PARALLEL SEQUENCE MINING ON DISTRIBUTED-MEMORY SYSTEMS

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
Embiya KARAPINAR
February, 2001

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.

Asst. Prof. Dr. Atilla Gürsoy (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.

Assoc. Prof. Dr. Özgür Ulusoy

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.

Asst. Prof. Dr. Uğur Doğrusöz

Approved for the Institute of Engineering and Science:

Prof. Dr. Mehmet Baray
Director of Institute of Engineering and Science

ABSTRACT

PARALLEL SEQUENCE MINING ON DISTRIBUTED-MEMORY SYSTEMS

Embiya KARAPINAR
M.S. in Computer Engineering
Supervisor: Asst. Prof. Dr. Atilla Gürsoy
February, 2001

Discovering all the frequent sequences in very large databases is a time consuming task. However, large databases forces to partition the original database into chunks of data to process in main-memory. Most current algorithms require as many database scans as the longest frequent sequences. *Spade* is a fast algorithm which reduces the number of database scans to three by using lattice-theoretic approach to decompose origional problem into small pieces(equivalence classes) which can be processed in main-memory independently.

In this thesis work, we present *dSpade*, a parallel algorithm, based on *Spade*, for discovering the set of all frequent sequences, targeting distributed-memory systems. In *dSpade*, *horizontal database partitioning method* is used, where each processor stores equal number of customer transactions.

dSpade is a synchronous algorithm for discovering frequent 1-sequences (F_1) and frequent 2-sequences (F_2). Each processor performs the same computation on its local data to get local support counts and broadcasts the results to other processors to find global frequent sequences during F_1 and F_2 computation. After discovering all F_1 and F_2, all frequent sequences are inserted into lattice to decompose the original problem into equivalence classes. Equivalence classes are mapped in a greedy heuristic to the least loaded processors in a round-robin manner. Finally, each processor asynchronously begins to compute F_k on its mapped equivalence classes to find all frequent sequences.

We present results of performance experiments conducted on a 32-node Beowulf Cluster. Experiments show that *dSpade* delivers good speedup and scales linearly in the database size.

Keywords: Lattice, equivalence class,horizontal database partitioning method.

ÖZET

DAĞITIK BELLEKLİ SİSTEMLERDE PARALEL DİZİ MADENCİLİĞİ

Embiya KARAPINAR
Bilgisayar Mühendisliği, Yüksek Lisans
Tez Yöneticisi: Yrd. Doç. Dr. Atilla Gürsoy
Şubat, 2001

Çok büyük veritabanlarında tüm sık dizileri bulmak çok zaman alan bir görevdir. Bununla birlikte, çok büyük veritabanları orjinal veritabanını birden çok veri yığınına parçalayarak ana bellekte işlemeyi zorunlu kılar. Çoğu güncel algoritmalar en uzun sık dizinin uzunluğu adedince veritabanını okumayı gerektirir. *Spade*, kafes-kuramı yaklaşımını kullanarak orjinal problemi ana hafızada işlenebilen küçük parçalara (eşdeğer sınıflara) ayıran ve vertabanını üç kere okuyan çok hızlı bir algoritmadır.

Bu tez calışmasında, dağıtık bellekli sistemler için sık diziler kümesinin tamamını bulan ve *Spade* algoritmasını baz alan *dSpade* adlı paralel algoritmayı öneriyoruz. *dSpade* algoritması her işlemcinin eşit miktarda müşteri hareketi sakladığı yatay veritabanı parçalama metodunu kullanır.

dSpade birli ve ikili sık dizileri bulan F_1 ve F_2 fazları süresince anauyumlu bir algoritmadır. Her işlemci F_1 ve F_2 fazları süresince yerel verileri üzerinde yerel destek sayılarını bulur ve genel birli ve ikili sık dizileri bulmak için bu destek sayılarını diğer işlemcilere yayımlar. Birli ve ikili sik dizileri bulduktan sonra tüm sık diziler kafes içine yerleştirilir ve orjinal problemi kücük parçalara bölmek amacıyle kafes eşdeğer sınıflara ayrıştırılır. Eşdeğer sınıflar açgözlü kurami yontemiyle en az görev yükü olan işlemciye döngüsel bir sırayla eşleştirilir. Bu aşamadan sonra, her işlemci zaman uyumsuz olarak kendisine eşleştirilen eşdeğer sınıfları üzerindeki tüm artan uzunluktaki sık dizileri, F_k, bulur.

Sonuçlarını açıkladıgımız başarım deneylerini 32-düğümlü Beowulf kümesinde yürüttük. Deneyler gösterdi ki, *dSpade* iyi bir hız oranı ve veritabanı boyutuna bağlı olarak lineer ölçekle artan sonuçlar verir.

Anahtar Sözcükler: Kafes, eşdeğer sınıf, yatay veritabanı parçalama metodu .

To my dear wife,Zeynep,

ACKNOWLEDGMENTS

I would like to express my gratitude to Dr. Atilla Gürsoy from whom I have learned a lot, due to his supervision, suggestions, and support during this research.

I am also indebted to Dr. Özgür Ulusoy and Dr. Uğur Doğrusöz for showing keen interest to the subject matter and accepting to read and review this thesis.

I would like to express my thanks to Dr. Uğur Doğrusöz due to his suggestions, extra teaching hours and support during my postgraduate education.

I would like to express my thanks to Dr.Mohammed Zaki for providing us the sourcecode of Spade.

I would like to express my thanks to Bora Uçar for his help on preparing the thesis document.

I would like to thank to Col.Abdülkadir VAROĞLU and Capt.Güner Gürsoy for their full support and contributions to my postgraduate education.

I would like to thank to my dear wife for her morale support and for many things.

This thesis was supported by Turkish Land Forces and Turkish Army Academy.

Contents

List of Figures

Chapter 1

Introduction

The problem of mining sequential patterns in a large database of customer transactions was introduced in [1]. A transaction data typically consists of a customer identifier, a transaction identifier, a transaction time associated with each transaction and the bought items per transaction. For example, consider the sales database of a bookstore, where the objects represent customers and the attributes represent authors or books. Let's say that the database records are the books bought by each customer over a period of time. The discovered patterns are the sequences of books most frequently bought by the customers. An example could be that "60 % of the people who buy Orhan Pamuk's Benim Adım Kırmızı also buy Ahmet Altan's Kılıç Yarası Gibi within 2 months." Stores can use these patterns for promotions, shelf placement, etc. Consider another example of a web access database at a popular site, where an object is a web user and an attribute is a web page. The discovered patterns are the sequences of most frequently accessed pages at that site. This kind of information can be used to restructure the web site, or to dynamically insert relevant links in web pages based on user access patterns.

The task of discovering all frequent sequences in large databases is a time consoming task. The search space is extremely large. For example, with m attributes there are $O(m^k)$ potentially frequent sequences of length k. However, large databases forces us to partition the origional database into chunks of data to process in main-memory. Most current algorithms require as many database scans as the longest frequent sequence.

Several algorithms have been proposed to find sequential patterns. The first algorithm for finding all sequential patterns, named *AprioriAll*, was presented in [1]. First, *AprioriAll* discovers all the sets of items with a user-specified minimum support (large itemset), where the support is the percentage of customer transactions that contain the itemsets. Secondly, the database is transformed by replacing the itemsets in each transaction with the set of all large itemsets.

Lastly, it finds the sequential patterns. It is costly to transform the database. In [2], *GSP* (Generalized Sequential Pattern) algorithm that discovers generalized sequential patterns was proposed. *GSP* finds all the frequent sequences without transforming the database. *GSP* algorithm outperformed AprioriAll by up to 20 times. Besides, some generalized definitions of sequential patterns are introduced in [2]. First, time constraints are introduced. Users often want to specify maximum and/or minimum time period between adjacent elements. Second, flexible definition of a customer transaction is introduced. It allows a user-specified window-size within which the items can be present. Third, given a user-defined taxonomy (is-a hierarchy) over the data items, the generalized sequential pattern, which includes items span different levels of the taxonomy, is introduced.

The problem of finding frequent episodes in a sequence of events was presented in [6]. An episode consists of a set of events and an associated partial order over the events. The definition of a sequence used in dSpade can be expressed as an episode, however their work is targeted to discover the frequent episodes in a single long event sequence, while we are interested in finding frequent sequences across many different customer sequences. They further extended their framework in [8] to discover generalized episodes, which allows one to express arbitrary unary conditions on individual episode events, or binary conditions on event pairs.

Zaki presented a new algorithm in his paper[12] which is called *Spade* (Sequential PAttern Discovery using Equivalence classes), for discovering the set of all frequent sequences. The key features of his approach are as follows:

1. He used a vertical ctid_list database format. He showed that all frequent sequences can be enumerated via simple ctid_list intersections.
2. He used a lattice-theoretic approach to decompose the original search space (lattice) into smaller pieces (sub-lattices) which can be processed independently in main-memory. His approach usually requires three database scans, or only a single scan with some pre-processed information, thus minimizing the I/O costs.
3. He decoupled the problem decomposition from the pattern search. He proposed two different search strategies for enumerating the frequent sequences within each sub-lattice: breadth-first and depth-first search.

Spade not only minimizes I/O costs by reducing database scans, but also minimizes computational costs by using efficient search schemes. The vertical ctid_list based approach is also insensitive to data-skew (see [5] for a good introduction on *data-skew*). *Spade* outperforms previous approaches by a factor of two. Furthermore, *Spade* scales linearly in the database size, and a number of other database parameters.

All the previous algorithms for finding sequential patterns mentioned above are serial algorithms. The problem of discvering sequential patterns has to handle a large amount of customer transaction database and requires multiple passes over the database which takes long computation time. Thus, its computational requirements are too large for a single processor to have a reasonable response time. In the literature, up to this time there exists two proposed work for parallel sequence mining. The first work on parallel sequence mining has looked at distributed-memory machines [10]. In this paper, they consider the parallel algorithms for mining sequential patterns on a shared-nothing environment. Three parallel algorithms (Non Partitioned Sequential Pattern Mining, Simply Partitioned Sequential Pattern Mining and Hash Partitioned Sequential Pattern Mining) are proposed. In *NPSPM*, the candidate sequences are just copied among all the nodes. The remaining two algorithms partition the candidate sequences over the nodes, which can efficiently exploit the total system's memory as the number of nodes is increased. If the candidate sequences are partitioned simply, customer transaction data has to be broadcasted to all nodes. *HPSPM* partitions the candidate itemsets among the nodes using hash function, which eliminates the customer transaction data broadcasting and reduces the comparison workload. Among three algorithms, *HPSPM* attains best performance.

