k-way transaction set partition with a single redistribution of the database as implied by Lemma 9. The redundant redistributions can be prevented by a more sophisticated algorithm that computes the item set partition for each processor and a redistribution algorithm that can split the transactions into k parts rather than 2. The given algorithm is structured the way it has been for clarity[2], however

[2] In that it is a direct recursive application of 2-way partitioning.

this optimization is significant. The optimized versions of the original algorithms are given in Algorithm 15 and Algorithm 16.

K-WAY-PARTITION* delays the redistribution to the basis of recursion, and broadcasts all partitioning information to all processors for redistribution. The new redistribution algorithm REDISTRIBUTE-DB* generalizes the two way redistribution to k-way case.

Besides being faster, the optimized algorithms express the logic more clearly. We first compute a partitioning of item sets, and successively redistribute the database according to the item set partitioning. When the current part has only a single processor in K-WAY-PARTITION*, we will have found the item set P_i for part i. Each processor broadcasts P_i to every other processor stored in an array $Parts$. After the collective communication, $Parts[i]$ holds the item set for part i of the partitioning. This information is utilized to compute the projection for each part of Π_{TS} in REDISTRIBUTE-DB*. Note that we map $Parts[i]$ to processor i for we have determined the processor groups according to the load estimate function in parallel recursive computation of item set partition.

Algorithm 15 K-WAY-PARTITION*$(T_{local}, \epsilon, G_{F_2}, Processors)$

1: $(P_1, P_2) \leftarrow$ 2-WAY-PARTITION(G_{F_2})
2: $(Processors_1, Processors_2) \leftarrow$ PARTITION-PROCESSORS$(P_1, P_2, Processors)$
3: Determine i such that $pid \in Processor_i$
4: **if** $|Processors_i| = 1$ **then**
5: All-to-all broadcast P_i to $Parts$ ▷ $Parts$ is an array of sets
6: $T_{part} \leftarrow$ REDISTRIBUTE-DB*$(T_{local}, Parts)$
7: **return** T_{part}
8: **else**
9: Broadcast G_{F_2} from $min(Processors)$ to $Processors$
10: $G'_{F_2} \leftarrow$ VERTEX-INDUCED-SUBGRAPH(G_{F_2}, P_i)
11: **return** K-WAY-PARTITION$(T_{local}, \epsilon, G'_{F_2}, Processors_i)$
12: **end if**

Algorithm 16 REDISTRIBUTE-DB$^*(T_{local}, Parts)$

```
1: for all X ∈ T_local do
2:    for all i ∈ Processors do
3:       P_i ← Parts[i]
4:       Xi ← X ∩ P_i
5:       if |X_i| > 2 then
6:          D(pid, i) ← D(pid, i) ∪ X_i
7:       end if
8:    end for
9: end for
10: AAPC D(i, j) holds messages from processor i to processor j
11: T_part ← ∅
12: for all i ∈ Processors do
13:    T_part ← T_part ∪ D(i, pid)
14: end for
15: return T_part
```

4.4.3 Distributed Graph for G_{F_2} and Local Pruning

Local pruning which is proposed in [18] can reduce the number of candidate patterns and communication volume dramatically in CANDIDATE-DISTRIBUTION based parallel frequency mining algorithms. It has been used in the FDM algorithm for designing an algorithm suitable for a distributed system. The main observation in local pruning may be stated as follows. If a pattern is frequent it will have to be *locally* frequent at a processor, assuming that the local databases have equal number of transactions. In local pruning each processor discards locally infrequent candidate item sets. The union of all locally frequent candidate patterns will provide the correct set of frequent candidate patterns. For approximately equal initial partitioning a tolerance margin may be introduced to the pruning test.

In PAR-FREQ we can apply local pruning to optimize the computation of G_{F_2} graph. We use an upper triangular matrix and accumulate local counts in COUNT-2-ITEMS (Algorithm 8) with a global reduction operation. This is suboptimal because both the memory requirement and the communication volume becomes $O(|I|^2)$ size which limits the scalability of the algorithm in the number of items. When local pruning is employed, the algorithm will have effectively

constructed a local G_{F_2} graph at each node which will bear sparsity like the global G_{F_2}. Therefore, a reduction of the local G_{F_2} graphs with a graph unification operation will compute the global G_{F_2} graph.

This approach has several other advantages. With the distributed G_{F_2} graph, there is no need to broadcast the graph in K-WAY-PARTITION and its optimized version since this may be achieved with a multinode accumulation in COUNT-2-ITEMS. Moreover, we may consider a parallel GPVS algorithm in 2-WAY-PARTITION and a parallel algorithm for computing the vertex induced subgraph thus eliminating the need for reduction altogether.

4.4.4 Compact Structures and Buffering for Communication

In the communication routines, it would be assumed that flat message buffers are used for communication of the sets in question. This, however, places a limit on the scalability in number of transactions. In order to remove this restriction, we can take advantage of trie structures. For doing this, the database redistribution algorithm can encode the transaction sets as compressed structures.

More importantly, buffering and asynchronous communication as in [82] should be employed to make it possible to redistribute very large datasets for which the representation of local databases may not fit into main memory, and for enabling communication/database reading overlap.

4.5 Concurrent Mining of Partitions

In the last step of PAR-FREQ, a given serial algorithm is used for local mining at each processor. Since the partitions can be mined independently, there is no need for any communication during the mining process.

In the following text, we present our version of a serial mining algorithm we

have chosen for its novel search strategy and appropriate use of data structures resulting in efficient and scalable mining. The modifications we make include a correction to pattern output and an optimization which improves both running time and storage requirements of the algorithm.

4.5.1 An Improved Version of FP-Growth

We propose FP-Growth* (Algorithm 17) which is an enhancement of FP-Growth featuring a correction, an optimization and minor improvements. The correction fixes an assumption which prevents correct output in many cases. An important optimization eliminates the need for intermediate conditional pattern bases.

A minor improvement comes from not outputting all combinations of the single path in the basis of recursion. Instead, we output a representation of this task since subsequent algorithms can take advantage of a compact representation for generating association rules and so forth. Another improvement is pruning the infrequent nodes of the single path and only outputting a compact "all patterns" representation when the pruned single path is non-empty.

In the following subsections, the remaining changes are discussed.

4.5.2 A Correction To FP-Growth Algorithm

There is a missing condition in FP-Growth. Consider the transaction set in Table 4.1. The algorithm must discover $\mathcal{F} = \{\{b, i, c\}, \{h, i, c\}\}$ with $\epsilon = 2$. However, FP-Growth can not find the support of $\{c, i, h\}$ because there is no way to determine the support of a pattern if the algorithm has hit the single path condition. The mistake is as follows: if the minimum support in β is sufficient to pass support threshold, there is no problem since the minimum support cannot be larger than α's support. However, there is no way to know the support of α alone. In this case α can only be generated together with β, which is insufficient

Algorithm 17 FP-GROWTH*$(Tree, \alpha)$

```
1: if Tree contains a single path P then
2:    prune infrequent nodes of P
3:    if |P| > 0 then
4:       output "all patterns in 2^P and α"
5:    end if
6: else
7:    for all a_i in header table of Tree do
8:       Tree_β ← CONS-CONDITIONAL-FP-TREE(Tree, a_i)
9:       output pattern β ← a_i ∪ α with count = f(a_i)  ▷ f(x) of Tree
10:      if Tree_β ≠ ∅ then
11:         FP-GROWTH(Tree_β, β)
12:      end if
13:   end for
14: end if
```

to pass support threshold. Our proposed solution is to determine β's count from $f(a_i)$ rather than $count[a_i]$ for the frequency of pattern β is the total frequency of nodes labeled a_i. In line 9, $f(a_i)$ is the $f(x)$ that belongs to $Tree$.

Transaction	a	b	c	d	e	f	g	h	i
$t_1 = \{h, b, i, c\}$		×	×					×	×
$t_2 = \{b, i, c\}$		×	×						×
$t_3 = \{h, i, c\}$								×	×

Table 4.1: A Transaction Set T with $I = \{a, b, c, d, e, f, g, h, i\}$

4.5.3 Eliminating Conditional Pattern Base Construction

The conditional tree $Tree_\beta$ can be constructed directly from $Tree$ without an intermediate conditional pattern base. The conditional pattern base in FP-GROWTH can be implemented as a set of *patterns*. A pattern in FP-GROWTH consists of a set of *symbols* and an associated *count*. With a counting algorithm and retrieval/insertion of patterns directly into the FP-Tree structure, we can eliminate the need for such a pattern base. Algorithm 18 constructs a conditional FP-Tree from a given $Tree$ and a symbol s for which the transformed prefix paths are computed.

Algorithm 18 CONS-CONDITIONAL-FP-TREE($Tree, s$)

```
1: table ← itemtable[Tree]
2: list ← table[symbol]
3: Tree' ← MAKE-FP-TREE
4: ▷ Count symbols without generating an intermediate structure
5: node ← list
6: while node ≠ null do
7:    COUNT-PREFIX-PATH(node, count[Tree])
8:    node ← next[node]
9: end while
10: for all sym ∈ range[count] do
11:    if count[sym] ≥ ϵ then
12:       F[Tree'] ← F[Tree'] ∪ sym
13:    end if
14: end for
15: ▷ Insert conditional patterns to Tree_β
16: node ← list
17: while node ≠ null do
18:    pattern ← GET-PATTERN(node)
19:    INSERT-PATTERN(Tree', pattern)
20:    node ← next[node]
21: end while
22: return Tree'
```

The improved procedure first counts the symbols in the conditional tree without generating an intermediate structure and constructs the set of frequent items. Then, each transformed prefix path is computed as patterns retrieved from $Tree$ and are inserted in $Tree_\beta$.

COUNT-PREFIX-PATH presented in Algorithm 19 scans the prefix paths of a given node. Since the pattern corresponding to the transformed prefix path has the count of the node, it simply adds the count to the count of all symbols in the prefix path. This step is required for construction of a conditional FP-Tree directly since an FP-Tree is based on the decreasing frequency order of F. This small algorithm allows us to compute the counts of the symbols in the conditional tree in an efficient way, and was the key observation in making the optimization possible.

Algorithm 19 COUNT-PREFIX-PATH($node, count$)

1: $prefixcount \leftarrow count[node]$
2: $node \leftarrow parent[node]$
3: **while** $parent[node] \neq null$ **do**
4: $\quad count[symbol[node]] \leftarrow count[symbol[node]] + prefixcount$
5: $\quad node \leftarrow parent[node]$
6: **end while**

Algorithm 20 GET-PATTERN($node$)

1: $pattern \leftarrow$ MAKE-PATTERN
2: **if** $parent[node] \neq null$ **then**
3: $\quad count[pattern] \leftarrow count[node]$
4: $\quad currnode \leftarrow parent[node]$
5: $\quad$ **while** $parent[node] \neq null$ **do**
6: $\quad\quad symbols[pattern] \leftarrow symbols[pattern] \cup symbol[currnode]$
7: $\quad\quad currnode \leftarrow parent[currnode]$
8: $\quad$ **end while**
9: **else**
10: $\quad count[pattern] \leftarrow 0$
11: **end if**
12: **return** $pattern$

Algorithm 20 retrieves a transformed prefix path for a given node excluding node itself and Algorithm 21 inserts a pattern into the FP-Tree. GET-PATTERN

computes the transformed prefix path as described in [35]. INSERT-PATTERN prunes the items not present in the frequent item set F of $Tree$[3] and sorts the pattern in decreasing frequency order to maintain FP-Tree properties and adds the obtained string to the FP-Tree structure. The addition is similar to insertion of a single string, with the difference that insertion of a pattern is equivalent to insertion of the symbol string of the pattern $count[pattern]$ times.

Algorithm 21 INSERT-PATTERN($Tree, pattern$)

1: $pattern \leftarrow pattern \cap F[Tree]$
2: Sort pattern in a predetermined frequency decreasing order
3: Add the pattern to the structure

The optimization in Algorithm 18 makes FP-GROWTH more efficient and scalable by avoiding additional iterations and cutting down storage requirements. An implementation that uses an intermediate conditional pattern base structure will scan the tree once, constructing a linked list with transformed prefix paths in it. Then, it will construct the frequent item set from the linked list, and in a second iteration insert all transformed prefix paths with a procedure similar to INSERT-PATTERN. Such an implementation would have to copy the transformed prefix paths twice, and iterate over all prefix paths three times, once in the tree, and twice in the conditional pattern list. In contrast, our optimized procedure does not execute any expensive copying operations and it needs to scan the pattern bases only twice in the tree. Besides efficiency, the elimination of extra storage requirement is significant because it allows FP-GROWTH to mine more complicated data sets with the same amount of memory.

[3] Which does not have to be identical to the F of calling procedure.

Chapter 5

Implementation

Our implementation is a straightforward translation of PAR-FREQ and FP-GROWTH of Chapter 4 except the parallel optimizations suggested. Algorithm 22 conveys use of PAR-FREQ with FP-GROWTH.

Algorithm 22 PAR-FP-GROWTH(T_i, ϵ)

1: PAR-FREQ(T_i, ϵ, FP-MINE)

Algorithm 23 FP-MINE(T_i, ϵ, G_{F_2})

1: $(F, F_2) \leftarrow G_{F_2}$
2: $Tree \leftarrow$ MAKE-FP-TREE(T_i, ϵ, F)
3: FP-GROWTH*($Tree, \emptyset$)

In the following sections, we will first give an overview of our parallel system and then discuss the low level implementation details not addressed in Chapter 4.

