A SAMPLING-BASED FRAMEWORK FOR PARALLEL MINING FREQUENT PATTERNS

BY

SHENGNAN CONG

B.E., Tsinghua University, 2000
M.S., University of Illinois at Urbana-Champaign, 2003

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2006

Urbana, Illinois

Abstract

Data mining is an emerging research area, whose goal is to discover potentially useful information embedded in databases. Due to the wide availability of huge amounts of data and the imminent need for turning such data into useful knowledge, data mining has attracted a great deal of attention in recent years.

Frequent pattern mining has been a focused topic in data mining research. The goal of frequent pattern mining is to discover the patterns whose numbers of occurrence are above a predefined threshold in the datasets. Depending on the different definition of pattern, frequent pattern mining stands for various mining problems, such as frequent itemset mining, sequential pattern mining and so on. Frequent pattern mining has numerous applications, such as the analysis of customer purchase patterns, web access patterns, natural disasters or alarm sequences, disease treatments and DNA sequences.

Many algorithms have been presented for mining frequent patterns since the introduction of the problem. Most of them are sequential algorithms and their execution time is constrained by the computing power, I/O speed and/or memory space of the uni-processor system where their implementations execute. The irregular nature of the datasets and the requirement to reach global decision make it challenging to efficiently mining frequent patterns in parallel.

In this dissertation, we propose a framework for parallel mining frequent patterns. Our parallelizing framework targets a distributed memory system. It was applied to the parallelization of frequent itemset mining, sequential pattern mining and closed-sequential pattern mining. Our parallel algorithms are based on the most efficient serial algorithms.

We devised a partitioning strategy to distribute the work to avoid the unnecessary inter-processor communication. Our task partition is based on the divide-and-conquer property of frequent pattern mining problem so that the processors can work asynchronously and independently during mining.

We discuss the load balancing problem inherent in the divide-and-conquer property and then present a sampling technique, called *selective sampling*, to address the load balance problem. Selective sampling strategy allows us to predict the relative time required to mine the projections and in this way enable us to identify large tasks, decompose them, and evenly distributed them across processors to achieve load balancing.

We implemented parallel algorithms for mining frequent itemsets, sequential patterns and closed-sequential patterns following our framework. A comprehensive performance study has been conducted in our experiments on both synthetic and real-world datasets. The experimental results have shown that our parallel algorithms have achieved good speedups on various datasets and the speedups are scalable up to 64 processors on our 64-processor system.

To Gang, and my parents.

Acknowlegements

First and foremost, my deepest gratefulness goes to my advisor, Professor David Padua. I would like to sincerely thank him for providing me with invaluable consistent guidance and support throughout my whole graduate study. I benefited tremendously from his help and insights in many aspects of my life as a graduate student. He has provided me with a perfect balance between guidance and freedom, which allows me to pursue my own ideas along the right direction. I will always be grateful for his consistent encouragement.

I greatly appreciate Professor Jiawei Han for his support and collaboration during my PhD study. Thank you for all the interesting discussions on my research work. I would like to thank the honorable members of my thesis committee, Dr. Jay Hoeflinger, Professor Marc Snir and Professor Laxmikant Kale, for their precious discussions on the ideas in my work. I am also indebted to them for reading my dissertation and providing valuable feedbacks.

I would like to extend my thanks to Professor Maria Garzaran for her assistance during my thesis research and for her warm help during my thesis preparation and job search.

My gratitude also goes to all current and previous members of Polaris group, especially Jianxin Xiong, Peng Wu, Jiajing Zhu, Zehra Sura, Xiaoming Li, David Wong and Ganesh Bikshadi. I also thank former members of I-ACOMA group, Michael Huang, Jose Renau, Yan Solihin and Milos Prvulovic for answering my questions and giving me advice.

I am grateful to Sheila Clark for her great administrative support during my graduate study.

Finally, I am deeply thankful to my parents for their endless encouragement and love. Last but not least, I thank my dear husband, Gang, for bringing so much fun to my life and sharing every moment of my success and frustration. You are my true companion in both work and life. Your love and support have been and will always be my most precious possession.

Table of Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

Data mining is a process to extract interesting and potentially useful patterns from a collection of data [16]. Due to the wide availability of datasets and the need for turning such data into useful information, data mining has attracted a great deal of attention in recent years.

One of the canonical tasks in data mining is to discover association rules. Association rule mining is the foundation of many essential data mining tasks, such as correlation [8], sequential patterns [5], cluster analysis [7] and associative classification [21].

A typical application of association rule mining is the analysis of sales data [45]. An association rule identifies the set of items that are most often purchased with another set of items. For example, an association rule may state that "90% of customers who bought milk and bread also bought eggs." This type of information may be used to decide catalog design, store layout, product placement, target marketing and other marketing strategy problems.

The task of mining association rules can be decomposed into two steps [4]. The first step is to count the number of occurrences of all the patterns that appear frequently in the dataset. The second step is to find the implication association rules among the frequent patterns discovered in the first step. For example, if there are 100 customers who bought A and B and there are 90 customers who bought A, B and C together, we can obtain the association rule, "90% of the costomers who buy A and B also buy C at the same time". Due to the exponential search space of the first step, most research efforts focus only on the first step, referred as *frequent pattern mining.* The second step is relatively straightforward and not as computationally intensive as the first step.

Frequent pattern mining actually refers to a series of different data mining problems. Depending on the different definition of *pattern*, frequent pattern mining stands for various mining problems, such as *frequent itemset mining*, *sequential pattern mining*, *closed pattern mining*, *maximal pattern mining* and so on. The example discussed above is a typical frequent itemset mining problem where pattern refers to a set of items.

Frequent pattern mining problems have broad applications such as the analysis of customer purchase patterns, web access patterns, natural disasters or alarm sequences, disease treatments and DNA sequences. In addition, frequent pattern mining also plays an essential role in many other data mining tasks, such as correlations [8], cluster analysis [7] and classification [21].