The second work is *pSpade* presented by Zaki in [13], a parallel algorithm for fast discovery of frequent sequences in large databases targeting shared-memory systems. *pSpade* decomposes the original search space into smaller suffix-based classes. Each class can be solved in main-memory using efficient search techniques, and simple join operations. Further each class can be solved independently on each processor requiring no synchronization. However, dynamic inter-class and intra-class load balancing must be exploited to ensure that each processor gets an equal amount of work.

In this thesis work, we present *dSpade*, a parallel algorithm, based on *Spade*, for discovering the set of all frequent sequences, targeting distributed-memory systems. In *dSpade*, *horizontal database partitioning method* is used, where each processor stores equal number of customer transactions.

dSpade is a synchronous algorithm for discovering frequent 1-sequences (F_1) and frequent 2-sequences (F_2). Each processor performs the same computation on its local data to get local support counts and broadcasts the results to other processors to find global frequent sequences during F_1 and F_2 computation. After discovering all F_1 and F_2, all frequent sequences are inserted into lattice to decompose the origional problem into equivalence classes. Equivalence classes are mapped in a greedy heuristic to the least loaded processors in a round-robin manner. Finally, each processor asynchronously begins to compute F_k on its mapped equivalence classes to find all frequent sequences.

We present results of performance experiments conducted on a 32-node Beowulf Cluster. Experiments show that *dSpade* delivers good speedup and scales linearly in the database size.

1.1 Problem Statement

Here we will discuss the problem of mining sequential patterns, stated as in [1]. Let I ={ i_1, i_2, ..., i_m }be a set of m distinct attributes, also called items. An itemset is a nonempty unordered collection of items (without loss of generality, we assume that items of an itemset are sorted in lexicographic order). A sequence is an ordered list of itemsets. An itemset i is denoted as(i_1, i_2, ..., i_k), where i_j is an item. An itemset with k items is called a k-itemset. A sequence α is denoted as ($\alpha_1 \mapsto \alpha_2 \mapsto \ldots \mapsto \alpha_q$), where the sequence element α_j is an itemset. A sequence with k items

$$k = \sum_{j=1}^{q} \alpha_j$$

is called a k-sequence. For example, (B $\mapsto$ AC) is a 3-sequence. An item can occur only once in an itemset, but it can occur multiple times in different itemsets of a sequence.

A sequence α=($\alpha_1 \mapsto \alpha_2 \mapsto \ldots \mapsto \alpha_n$) is a subsequence of another sequence β=($\beta_1 \mapsto \beta_2 \mapsto \ldots \mapsto \beta_m$) , denoted as $\alpha \leq \beta$, if there exist integers ($i_1 < i_2 < \ldots < i_n$) such that $a_j < b_{ij}$ for all a_j . For example the sequence (B $\mapsto$ AC) is a subsequence of (AB $\mapsto$E$\mapsto$ ACD), since the sequence elements B $\subseteq$ AB, and AC $\subseteq$ ACD. On the other hand the sequence (AB $\mapsto$ E) is not a subsequence of (ABE), and vice versa. We say that α is a proper subsequence of β, denoted $\alpha \prec \beta$ if $\alpha \leq \beta$ and $\beta \not\leq \alpha$. A sequence is maximal if it is not a subsequence of any other sequence. A subsequence of length k is called a k-subsequence.

A transaction T has a unique identifier and contains a set of items, i. e., $T \subseteq I$. A *customer* C has a unique identifier and has associated with it a list of transactions $\{T_1, T_2, \ldots, T_n\}$. We assume that no customer has more than one transaction with the same time-stamp, so that we can use the transaction-time as the transaction identifier. We also assume that a customer's transaction list is sorted by the transaction-time, forming a sequence $T_1 \mapsto T_2 \mapsto \ldots \mapsto T_n$ called the *customer-sequence*. The database D consists of a number of such customer-sequences.

A customer-sequence, C is said to contain a sequence α, if $\alpha \leq C$, i. e., if α is a subsequence of the customer-sequence C. The support or frequency of a sequence, denoted $\sigma(\alpha)$, is the the total number of customers that contain this sequence. Given a user-specified threshold called the minimum support

DATABASE		
Customer-ID	Transaction-Time	Items
1	10	CD
1	15	ABC
1	20	ABF
1	25	ACDF
2	15	ABF
2	20	E
3	10	ABF
4	10	DGH
4	20	BF
4	25	AGH

Figure 1.1: Original Customer-Sequence Database

(denoted as min_sup), we say that a sequence is frequent if occurs more than min_sup times. The set of frequent k-sequences is denoted as F_k.

Given a database D of customer sequences and min_sup, the problem of mining sequential patterns is to find all frequent sequences in the database. For example, consider the customer database shown in Figure 1.1 (used as a running example throughout this paper). The database has 8 items (A to H), 4 customers, and 10 transactions in all. The Figure 1.2 shows all the frequent sequences with a minimum support of 50% or 2 customers. In this example we have a unique maximal frequent sequence D$\mapsto$ BF $\mapsto$ A.

The organization of this thesis is as follows: Chapter 2 presents a brief description of subsequence lattice theory and the *Spade* algorithm. The terminology described in this chapter will be used throughout this document. In Chapter 3, we will give detailed information about *dSpade* algorithm and discuss important issues. In Chapter 4, experimental results will be presented along with the comments. Finally, directions for future work and a conclusion will be presented.

FREQUENT SEQUENCES		
F_k	Sequences	Frequency
1	A	4
1	B	4
1	D	2
1	F	4
2	AB	3
2	AF	3
2	B↦A	2
2	BF	4
2	D↦A	2
2	D↦B	2
2	D↦F	2
2	F↦A	2
3	ABF	3
3	BF↦A	2
3	D↦BF	2
3	D↦B↦A	2
3	D↦F↦A	2
4	D↦BF↦A	2

Figure 1.2: Frequent sequences with a minimum support of 2

Chapter 2

The *Spade* Algorithm

In this chapter, the most important issues of the *Spade* algorithm are discussed to make the reader more familiar with the thesis subject.

2.1 Vertical Database Layout

Most of the current sequence mining algortihms assume a *horizontal database layout*, where each customer-transaction identifier, *(cid-tid)*, is stored, along with the items contained in the transaction. In *Spade* , *vertical database layout* is used, where each item X is associated with its ctid_list, denoted L(X), which is a list of all customer-transaction identifiers, *(cid-tid)*, containing the item.

2.2 Subsequence Lattice Approach

We assume that the reader is familiar with basic concepts of lattice theory (see [3] for a good introduction).

The bottom element of the sequence lattice S is {}, but the top element is undefined. However, in practical cases it is bounded. The set of *items* of lattice S are defined to be the immediate upper neighbors of the bottom element. For example, consider Figure 2.2 which shows the sequence lattice induced by the maximal frequent sequence D $\mapsto$ BF $\mapsto$ A for our example database. The set of the frequent items is {A, B, D, F}.

It is obvious that the set of all frequent sequences forms a meet-semilattice. So, we observe that all subsequences of a frequent sequence are frequent.

HORIZONTAL	
CID-TID	ITEMS
1-10	AC
1-20	BD
1-30	ACD
2-20	ABCD
2-25	AB
3-15	BC
3-25	AD
4-10	AB
4-30	BCD

VERTICAL	
A	
CID	TID
1	10
1	30
2	20
2	25
3	25
4	10
B	
CID	TID
1	20
2	20
2	25
3	15
4	10
4	30
C	
CID	TID
1	10
1	30
2	20
3	15
4	30
D	
CID	TID
1	20
1	30
2	20
3	25
4	30

Figure 2.1: Database Layout

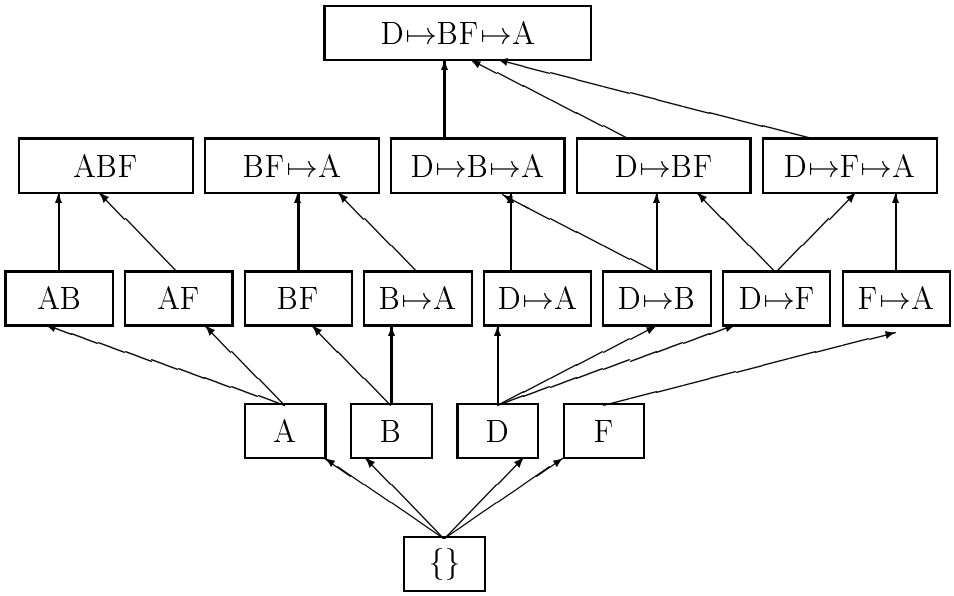

Figure 2.2: Lattice Induced by Maximal Sequence D ↦ BF ↦ A.

A		B		D		F	
CID	TID	CID	TID	CID	TID	CID	TID
1	15	1	15	1	10	1	20
1	20	1	20	1	25	1	25
1	25	2	15	2	25	2	15
2	15	3	10	4	10	3	10
3	10	4	20	4	25	4	20

Figure 2.3: Ctid_lists for the items

2.3 Support Counting

Each item X in the sequence lattice have its vertical ctid_list, denoted *L(X)*, which is a list of all customer *(cid)* and transaction identifier *(tid)* pairs containing the item. Figure 2.3 shows the ctid_lists for the items in our example database. For example, consider the item D. In Figure 1.1, we observe that D occurs in the following customer-transaction identifier pairs {(1, 10), (1, 25), (2, 25), (4, 10), (4, 25)}.

We scan the vertical ctid_list of item D and count different *cid*s encountered. If this count is equal or larger than the minimum support value, then item D is inserted into lattice S.