5.1 System Hardware and Software

The program was tested on a Beowulf class supercomputer [69] comprised of 32 compute nodes, an interconnection network and an interface node. Each node has a 400Mhz Pentium-II processor, 128MB memory and a 6GB local disk. The

interface node is a 500Mhz Pentium-II with 512MB memory and over 60GB disk space. The interconnection network is comprised of a 3COM SuperStack II 3900 managed switch connected to Intel Ethernet Pro 100 Fast Ethernet network interface cards at each node and a Gigabit Ethernet uplink connected to the interface node.

The system runs Linux kernel 2.4.14 and Debian GNU/Linux 3.0 distribution. For message passing software we primarily use LAM/MPI implementation.

5.2 Code

The implementation language was C++ on GNU g++ compiler version 2.95.4. LAM/MPI 6.5.6 was used for message passing library [30]. Association rule generator [7] was used for synthetic data generation. ONMETIS in METIS partitioning package [47] was used for the serial graph bisection by vertex separator algorithm used in 2-WAY-PARTITION (Algorithm 10). We have employed the unweighted GPVS routine of ONMETIS for graph partitioning which is inexact. The GPVS should take into account the vertex weights of G_{F_2} for minimization of data replication. Our ongoing research includes efforts on an efficient weighted GPVS implementation.

PAR-FREQ (Algorithm 5) using optimized K-WAY-PARTITION* (Algorithm 15) and REDISTRIBUTE-DB* (Algorithm 16) and the improved FP-GROWTH* (Algorithm 17) was implemented. Remaining parallel optimizations for PAR-FREQ were omitted. We have realized PAR-FP-GROWTH (Algorithm 22) for the performance study.

In the current implementation there are 8305 lines of C++ code and several auxiliary scripts the development of which has taken approximately 6 months including a test/performance suite, utilities, libraries, graphs and changed ideas. Initially we had implemented a three-way partitioning scheme which performed poorly. Upon a more rigorous analysis, the two-way scheme was seen to be superior.

5.2.1 Communication Routines

All communication routines are wrapped for more convenient access in the program. In particular, we maintain a global MPI communicator for handling processor groups and provide routines for initialization/finalization, parallel logging, diagnostics.

The communication routines sometimes provide new functionality as we shall see shortly, otherwise we will refer to MPI names.

5.2.2 Data Structures

For variable length arrays, we use `vector<T>` in standard library. For storing transactions, patterns and other structures representable as strings we use efficient variable length arrays. We also use `set<int>` to store the set of frequent item sets in certain places where it would be fast to do so, otherwise we use sorted arrays to implement sets. The graph data structure implements adjacency list representation which stores adjacency lists as efficient growable arrays. The graph data structure also implements a means to encode/decode graphs to message buffers. Transaction sets in memory are represented as a linked list of transactions using `list<T>`.

In the code, mostly an object based design was followed so related data was encapsulated in the same class. For instance, all transaction set counting information is stored in `TS_Counter` class and all information related to 2-way partitioning is stored in `Twoway_Partition`.

5.3 Initial Distribution

The initial partitioning is accomplished with a simple program that reads each transaction set and writes it in a database file which is distributed in a cylic way, i.e. to split the database in 4 it opens 4 files to which transactions are written in

cycling order as they are read. The files are then copied to the working directory on compute nodes.

We have not analyzed the effects of differing initial distributions on the performance of our algorithm. The partitioning scheme we have developed is indifferent to initial distribution. However, the communication volume may change slightly. We do not predict a significant effect on performance at any rate as such a change would only effect the redistribution algorithm.

5.4 Computing F and G_{F_2}

In both count COUNT-1-ITEMS and COUNT-2-ITEMS the database file is read sequentially to compute the counts. This is necessary as we cannot assume that the local database partition may fit in memory; it is not possible to cache the database in the first pass. However, system level optimizations may be facilitated. We have used standard I/O routines for all data access. A better implementation would be to use an efficient database system such as Berkeley DB which implements memory-mapped regions and possibly other system level optimizations. The alternative is to use POSIX `mmap` directly or to use low level asynchronous I/O routines to overlap computation and disk reads.

The sum reduction to all processors in COUNT-1-ITEMS is implemented with a call to `MPI::Allreduce`. Likewise, the sum reduction in COUNT-2-ITEMS is done with a single `MPI::Reduce`.

5.5 Partitioning

In the vertex bisection routine an overhead is due to a necessary conversion of G_{F_2} to METIS graph data structure. We also have to translate the results back. Unfortunately, a work around is not possible since we need flexible graph structures in the implementation which is binary incompatible with METIS.

For load balancing, all three load estimate functions have been implemented. The default estimate function is Equation 4.3 which we have used for our experiments.

The message buffers in REDISTRIBUTE-DB is implemented as a matrix of variable length arrays. We do not write directly to message buffers, but first write to a matrix of transaction sets which are encoded into the message buffers. Consequently, the messages are exchanged with an all-to-all personalized communication. We have written a personal communication routine here which first communicates the length of message buffers and then the messages themselves using MPI non-blocking send/receive commands.[1] A similar routine has been written for broadcasting the sets P_i to $Parts$ in K-WAY-PARTITION* (Algorithm 15).

For tracking processor groups, we use MPI::Split to identify which processors are continuing parallel recursion and to divide the processor groups in two.[2] Broadcasting the graph can be managed with a plain `MPI::Bcast` since we can decode/encode the graph from/to flat buffers.

5.6 FP-Growth Implementation

Not much to say about FP-GROWTH is left from the detailed algorithms of Chapter 4. The development of the algorithm proceeded from a regular Trie structure to a multi-Trie structure to eventually the FP-Tree structure and the top level algorithm. One thing we have noticed about this algorithm is that it does not seem to be as memory efficient as implied in [35]. Even if the number of nodes is kept small, we store many fields per node and the algorithm consumes a lot of memory in practice.

The algorithm has one detail which required a special code: sorting the frequent items in a transaction according to an order L, in line 2 of Algorithm 2 and

[1] In a way similar to LAM's all-to-all communication implementation.

[2] MPI::Split is a collective operation designed exactly for this purpose. See the MPI standard for further information.

line 2 of Algorithm 21. For preserving FP-Tree properties all transactions must be inserted in the very same order.[3] Figure 5.1 shows the top level C++ code that generates such a unique frequency decreasing order making use of indices when there are items with the same frequency. Using this procedure, we are not obliged to maintain an L.

```
void FP_Tree::sort_decreasing_freq(vector<int>& A)
{
  // sort in order of decreasing frequency
  count_quiksort(A, 0, A.size()-1);
  // scan for blocks with same count
  int i = 0;
  int n = A.size()-1;
  while (i < n) {
    int x = A[i];
    int j = i ;
    // find the last element with same count as x
    do {
      j++;
    } while (j<=n && count[A[j]] == count[x]);
    j--;
    if (j>i)
      quiksort(A,i,j);          // order in increasing indices
    i++;
  }
}
```

Figure 5.1: C++ code to sort transactions in a unique decreasing order

[3]For patterns also in our implementation.

Chapter 6

Performance Study

In this chapter we report on our experiments demonstrating the performance of Par-FP-Growth (Algorithm 22) under various parameter changes.

We have measured the performance of Algorithm 22 on a 32-node Beowulf cluster described in Section 5.1. Par-FP-Growth was run on four synthetic databases with varying number of processors and support threshold.

We describe the data sets used for our experiments in the next section. Following that, we present our performance experiments. In Section 6.5 we interpret the results, and the last section briefly compares the performance of Par-FP-Growth to Par-Eclat.

6.1 Data

We have used the association rule generator described in [7] for constructing all experiment data. Synthetic databases in our evaluation have been selected from [82] and [81] for comparison of our results to the performance of parallel Eclat. These databases have been derived from previous studies [7, 66, 5]. Table 6.1 explains the symbols we use for denoting the parameters of Association Rule Generator tool. The experimental databases are depicted in Table 6.2.

In all experiments, the number of items is 1000, and the number of maximal potentially frequent patterns is 2000. T15.I4.367K was reduced in transaction size from 1471 in T15.I4.D1471K used in [82] for a more comparable run time to other experiments. Remaining experiments are intact. Our serial algorithm could not compress T20.I6.D1137 in the physical memory even when number of transactions was decreased, therefore it was not included in the experiment set as we are interested in observing the speedup.

$\|T\|$	Number of transactions in transaction set
$\|t\|_{avg}$	Average size of a transaction t_i
$\|f_m\|_{avg}$	Average length of maximal pattern f_m

Table 6.1: Dataset parameters

Name	$\|T\|$	$\|t\|_{avg}$	$\|f_m\|_{avg}$	Size
T10.I6.800K	8×10^5	10	6	38.7MB
T10.I6.1600K	1.6×10^6	10	6	77.4MB
T10.I4.1024K	1.024×10^6	10	4	50.5MB
T15.I4.367K	3.67×10^5	15	4	25.1MB

Table 6.2: Synthetic data sets

6.2 Running Time

The algorithm has been run on differing number of processors ranging from a single processor up to 28 processors.[1] For single processor runs, we have used the interface node of our Beowulf system which has a 500Mhz Pentium-II processor, 512MB memory and a more capable disk[2] as the sequential trials require more physical memory than 128MB. The running times on 500Mhz interface node have

[1] $1, 2, 4, 8, 12, 16, 20, 24, 28$ processors to be exact, with exceptions in cases when low number of processors was not possible.

[2] 7200 RPM Quantum disk instead of 5400 RPM Seagate disks at nodes.

been scaled by 5/4 to reflect the performance difference of processors.[3] Each experiment has been repeated with varying support thresholds within the range 0.75% to 0.25%. For lower thresholds we have not been able to conduct meaningful experiments as the memory requirements of FP-Growth exceeded all physical RAM. Running time results have been plotted in Figure 6.1, Figure 6.2, Figure 6.3, Figure 6.4, Figure 6.5, and Figure 6.6. We have plotted linear speedup function with respect to the running time on a single processor for comparison to the ideal case in these figures. Alternatively, running time surfaces for the experiment data sets are plotted in Figure A.1 and Figure A.2 of Appendix A. These plots feature contours at the base of *Processors* and *Support* axises. Similar performance patterns are easily spotted in the running time contours.

[3]In our experience, this scaling has been quite accurate when compared to instances of the serial program that could run on nodes.