Many algorithms have been introduced for frequent itemset mining, sequential pattern mining, and other frequent pattern mining problems[3, 22, 18, 17, 23, 6, 31, 12, 39, 40, 43]. Most of them are sequential algorithms and their execution time is constrained by the computing power, I/O speed and/or memory space of the uni-processor system where their implementations execute. Sometimes the target machine lacks the power to process large datasets from the real world in a reasonable time.

Distributed memory systems, such as clusters, have become widely available and affordable. Distributed memory systems have high flexibility, scalability, low cost performance ratio and easy connectivity. Consequently, it is important to investigate parallel data mining algorithms to take advantage of the computation and I/O power of distributed memory systems, as well as their aggregate memory spaces. This is the objective of this thesis.

1.2 Challenges

Data mining requires progressive information collection and manipulation. The information gathering process usually calls for numerous iterations of data comparison and analysis and requires a lot of computational resources. This makes the information gathering process an ideal candidate for parallel processing. However, to achieve efficient parallel data mining is not straightforward.

First, the amount of data communication between computing nodes required to obtain a global count for each (frequent) pattern can be prohibitively large to the point of eliminating the benefits gained from parallelization.

Second, it is challenging to evenly distribute the mining work to the processors in order to attain load balancing. Due to the irregular characteristics of databases, distributing and balancing the mining tasks between the processors without jeopardizing the global solution is a difficult problem..

In general, there are several problems that must be addressed in the development of parallel algorithms for data mining problems.

1. ***Minimizing synchronization and communication***. An ideal parallel data mining algorithm allows all nodes to operate asynchronously, without having to stall frequently for global barriers or communication. To minimize data communication between nodes, it is better for frequent pattern mining problems to be implemented in parallel by assigning a different task to each node instead of naively partitioning the data across the distributed system.

2. ***Good data decomposition***. One of the benefits of parallel frequent pattern mining is that each node can potentially work on a reduced-size subset of the total database so that the space needed for each processor can be reduced, which is very important since some of most efficient sequential algorithms require a large memory space.

3. ***Effective load balancing***. To maximize the performance of parallel algorithms, each node needs to have approximately the same amount of work to do so that the work load tends to be even across the processors. Load balancing is one of the most difficult problems in parallel frequent data mining.

4. ***Minimizing I/O***. Parallel data mining algorithms tends to reduce I/O overhead if I/O could also be parallelized. However, to achieve the maximum performance on parallelized I/O operations, it important to arrange I/O operations carefully to avoid the competition for I/O bandwidth over a shared system bus.

Since the above aspects are not isolated from each other, it is extremely difficult to achieve all of these goals on parallel frequent pattern mining. Although a few research results have been reported on parallelizing frequent pattern mining problems, much requires to be done in this area.

1.3 Contributions

In this dissertation, we propose a framework for parallel mining frequent patterns. Our parallelizing framework targets a distributed memory system. It was applied to the parallelization of frequent itemset mining, sequential pattern mining and closed-sequential pattern mining. Our parallel algorithms are based on the most efficient serial algorithms.

We devised a partitioning strategy to distribute the work to avoid the unnecessary interprocessor communication. Our task partition is based on the divide-and-conquer property of frequent pattern mining problem so that the processors can work asynchronously and independently during mining.

The task partition following the divide-and-conquer property may result in imbalanced workload because the mining time of the large subtasks may be too large relative to the overall task mining time. The imbalanced workload greatly restricts the scalability of parallelism.

After discussing how the proposed strategy makes use of the divide-and-conquer strategy for parallelization, we discuss the load balancing problem inherent in the divide-and-conquer property and then present a sampling technique, called *selective sampling*, to address the load balance problem. Selective sampling strategy allows us to predict the relative time required to mine the projections and in this way enable us to identify large tasks, decompose them, and evenly distributed them across processors to achieve load balancing.

We implemented parallel algorithms for the frequent itemset mining problem, sequential pattern mining problem and closed-sequential-pattern mining problem following the framework proposed. Our implementation used MPI [24] and was evaluated on a Linux cluster with 64 processors. The performance study used both synthetic and real world datasets. The experimental results show that our parallel algorithms have good speedups on various datasets and the speedups are scalable up to 64 processors on our 64-processor system.

To summarize, the main contributions of this thesis are:

- We propose a framework for parallel mining frequent patterns based on the divide-and-conquer property of the frequent pattern mining problem.
- We describe the load balancing problem arising from the divide-and-conquer strategy of our parallel algorithms and present a sampling technique that addresses this problem and greatly enhances the scalability of the parallel algorithms.
- We implemented parallel algorithms for mining frequent itemsets, sequential patterns and closed-sequential patterns following our framework. We achieve fairly good speedups with various databases.

1.4 Dissertation Organizations

The rest of the dissertation is organized as follows. Chapter 2 describes data mining including the definition of the problems to be considered in this thesis and serial algorithms to solve them. We introduce our framework for parallelization in Chapter 3. In Chapter 4, we attack the load balance problem and discuss the sampling technique to achieve load balance in our framework. We will discuss other issues that must be considered to achieve performance scalability on a large number of processors in Chapter 5. Chapter 6 gives the description of the parallel frequent pattern mining algorithms in detail by applying the techniques we proposed. The experimental and performance results are presented in Chapter 7. We review related work in Chapter 8. Finally, Chapter 9 concludes this dissertation.

Chapter 2

Background

This chapter presents the formal definitions of three specific frequent pattern mining problems and introduces some popular serial algorithms for these problems.

2.1 Problem definition

Depending on the definitions of *pattern*, the term "'frequent pattern mining"' stands for one of a set of specific problems such as frequent itemset mining, sequential-pattern mining, closed-sequential pattern mining and so on. Their formal definitions are given in this section.