D↦A	
CID	TID
1	15
1	20
1	25
4	25

D↦B	
CID	TID
1	15
1	20
4	20

D↦F	
CID	TID
1	20
1	25
4	20

D↦B↦A	
CID	TID
1	20
1	25
4	25

D↦BF	
CID	TID
1	20
4	20

D↦BF↦A	
CID	TID
1	25
4	25

Figure 2.4: Computing Support Count via Ctid_list Intersections

2.4 Ctid_list Intersection

We now describe how the actual ctid_list intersection is performed. Consider Figure 2.4, which shows the example ctid_lists for the sequence atoms D $\mapsto$ A, D $\mapsto$ B and D $\mapsto$ F. To compute the new ctid_list for the resulting itemset atom D $\mapsto$ BF, we simply need to check for equality of (cid, tid) pairs. In our example, the only matching pairs are {(1, 20), (4, 20)}. This forms the ctid_list for D $\mapsto$ BF. To compute the ctid_list for the new sequence atom D $\mapsto$ B $\mapsto$ A, we need to check for a follows relationship, i. e., for a given pair (cid, tid_1) in L(D $\mapsto$ A), we check whether there exists a pair (cid, tid_2) in L(D $\mapsto$ B) with the same cid, but with $tid_1 > tid_2$. If this is true, it means that the item A follows the item B for customer cid. In other words, the customer cid contains the pattern D $\mapsto$ B $\mapsto$ A, and the pair (cid, tid_1) is added to its ctid_list. Since we only intersect sequences within a class, which have the same prefix, we only need to keep track of the last tid for determining the equality and follows relationships.

2.5 Lattice Decomposition: Prefix-Based Classes

If we had enough main-memory, we could enumerate all the frequent sequences by traversing the lattice, and performing intersections to obtain sequence supports. In practice, we only have a limited amount of main-memory, and all the intermediate vertical ctid_lists will not fit in memory. This problem is solved by decomposing the original lattice into smaller pieces which are called as equivalence classes such that each equivalence class can be solved independently in main-memory.

An equivalence relation on a set is a reflexive, symmetric and transitive bi-

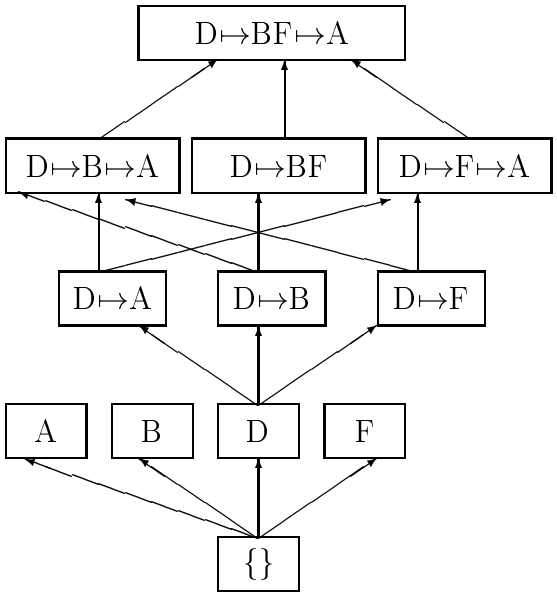

Figure 2.5: Equivalence Classes Induced by θ_1, on S and θ_2, on $[D]_{\theta_1}$

nary relation. An equivalence relation partitions the set (lattice) into disjoint subsets, called equivalence classes (sublattices). Define an equivalence relation θ_k on the lattice S as follows: two sequences are in the same class if they share a common k-length prefix. We therefore call θ_k a prefix-based equivalence relation. Figure 2.5 shows the lattice induced by the equivalence relation θ_k where we collapse all sequences with a common k-length prefix into an equivalence class. Figure 2.5 shows the equivalence classes induced by θ_1 on S, namely, $\{[A]_{\theta_1}, [B]_{\theta_1}, [D]_{\theta_1}, [F]_{\theta_1}\}$

We can compute all the support counts of the sequences in each class (sublattice) by intersecting the ctid_list of items or any two subsequences at the previous level.

In practice it is found that the one level decomposition induced by θ_1 is sufficient. However, in some cases, a class may still be too large to be solved in main-memory. In this case, equivalence class decomposition is applied recursively. Let's assume that [D] is too large to fit in main-memory. Since [D] is itself a lattice, it can be decomposed using θ_2. Figure 2.5 shows the classes induced by applying θ_2 on [D] (after applying θ_1 on S). Each of the resulting six classes, [A], [B], [D $\mapsto$ A], [D $\mapsto$ B], [D $\mapsto$ F], and [F], can be solved independently.

```
Spade(Min_sup, Data)
    F_1={frequent items};
    F_2={frequent 2-sequences};
    ε = {Equivalence classes [X]_θ1};
    for all [X]_θ1 ∈ ε do
        Enumerate_Frequent_Sequences([X]);
Enumerate_Frequent_Sequences(T)
        for all atoms A_i ∈ S do
            T_i = ∅;
            for all atoms A_j ∈ S, with j > i do
                R=A_i ⋃ A_j;
                if (Prune(R)==FALSE) then
                    L(R)=L(A_i) ⋂ L(A_j);
                    if σ(R) ≥ Min_sup then
                        T_i = T_i ⋃{R};
                        F_|R| = F_|R| ⋃{R};
            if (DFS) then Enumerate_Frequent_Sequences(T_i)
        if (BFS) then
          for all T_i ≠ ∅ do Enumerate_Frequent_Sequences(T_i)
```

Figure 2.6: Pseudocode of the *Spade* algorithm

2.6 The Serial *Spade* Algorithm

Figure 2.6 shows the high level structure of the algorithm. The main steps include the computation of the frequent 1-sequences and 2-sequences, the decomposition into prefix-based equivalence classes, and the enumeration of all other frequent sequences via Breadth-First Search or Depth-First Search within each class.

Chapter 3

The Parallel *dSpade* Algorithm

3.1 Introduction

In this chapter, the design of the parallel *dSpade* algorithm and its implementation on distributed-memory systems is presented. First, a brief description of the distributed-memory multicomputers and the parallel programming model will be given since the parallel design (of the *Spade* algorithm) depends significantly on the underlying machine model. Then, major decisions about the parallelization of the *Spade* algorithm, such as data partitioning, will be discussed. Finally, detailed description and implementation of major phases of the parallel *dSpade* will be given. The reader is referred to [11] for more information on parallel computers and programming models. We will only describe basic characteristics of message passing systems and programming as needed in our design.

Distributed memory machines are cost-effective and scalable form of paralel computers. Generally, a distributed memory multicomputer is a collection of processing nodes interconnected via a fast communication network. Each processing node has a processor, local memory and cache, and communication subsystem which handles communication through the network. The most significant characteristic of these systems is that a processor cannot access directly local memory of other processing nodes. Therefore, these systems are called shared-nothing or message-passing systems as well. The data or information is exchanged by messages. In order to get some data or result of a computation done at another processor, a processor requests the data by sending explicitly a message to the remote processor. When the remote processor receives the request (which must post a receive command explicitly), and if the data is ready, a reply message will be send back to the requester with data. This is the simple request-reply mechanism. However, depending on the design of the parallel algorithm and type of interaction, there can be other forms of commu-

nication. For example, if the producer of the data knows that the data will be needed by another processor, the producer can send the data without waiting for the request. This improves the performance since it eliminates one message send-receive phase. Or, sometimes, a particular data is needed by every processor. Then, another form of communication, a collective communication, broadcast is done. Instead of sending the data to each processor with separate messages, a broadcast operation is provided by the programming environment, possibly, implemented in a more efficient way depending on the architecture. The parallel algorithm designer, therefore, must use such services as much as possible instead of using simple send-receive all the time.

The parallel programming model that we have used is explicit message passing and SPMD (single program multiple data). In this model, the same program is loaded and executed on processors. However, each processor has its own data (multiple data), and processor can take differing actions by checking their processor identification. For example, in the program, only one particular processor can be given right to read user input, say processor with id 0, by coding "if my id is zero then read user input else receive input message". These programs is written usually in traditional sequential languages such as C++ or C, and linked with a message-passing library which supports loading program to each processor, setting up communication between processors, and performing message passing. MPI [4, 9] is one of the popular message-passing libraries that we used in our implementation. MPI was developed by a group of computer companies and universities and it is avalible on a wide range of machines. MPI contains a rich set of communication calls including many collective operations such as broadcast, reduce, gather, and more that we will be sing in our code.

Development of parallel algorithms for distributed memory machines, in general, follow certain steps. First, one must partition the data among processors and map the computations to processors. The data partitioning result in mapping the computations to processors in our case because we follow "owner computes" rule. The computations are associated with data and the processor that owns the data perform the computation also. In this way, the computations are distributed to processors but one must be careful about the distribution. For an efficient parallel execution, each processor must have equal amount of computational load, and also communication across processors must be low. In general, load balancing is a difficult issue. Depending on the problem, the load can be balanced at the beginning of the computation if the computational load can be estimated in advance and does not change during the execution. This is called static load balancing. If the computational load changes dynamically during the execution, then the data distribution and mapping of computations must be adjusted dynamically. Dynamic load balancing thus brings additional costs for task/data movement and also mechanisms to detect whether there is an imbalance. In our implementation, we try to guess the

ENTIRE DATABASE				
ITEMS	A	B	C	D
cid-tid	1 - 10	1 - 10	1 - 10	1 - 10
cid-tid	2 - 20	2 - 20	2 - 20	2 - 20
cid-tid	3 - 30	3 - 30	3 - 30	3 - 30
cid-tid	4 - 40	4 - 40	4 - 40	4 - 40

Figure 3.1: Entire Database

computational load in advance and partition data to do static balancing at the beginning.

In he next section, we will discuss partitioning of the input database. Then, we will explain the parallelization of each major phase of the *Spade* algorithm, F_1, F_2, and F_k phases. Each phase produces their own partial results. And we will discuss how the partial results are combined to continue with the next phase.

3.2 Database Partitioning Methods

There are two methods for partitioning the entire database among P processors. In *vertical database partitioning method*, each processor has a subset of items for all customers such that the number of items is roughly equal among processors. In *dSpade*, *horizontal database partitioning method* is used, where each processor stores equal number of customer transactions.

"Why we did not use *vertical database partitioning method* ?" is a naturally upcoming question. Since each processor holds the complete ctid_lists of items, in the computation of F_2 each processor needs all of the ctid_lists of all items to compute any candidate 2-sequence is frequent or not. Thus, it requires multiple pass on database and extra communication overhead to compute all the set of F_2.