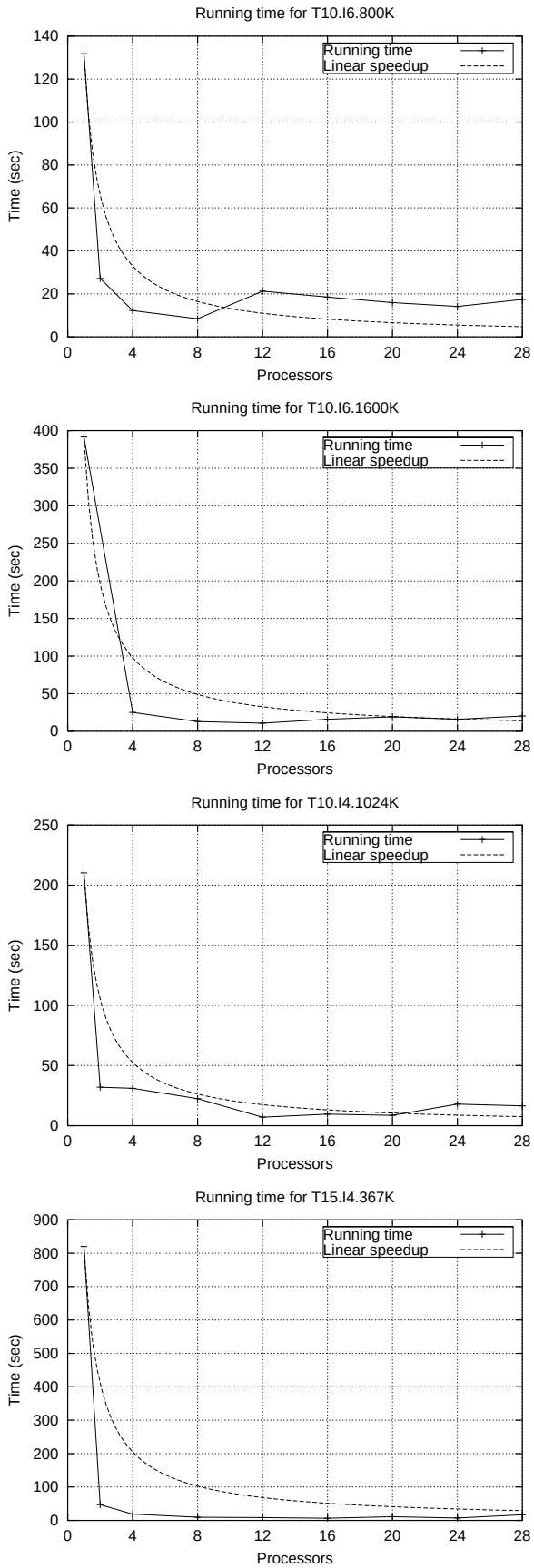

Figure 6.1: Running time for support threshold 0.75%

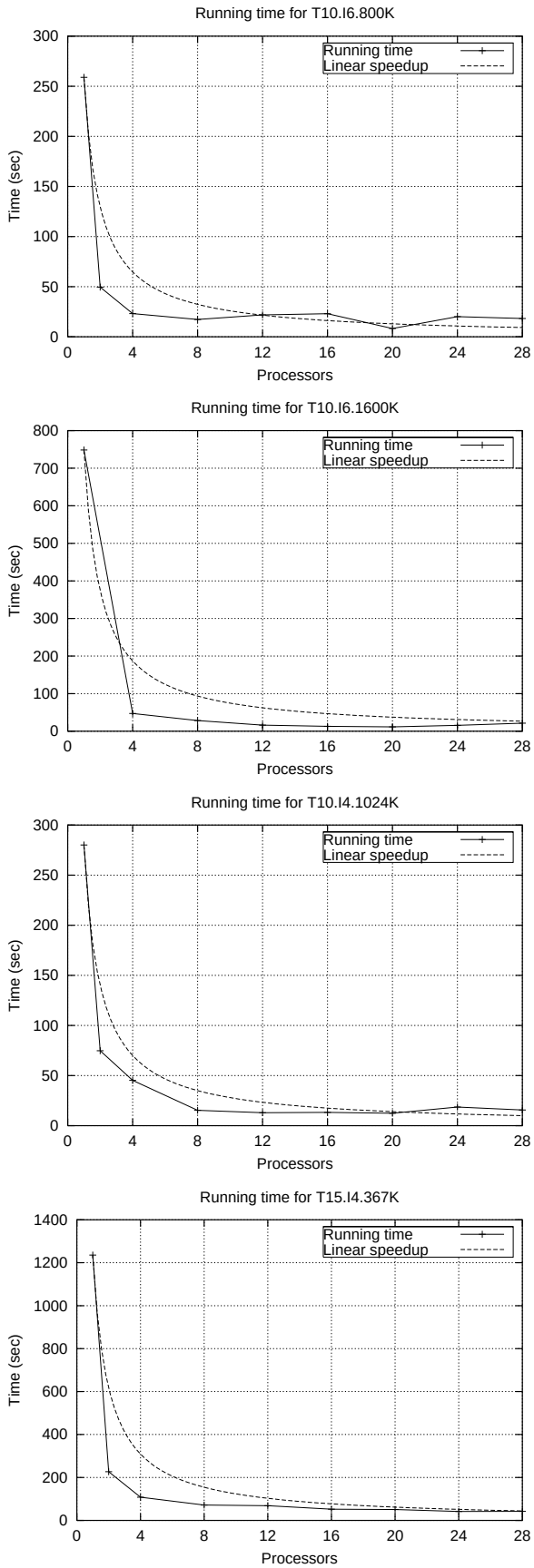

Figure 6.2: Running time for support threshold 0.45%

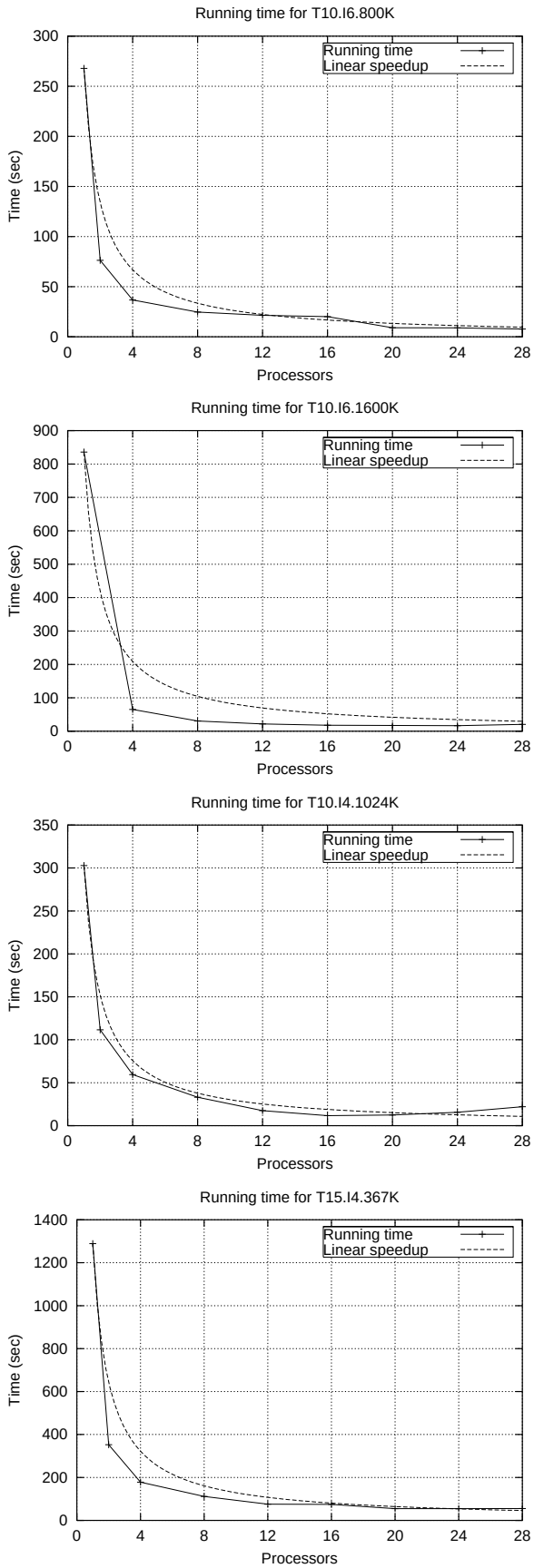

Figure 6.3: Running time for support threshold 0.40%

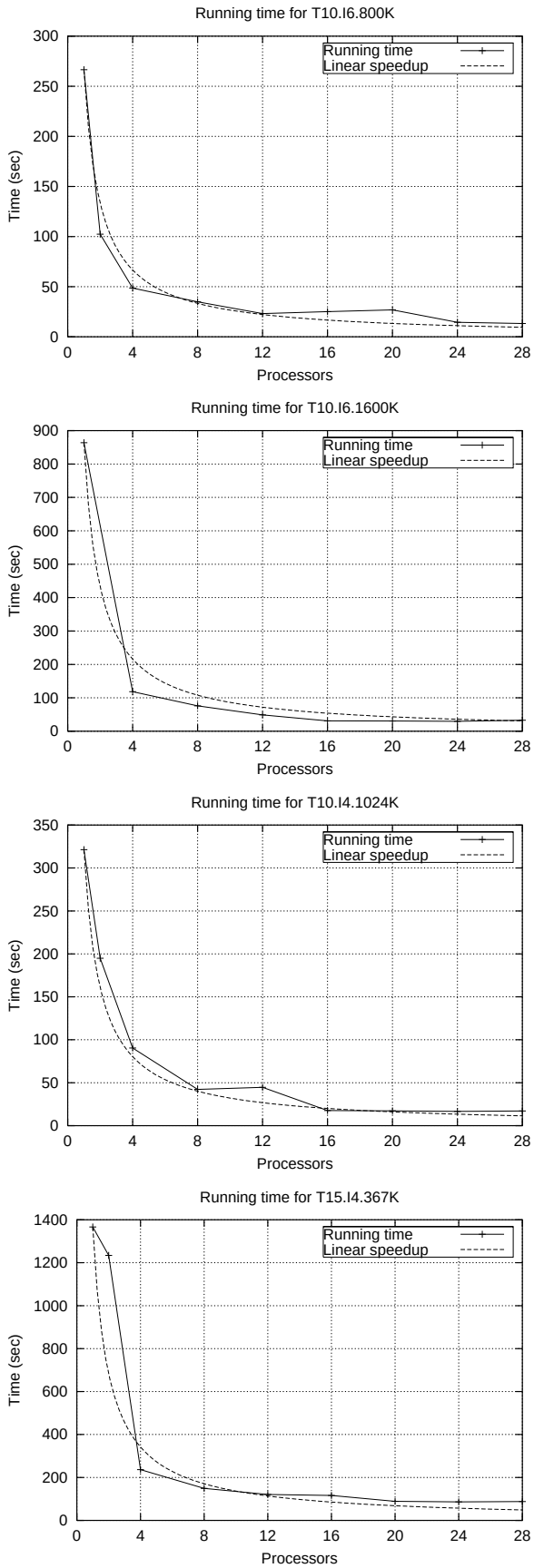

Figure 6.4: Running time for support threshold 0.35%

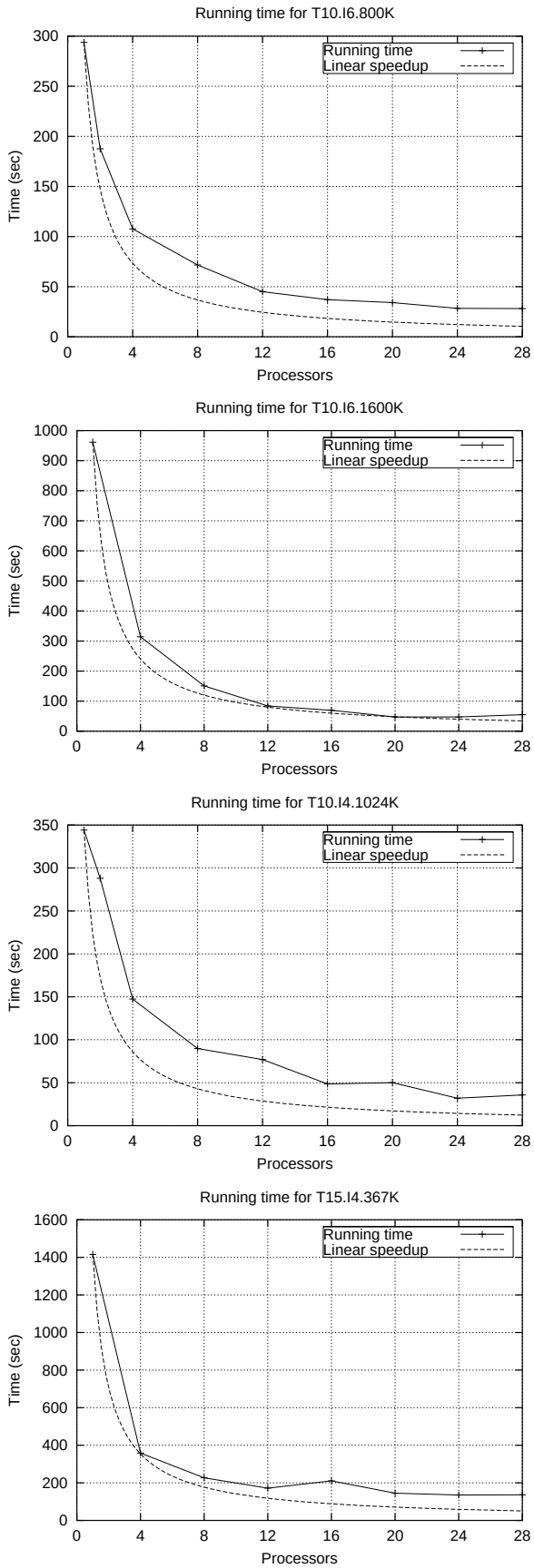

Figure 6.5: Running time for support threshold 0.30%

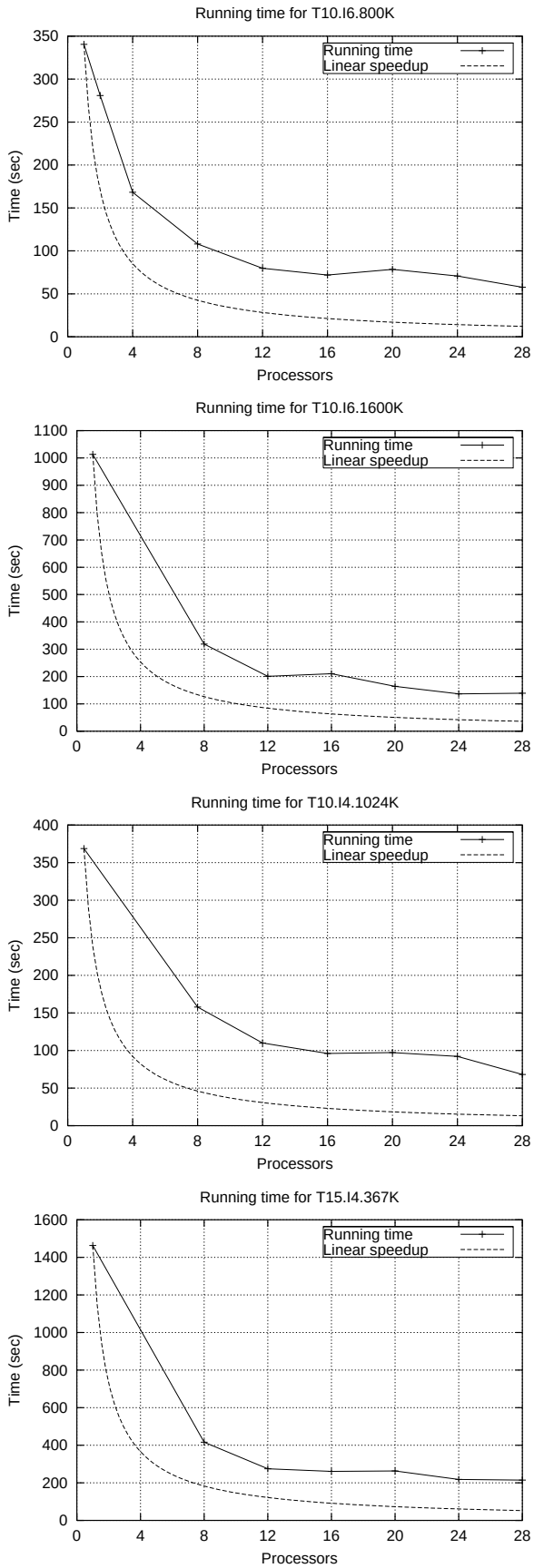

Figure 6.6: Running time for support threshold 0.25%

6.3 Speedup

The speedups are calculated with respect to the running time on a single processor. The plots in Figure 6.7 to Figure 6.12 convey the speedup results for the experiments presented in the previous section.