2.1.1 Frequent itemset mining

The frequent pattern mining problem was first introduced by R. Agrawal, et al. in [4] as mining association rules between sets of items. Thus we first define frequent itemset mining problem as follows.

Let $I = \{a_1, a_2, ..., a_m\}$ be a set of *items*. An *itemset* $X \subseteq I$ is a subset of items. An itemset with l items is called an *l-itemset*.

A *transaction* $T = (tid, X)$ is a tuple where *tid* is a transaction identifier and X is an itemset. A transaction $T = (tid, X)$ is said to *contain* itemset Y if $Y \subseteq X$.

A *transaction database* DB_T is a set of transactions. The *support* of an itemset X in transaction database DB_T, denoted as $sup(X, DB_T)$, is the number of transactions in DB_T containing X, i.e.,

$sup(X, DB_T) = |\{(tid, Y)|((tid, Y) \in DB_T) \wedge (X \subseteq Y))\}|$

Transaction	Items
T1	Milk, bread, cookies, juice
T2	Milk, juice
T3	Milk, eggs
T4	Bread, cookies, coffee

Figure 2.1: Simple example for frequent itemset mining

If there is only one transaction database used in the discussion, we will often ignore the second parameter of the *sup* operator and represent it as $sup(X)$.

Given a user-specified support threshold *min_sup*, X is called a *frequent itemset*, if $sup(X) \geq min\_sup$.

Definition 1 *The problem of mining frequent itemsets is to find all the frequent itemsets (or complete set) in a transaction database DB_T with respect to a given support threshold min_sup.*

Here is a simple example. Figure 2.1 is a database containing 4 transactions. The set of items in the database is $\{milk, bread, cookies, juice, eggs, coffee\}$. Given a support threshold as 2, frequent itemset mining problem is to find the following list of itemsets: $\{milk\}$:3, $\{bread\}$:2, $\{cookies\}$:2, $\{juice\}$:2, $\{milk, juice\}$:2 and $\{bread, cookies\}$:2, where the number after the column is the *support* of each itemset. For example, $\{milk\}$ is a 1-itemset with the support of 3 while $\{milk, juice\}$ is a 2-itemset with the support of 2.

Association rules can be derived from frequent patterns. An *association rule* is an implication of the form $X \Longrightarrow Y$, where X and Y are itemsets[1] and $X \cap Y = \emptyset$. The rule $X \Longrightarrow Y$ has *support* s in a dataset DB_T if $sup(X \cup Y) = s$. The rule $X \Longrightarrow Y$ holds in the transaction DB_T with *confidence* c where $c = \frac{sup(X \cup Y)}{sup(X)}$.

For example, in the example of Figure 2.1, the confidence of the association rule: $milk \Longrightarrow milk, juice$ is $\frac{2}{3}$ and the confidence of the association rule: $juice \Longrightarrow milk, juice$ is 1. This

[1]To simplify the description, we assume the pattern here refers to a set of items. The pattern can also be other type of objects, such as a sequence of itemsets.

implies that 66% of the customer who bought *milk* also bought *juice*, while all the costomer (100%) who bought *juice* also bought *milk*.

Given a dataset DB_T, a support threshold *min_sup* and a confidence threshold *min_conf*, the problem of *association rule mining* is to find the set of all association rules that have support and confidence no less than some user-specified thresholds.

Association rule mining can be divided into two steps. First, frequent patterns with respect to support threshold *min_sup* are mined. Second, association rules are generated with respect to confidence threshold *min_conf*. Since the first step, mining frequent patterns, cost significantly more time than the second step, most previous studies (e.g., [4]) and this thesis focus on the frequent pattern mining step.

2.1.2 Sequential-pattern mining

Sequential-pattern mining can be treated as an extension of frequent-itemset mining in the temporal domain. The problem of sequential-pattern mining arises naturally in many applications and was first introduced in [5]. The goal of sequential-pattern mining is to discover a sequence of attributes across time shared among a large number of objects in a given database.

For example, consider a database of web access, where an object is a web user and an attribute is a web page. The patterns to be discovered are the order in which the most frequently accessed sequences of pages at that site. This kind of information can be used to restructure the website. Similarly, marketing and sales data collected over a period of time provide sequences that can be analyzed and used for projections and forecasting. Moreover, many important knowledge discovery tasks in genomics require the analysis of DNA and protein sequences to discover frequently occurring motifs that correspond to evolutionary conserved functional units.

The problem of mining sequential patterns can be formally stated as follows:

Let $I = \{a_1, a_2, ..., a_n\}$ be a set of *items*. An *itemset* is a subset of items. A *sequence* is an ordered list of itemsets. A sequence s is denoted by $\langle s_1 s_2 ... s_l \rangle$, where s_j is an itemset, i.e., $s_j \subseteq I$ for $1 \leq j \leq l$. s_j is also called an *event* of the sequence, and denoted as $(x_1 x_2 ... x_m)$, where x_k is an item, i.e., $x_k \in I$ for $1 \leq k \leq m$. For brevity, the brackets are omitted if an event has only one item. That is, event (x) is written as x.

An item can occur at most once in an event of a sequence because an event is a set, but can occur multiple times in different events of a sequence. The number of items in a sequence is called the *length* of the sequence. A sequence with length l is called an *l-sequence.*

A sequence $\alpha = \langle a_1, a_2 ... a_n \rangle$ is called a **subsequence** of another sequence $\beta = \langle b_1, b_2 ... b_m \rangle$, denoted as $\alpha \sqsubseteq \beta$ or $\alpha \sqsubseteq \beta$, if there exist integers $1 \leq j_1 \leq j_2 \leq ... \leq j_n \leq m$ such that $a_1 \subseteq b_{j_1}$, $a_2 \subseteq b_{j_2}$,...,$a_n \subseteq b_{j_n}$.