Figure 3.1 shows the entire database before partitioned among processors. In this example, we assume that entire database holds 4 customer transactions and each customer buys 4 different items in one transaction. For each item, its own vertical ctid_list is formed and stored in database. Figure 3.2 shows database partitioning methods in more detail. In *vertical database partitioning method*, each of 4 processor holds an entire vertical ctid_list of the corresponding item. In *horizontal database partitioning method*, each of 4 processor holds its corresponding portion of vertical ctid_lists for the all items. In a real dataset example, we design a dataset such that it holds 10,000 items with 200,000 customer transactions. Also, we assume that number of processors is 8. In

VERTICAL PARTITION				
PROCESSORS	1	2	3	4
ITEMS	A	B	C	D
cid-tid	1 - 10	1 - 10	1 - 10	1 - 10
cid-tid	2 - 20	2 - 20	2 - 20	2 - 20
cid-tid	3 - 30	3 - 30	3 - 30	3 - 30
cid-tid	4 - 40	4 - 40	4 - 40	4 - 40

HORIZONTAL PARTITION				
ITEMS	A	B	C	D
PROCESSORS	cid-tid	cid-tid	cid-tid	cid-tid
1	1 - 10	1 - 10	1 - 10	1 - 10
2	2 - 20	2 - 20	2 - 20	2 - 20
3	3 - 30	3 - 30	3 - 30	3 - 30
4	4 - 40	4 - 40	4 - 40	4 - 40

Figure 3.2: Data Partitioning Methods

vertical database partitioning method, we will divide the whole dataset into 8 chunks of data such that every chunk of data stores whole vertical ctid_list of 1,250 items. In *horizontal database partitioning method*, we will divide the whole dataset into 8 chunks of data such that every chunk of data stores 25,000 customer transaction portion of each vertical ctid_list for all items.

3.3 The Parallel *dSpade* Algorithm

Figure 3.3 shows the high level structure of the *dSpade* algorithm. The main steps include the computation of the frequent 1-sequences and 2-sequences, decomposition of lattice into prefix-based equivalence classes, partition of total task among processors, the broadcasting of vertical ctid_lists of all elements of each equivalence class and the enumeration of all other frequent sequences via BFS or DFS search within each class asynchronously by each processor. We will now describe each step in some more detail.

3.3.1 Computing Frequent 1-Sequences F_1

Given the horizontal partitioned database to each processor, all frequent 1-sequences can be computed in a single database scan. For each database item, every processor reads its ctid_list from the local disk into its memory, then scans the ctid_list, increments the support for each new *cid* encountered and inserts the support count into a 1-dimensional array indexed by item *id* number for the all items. After each processor completes support count for the all

```
dSpade(min_sup, D)
        GenF1(min_sup, D);
        GenF2(min_sup, D);
        C={parent equivalence classes Ci=[Xi]};
        Sort_on_Weight(C);
        Partition_Work(C);
        for all items i ∈ D do
            Form_Fk_Ctid_List(i);
        for all items i ∈ Ci do
            Enumerate_Frequent_Sequences(i);
        end
```

Figure 3.3: Pseudocode of the parallel *dSpade* algorithm

items, each processor broadcasts the count array to other processors in a single communication. After reducing the counts by summation into a 1-dimensional array indexed by only frequent item *id* numbers, each processor computes frequent 1-sequences. At this level, each processor does the same computation on its local data and stores the global F_1 frequent items for use in computation of F_2. Figure 3.4 shows the high level structure of the GenF_1. GenF_1 produces a 1-dimensional array shown in Figure 3.5 to represent frequent 1-sequences.

3.3.2 Computing Frequent 2-Sequences F_2 :

Vertical data layout using ctid_lists increases cost of computing F_2, which is basically a self join on F_1. For each item $X \in F_1$, we can read its vertical ctid_list from disk into memory. Then for all items $Y \in F_1$, such that $Y \geq X$, we can read their ctid_lists and intersect them with X. A single intersection is sufficient to determine whether any of the sequences (XY), (X $\mapsto$ Y), or (Y $\mapsto$ X) is frequent. If we use this approach, then item i is scanned i times from the disk. If $\mid F_1 \mid = n$, then the total number of ctid_list scans is given by the sum:

$$\sum_{i=1}^{n} i = n(n+1)/2$$

The average number of times an item's ctid_list is scanned asymptotically O(n). Thus, using the vertical data layout to compute F_2 requires n database scans, whereas in the horizontal format this can be done in a single pass. Then we recover the horizontal format on-the-fly from the vertical data layout, and

```
Gen F1(min_sup, D)
    I=Maximum item number in database D;
    F1=int [| I |];
    Frequent_F1=int [ ];
    for all database items i ∈ I do
        L=Read_ctid_list(i);
        for all distinct cid ∈ L do i_sup=i_sup+1;
        F1[i]=i_sup;
    Reduce local support count array F1 by summation;
    for all items i ∈ F1 do
        if (F1[i] ≥ min_sup) then Frequent_F1 = Frequent_F1 ⋃{i};
    return Frequent_F1;
    end
```

Figure 3.4: Pseudocode of the GenF_1

FREQUENT 1-SEQUENCES														
index	0	1	2	3	4	5	6	7	8	9	...	n-2	n-1	n
items	A	B	D	F	G	H	K	L	M	S	...	W	Y	Z

Figure 3.5: Array for frequent 1-sequences

use this new format to compute F_2. How to achieve this efficiently is discussed below. This is done in the same way with *Spade*.

Optimized F_2 Computation:

There are four main steps in the optimized F_2 calculation:

- Invert the vertical data layout to obtain the horizontal format,
- Create S_1 and S_2 matrices for candidate generation,
- Use the new format to compute F_2.
- Insert frequent 2-sequences into lattice

After each processor completes the first three steps mentioned above for its local data, each processor broadcasts the counts to other processors at a manageable communication cost. Then each processor computes frequent 2-sequences by a simple comparison of all elements of S_1 and S_2 matrices with minimum support value. The structure of S_1 and S_2 matrices is discussed in page 25.

GenF_2(min_sup, D)
$I_D = Invert(D)$;
Generate candidate matrices S_1 and S_2;
for all rows of S_1 and S_2 do
$CompF_2(I_D)$;
Broadcast S_1 and S_2 and reduce by summation;
for all $X, Y \in S_1$ and S_2 do
if$(S_1[X][Y] \geq min_sup) then F_2 = F_2 \bigcup \{(X \mapsto Y)\}$;
if$(S_1[Y][X] \geq min_sup) then F_2 = F_2 \bigcup \{(Y \mapsto X)\}$;
if$((Y > X) and (S_2[X][Y] \geq min_sup)) then F_2 = F_2 \bigcup \{(XY)\}$;
insert all elements of F_2 into equivalence classes graph;
end

Figure 3.6: Pseudocode of the GenF_2

Invert(D)
for all frequent items $i \in F_1$ do
L=Process_ctid_list(i);
for all (cid, tid) pairs in L do
n=cid-mincid;
$I_D[n] = I_D[n] \bigcup$ (i, tid);
end

Figure 3.7: Pseudocede of Invert Database

Finally, each processor inserts frequent 2-sequences into the lattice. Figure 3.6 shows the high level structure of the GenF_2. Algorithms used at each step are discussed in full detail below.

Database Inversion:

The inversion method is shown in Figure 3.7. The vertical input database is denoted as D. We assume that the inverted database, denoted as I, fits in memory. In the figure, I_D[n], denotes the set of transactions belonging to the n-th customer. Each element of this set is of the form *(item, tid)*, i.e., an item and its associated transaction identifier *(tid)*. The inversion process is quite straight-forward. For each item, i, we scan its ctid_list from disk. Each element of the ctid_list is a *(cid, tid)* pair. Using *cid* to compute the offset n, we insert into I_D[n] the pair *(i, tid)*. Figure 3.8 shows the inverted database obtained from vertical database.

ON-THE-FLY TRANSFORMATION	
cid	(item, tid) pairs
1	(A 15)(A 20)(A 25)(B 15)(B 20)(C 10)(C 15)(C 25)(D 10)(D 25)(F 20)(F 25)
2	(A 15)(B 15)(E 20)(F 15)
3	(A 10)(B 10)(F 10)
4	(A 25)(B 20)(D 10)(F 20)(G 10)(G 25)(H 10)(H 25)

Figure 3.8: Vertical-to-Horizontal Database Recovery

Candidate array generation for F_2 :

S_1 MATRIX FOR $(X \mapsto Y)$								
index	A	B	C	D	E	F	...	Z
A	0	0	0	0	0	0	...	0
B	0	0	0	0	0	0	...	0
C	0	0	0	0	0	0	...	0
D	0	0	0	0	0	0	...	0
E	0	0	0	0	0	0	...	0
F	0	0	0	0	0	0	...	0
...	0	0	0	0	0	0	...	0
Z	0	0	0	0	0	0	...	0

S_2 MATRIX FOR (XY)								
index	A	B	C	D	E	F	...	Z
A	-	0	0	0	0	0	...	0
B	-	-	0	0	0	0	...	0
C	-	-	-	0	0	0	...	0
D	-	-	-	-	0	0	...	0
E	-	-	-	-	-	0	...	0
F	-	-	-	-	-	-	...	0
...	-	-	-	-	-	-	-	0
Z	-	-	-	-	-	-	-	-

Figure 3.9: S_1 and S_2 matrix for candidate 2-sequences

Let $\mid F_1 \mid =$ n, and $X, Y \in F_1$. Each processor forms two matrices of (n*n) dimensions indexed by the frequent items of F_1. We setup a matrix denoted S_1 for counting sequences of the form $(X \mapsto Y)$, and another matrix denoted S_2 of dimensions (n*(n-1)/2) for counting sequences of the form (XY). Figure 3.9 shows an example of these two 2-dimensional arrays. Each cell of these arrays containing "0" is used to count 2-sequences. In S_2 array, each cell containing "-" is not created and not used, since (XY) and (YX) represents the same itemset.

```
Comp$F_2$(min_sup, $H_D$)
    for all $C \in H_D$ do
        for all distict items $X \in C$ do
            $X.L$=Get_Pairs_with_Item($X$);
            for all distict items $Y \in C$, with $Y \geq X$ do
                $Y.L$=Get_Pairs_with_Item($Y$);
                Contains($X$, $Y$, $S_1[X][Y]$, $S_1[Y][X]$, $S_2[X][Y]$);
    end
```

Figure 3.10: Pseudocede of Compute F_2

Compute F_2:

We use the recovered horizontal database to count the support of all 2-sequences. We assume that the candidate count matrices fit in memory. For an item X, let X.L denote the list of all (X, tid) pairs for the current customer. For each item pair X, and Y (with $Y \geq X$), we first form their lists, X.L and Y.L, and then call the *Contains()* routine to increment the count of the sequences (XY), (X$\mapsto$ Y), or (Y $\mapsto$ X) if any of them are present.