Figure 6.13 pictures the variation of speedup versus support for 16 processors.

The renderings in Figure A.3, and Figure A.4 depict speedup surfaces for illustrating the patterns in variation of number of processors and support threshold. The three dimensional plots have useful contours drawn at the base like running time plots.

6.4 Load Balancing

Figure 6.14 compares the performances of load estimate functions $load_1$ (Equation 4.1), $load_2$ (Equation 4.2) and $load_3$ (Equation 4.3) for two databases at 0.25% support threshold. It is seen that Equation 4.3 performs slightly more consistent and better than Equation 4.1.

6.5 Interpretation

Our experiments demonstrate that Par-FP-Growth performs quite efficiently despite important optimizations omitted from the implementation. For all experiment data, the running time follows a decreasing trend as more processors are utilized. The most significant reduction in run time is seen in 2 and 4 processor cases indicating that our partitioning scheme is most effective when there is little overhead from re-distributing the data. Decreasing the support amplifies the number of frequent patterns found, however the savings are in general as well as in higher thresholds, and better for smaller number of processors. The discontinuities in running time plots suggest that the simple load estimation function we

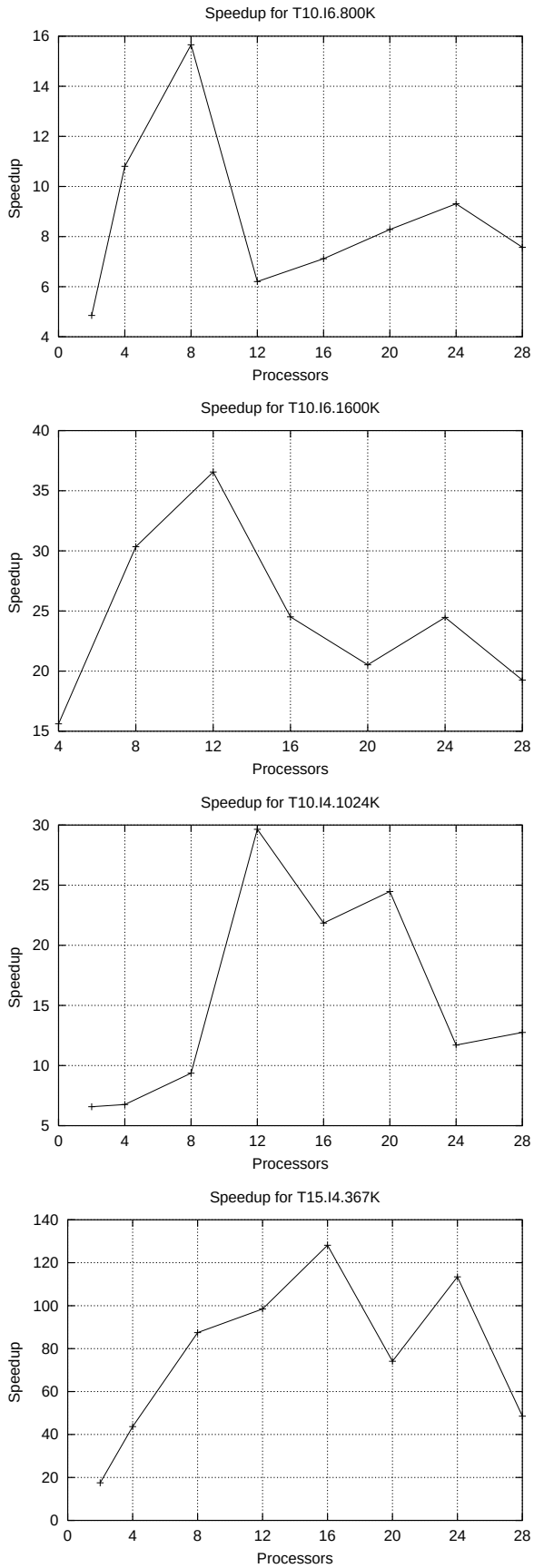

Figure 6.7: Speedup for support threshold 0.75%

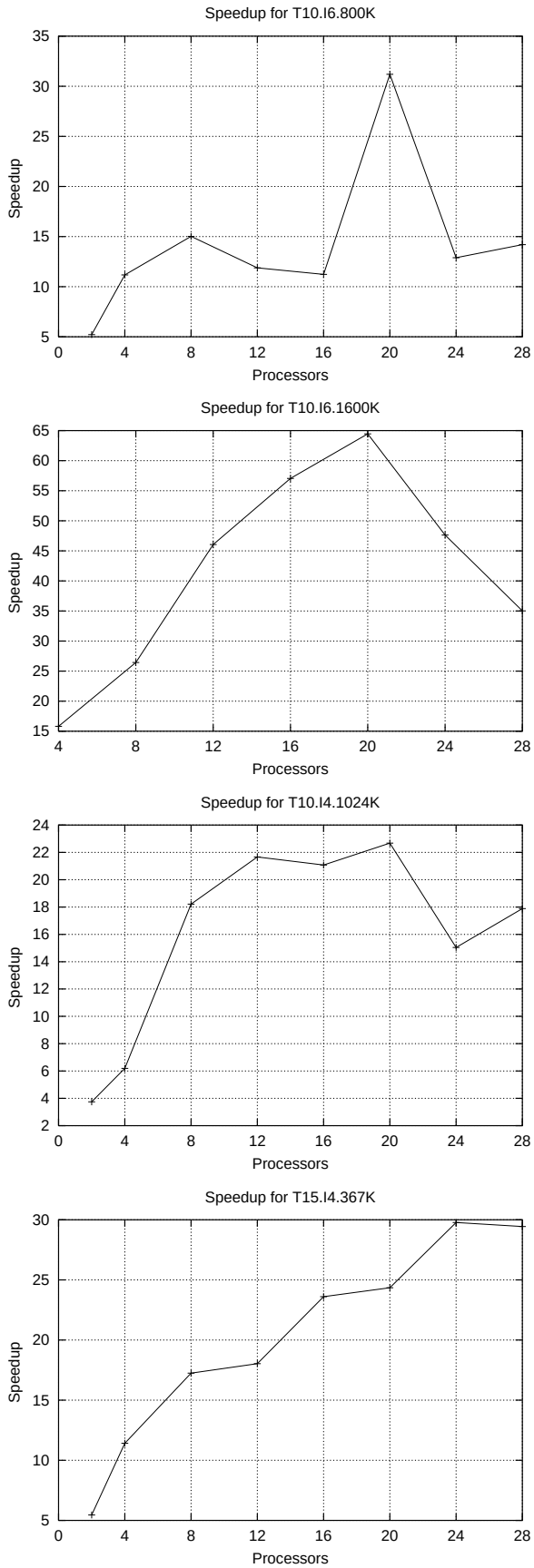

Figure 6.8: Speedup for support threshold 0.45%

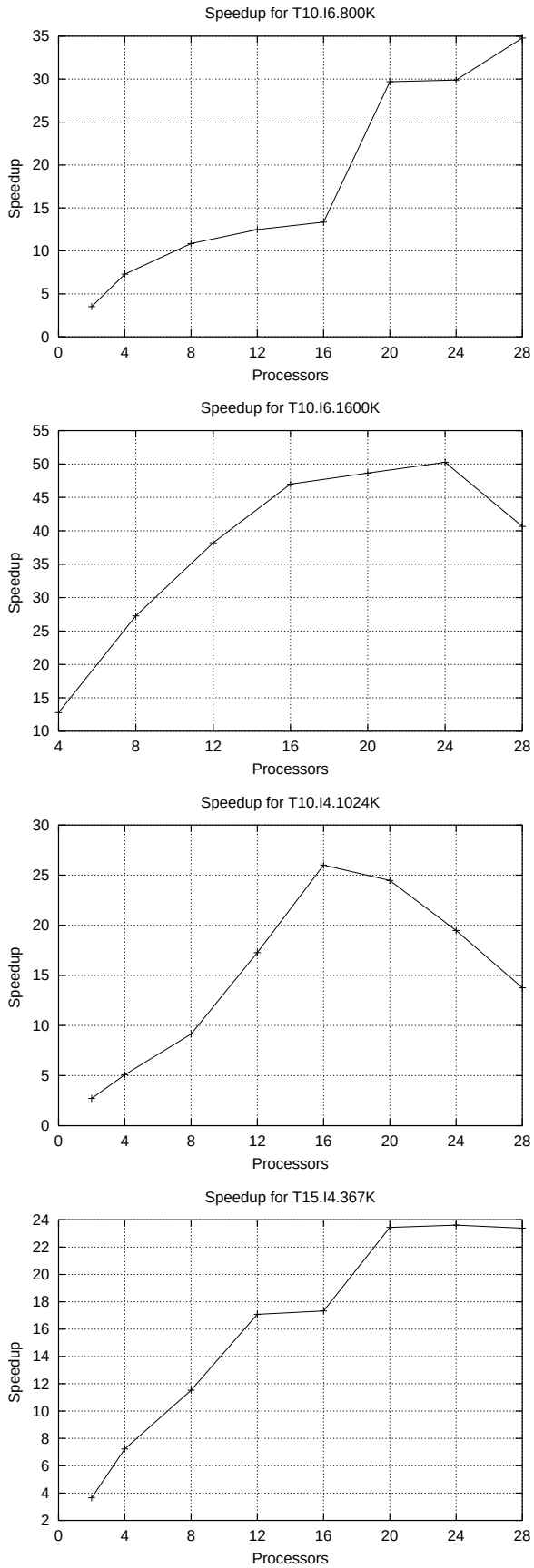

Figure 6.9: Speedup for support threshold 0.40%

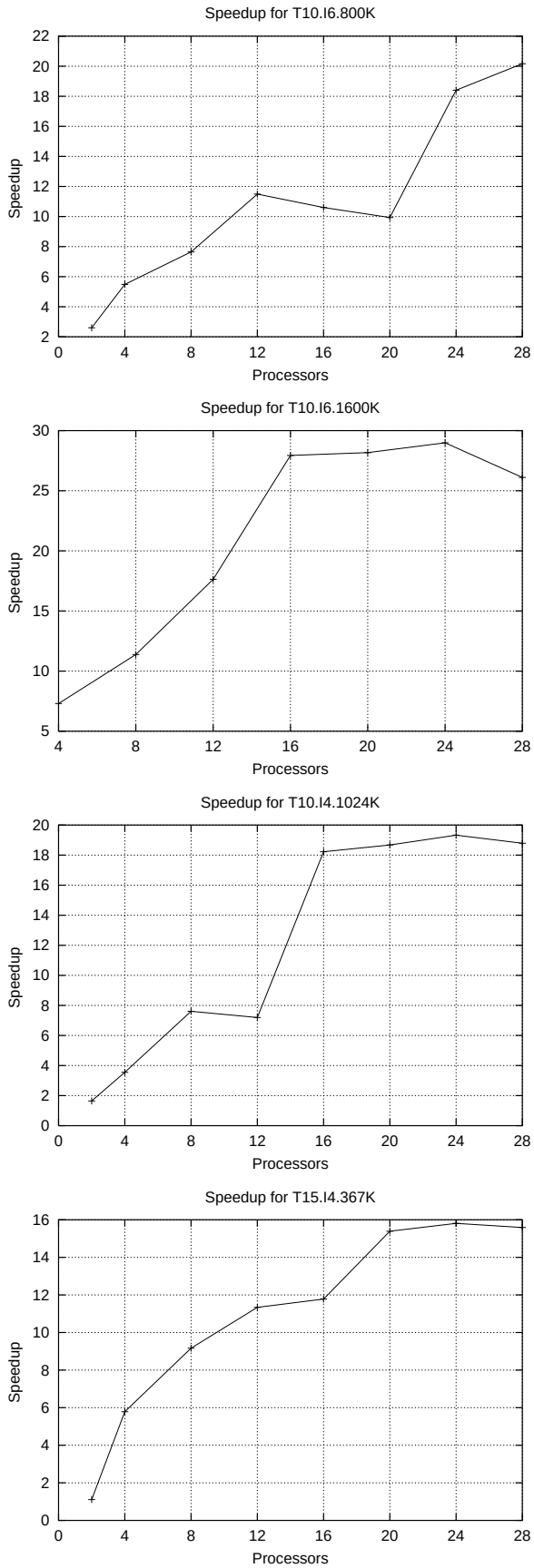

Figure 6.10: Speedup for support threshold 0.35%

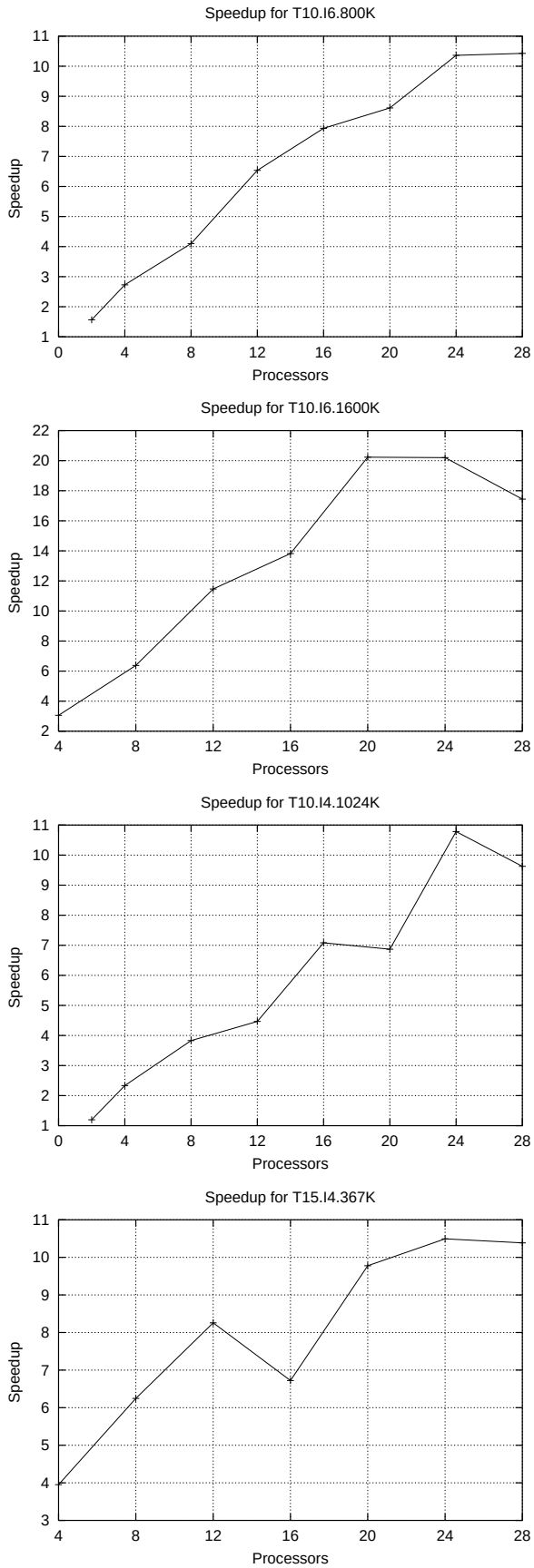

Figure 6.11: Speedup for support threshold 0.30%

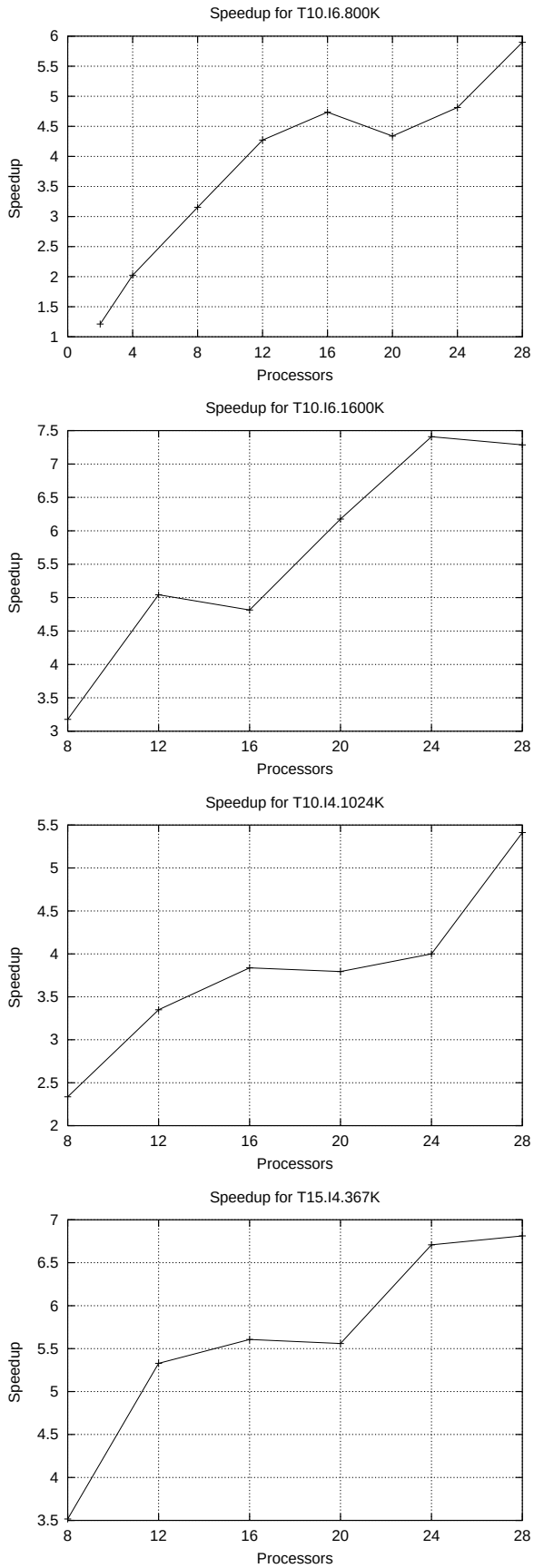

Figure 6.12: Speedup for support threshold 0.25%

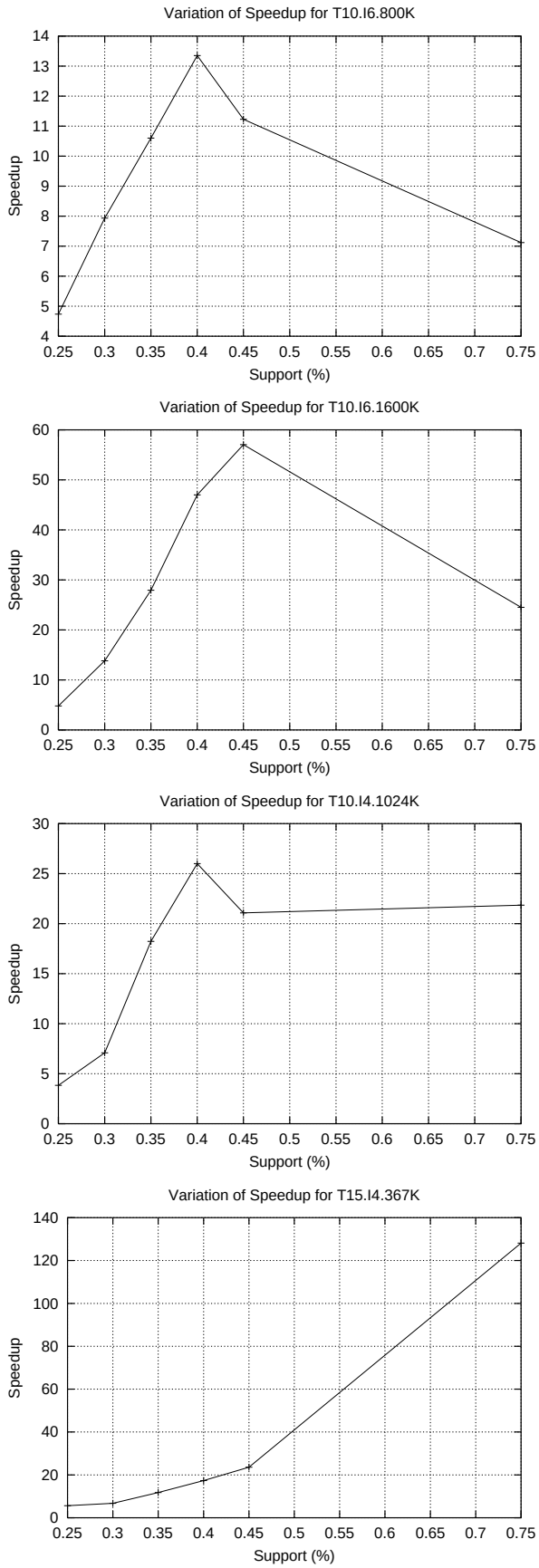

Figure 6.13: Speedup vs. support on 16 processors.

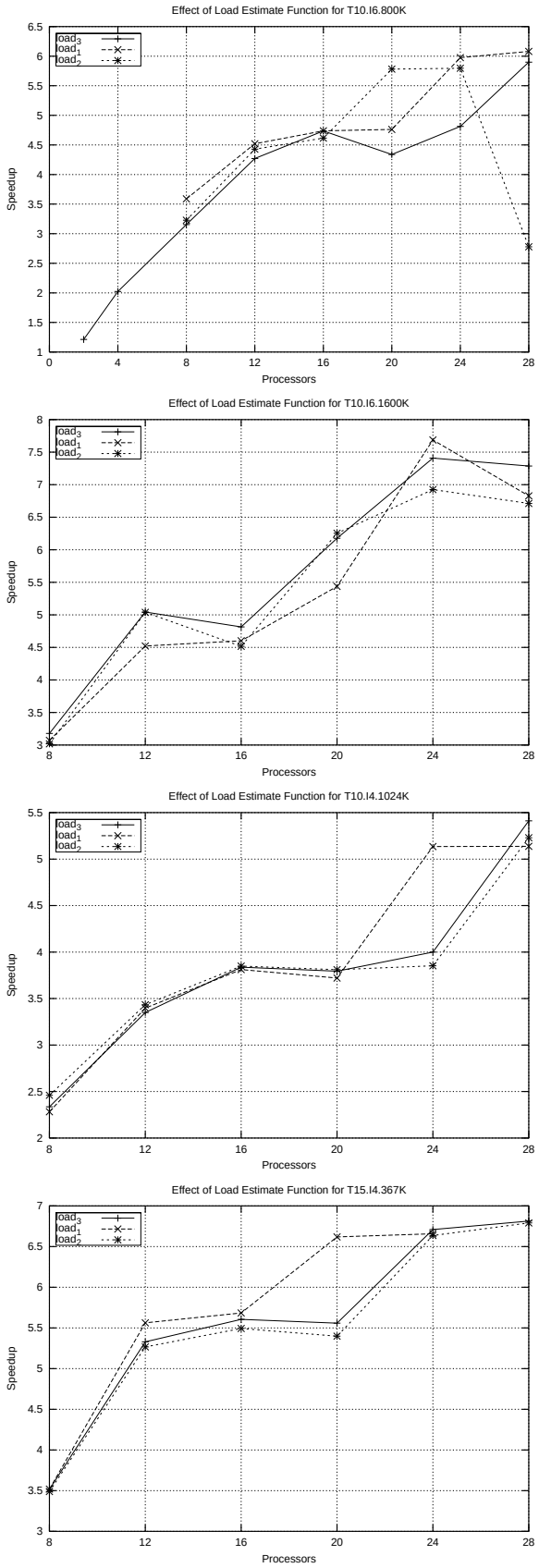

Figure 6.14: Comparative performance of three load estimate functions for support threshold of 0.25%

employ does not yield a precise load balance among processors. Another factor that contributes to the load imbalance is the recursive partitioning scheme used by our k-way partitioning algorithm. Recursive item set partitioning is not very accurate for small number of processors. It is also seen that the running time decreases dramatically as support and number of processors are increased.