A *sequence database* DB_S is a set of tuples $\langle sid, s \rangle$, where *sid* is a *sequence id* and s is a sequence. A tuple $\langle sid, s \rangle$ is said to *contain* a sequence α, if α is a subsequence of s, i.e., $\alpha \sqsubseteq s$.

The *support* of a sequence α in a sequence database DB_S, denoted as $sup(\alpha)$, is the number of tuples in the database containing α, i.e., $sup(\alpha) = |\{\langle sid, s \rangle | (\langle sid, s \rangle \in S) \wedge (\alpha \sqsubseteq s))\}|$

Given a user-specified value *min_sup* as the *support threshold*, a sequence α is called a *sequential pattern* in a sequence database DB_S if α is contained by at least *min_sup* sequences in the database, i.e., $sup(\alpha) \geq min\_sup$.

Definition 2 *Given a sequence database and a min_sup threshold, the problem of sequential pattern mining is to find the set of all sequential patterns in the database.*

2.1.3 Closed-sequential pattern mining

Since a long sequence contains a combinatorial number of subsequences, sequential pattern mining will generate an explosive number of frequent subsequences for long patterns.

For instance, assume the database contains only one long sequence $\langle (a_1)(a_2)...(a_{100}) \rangle$. If the minimum support threshold is 1, it will generate $2^{100}-1$ sequential patterns. However, all of these sequential patterns have the same support value as the longest one $\langle(a_1)(a_2)...(a_{100})\rangle$. To list all of these sequential patterns is prohibitively expensive in both time and space.

An interesting solution, called *closed-sequential patterns mining*, has been proposed to overcome this difficulty [28, 40]. A closed-sequential pattern is a sequential pattern that has no super-sequence with the same occurrence frequency.

The set of *closed-sequential pattern* is defined as follows.

Assume FS is the set of all sequential patterns and CS is the set of *closed-sequential patterns*. Then CS =$\{\alpha|\alpha \in FS \ and \ \nexists\beta \in FS \ such \ that \ \alpha \sqsubseteq \beta \ and \ sup(\alpha) = sup(\beta)\}$. The problem of *closed-sequential pattern mining* is to find *CS* with support value no less than the user-specified minimum support threshold.

Figure 2.2 gives an example of sequence database. The database has 3 sequences and 5 distinct items. If the minimum support threshold is set as 2, the CS set includes 4 subsequences: $\langle(ae)c\rangle$:2, $\langle(ae)d\rangle$:2, $\langle da\rangle$:3, $\langle dab\rangle$:2, while the corresponding *FS* set has 16 subsequences: $\langle a\rangle$:3, $\langle d\rangle$:3, $\langle b\rangle$:2, $\langle c\rangle$:2, $\langle e\rangle$:2, $\langle(ae)\rangle$:2, $\langle ac\rangle$:2, $\langle ad\rangle$:2, $\langle ec\rangle$:2, $\langle ed\rangle$:2, $\langle ab\rangle$:2, $\langle db\rangle$:2, $\langle dab\rangle$:2, $\langle(ae)d\rangle$:2, $\langle(ae)c\rangle$:2, $\langle da\rangle$:3. *CS* contains exactly the same information as *FS*, but includes much fewer number of subsequences.

Sequence_id	**Sequence**
10	*<(ae)cda>*
20	*<dab>*
30	*<d(abe)(bcd)>*

Figure 2.2: An example of sequence database

Mining closed-sequential pattern has been proved to be as powerful a solution to the mining of sequential patterns as the mining the complete set of sequential patterns [14, 28, 40]. Using the set of closed-sequential-patterns, a reduced set of association rules can be generated directly. Since there can be thousands of association rules that hold in a database, reducing the number of rules produced without information loss is an important improvement for the understandability of the result.

2.2 Serial algorithms for frequent pattern mining problems

In this section, we describe some popular serial algorithms for the frequent pattern mining problems, including frequent itemset mining, sequential pattern mining and closed-sequential pattern mining, all of which were defined in the previous section.

2.2.1 Serial algorithms for frequent itemset mining

Given d items in the dataset, there are 2^d possible candidate itemsets. A naive approach is to match each transaction in the dataset against every candidate itemset, count the number of occurrence for each itemset and then identify those appearing frequently enough. However, since the search space is exponential to the number of items occurring in the dataset and the dataset could be massive, containing millions of transactions, such a brute-force method is too computationally expensive and may not complete within a reasonable period of time.

The search space of all itemsets can be represented by a *subset-lattice*, with the empty itemset at the top and the set containing all items at the bottom. For instance, if the dataset has 5 items: A,B,C,D and E. The search space for mining the frequent itemset makes up the subset lattice as Figure 2.3 shown.

Notice that the support value of an itemset is greater than or equal to the support values of its supersets. This means that the support values of the itemsets are monotonically decreasing when moving from the top of the lattice to the bottom. Therefore, if an itemset is infrequent, all of its supersets must be infrequent and can be pruned from the search space.

There are basically two types of algorithms to mine frequent itemsets, *breadth-first search*