Insertion of Frequent 2-sequences into Lattice:

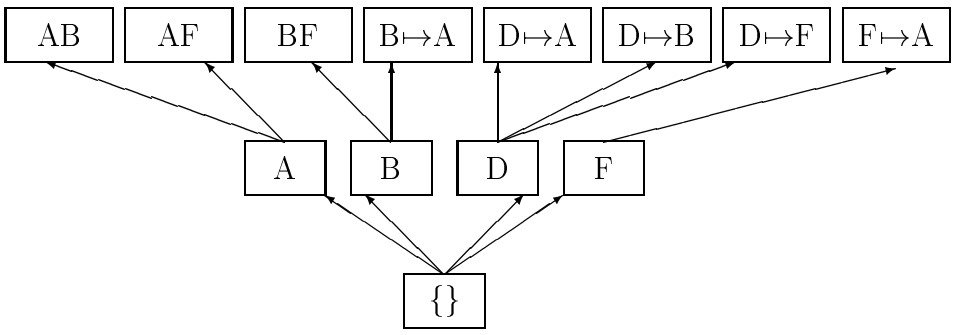

Figure 3.11: Lattice Formed by Insertion of Frequent 2-sequences.

Each processor inserts frequent 2-sequences into the lattice and form equivalence classes. Every frequent 2-sequence is inserted into the equivalence class according to its prefix subclass. For example, B$\mapsto$A is inserted into equivalence class induced by B as shown in Figure 3.11. Arrows in Figure 3.11 make it easy to understand the structure of lattice implemented. Every new k-length frequent sequence is inserted into equivalence class of k-1 length prefix sequence.

3.3.3 Computing Frequent k-Sequences, k$\geq$ 3, F_k

We decompose lattice [C] formed after insertion of all elements of F_2 into independent equivalence classes. Each equivalence class is weighted according to number of its elements. By using static load balancing approach these equivalence classes are shared among processors such that each processor is assigned nearly equal weighted amount of task.

From the $GenF_2()$ routine we know the elements of lattice [C], but not their vertical ctid_lists. The first step is to construct the ctid_lists for the elements (Cx)$\in$ [C], or (C $\mapsto$ x) $\in$ [C]. This is discussed in full detail below.

Finally, each processor asynchronously begins to compute F_k on its assigned equivalence classes bu using the Enumerate_Frequent_Sequences(C) algorithm shown in Figure 3.19. To compute new frequent k-sequences we use three rules of candidate generation with simple ctid_list intersections, check for containment and insert frequent items into equivalence classes to recursively generate new classes of increasing lengths of sequence until all frequent sequences with prefix C is found.

We will now discuss important steps related to computation of F_k in detail.

Equivalence Classes :

Given the set of frequent k-sequences, F_k, it is said that any two sequences belong to the same equivalence class if they share a common k-1 length sequence prefix. More formally, let $P_{k-1}(X)$ denote the k-1 length sequence prefix of the k-sequence X.Since X is frequent, $P_{k-1}(X) \in F_{k-1}$. An equivalence class is defined as follows:

$$[C \in F_{k-1}] = \{X \in F_k \mid P_{k-1}(X) = C\}$$

Each equivalence class has two kinds of elements, $[C].S_1 = \{(C \mapsto x)\}$, or $[C].S_2 = \{(Cx)\}$, depending on the sequence pattern.

The motivation for this definition is that it leads to a very natural partition of the k-sequences into equivalence classes which can be processed independently. A class $[C]$ has all information for generating all sequences with the prefix C.

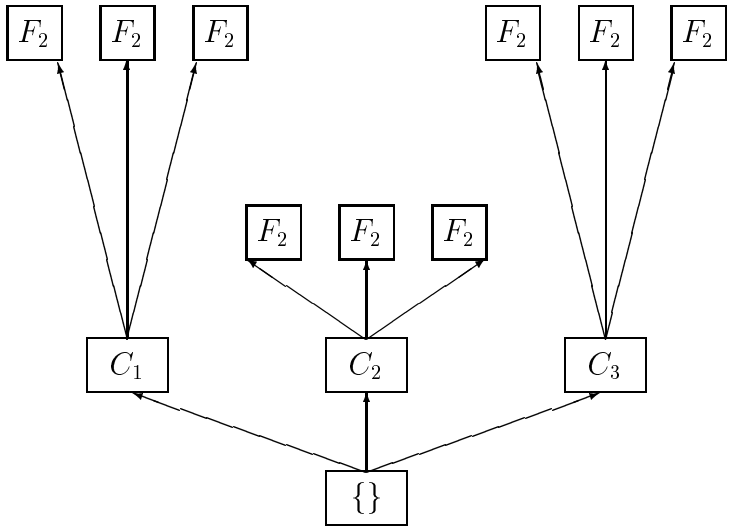

Figure 3.12: Computation tree of classes

Problem Decomposition using Equivalence Classes :

The *dSpade* algorithm begins by calling $GenF_1$, and $GenF_2$. At the end of this process, we have available the sets of frequent sequences, F_1 and F_2. The elements of F_1 actually belong to a single equivalence class, $[\emptyset]$, with a null prefix. However, this class corresponds to the entire sequential pattern discovery problem. The equivalence classes of F_2, on the other hand, provide a natural partition of the problem into the subclasses, [C], where $C \in F_1$. Each subclass can be processed independently, since all frequent k-length sequences with the prefix C is produced with intersection of k-1 length sequences of the subclass. These equivalence classes can be solved independently.

Static Load Balancing

Let $C = \{C_1, C_2, C_3\}$ represent the set of the parent equivalence classes as shown in Figure 3.12. We need to assign the classes into the processors to minimize load imbalance during computation of F_k. As in Zaki's approach [13], an entire class is scheduled on one processor. Each equivalence class is weighted according to the number of elements in the class. Since the *dSpade* algorithm will use all pairs of items for the next iteration, weight ω_i is assigned

$$\omega_i = \binom{|C_i|}{2}$$

to the class C_i.

```
Form_F_k_Ctid_List(i)
        Read_Ctid_List(i);
        Gather_Ctid_List(i);
        for all elements j ∈ F_2 and j ∈ C_i do
            Read_Ctid_List(j);
            Gather_Ctid_List(j);
            Intersect(i, j);
        end
```

Figure 3.13: Pseudocode of Form_F_k_Ctid_List(i)

After assigning the weights, classes are scheduled using a greedy heuristic. The classes are sorted on the weights (in decreasing order), and assigned in a round-robin manner to the least loaded processor.

Once the classes have been scheduled, the computation proceeds in a purely asynchronous manner since there is never any need to synchronize or share information among the processors. If we apply Weight Function ω to the class tree shown in Figure 3.12, we get $\omega_1 = \omega_2 = \omega_3 = 3$. Using the greedy scheduling scheme on two processors, P_0 gets the classes C_1 and C_3, and P_1 gets the class C_2.

Sort_on_Weight(C) and Partition_Work(C) routines in Figure 3.3 represent used static load balancing approach in *dSpade* algorithm.

Form_F_k_Ctid_List(i):

We use static load balancing to decompose the entire lattice among processors. After scheduling classes to the processors, each processor needs the vertical ctid_lists of elements which are belong to the assigned equivalence classes. From the $GenF_2()$ routine we know the elements of [C], but not their vertical ctid_lists. The first step is to construct the vertical ctid_lists for the elements (Cx)$\in$ [C], or (C $\mapsto$ x) $\in$ [C]. This is done by Form_F_k_Ctid_List(i) routine in Figure 3.13.

Firstly, each processor reads partial ctid_list of item C from local disk and broadcasts to other processors to gather the complete ctid_list of item C. Now, all processors have vertical ctid_list of item i. Then, all processors do the same things for item x. Finally, all processors perform the intersection by scanning the two ctid_lists via looking for matching customer identifiers. We then call the Contains() subroutine to determine frequent sequences.

C	
CID	TID
1	30
1	40
2	60
3	40
4	10
4	30
4	50
4	80
5	10
5	50
5	70
8	50
8	60
8	70

x	
CID	TID
1	70
1	80
2	60
3	40
4	30
4	40
4	50
4	70
5	10
6	50
6	65
7	20
7	35
8	50

C↦x	
CID	TID
1	70
1	80
4	30
4	40
4	50
4	80

Cx	
CID	TID
2	60
3	40
4	20
4	15
4	50
8	50

Figure 3.14: Example of ctid_list Intersection for Form_F_k_Ctid_List(C) step

The whole *Intersection* process is shown by means of an example in Figure 3.14. This approach has important drawbacks such that all processors gathers the ctid_lists for all items of F_1 in lattice and makes intersections to form F_2 vertical ctid_lists for use in computation of F_k. But each processor needs only F_2 vertical ctid_lists for elements of its assigned equivalence classes. This redundant work effects the performance of *dSpade*.

Candidate Generation Rules:

New candidate sequences are constructed in three steps:

1. Self-Join ($[C].S_1 * [C].S_1$) : Each element, $(C \mapsto x) \in [C].S_1$, generates a new equivalence class $[\Phi] = [(C \mapsto x)]$. To generate the different classes we simply consider all pairs of elements in $[C].S_1$, say $(C \mapsto x)$ and $(C \mapsto y)$. With only one intersection of their corresponding ctid_lists we determine whether any one of the sequences $(C \mapsto x \mapsto y)$, $(C \mapsto y \mapsto x)$, or $(C \mapsto xy)$ is frequent, and insert it in the appropriate equivalence class.

2. Self-Join ($[C].S_2 * [C].S_2$) : Each element, $(Cx) \in [C].S_2$, generates a new equivalence class $[\Phi] = [(Cx)]$. To generate the different classes we simply consider all pairs of elements in $[C].S_2$, say (Cx) and (Cy). Joining them can produce only one possible candidate, (Cxy), which belongs to the list $[\Phi].S_2$. A simple intersection of ctid_lists is performed to check if the candidate is frequent.

3. Cross-Join ($[C].S_2 * [C].S_1$) : To obtain class $[\Phi].S_1$ for $[\Phi] = [(Cx)]$, we need to join $(Cx) \in [C].S_2$, with all elements, $(C \mapsto y) \in [C].S_1$. This produces only the candidate, $(Cx \mapsto y)$, which belongs to the list $[\Phi].S_1$. A simple intersection of ctid_lists is performed to check if the candidate is frequent.

Let $[\aleph]$ be an equivalence class of frequent k-sequences. Then all frequent sequences with the prefix $\aleph$ are generated from $[\aleph]$ by applying the candidate generation rules.

The candidate F_3 sequences are produced by applying the rules above on the equivalence classes generated by $GenF_1$ and $GenF_2$. Each k-length candidate sequence's support is computed with only one intersection of (k-1)-length prefix subsequences. Only the frequent sequences are inserted into the appropriate equivalence classes.