The speedup plots reveal that we attain super linear speedup towards 0.75% support although load balance is perturbed in large number of processors. This superlinear behavior may be expected in any successful partitioning scheme due to the fact that the problem has an exponential running time tendency in the number of items. The size of the state space will decrease dramatically if item set partitioning can separate the item set I into small sets of items. We observe a more consistent speedup behavior at 0.40%. As the support is lowered to 0.25%, the speedup falls below linearity. Nevertheless, it follows an increasing trend as we grow the number of processors and lower the support, except in a few cases where there is slight decrement in the speedup. Also note that the communication volume will increase as the number of processors is increased, limiting total speedup. We observe this behavior especially in the harder instances of the problem where there are more frequent items enlarging the communication volume.

The databases T10.I6.800K and T10.I6.1600K differ only in the number of transactions, therefore they may help us to reason about the scalability of the algorithm. Although we have not conducted this specific set of experiments to measure scalability in the number of transactions and items, it is seen in the speedup graphs that our algorithm yields larger speedup for the latter database. The serial algorithm takes almost thrice the running time of T10.I6.800K on T10.I6.1600K at support 0.25% indicating that the mining algorithm will have to consider many more patterns; the difference in running time cannot be solely caused by greater number of transactions. This shows that the algorithm is inclined towards better savings with increasing number of transactions. An implementation that incorporates scalability optimizations should provide favorable scalability in both parameters.

We observe a remarkable speedup of 128.15 on T15.I6.376K database with a support threshold of 0.75% and 16 processors. This speedup supports our reasoning that it is possible to obtain superlinear speedup in frequency mining due to the nature of the problem, in some cases.

The variation of speedup versus support in Figure 6.13 shows that speedup has a decreasing behavior as support is lowered at 16 processors. An interesting feature of this cross-section is that it shows a peak in speedup at around 0.4% support in first three databases, which shows that for lower support values, there may be a load imbalance preventing further speedup.

Three dimensional plots of the running time and speedup results are provided in Appendix A for inspecting performance patterns.

6.6 Comparison with Parallel Eclat

We do not have access to a parallel version of ECLAT for comparing our implementation on our Beowulf system.[4] Nevertheless, it is still possible to make a comparison with parallel ECLAT.