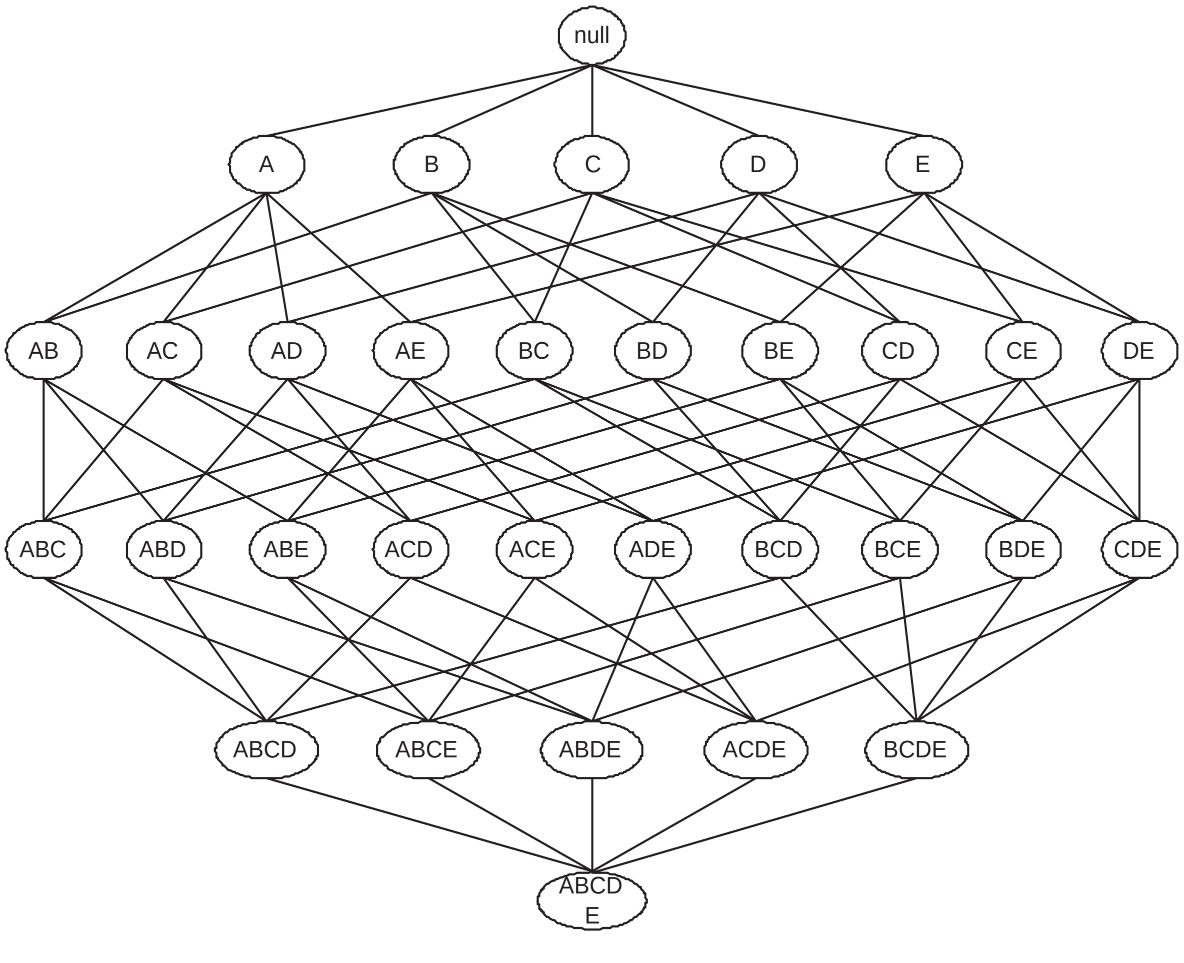


Figure 2.3: An example of subset-lattice

algorithms and *depth-first search* algorithms. The breadth-first algorithms scan the database testing all candidate itemsets level by level for frequency from the top of the lattice, while the depth-first algorithms search the lattice from a singleton itemset i and larger candidate sets are generated by adding one item at a time.

Breadth-first search algorithm

The best-known breadth-first algorithm is called *Apriori* algorithm [4]. The Apriori algorithm works in the following way:

1. A first scan of the dataset finds all of the frequent-1 itemsets.
2. The k-th ($k > 1$) scan uses the *seed set* of length-($k-1$) frequent itemsets found in the previous pass to generate new potential length-k itemsets, called *candidate itemsets.* A candidate frequent itemset is generate only if *all* of its subsets are frequent.
3. The k-th scan of the dataset finds the support value of every length k candidate itemsets. The candidates which pass the minimum support threshold are identified as frequent itemsets and become the seed set for the next pass.
4. The computation terminates when there is no frequent itemset found or there is no candidate itemset that can be generated in any pass.

Apriori algorithm was the first non-trivial algorithm developed for mining frequent itemsets. Later many variants [27, 35, 26, 25, 9] were proposed to optimize certain steps within the algorithm. However, the Apriori-like algorithms may bear the following costs, independent of their implementation techniques:

- The number of candidates to be generated may be huge, especially when the length of the frequent itemset is long. For example, to generate one frequent itemset $\{a_1, a_2, ..., a_{100}\}$, the number of candidates that has to be generated will be $2^{100} - 1 \approx 10^{30}$.

- Each scan of the dataset examines the entire dataset against the whole set of candidates, which is quite costly when the dataset is large and the number of candidates is numerous.

So the large number of candidates and the multi-scans of the datasets usually limit the performance of the breadth-first search algorithms.

Depth-first search algorithm

The depth-first search algorithms usually perform better than the breadth-first algorithms, especially when there exist long patterns or using low minimum support thresholds. Among the depth-first algorithms, *FP-growth algorithm* [18] is one of the fastest and best-known.

The FP-growth algorithm uses a tree structure, called *FP-tree*, to store the information about frequent itemsets. Every path from the root to a leaf node of the FP-tree represents a transaction in the database[2]. The items are organized in the tree in descending order of their frequencies. The items with higher frequency are located closer to the root. The FP-tree has a *header table* which contains all frequent items in the database (and their occurrence counts), represented as *Header(T)*. An entry in the header table is the starting point of a list that links all the nodes in the FP-tree referring to an item.

Here is a simple example of the FP-tree representation. Let the transaction database DB_T be as shown in Figure 2.4.(a) and the minimum support threshold be 3. The first scan of DB_T derives a list of frequent single items with their frequency, $\langle (f:4),(c:4),(a:3),(b:3),(m:3),(p:3)\rangle$. These items are ordered in frequency descending order. This ordering is important since the items will appear in the branches of the FP-tree following this order.