Ctid_list Intersection :

The algorithms for Intersect() and Contains() routines is presented in Figure 3.15 and the whole intersection process is shown by means of an example in Figure 3.16.

We will now describe how to perform the ctid_list intersection for two sequences within an equivalence class. Depending on the pattern of the two sequences there may be three possible frequent candidates. Only one intersection is sufficient to determine which of the three are frequent. These three cases correspond to the candidate generation rules presented above. To perform the intersection we first scan the vertical ctid_lists of two items and call Contains() routine to determine whether the candidate sequences are present in the vertical ctid_lists of items for incrementing the count of candidate sequences if they are.

Checking for Containment :

The Contains() routine checks whether a given sequence is present in a customer transaction. Given X and Y, the two vertical ctid_lists composed of *(cid, tid)* pairs with the same *cid*, we need to check for two kinds of relationships among the *tid* entries:

1. Equality: for XY
2. Follows : for X $\mapsto$ Y and Y $\mapsto$ X

```
Intersect(α, β, Φ_{x↦y}, Φ_{y↦x}, Φ_{xy})
        for all distinct cids C_α ∈ α.ctid_list do
            for all distinct cids C_β ∈ β.ctid_list do
                if (C_α == C_β) then
                    α.L = Get_pairs_with_cid(C_α);
                    β.L = Get_pairs_with_cid(C_β);
                    Contains(α, β, Φ_{x↦y}, Φ_{y↦x}, Φ_{xy})
        end

Contains(α, β, Φ_{x↦y}, Φ_{y↦x}, Φ_{xy})
        if(Φ_{x↦y} ≠ ∅) then
          for all tids t_b ∈ β.L do
              if ∃ tid t_a ∈ α.L such that t_b > t_a then
                  Φ_{x↦y}.Add_ctid_list(cid, t_b);

        if(Φ_{y↦x} ≠ ∅) then
          for all tids t_a ∈ α.L do
              if ∃ tid t_b ∈ β.L such that t_a > t_b then
                  Φ_{y↦x}.Add_ctid_list(cid, t_a);

        if(Φ_{xy} ≠ ∅) then
          for all tids t_b ∈ β.L do
              if ∃ tid t_a ∈ α.L such that t_b = t_a then
                  Φ_{xy}.Add_ctid_list(cid, t_b);
```

Figure 3.15: Pseudocodes of Intersection() and Contains() routines.

P↦X	
CID	TID
1	30
1	40
2	60
3	40
4	10
4	30
4	50
4	80
5	10
5	50
5	70
8	50
8	60
8	70

P↦Y	
CID	TID
1	70
1	80
2	60
3	40
4	30
4	40
4	50
4	70
5	10
6	50
6	65
7	20
7	35
8	50

P↦X↦Y	
CID	TID
1	70
1	80
4	30
4	40
4	50
4	80

P↦Y↦X	
CID	TID
4	50
4	80
5	50
5	70
8	60
8	70

P↦XY	
CID	TID
2	60
3	40
4	20
4	15
4	50
8	50

Figure 3.16: Ctid_list Intersections

```
Prune(β)
        for all (k-1)-subsequences, α ≺ β, do
            if([α_1]) has been processed, and α ∉ F_{k-1} then
                    return TRUE;
        return FALSE
end
```

Figure 3.17: Pseudocode of the Prune algorithm

For the *equality* check we simply traverse the two ctid_lists and insert matching *(cid, tid)* pairs into XY.ctid_list. For the *follows* check for X $\mapsto$ Y, we insert into X $\mapsto$ Y.ctid_list all *tids* $\in X$ greater than some *tid* in Y for the matching *cid*s. Finally, for the *follows* check for Y $\mapsto$ X, we insert into Y $\mapsto$ X.ctid_list all *tid*s in Y greater than some *tid* in X.

Pruning Candidates

Equivalence classes are processed in descending order to facilitate candidate pruning. The pruning algorithm is shown in Figure 3.17. We know that all subsequences of a frequent sequence are frequent. If we can determine that any subsequence of an candidate sequence is not frequent, then we do not perform Intersection() for that candidate sequence and go on for the next candidate sequence. This speeds up the Enumerate_Frequent_Sequences algorithm. Let's examine the Prune() algorithm:

Let α_1 denote the first item of sequence α. Before generating the ctid_list for a new k-sequence β, we check whether all the subsequences, $\alpha \prec \beta$, of length k-1 are frequent. If they all are frequent, then we perform the ctid_list intersection. Otherwise, β is dropped from computation.

For example consider a sequence β=(D $\mapsto$ BF $\mapsto$ A). The 3-length subsequences (D $\mapsto$ BF), (D $\mapsto$ B $\mapsto$ A), and (D $\mapsto$ F $\mapsto$ A) are all elements of the class [D]. So, if any of them is not present in equivalence class [D], then β=(D $\mapsto$ BF $\mapsto$ A) is not frequent also.

Search for Frequent Sequences

We will discuss two main strategies for enumerating the frequent sequences within each equivalence class: breadth-first and depth-first search.

1. *Breadth-First Search (BFS)*: In breadth-first search, the equivalence classes generated by Enumerate_Frequent_Sequences() routine recursively is pro-

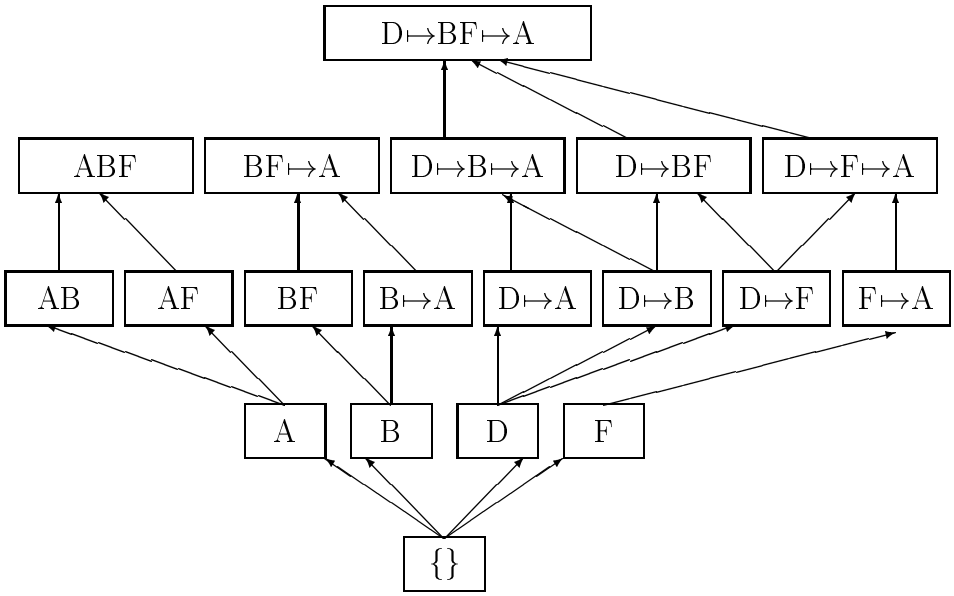

Figure 3.18: Lattice Induced by Maximal Sequence D $\mapsto$ BF $\mapsto$ A.

cessed in a bottom-up manner. All the (k-1)-length sequences are processed before moving on to the k-length sequences. For example in Figure 3.18, we process the equivalence classes {[D $\mapsto$ A], [D $\mapsto$ B], [D $\mapsto$ F]}, before moving on to the classes {[D $\mapsto$ B $\mapsto$ A], [D $\mapsto$ BF], [D $\mapsto$ F $\mapsto$ A]}, and so on.

2. *Depth-First Search (DFS)*:

   In a depth-first search, all sequences of any k-length of an equivalence class are completely processed along one path before moving on to the next path. For example, we process the classes in the following order [D $\mapsto$ A], [D $\mapsto$ B], [D $\mapsto$ B $\mapsto$ A], [D $\mapsto$ BF], [D $\mapsto$ BF $\mapsto$ A], and so on.

The advantage of BFS over DFS is that we have more information available for pruning. For example, we know the set of 2-sequences before constructing the 3-sequences, while this information is not available in DFS. On the other hand DFS requires less main-memory than BFS.

Enumerating Frequent Sequences

Basically all processors does the same computation for computing F_k, but do not synchronize with other processors and work over only its own input data. The input to the Enumerate_Frequent_Sequences(S) routine is the equivalence class S which is composed of F_1 and F_2 sequences and their vertical ctid_lists.

```
Enumerate_Frequent_Sequences(S)
        for all atoms A_i ∈ S do
            T_i = ∅;
            for all atoms A_j ∈ S, with j > i do
                R=A_i ⋃ A_j;
                if (Prune(R)==FALSE) then
                    L(R)=Intersect(L(A_i), L(A_j));
                    if σ(R) ≥ Min_sup then
                        S_i = S_i ⋃{R};
                        F_|R| = F_|R| ⋃{R};
            end;
            if (DFS) then Enumerate_Frequent_Sequences(T_i)
        end
        if (BFS) then
          for all T_i ≠ ∅ do Enumerate_Frequent_Sequences(T_i)
        end
```

Figure 3.19: Pseudocode of computing F_k

We then enter the iterative processing phase. At each new level, we firstly generate candidate sequences by using candidate sequence generation rules on F_{k-1}. Frequent sequences, F_k, are determined by intersecting the vertical ctid_lists of F_{k-1} elements of F_k and checking the resulting vertical ctid_list against minimum support value. Before intersection step, Prune() routine is called for ensuring that all the subsequences of the processed candidate sequence are frequent. If Prune() returns "false", then we go ahead with the next candidate sequence. The frequent sequences are inserted into equivalence class S to recursively generate new frequent sequences of increasing sequence lengths until all frequent sequences with prefix S is found.

The depth-first search requires to store vertical ctid_lists of processed candidate sequence and its subsequences. Breadth-first search needs the all sequences of F_{k-1} and omits all vertical ctid_lists of F_{k-1} after computing F_k

3.3.4 Disk Scans

During GenF_1, all the item ctid_lists are scanned from local disk into memory in one pass on the database. During GenF_2, only the ctid_lists of frequent 1-sequences are inverted into horizontal format in one pass on the database. To compute all frequent sequences which have length 3 or more, only once the database is accessed during process of Form_F_k_Ctid_List() routine. Thus,

it is claimed that *dSpade* algorithm will require a single database scan after computing F_2, in contrast to the approaches in [10] which require multiple scans.

Chapter 4

Experiments and Results

In this chapter, results of various experiments that have been conducted are presented in order to show the effects of size of data and minimum support on the parallel performance. The first section describes the synthetic datasets used in experiments. Then, the implementation details are presented. Finally, in the last section results of various experiments are discussed.