Before proceeding, we should outline the major differences between PAR-FP-GROWTH and PAR-ECLAT with respect to performance. First, ECLAT assumes that F_2 has been computed in a preprocessing step; our algorithm starts with local database partitions and computes every bit of information required in the run time. In addition, the hardware platforms differ so much that any sort of running time comparison would be be insensible. The sequential running times in [82] show that the nodes in the DEC Alpha cluster may actually have better integer performance and disk throughput than our system. Otherwise, the differences in serial algorithm must be attributed to the efficacy of their algorithm for the studied parameters. As for the interconnection network, Zaki's experiments use the custom DEC Memory Channel network which has 30MB/s point-to-point

[4] Implementing it would be difficult due to the fact that almost every component of ECLAT is different.

bandwidth compared to a certain unknown capacity that is smaller than 10MB/s in our system.[5] Moreover, this implementation is not intended to be the final version of our software. Certain optimizations outlined in Chapter 4 were omitted, most notably local pruning for computing a distributed G_{F_2}.

At any rate, speedup of both algorithms at the same support threshold (0.25%) may allow us to make a comparison. The algorithms have increasing speedup with increasing number of processors. For T10.I4.2048K, Par-Eclat reaches 3.5 speedup on 32 processors while Par-FP-Growth reaches 7.28 speedup for T10.I4.1600K on 28 processors. Par-Eclat achieves about 3.8 speedup for T15.I4.D1471K on 32 processors while our algorithm attains 6.81 speedup for T15.I4.367K on 28 processors. The running times for the most number of processors are close for both algorithms. This comparison shows that, despite the disadvantages of our current implementation and the performance of the serial algorithm and the capacity of our hardware, Par-FP-Growth is clearly superior to Par-Eclat in terms of speedup.

[5] We do not have exact knowledge of the architecture and capacity of our network switch.

Chapter 7

Conclusions

We have described the frequency mining problem formally in Chapter 1 and in Chapter 2 we have surveyed relevant work in the literature.

We have proposed a novel transaction set partitioning scheme with which we have crafted a generic parallel frequency mining algorithm. The transaction set partitioning method depends on theoretical observations presented in Chapter 3. We follow from a simple theoretical observation that identifies lack of cliques among two sets of items in G_{F_2} graph to derive a scheme that makes it possible to decompose the mining problem in two independent subproblems. Due to the nature of the problem, any reduction in the number of items becomes significant for efficient parallel algorithms. Another significant aspect of our method is that it does not find a random decomposition of the problem, but finds one that minimizes data replication in the resulting partition while possibly maintaining load balance.[1] All these desirable optimization properties arise from the application of graph partitioning problem which has been previously used in parallelization of problems with detailed structure. Our scheme maps a graph partitioning model on a level of information that is easily computed (G_{F_2}) to the frequency mining problem and therefore is suitable for parallelization. We have shown that this scheme can be extended to a k-way partitioning by recursive bisectioning and

[1] Since maintaining load balance of frequency mining is a difficult problem in itself, we cannot claim that our scheme solves it decisively.

that there is no need to compute the output transaction set partition to do so; computing an item set partition and a G_{F_2} graph is sufficient.

Based on our partitioning scheme, we have designed PAR-FREQ which is a generic frequency mining algorithm in Chapter 4. Since our transaction set partitioning method does not assume a particular representation or operation in the local mining phase we can construct an algorithm that is fairly independent of the underlying sequential mining algorithm. In the algorithm we have presented, it suffices for the sequential mining algorithm to accept a transaction set in horizontal layout.[2] We describe a basic algorithm that is a straightforward translation of our k-way partitioning scheme and optimizations that may be used for a very efficient version of the same algorithm. First phase of the algorithm computes F and G_{F_2} which is required in our scheme. In the basic algorithm, k-way partitioning executes two-way partitioning in a recursive parallel fashion. The algorithm starts with the whole set of processors and in each partitioning step divides the set of processors into two as well as the transaction set. For this task, a load estimate of the partition is computed to distribute the computational load accordingly among the current set of processors. The database is redistributed at the end of the step so that each set of processors can resume parallel mining recursively. The recursion lasts until each processor group has a single processor. The single processor can then mine its local transaction set without further communication. That is similar in spirit to CANDIDATE-DISTRIBUTION and PAR-ECLAT as each processor can perform frequency mining independently once the database has been redistributed according to the k-way partition. Optimizations address computation of G_{F_2} which is done in a way similar to COUNT-DISTRIBUTION and redistribution phase. The most important one of them is reducing the number of, intuitively[3] $O(logn)$, redistributions to a single redistribution as in PAR-ECLAT. In the optimized version, the algorithm does the partitioning with recursive bisection again, but does not redistribute the database at the end of each two-way partitioning step. Instead it postpones the redistribution to the basis of the algorithm when

[2]Most mining algorithms use the horizontal database layout. However, our algorithm can be easily modified to work with other types of serial algorithms since the generality comes from theory.

[3]In reality, the number of redistributions depend on how good the graph partitioning is.

the item sets for all processors have been determined. The partitioning information is broadcast to all processors, after which a single redistribution achieves k-way transaction set partitioning.

In Section 4.5.1 we present our version of FP-Growth which sports multiple improvements. The published algorithm has a bug in its pattern output which is corrected, and a large intermediate structure required in the recursive step is eliminated.

Chapter 5 presents certain details of our implementation that are worth documenting including a sorting routine that efficiently determines a unique decreasing frequency order for use with FP-Growth. We describe our implementation platform and properties of the code with respect to important tasks such as counting items and partitioning. Our implementation realizes Par-FP-Growth which consists of the basic Par-Freq algorithm with the improved serial FP-Growth.

In Chapter 6, we report the results of our performance experiments on four sythetic databases. These databases have been used in other parallel frequency mining benchmarks and thus constitute a point of reference for the performance of our algorithm. Since the main goal of a parallel algorithm is performance, we should establish the validity of our approach with proved gains. The experiments demonstrate the performance of the algorithm with changing number of processors and support threshold on the experiment data. The experiments show that the algorithm has a similar performance behavior on all data sets. It gains superlinear speedup in high threshold, and with increasing number of processors the increase in speedup decreases below linearity. Due to inexact load balancing, largely caused by the inaccuracy of load estimate function and recursive bisection scheme, we see that it cannot attain all the speedup it could have achieved. However, it always tends to scale up successfully as number of processors is increased. We have also observed that it scales well in the number of transactions. Its performance in the largest number of processors we have run our experiments are satisfactory, especially taking into account the fact that our implementation omits some of the parallel optimizations of Section 4.4. On one of the databases, we reach 128.15 speedup at 0.75% support threshold. In the hardest instances of

the problem we have benchmarked, the algorithm attains 5.41 to 7.28 speedup which is better than the speedup behavior of PAR-ECLAT.

For future research we consider working on the following issues.

1. Realizing the omitted optimizations in PAR-FREQ. Local pruning explained in Section 4.4.3 will enable scalability in the number of items while communication improvements of Section 4.4.4 will make the algorithm scalable in the number of transactions. Another optimization is embedding the first database pass of the serial algorithm inside the communication routine.[4]

2. Implementing other serial mining algorithms such as APRIORI for use with PAR-FREQ. We would like to see how well other serial algorithms perform with our generic parallel algorithm and the impact on parallel performance.

3. Using large real world transaction data in our experiments. A benchmark study states that the algorithms overfit synthetic data [12]. It would be thus beneficial to observe the performance characteristics of our algorithm on real world data.

[4]This would speed up the current PAR-FP-GROWTH implementation.

Bibliography

[1] P. L. A. Abu-Hanna. Prognostic models in medicine, AI and statistical approaches. *Methods of Information in Medicine*, 40:1–5.

[2] S. Agarwal, R. Agrawal, P. M. Deshpande, A. Gupta, J. F. Naughton, R. Ramakrishnan, and S. Sarawagi. On the computation of multidimensional aggregates. In *Proc. 22nd Int. Conf. Very Large Databases, VLDB*, pages 506–521. Morgan Kaufmann, 3–6 1996.

[3] R. Agrawal, J. Gehrke, D. Gunopulos, and P. Raghavan. Automatic subspace clustering of high dimensional data for data mining applications. *SIGMOD Record ACM Special Interest Group on Management of Data*, pages 94–105, 1998.

[4] R. Agrawal, T. Imielinski, and A. N. Swami. Mining association rules between sets of items in large databases. In P. Buneman and S. Jajodia, editors, *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, pages 207–216, Washington, D.C., 26–28 1993.

[5] R. Agrawal and J. C. Shafer. Parallel mining of association rules. *IEEE Trans. On Knowledge And Data Engineering*, 8:962–969, 1996.

[6] R. Agrawal and J. C. Shafer. Parallel mining of association rules: Design, implementation and experience. Technical report, IBM Almaden Research Center, IBM Corp, Almaden Res Ctr, 650 Harry Rd, San Jose, Ca, 95120, 1996.

[7] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In J. B. Bocca, M. Jarke, and C. Zaniolo, editors, *Proc. 20th Int. Conf. Very Large Data Bases, VLDB*, pages 487–499. Morgan Kaufmann, 12–15 1994.

[8] R. Agrawal and R. Srikant. Mining sequential patterns. In P. S. Yu and A. L. P. Chen, editors, *Proc. 11th Int. Conf. Data Engineering, ICDE*, pages 3–14. IEEE Press, 6–10 1995.

[9] O. Albert Y. Zomaya, Tarek El-Ghazawi. Parallel and distributed computing for data mining. *IEEE Concurrency*, 7(4):11–13, 1999.

[10] K. Ali, S. Manganaris, and R. Srikant. Partial classification using association rules. In *Knowledge Discovery and Data Mining*, pages 115–118, 1997.

[11] K. Alsabti, S. Ranka, and V. Singh. CLOUDS: A decision tree classifier for large datasets. In *Knowledge Discovery and Data Mining*, pages 2–8, 1998.

[12] Z. Z. Blue. Real world performance of association rule algorithms. In *KDD 2001.*

[13] S. Brin, R. Motwani, and C. Silverstein. Beyond market baskets: generalizing association rules to correlations. pages 265–276, 1997.

[14] S. Brin, R. Motwani, J. D. Ullman, and S. Tsur. Dynamic itemset counting and implication rules for market basket data. In J. Peckham, editor, *SIGMOD 1997, Proceedings ACM SIGMOD International Conference on Management of Data, May 13-15, 1997, Tucson, Arizona, USA*, pages 255–264. ACM Press, 05 1997.

[15] A. Buja, D. Cook, and D. Swayne. Interactive high-dimensional data visualization. *Journal of Computational and Graphical Statistics*, 5:78–99, 1996.

[16] M.-S. Chen, J. Han, and P. S. Yu. Data mining: an overview from a database perspective. *IEEE Trans. On Knowledge And Data Engineering*, 8:866–883, 1996.

[17] S. Cheng. Statistical approaches to predictive modeling in large databases. Master's thesis, Simon Fraser University, Canada, January 1998.

[18] D. W. Cheung, V. T. Ng, A. W. Fu, and Y. J. Fu. Efficient mining of association rules in distributed databases. *IEEE Trans. On Knowledge And Data Engineering*, 8:911–922, 1996.

[19] D. J. Cook, L. B. Holder, S. Su, R. Maglothin, and I. Jonyer. Structural mining of molecular biology data. *IEEE Engineering in Medicine and Biology, special issue on Advances in Genomics*, 2001.

[20] R. Cooley, J. Srivastava, and B. Mobasher. Web mining: Information and pattern discovery on the world wide web. In *Proceedings of the 9th IEEE International Conference on Tools with Artificial Intelligence (ICTAI'97)*, November 1997.

[21] L. Dehaspe, H. Toivonen, and R. D. King. Finding frequent substructures in chemical compounds. In R. Agrawal, P. Stolorz, and G. Piatetsky-Shapiro, editors, *4th International Conference on Knowledge Discovery and Data Mining*, pages 30–36. AAAI Press., 1998.

[22] S. Djoko, D. Cook, and L. Holder. An empirical study of domain knowledge and its benefits to substructure discovery. In *IEEE Transactions on Knowledge and Data Engineering*, volume 9, 1997.

[23] A. E. Eiben, T. J. Euverman, W. Kowalczyk, and F. Slisser. Modelling customer retention with statistical techniques, rough data models and genetic programming. In A. Skowron and S. K. Pal, editors, *Fuzzy Sets, Rough Sets and Decision Making Processes*. Springer-Verlag, Berlin, 1998.

[24] C. Faloutsos, M. Ranganathan, and Y. Manolopoulos. Fast subsequence matching in time-series databases. In *Proc. ACM SIGMOD*, pages 419–429, 1994.

[25] U. M. Fayyad, D. Haussler, and P. E. Stolorz. KDD for science data analysis: Issues and examples. In *Knowledge Discovery and Data Mining*, pages 50–56, 1996.

[26] U. M. Fayyad, D. Haussler, and P. E. Stolorz. The KDD process for extracting useful knowledge from volumes of data. *Communications of the ACM*, pages 27–34, November 1996.

[27] U. M. Fayyad, G. Piatetsky-Shapiro, and P. Smyth. From data mining to knowledge discovery in databases. *AI Magazine*, 17(3):37–54, 1996.

[28] U. M. Fayyad, G. Piatetsky-Shapiro, and P. Smyth. Knowledge discovery and data mining: Towards a unifying framework. In *Knowledge Discovery and Data Mining*, pages 82–88, 1996.

[29] F. Geerts, B. Goethals, and J. V. den Bussche. A tight upper bound on the number of candidate patterns. In *Proceedings of the First IEEE International Conference on Data Mining*.

[30] J. V. Greg Burns, Raja Daoud. LAM: An open cluster environment for MPI. In J. W. Ross, editor, *Proceedings of Supercomputing Symposium*, pages 379–386, 1994.