The second DB scan constructs the FP-tree for DB_T. For each transaction, its frequent items are first sorted in descending order of frequency (Figure 2.4.(b)) and then inserted as a branch into the tree. Unless the tree is empty, the insertion is conducted by coalescing

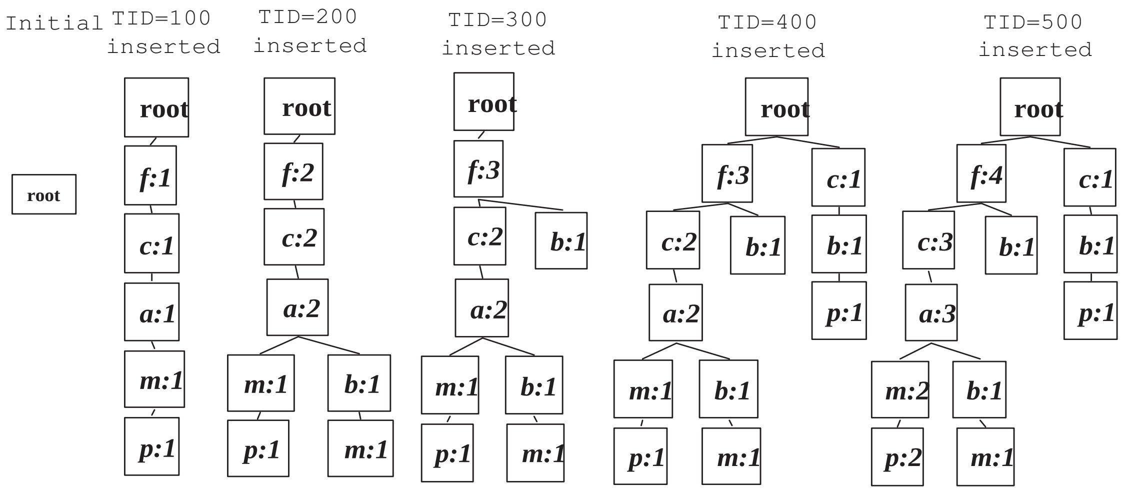
[2]Some transactions are corresponding to a path from the root to a non-leaf node. See the example in Figure 2.4 for details.

TID	Items bought
100	{f, a, c, d, g, i, m, p}
200	{a, b, c, f, l, m, o}
300	{b, f, h, j, o}
400	{b, c, k, s, p}
500	{a, f, c, e, l, p, m, n}

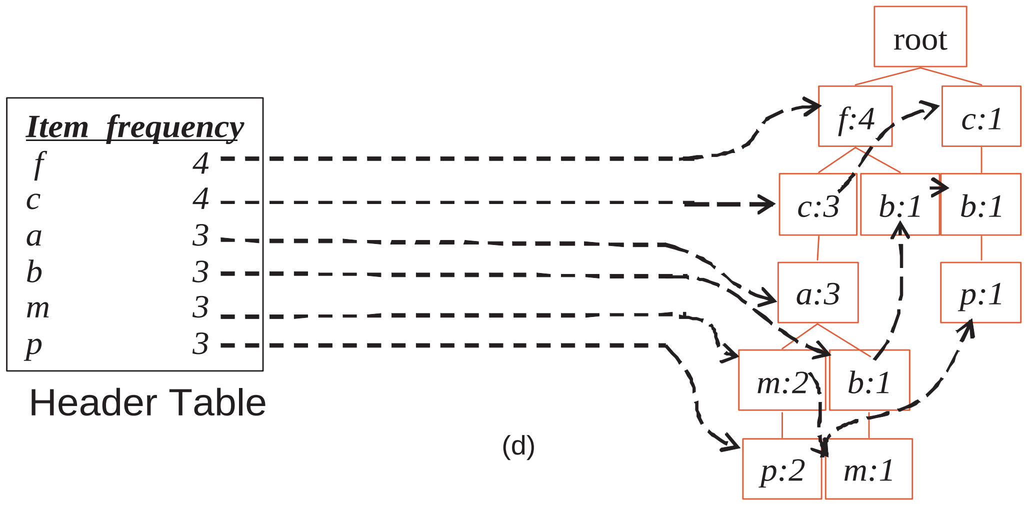
(a)

Ordered Frequent Items
{f, c, a, m, p }
{f, c, a, b, m }
{f, b }
{c, b, p }
{f, c, a, m, p }

(b)



(c)



(d)

Figure 2.4: Example for FP-tree construction

the shared (existing) prefix path and updating the counters of the nodes on the path (Figure 2.4.(c)). In addition, whenever there is a new node created, the node is added to the appropriate linked list of its item maintained by the header table. The complete FP-tree for the example database is as shown in Figure 2.4.(d).

After the FP-tree is built, it is mined in a divide-and-conquer way.

Suppose $\{X_1, X_2, ..., X_n\}$ is a set of itemsets and i is an item. We define an operator $\oplus$ as:

$i \oplus \{X_1, X_2, ..., X_n\} \equiv \{\{i\} \cup X_1, \{i\} \cup X_2, ..., \{i\} \cup X_n\}$.

The mining of the FP-tree T following the FP-growth algorithm can be formalized as function *F*() below where T is the FP-tree for the whole database and *P(i,T)* represents the *projected database* of i (i appears in the header table of T, that is, $i \in Header(T)$).

```
**function** *F*(*T*)
**begin**
    **if** (*T* is a chain) **return** set of all combinations of elements in *T*;
    **else return** $\bigcup_{i \in Header(T)}((i \oplus F(P(i,T))) \cup \{i\})$;
**end**
```

The projected database $P(i, T)$ consists of the transactions in the database containing i, with i and the items less frequent than i deleted. Given an item i in the header table, by following the linked list for i, all nodes representing i in the tree are visited. Then tracing the branches from i's nodes back to the root will form i's projected database. The items appearing in the projected database are sorted by frequency again and then the transactions in the projected database are built in an FP-tree structure with the infrequent items removed. The same support threshold is used when building the FP-tree for the projected database.