4.1 Synthetic Datasets

We used the publicly available dataset generation code from the IBM Quest data mining project [7]. These datasets mimic real-world transactions, where people buy a sequence of sets of items. Some customers may buy only some items from the sequences, or they may buy items from multiple sequences. The customer sequence size and transaction size are clustered around a mean and a few of them may have many elements. The different dataset generation parameters are listed in Figure 4.1.

Parameter	Description
D	Number of customers
C	Average number of transactions per customer
T	Average number of items per transactions
S	Average number of itemsets in maximal potential frequent sequences
I	Average number of items in maximal potential frequent itemsets
N	Number of items
N_S	Number of maximal potential frequent sequence
N_I	Number of maximal potential frequent itemsets

Figure 4.1: Dataset Generation Parameters

Dataset	C	T	S	I	D	Size(MB)
C10-T2.5-S4-I1.25-D200K	10	2.5	4	1.25	200 000	39.8
C10-T2.5-S4-I1.25-D400K	10	2.5	4	1.25	400 000	81.5
C10-T2.5-S4-I1.25-D800K	10	2.5	4	1.25	800 000	163.2

Figure 4.2: Dataset Generation Parameters

The datasets are generated by following the steps listed below:

- N_I maximal itemsets of average size I are generated by choosing from N items.
- N_S maximal sequences of average size S are created by assigning itemsets from N_I to each sequence.
- A customer of average C transactions is created, and sequences in N_S are assigned to different customer elements, respecting the average transaction size of T.
- The generation stops when D customers have been generated.

The default values of $N_S = 5000$, $N_I = 25000$ and N = 10000 are selected. We refer the reader to [7] for detailed information on the datasets generation.

Figure 4.2 shows the datasets with their parameter settings. After generating the synthetic dataset in horizontal data layout by using the parameters listed above, the dataset is transformed into vertical data layout offline. The whole database is partitioned into chunks of data according to the number of processors. Thus, each processor computes on one chunk of data independently.

4.2 Implementation of *dSpade*

All of the algorithms and related data structures were implemented in C++ programming language and LAM implementation of MPI [9, 4]. LAM is a parallel processing environment and development system for a network of independent computers. It features the *Messape-Passing Interface (MPI)* programming standart, supported by extensive monitoring and debugging tools.

The experiments are conducted on a Beowulf Cluster. Beowulf systems are high performance parallel computers built with cheap commodity hardware connected with a low latency and high bandwith interconnection network, and equipped with free system software such as GNU/Linux or FreeBSD. The hardware of the *Borg* consists of three components:

C10-T2. 5-S4-I1. 25-D200K				
# of processors	F_1 time	F_2 time	F_k time	Total time
1	2.834795	35.317450	1.276176	39.461353
2	0.192478	19.371439	1.772814	21.365688
4	0.079038	18.065061	3.969298	22.138034
8	0.059240	21.222039	4.724835	26.082739
16	0.053348	23.580715	5.489792	29.229082

Figure 4.3: Min_Sup=0.5%

C10-T2.5-S4-I1.25-D800K				
# of processors	F_1 time	F_2 time	F_k time	Total time
4	2.831116	100.253172	14.879093	118.118775
6	1.332462	48.686530	20.145217	70.274802
8	3.060354	29.904967	23.205833	56.225599
10	2.115730	20.212355	27.471458	49.868859
12	1.010077	17.625880	30.375910	49.115820
14	0.093599	17.781143	33.564061	51.53164
16	0.087551	19.134109	34.795408	54.119033

Figure 4.4: Min_Sup=0.5%

1. *NODES:* There are 32 identical nodes with Intel Pentium II 400 Mhz CPU, 64 MB PC100 RAM, 6 GB UDMA IDE hard drive and Intel EtherExpress Pro 10/100 NIC.

2. *INTERCONNECTION NETWORK:* The interconnection network is a 3COM SuperStack II 3900 smart switch which has 100Base-TX ports and a Gigabit uplink. The ports connect to nodes and uplink connects to the interface computer.

3. *INTERFACE COMPUTER:* The interface computer is a workstation with Intel Pentium III 500 Mhz CPU, 512 MB RAM, 26 GB hard drive. It has a Gigabit NIC which connects to the uplink of a switch and fast Ethernet to connect to the Net. The Interface Computer provides communication with developers through console and network.

4.3 Experiments

Figure 4.5 and 4.6 shows the execution times for each step of *dSpade* algorithm for C10-T2.5-S4-I1.25-D800K dataset for 0.6% and 0.8% minimum support values. Figures 4.7 and 4.8 shows the total execution time and the speedup ratio charts for database C10-T2.5-S4-I1.25-D800K with Min_Sup=0.6(%). The

C10-T2.5-S4-I1.25-D800K				
# of processors	F_1 time	F_2 time	F_k time	Total time
4	2.351926	47.935984	5.144520	55.520032
6	4.680082	26.964325	6.219496	37.912134
8	3.898711	18.544761	7.240789	29.741549
10	2.116996	16.206067	8.964333	25.346578
12	1.102453	13.450797	7.477851	22.103976
14	0.090548	10.444257	9.952649	20.557574
16	0.087476	8.575592	9.262332	17.029163

Figure 4.5: Min_Sup=0.6%

C10-T2. 5-S4-I1. 25-D800K				
# of processors	F_1 time	F_2 time	F_k time	Total time
4	10.611414	25.339592	0.157043	36.162087
8	5.132765	14.034057	0.207893	19.411805
16	0.087098	10.530834	0.311820	11.028449

Figure 4.6: Min_Sup=0.8%

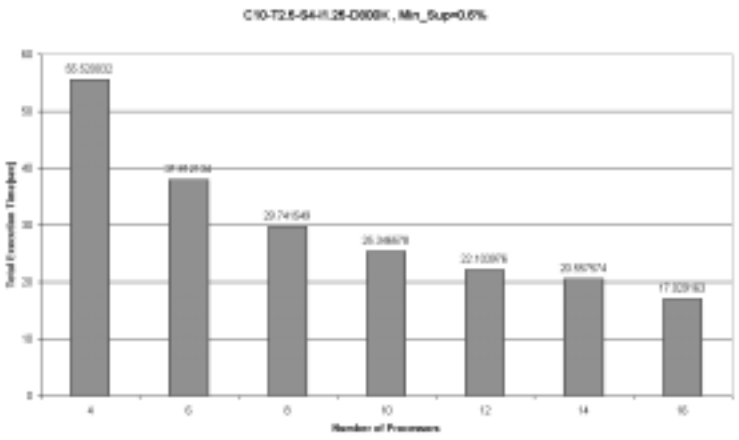

Figure 4.7: Execution Time [sec]

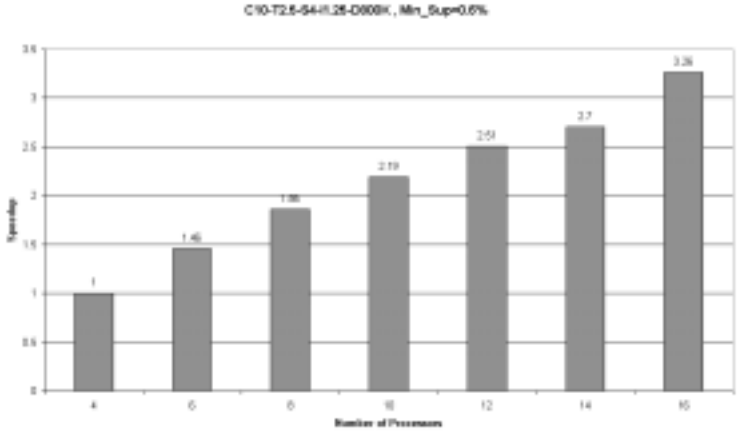

Figure 4.8: Speedup

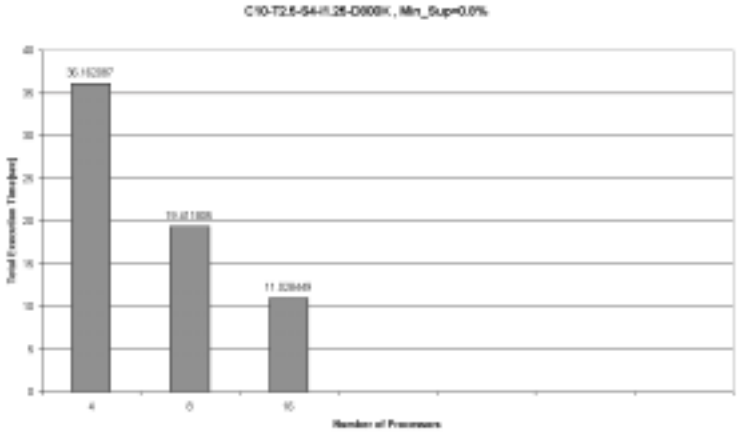

Figure 4.9: Execution Time [sec]

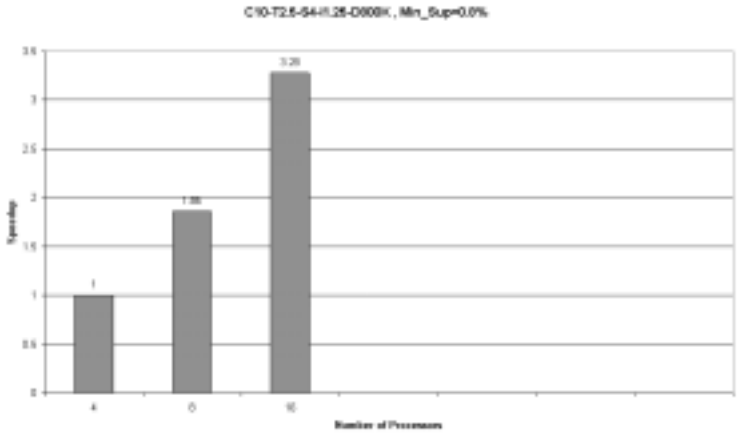

Figure 4.10: Speedup

C10-T2.5-S4-I1.25-D800K					
# of processors	F_1 time	F_2 time	F_k Comm_time	F_k time	Total time
4	9.081542	44.290284	3.066090	5.098326	58.534853
8	4.317009	21.500498	3.183651	6.524056	32.390020
16	0.086916	10.588627	4.483058	8.539685	19.309471

Figure 4.11: Min_Sup=0.6%

C10-T2.5-S4-I1.25-D800K					
# of processors	F_1 time	F_2 time	F_k Comm_time	F_k time	Total time
4	8.156739	106.246041	9.326660	13.784408	128.277421
8	4.490147	45.996040	10.022598	22.243071	72.803361
16	0.081041	29.211027	18.420772	35.372381	64.763782

Figure 4.12: Min_Sup=0.5%

chart is normalized with the execution time on 4 nodes. We used this normalization because at lower number of nodes than 4, the size of the processed data does not fit in memory and performs very poorly. To get reliable results on speedup ratio, we selected minimum optimal number of nodes as 4. We obtain good speedup ratio for 8 processors, ranging as high as 1.86. On 16 processors, we obtained a maximum of 3.26. Figure 4.9 and 4.10 shows the total execution time and the speedup ratio charts for database C10-T2.5-S4-I1.25-D800K with Min_Sup=0.8(%). We obtain good speedup ratio for 8 processors, ranging as high as 1.86. On 16 processors, we obtained a maximum of 3.28. As these charts indicate, *dSpade* achieves good speedup performance.