[31] S. Guha, R. Rastogi, and K. Shim. ROCK: A robust clustering algorithm for categorical attributes. *Information Systems*, 25(5):345–366, 2000.

[32] D. Gunopulos, R. Khardon, and H. M. et al. Data mining, hypergraph transversals, and machine learning (extended abstract). In *Proceedings of the symposium on Principles of Database Systems*, pages 209–215, 1997.

[33] A. Gupta. Fast and effective algorithms for graph partitioning and sparse matrix ordering. Technical report, IBM, 1996.

[34] E.-H. Han, G. Karypis, V. Kumar, and B. Mobasher. Clustering based on association rule hypergraphs. In *Research Issues on Data Mining and Knowledge Discovery*, 1997.

[35] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In W. Chen, J. Naughton, and P. A. Bernstein, editors, *2000 ACM SIGMOD Intl. Conference on Management of Data*, pages 1–12. ACM Press, May 2000.

[36] S. Handley, P. Langley, and F. A. Rauscher. Learning to predict the duration of an automobile trip. In *Knowledge Discovery and Data Mining*, pages 219–223, 1998.

[37] V. Harinarayan, A. Rajaraman, and J. D. Ullman. Implementing data cubes efficiently. In *Proc. ACM SIGMOD*, pages 205–216, 1996.

[38] M. Hearst. Untangling text data mining. In *Proceedings of ACL'99: the 37th Annual Meeting of the Association for Computational Linguistics*, 1999.

[39] T. Hellström and K. Holmström. Predicting the stock market. Technical report, Department of Mathematics and Physics, Mälardalen University, Sweden.

[40] B. Hendrickson and E. Rothberg. Improving the run time and quality of nested dissection ordering. *SIAM Journal on Scientific. Computing*, 20(2):468–489, 1998.

[41] C. Hidber. Online association rule mining. In *SIGMOD Conf.*, pages 145–156, 1999.

[42] J. Hipp, U. Güntzer, and G. Nakhaeizadeh. Algorithms for association rule mining – a general survey and comparison. *SIGKDD Explorations*, 2(1):58–64, July 2000.

[43] J. Itskevitch. Automatic hierarchical e-mail classification using association rules. Master's thesis, Simon Fraser University, 2001.

[44] M. Kamber, L. Winstone, W. Gon, and J. Han. Generalization and decision tree induction: Efficient classification in data mining. In *Proc. of 1997 Int. Workshop Research Issues on Data Engineering (RIDE)*, 1997.

[45] G. Karypis. Multilevel refinement for hierarchical clustering.

[46] G. Karypis, E.-H. S. Han, and V. Kumar. Chameleon: Hierarchical clustering using dynamic modeling. *Computer*, 32(8):68–75, 1999.

[47] G. Karypis and V. Kumar. *METIS A Software Package for Partitioning Unstructured Graphs, Paritioning Meshes an Computing Fill-Reducing Orderings of Sparse Matrices.*

[48] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific. Computing*, 1998.

[49] B. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *The Bell System Technical Journal*, September 1969.

[50] M. Kuramochi and G. Karypis. Frequent subgraph discovery. In *1st IEEE Conference on Data Mining*, 2001.

[51] J. W. H. Liu. A graph partitioning algortihm by node separators. *ACM Transactions on Mathematical Software*, 15(3):198–219, September 1989.

[52] H. Lu, J. Han, and L. Feng. Stock movement prediction and n-dimensional inter-transaction association rules. In *Proc. ACM SIGMOD Workshop on Research Issues on Data Mining and Knowledge Discovery*, pages pages 12:1–12:7, Seattle, Washington, June 1998.

[53] V. K. Mahesh Joshi Eui-Hong, George Karypis. Parallel algorithms in data mining.

[54] W. Maniatty and M. J. Zaki. A requirements analysis for parallel KDD systems. In *IPDPS Workshops*, pages 358–365, 2000.

[55] H. Mannila. Methods and problems in data mining. In *Proceedings of International Conference on Database Theory (ICDT)*, pages 41–55, 1997.

[56] H. Mannila, H. Toivonen, and A. I. Verkamo. Discovery of frequent episodes in event sequences. *Data Mining and Knowledge Discovery*, 1(3):259–289, 1997.

[57] M. Mehta, R. Agrawal, and J. Rissanen. SLIQ: A fast scalable classifier for data mining. In *Extending Database Technology*, pages 18–32, 1996.

[58] O. Z. Mohammad. Fast parallel association rule mining without candidacy generation. In *Proc. of the IEEE 2001 International Conference on Data Mining (ICDM'2001)*, San Jose, CA, USA, November 29–December 2 2001.

[59] A. Moore and J. Schneider. Cached sufficient statistics for automated mining and discovery from massive data sources. Technical report, The Auton Lab, Carnegie Mellon University Robotics Institute and School of Computer Science.

[60] A. Mueller. Fast sequential and parallel algorithms for association rule mining: A comparison. Technical Report CS-TR-3515, College Park, MD, 1995.

[61] NASA astronomical data center home page. `http://adc.gsfc.nasa.gov`.

[62] B. Olsson and K. Laurio. Discovery of diagnostic patterns from protein sequence databases. In *Principles of Data Mining and Knowledge Discovery*, pages 167–175, 1998.

[63] M. M. Özdal and C. Aykanat. Hypergraph models and algorithms for data-pattern based clustering.

[64] R. Rastogi and K. Shim. PUBLIC: A decision tree classifier that integrates building and pruning. *Data Mining and Knowledge Discovery*, 4(4):315–344, 2000.

[65] S. Russell and P. Norvig. *Artificial Intelligence A Modern Approach*, chapter 4 Informed Search Methods. Prentice-Hall Int., 1995.

[66] A. Savasere, E. Omiecinski, and S. B. Navathe. An efficient algorithm for mining association rules in large databases. In *The VLDB Journal*, pages 432–444, 1995.

[67] J. C. Shafer, R. Agrawal, and M. Mehta. SPRINT: A scalable parallel classifier for data mining. In T. M. Vijayaraman, A. P. Buchmann, C. Mohan, and N. L. Sarda, editors, *Proc. 22nd Int. Conf. Very Large Databases, VLDB*, pages 544–555. Morgan Kaufmann, 3–6 1996.

[68] C. Silverstein, S. Brin, and R. Motwani. Beyond market baskets: Generalizing association rules to dependence rules. *Data Mining and Knowledge Discovery*, 2(1):39–68, 1998.

[69] T. Sterling, D. Savarese, D. J. Becker, J. E. Dorband, U. A. Ranawake, and C. V. Packer. BEOWULF: A parallel workstation for scientific computation. In *Proceedings of the 24th International Conference on Parallel Processing*, pages I:11–14, Oconomowoc, WI, 1995.

[70] R. R. Tian Zhang and M. Livny. Birch: An efficient data clustering method for very large databases. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, pages 103–114, Montreal,Canada, 1996.

[71] P. S. U. Fayyad, G. Piatetsky-Shapiro and R. Uthurusamy. *Advances in Knowledge Discovery and Data Mining*. MIT Press, 1996.

[72] G. Weiss, J. Eddy, and S. Weiss. *Knowledge-Based Intelligent Techniques in Industry*, chapter 8, Intelligent Telecommunication Technologies. CRC Press., 1998.

[73] P. C. Wong and R. D. Bergeron. *30 Years of Multidimensional Multivariate Visualization*, pages 3–33. IEEE Computer Society Press., Los Alamitos, CA, 1997.

[74] D.-Y. Yang, A. Johar, A. Grama, and W. Szpankowski. Summary structures for frequency queries on large transaction sets. In *Data Compression Conference*, pages 420–429, 2000.

[75] M. L. Yaron. Knowledge discovery in time series databases. *IEEE Transactions on Systems, Man and Cybernetics - Part B: Cybernetics*, 2001.

[76] O. R. Zaïane, J. Han, and H. Zhu. Mining recurrent items in multimedia with progressive resolution refinement. In *Mining Recurrent Items in Multimedia with Progressive Resolution Refinement*, San Diego, CA, February 2000.

[77] M. Zaki and C. Hsiao. Charm: an efficient algorithm for closed association rule mining. Technical report, 1999.

[78] M. J. Zaki. Parallel and distributed association mining: A survey. *IEEE Concurrency*, 7(4):14–25, 1999.

[79] M. J. Zaki. Scalable algorithms for association mining. *Knowledge and Data Engineering*, 12(2):372–390, 2000.

[80] M. J. Zaki, N. Lesh, and M. Ogihara. Planmine: Sequence mining for plan failures. In *Knowledge Discovery and Data Mining*, pages 369–374, 1998.

[81] M. J. Zaki, S. Parthasarathy, and W. Li. A localized algorithm for parallel association mining. In *ACM Symposium on Parallel Algorithms and Architectures*, pages 321–330, 1997.

[82] M. J. Zaki, S. Parthasarathy, M. Ogihara, and W. Li. Parallel algorithms for discovery of association rules. *Data Mining and Knowledge Discovery*, 1(4):343–373, 1997.

Appendix A

Detailed Performance Results

In this appendix, we include performance plots of running time and speedup surfaces for observing the similar patterns in our performance experiments.