Figure 2.5 illustrates the projected database and the FP-tree for the projection of item m in the example in Figure 2.4. Item b does not appear in the FP-tree for m's projection because it is not *frequent* (b's support value with the projection is less than 3) in m's projected

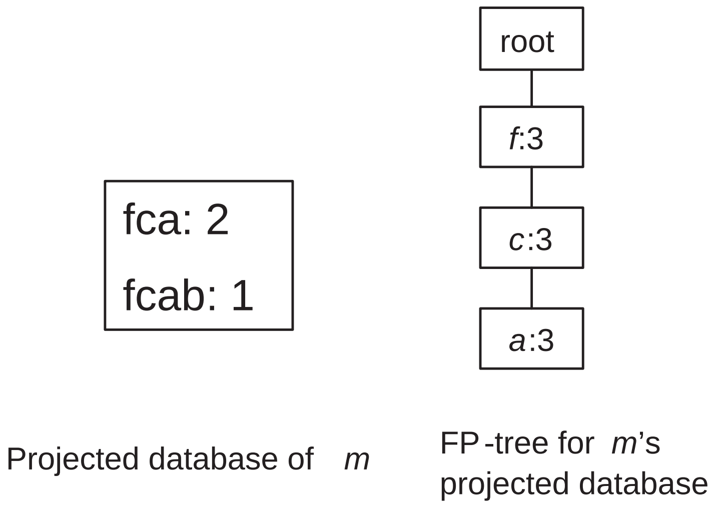


Figure 2.5: Example for tree projection

database.

After the FP-tree is built, FP-growth algorithm recursively mines the FP-tree until it becomes a chain, it returns the set of all the combinations of the elements of the chain. Notice that a FP-tree with a single item is a chain too.

In general, the worst-case time complexity of FP-growth algorithm is $O(2^N)$, where N is the number of items in the dataset, since it potentially needs to check all possible combinations of N items. In the process of recursive mining, the callee must store all the FP-tree structures of the callers in memory as well as build the FP-tree of the current level. Because the number of nodes in an FP-tree is exponential to the depth of the FP-tree in the worst case, the space complexity of FP-growth is at least exponential to the depth of the FP-tree.

2.2.2 Serial algorithms for sequential-pattern mining

Sequential-pattern mining problem is more complicated than frequent itemset mining problem since the items in a sequential-pattern are ordered and a single item may appear multiple times (in different events) in a single sequential-pattern.

On the other hand, similar to frequent itemset mining, the search space of sequential-pattern mining can be represented by a *subsequence-lattice*, with the empty sequence at the top and the length of the sequences increase by one each level from top down. The depth of the subsequence-lattice is decided by the maximum length of all sequences in the dataset.

For instance, if the dataset has 3 items, which are a, b, and c. The search space for mining the sequential patterns makes up the subsequence lattice shown in Figure 2.6. When moving from the top of the lattice to the bottom, the lengths of the sequences increase by one at each level. The sequences at level L are subsequences of the sequences at level $L+1$ of the lattice. The items within an event appear in a fixed order. We use alphabetical order in Figure 2.6.

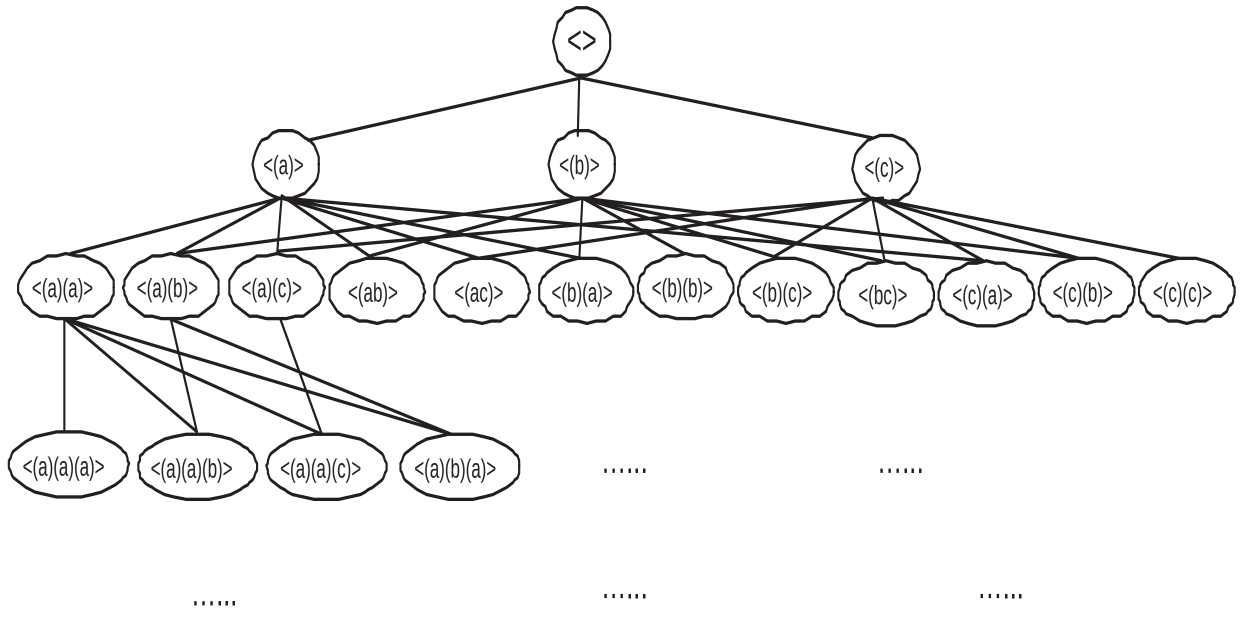


Figure 2.6: An example of subsequence-lattice

The support values of the sequences are still monotonically decreasing when moving top down within the lattice. Thus if a sequence is infrequent, all of its super sequences must be infrequent and can be pruned from the search space. Therefore, depending on the strategy to traverse the search space, the solutions for sequential-pattern mining are also classified into two categories: *breadth-first* and *depth-first*.