But, in some cases dSpade performs poorly. If we analyze the Figures 4.4 and 4.3, we will see two main problems with the performance of *dSpade*:

1. *dSpade* performs poorly if the size of database decreases. For C10-T2.5-S4-I1.25-D200K dataset with 0.5% minimum support value, as the number of processors increases the execution time also increases. In Figure 4.4, the execution time is nearly halved at 2-processors, but increased at 4, 8, 16-processors. Communication overhead defeats the gain from computation as the number of processors increases at low size of databases.

2. *dSpade* performs poorly at some experiments with decreasing minimum support value on very large databases. Every node keeps a piece of vertical ctid_list for all items in the database. The Form_F_k_Ctid_List() routine collects these vertical ctid_lists of frequent items, which are inserted into equivalence classes, from nodes and broadcasts the complete ctid_list of every frequent item to other nodes. As the number of nodes increases, the communication time increases and speedup ratio decreases.

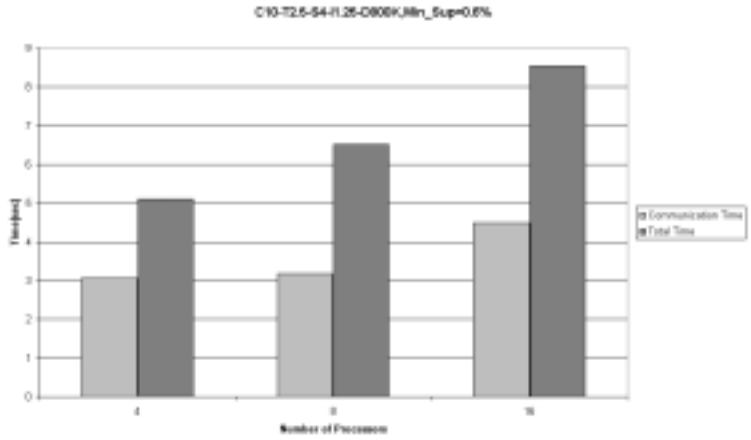

Figure 4.13: F_k Execution Time [sec]

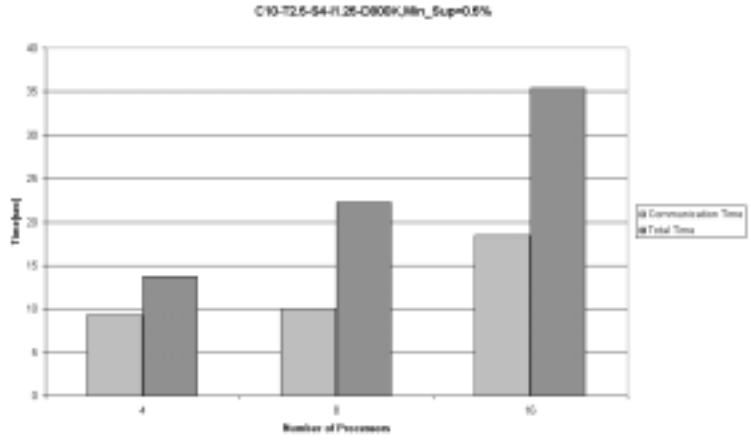

Figure 4.14: F_k Execution Time [sec]

```
Optimized_Form_F_k_Ctid_List(C_i)
        for all elements a ∈ C_i do
            Read_Ctid_List(a);
            only related processor Gather_Ctid_List(a);
            for all elements b ∈ F_2 and b ∈ C_i do
                Read_Ctid_List(b);
                only related processor Gather_Ctid_List(b);
                only related processor Intersect(a, b);
        end
```

Figure 4.15: Pseudocode for Optimized_Form_F_k_Ctid_List(i)

Figures 4.13 and 4.14 shows the percentage of communication time during F_k computation time. With decreasing minimum support value, *dSpade* finds increasing number of frequent sequences. The experiment conducted for C10-T2.5-S4-I1.25-D800K dataset with 0. 5% minimum support value, as shown in Figure 4.3, *dSpade* speeds down with increasing number of processors, since communication overhead increases with decreasing minimum support value and increasing number of processors.

We designed a new algorithm to overcome these drawbacks, but not implemented yet. We call this algorithm as Optimized_Form_F_k_Ctid_List(). The input to the Optimized_Form_F_k_Ctid_List() is a mapped equivalence class of the processor. Simply, in this algorithm each processor gathers ctid_lists of items from other processors to create ctid_lists of F_2 ,which are elements of the mapped equivalence classes to itself. Thus the moved data amount is decreased if compared with implemented algorithm.

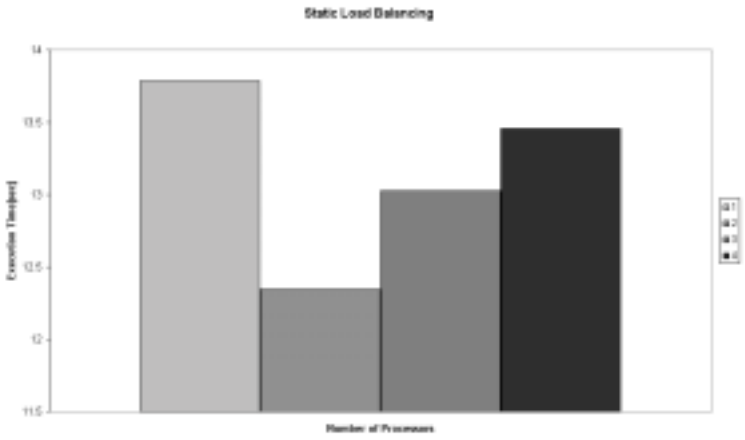

Figure 4.16: F_k Execution Time [sec]

C10-T2.5-S4-I1.25-D400K										
Min_Sup(%)	F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8	F_9	F_{10}
1	562	10	0	0	0	0	0	0	0	0
0. 75	1014	55	3	0	0	0	0	0	0	0
0. 5	1773	421	182	53	11	1	0	0	0	0
0. 25	3086	3550	3620	3228	2385	1429	589	131	15	1

Figure 4.17: Number of frequent sequences

We study the effect of static load balancing approach on the total elapsed execution time of each processor. It is shown in Figure 4.16 that static load balancing approach creates ignoreable amount of load imbalance and does not cause speeddown in execution time.

Finally, we study the effect of changing minimum support on the parallel performance, as shown in Table 4.17. The minimum support was varied from a high of 0.25% to a low of 1%. Table 4.17 shows the number of frequent sequences discovered at the different minimum support levels. The number of frequent sequences are not linear with respect to the minimum support. But the total execution time increases linearly with decreasing value of minimum support.

Chapter 5

Conclusion

In this thesis work, we presented *dSpade*, a parallel algorithm based on spade for discovering the set of all frequent sequences, targeting distributed-memory systems. In *dSpade*, *horizontal database partitioning method* is used, where each processor stores equal number of customer transactions.

dSpade is a synchronous algorithm for discovering frequent 1-sequences (F_1) and frequent 2-sequences (F_2). Each processor performs the same computation on its local data to get local support counts and broadcasts the results to other processors to find global frequent sequences during F_1 and F_2 computation. After discovering all F_1 and F_2, all frequent sequences are inserted into lattice to decompose the original problem into equivalence classes. Equivalence classes are mapped in a greedy heuristic to the least loaded processors in a round-robin manner. Finally, each processor asynchronously begins to compute F_k on its mapped equivalence classes to find all frequent sequences.

As in *Spade* algorithm, *dSpade* usually makes only three database scans. It has also good scaleup properties with parameters of minimum support and number of customer transaction.

5.1 Future Work

dSpade performs poorly at some experiments with decreasing minimum support value on very large databases. We designed Optimized_Form_F_k_Ctid_List() algorithm to to solve this problem, but not implemented yet.

Static load balancing approach creates ignoreable load imbalance and does not cause speeddown in execution time. But by using dynamic load balancing techniques, improvements on performance results can be achieved.

Bibliography

[1] R. Agrawal and R. Srikant. Mining sequential patterns. In *11th ICDE Conference*, 1995.

[2] R. Agrawal and R. Srikant. Mining sequential patterns:generalizations and performance improvements. In *5th Intl.Conf.Extending Database Technology*, March 1996.

[3] B. A. Davey and H. A. Priestly. *Introduction to Lattices and Order*. Cambridge University Press, 1990.

[4] Message Passing Interface Forum. *MPI:A Message-Passing Interface Standart*. Available by anonymous ftp from ftp.mcs.anl.gov, June 1995.

[5] Alex A. Freitas and Simon H. Lavington. *Mining very large databases with parallel processing*. Kluwer Academic Publishers, 1998.

[6] H. Toivonen H. Mannila and I. Verkamo. Discovering frequent episodes in sequences. In *1st Intl. Conf.KDD*, 1995.

[7] http://www.almaden.ibm.com/cs/quest/syndata.html. *Quest Project*. IBM Almaden Research Center,San Jose,CA 95120.

[8] H. Mannila and H. Toivonen. Discovering generalized episodes using minimal occurences. In *2nd Intl. Conf.KDD*, 1996.

[9] Peter S. Pacheco. *Parallel Programming with MPI*. Morgan Kaufmann Publishers,Inc., 1997.

[10] T. Shintani and M.Kitsuregawa. Mining algorithms for sequential patterns in parallel:hash based approach. In *Pacific-asia Conf. on KDD*, April 1998.

[11] Anshul Gupta Vipin Kumar, Ananth Grama and George Karypis. *Introduction to Parallel Computing: Design and Analysis of algorithms*. Benjamin-Cummings Addison-Wesley Publishing Company, 1994.

[12] Mohammed J. Zaki. Efficient enumeration of frequent sequences. In *7th Intl. Conf. on Information and Knowledge Management*, November 1998.

[13] Mohammed J. Zaki. Parallel sequence mining on shared-memory machines. In *Intl. Conf. of WPDM-KDD*, November 1999.

Index