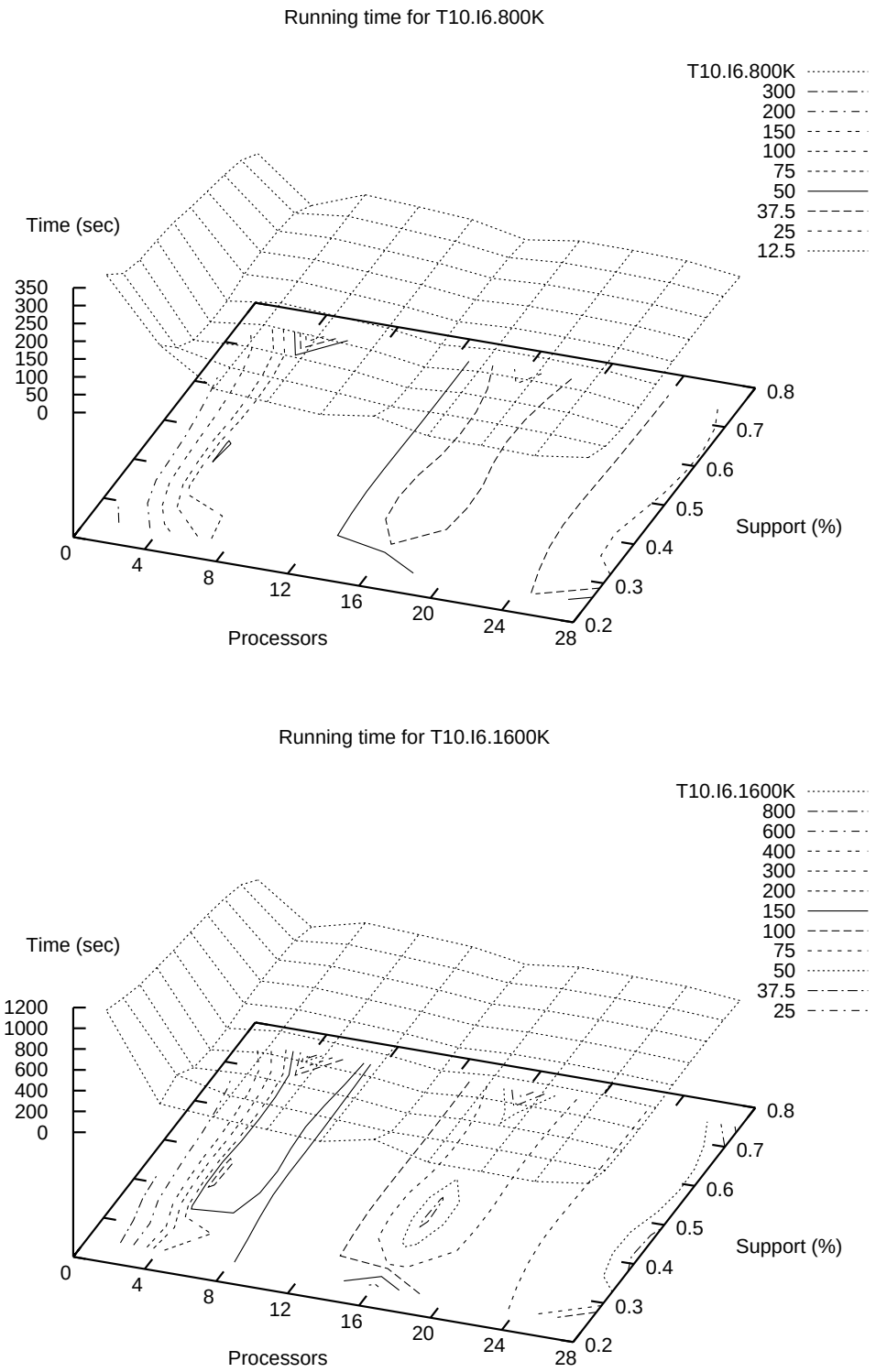

Figure A.1: Running time for T10.I6.800K and T10.I6.1600K

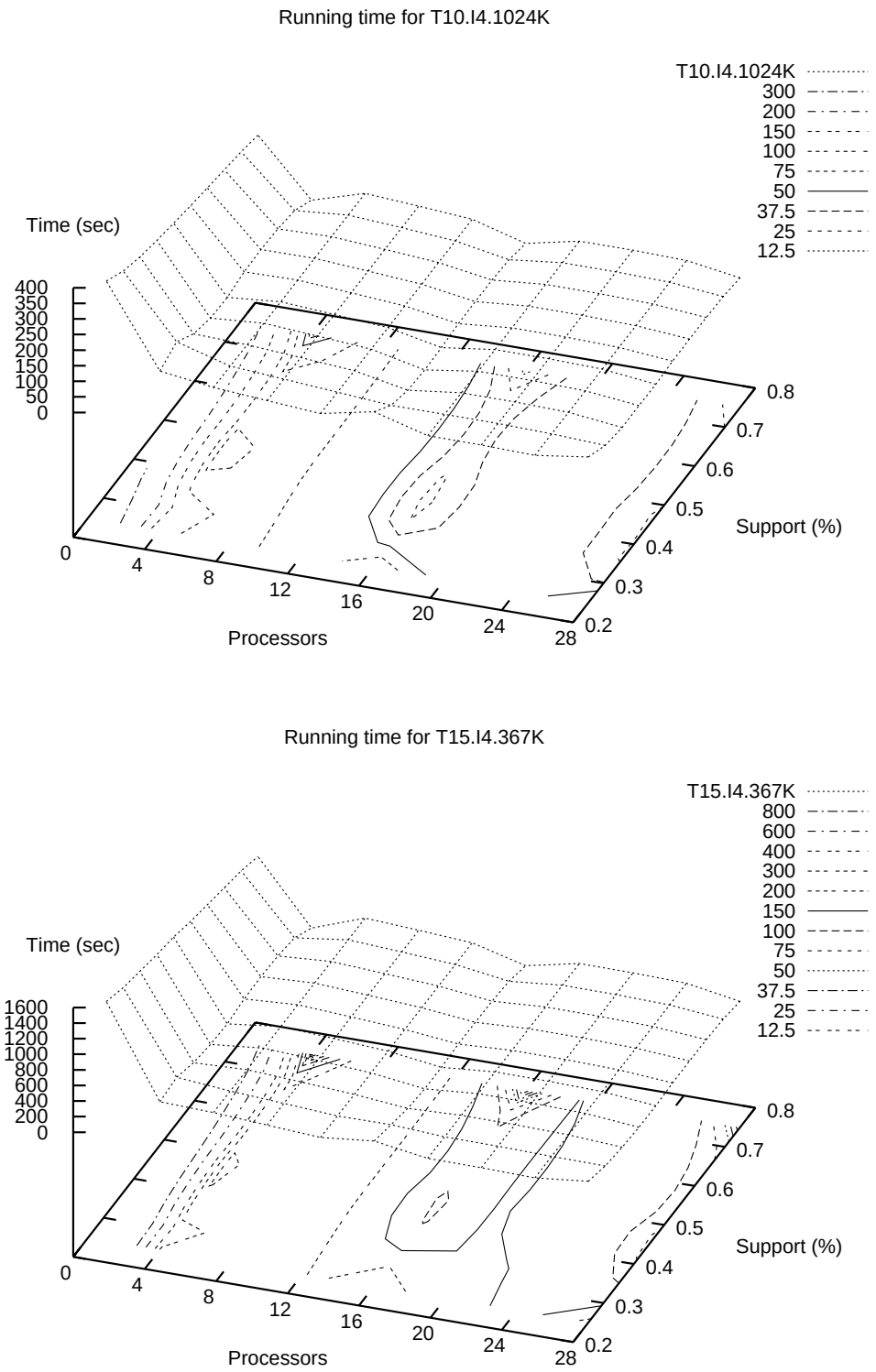

Figure A.2: Running time for T10.I4.1024K and T15.I4.367K

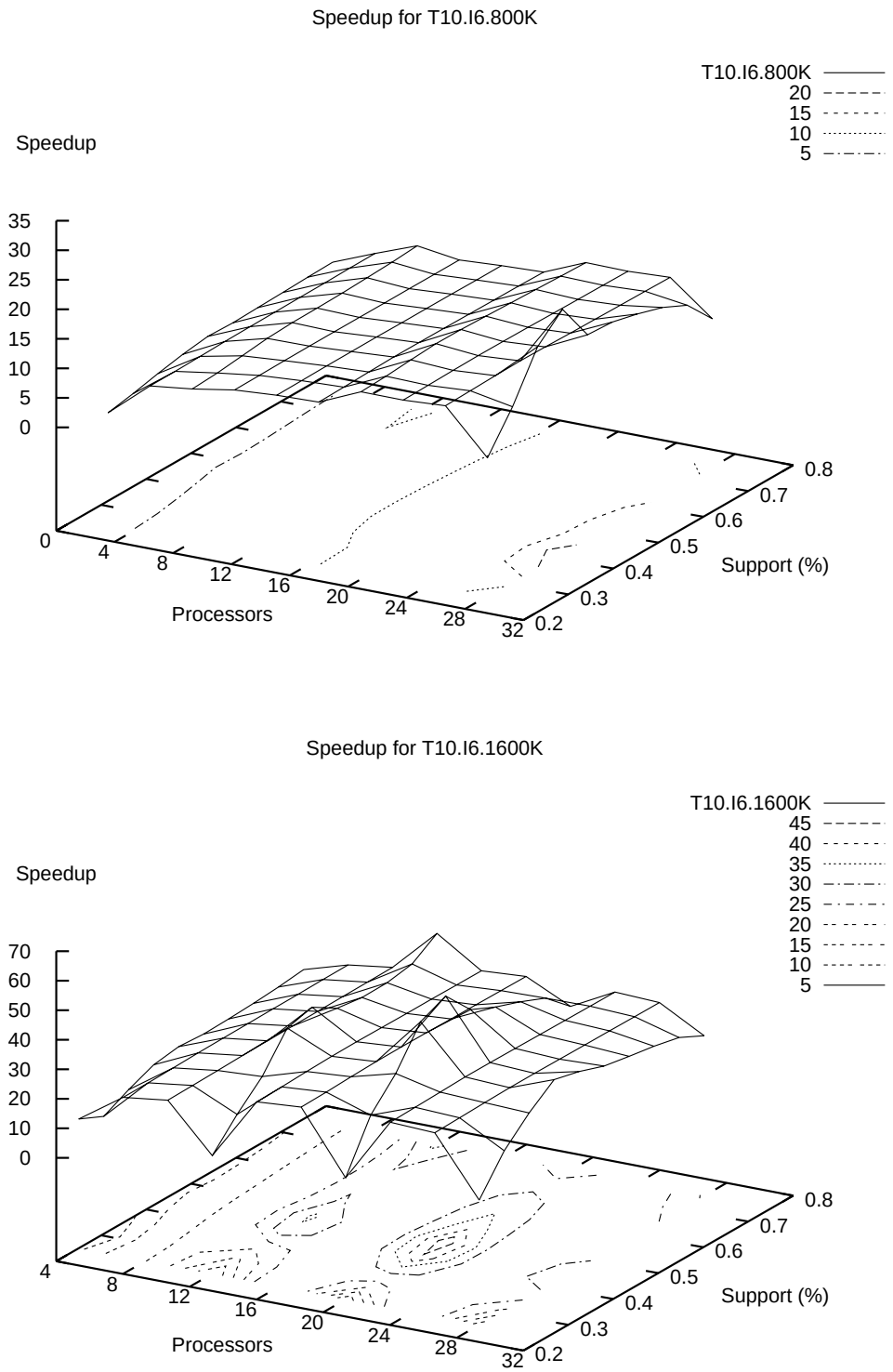

Figure A.3: Speedup for T10.I6.800K and T10.I6.1600K

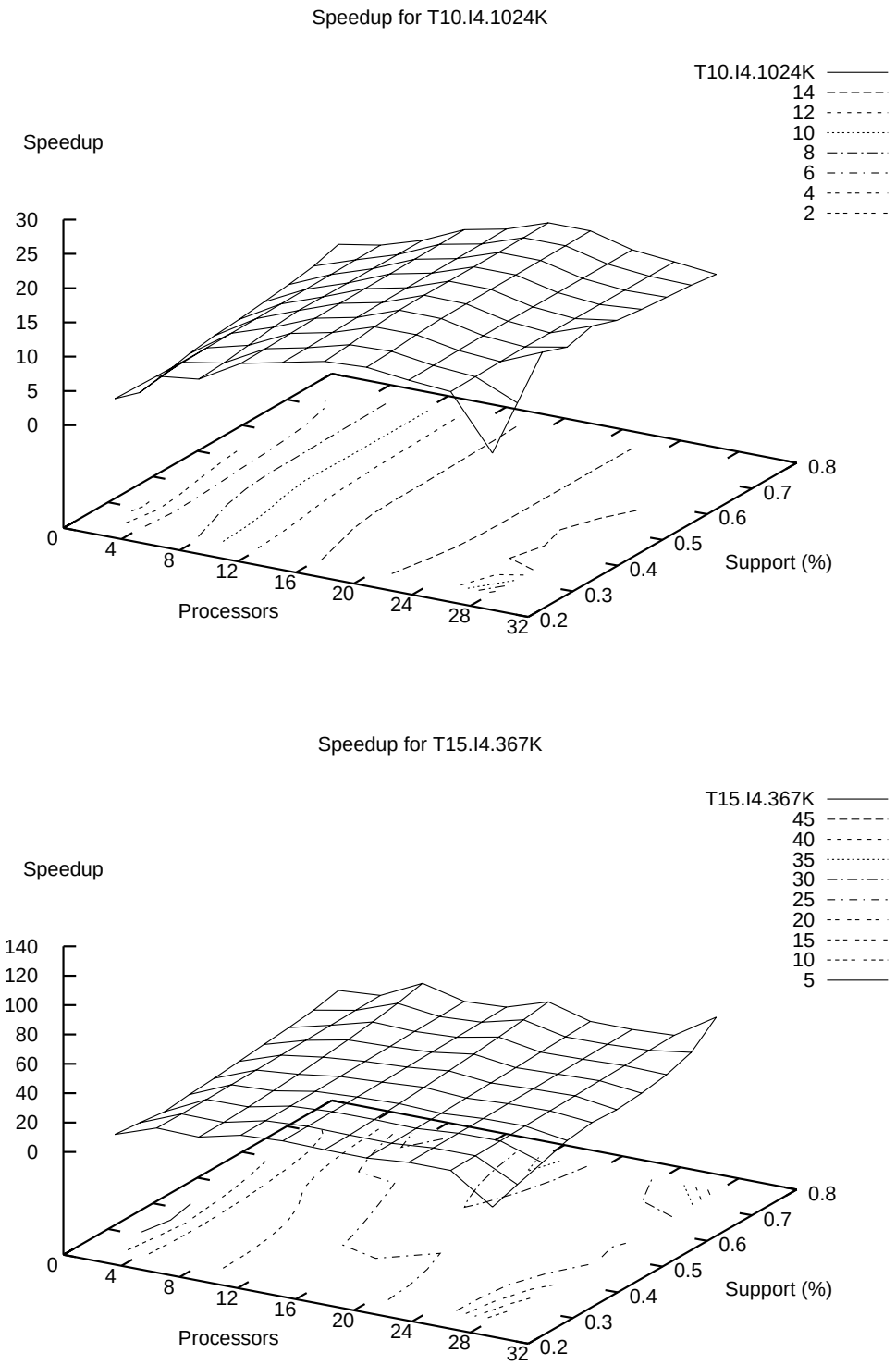

Figure A.4: Speedup for T10.I4.1024K and T15.I4.367K

Appendix B

Proof and Algorithm

Following proof pertains to Lemma 1.

Proof. The proof follows easily from definition of $\mathcal{F}$. Let $X \in \mathcal{F}(T, \epsilon)$. For all $Y \subset X$ we can ascertain $Y \in 2^I$ since $Y \subset X \subseteq I$. $\forall\ Y \subset X,\ o(Z, Y) \geq o(Z, X)$ while $X \subseteq Z \wedge Y \subset X \rightarrow Y \subset Z$. Therefore $\Sigma_{Z \in T} o(Z, Y) \geq \Sigma_{Z \in T} o(Z, X)$, that is $f(T, Y) \geq f(T, X)$. For $f(T, X) \geq \epsilon$, it is also the case that $f(T, Y) \geq \epsilon$. Hence $Y \in \mathcal{F}(T, \epsilon)$. This lemma is due to Agrawal [7], and it is known as the downward closure property. □

COUNT-2-ITEMS* is the optimized counting routine that uses an F_2 matrix of rank $|F|$ as explained in Section 4.4.1.

Algorithm 24 COUNT-2-ITEMS$^*(T_i, \epsilon, F)$

```
▷ A is an upper triangular matrix to hold local counts
A ← MAKE-UT-MATRIX(|F|)
map is an array of length |F|
invmap is an array of length |I|
x ← 0
for all i ∈ F do
    map[x] ← i
    invmap[i] ← x
    x ← x + 1
end for
for all X ∈ T_i do
    for all {u, v} ⊂ X ∩ F do
        u' ← invmap[u]
        v' ← invmap[v]
        if u' > v' then
            swap u' and v'
        end if
        a_{u'v'} ← a_{u'v'} + 1
    end for
end for
C ← MAKE-UT-MATRIX(|F|)
Reduce Sum L to C at pid 0  ▷ C is the global UT count matrix
if pid = 0 then
    for u ← 0 to |F| − 1 do
        for v ← u + 1 to |F| − 1 do
            if c_{uv} ≥ ε then
                E(G_{F_2}) ← E(G_{F_2}) ∪ (map[u], map[v])
            end if
        end for
    end for
end if
return G_{F_2}
```