An algorithm called *GSP* [36] is the representative of breadth-first search algorithm to mine sequential patterns. The basic structure of the GSP algorithm is very similar to the Apriori algorithm, except that GSP deals with sequences rather than itemsets. The differences between Apriori and GSP are in the details of candidate generation and counting the numbers of occurrence.

GSP adopts a multiple-pass, candidate generation-and-test approach. The first scan finds all the single item frequent sequences. Each subsequent pass starts with the set of sequential-patterns found in the previous pass and generates new potential *candidate sequences* based on the principle that *any super-pattern of an infrequent pattern cannot be frequent.* The algorithm terminates when no new sequential pattern is found in a pass, or when no candidate sequence can be generated.

PrefixSpan algorithm [31] is a depth-first search algorithm for sequential-pattern mining, with a philosophy similar to that of the FP-growth algorithm for frequent itemset mining. PrefixSpan has proven to be one of the most efficient algorithms for sequential pattern mining.

PrefixSpan recursively projects a sequence database into a set of projected databases and grows sequential patterns in each projected database by exploring only locally frequent fragments.

An item *i*'s projected database of *DB* is a set of subsequences, which is made up of the sequences in DB containing i with all items and itemsets (events) before the 1st occurrence of i deleted. The infrequent items are also deleted during the projection. If there are other items left in the event containing the 1st occurrence of an item i, then i is replaced with " _ ", otherwise the event is deleted.

Here is an example. Suppose we want to mine sequential patterns in a sequence database DB_S, shown in Figure 2.7.(a). If an event contains only one item, the bracket () is omitted. The support threshold is set to 2. First, the algorithm scans DB_S once to find the frequent items and their supports: $\{\langle a\rangle:4, \langle b\rangle:4, \langle c\rangle:4, \langle d\rangle:3, \langle e\rangle:3, \langle f\rangle:3\}$. Then the projected databases for the six frequent items are shown in Figure 2.7(b).

Sequence_id	**Sequence**
10	*<a(abc)(ac)d(cf)>*
20	*<(ad)c(bc)(ae)>*
30	*<(ef)(ab)(df)cb>*
40	*<eg(af)cbc>*

(a)

Prefix	**Projected (postfix) database**
<a>	*<(abc)(ac)d(cf)>, <(_d)c(bc)(ae)>, <(_b)(df)cb>, <(_f)cbc>*
<b>	*<(_c)(ac)d(cf)>, <(_c)(ae)>, <(df)cb>, <c>*
<c>	*<(ac)d(cf)>, <(bc)(ae)>, <b>, <bc>*
<d>	*<(cf)>, <c(bc)(ae)>, <(_f)cb>*
<e>	*<(_f)(ab)(df)cb>, <(af)cbc>*
<f>	*<(ab)(df)cb>, <cbc>*

(b)

Figure 2.7: Example for PrefixSpan algorithm

There are two ways to grow a sequential pattern: *intra-event expansion* and *inter-event expansion.* The former refers to expanding the pattern by adding a new item to the last event of the pattern, while the latter means adding a separate event containing a single new item at the tail of the pattern. In order to distinguish the two types of expansion of an item i, we denote i for inter-event expansion and $\_i$ for intra-event expansion.

Suppose (E) is an event. We define an operator $\bowtie$:

- $(E) \bowtie i$ represents appending i as an event (i) following (E). i.e. $(E) \bowtie i = (E)i$
- $(E) \bowtie \_i$ represents adding i into event (E). i.e. $(E) \bowtie \_i = (Ei)$

Suppose S $(S = S'(E))$ is a sequence where (E) is last event in S and S' is the prefix of (E). Then, $S \bowtie i = S'(E)(i)$ and $S \bowtie \_i = S'(Ei)$

For example, $\langle (ac)b(abd) \rangle \bowtie f = \langle (ac)b(abd)f \rangle$ and $\langle (ac)b(abd) \rangle \bowtie \_f = \langle (ac)b(abdf) \rangle$. Here $\langle (ac)b(abd)f \rangle$ and $\langle (ac)b(abdf) \rangle$ are two different patterns obtained by inter-event expansion and intra-event expansion respectively.

We denote $P(i, DB_S, Q)$ as i's *projected dataset* of a dataset DB_S where Q is the prefix pattern accumulated. Since no prefix pattern has been accumulated at the beginning, Q is NULL when DB_S is the whole dataset (not a projected dataset derived from the whole dataset).

When the accumulated prefix pattern Q is not NULL, we need to conduct both the intra-event expansion and inter-event expansion of Q when doing projections. Suppose DB_P is a dataset, j is an item and Q_j is the prefix pattern accumulated before making projections for j on DB_P. Let X be the last event of Q_j. For item j, we need to make projections for j and $\_j$ to explore both intra-event expansion and inter-event expansion of Q_j.

j's projected dataset of DB_P, denoted as $P(j, DB_P, Q_j)$, is a set of subsequences, which consists of the subsequences of DB_P, where the prefix before the 1st occurrence of j in each sequence is deleted.

$\_j$'s projection $P(\_j, DB_P, Q_j)$, is a set of subsequences, which consists of:

- the subsequences of DB_P where the $1st$ occurrence of $\_j$ in each sequence is deleted.

- the subsequences of DB_P where the prefix before the $1st$ occurrence of $X \bowtie \_j$ in each sequence is deleted.

For example, in Figure 2.7, a's projected dataset of the whole dataset is:
$DB_a = P(a, DB_S, NULL) = \{\langle(abc)(ac)d(cf)\rangle, \langle(\_d)c(bc)(ae)\rangle, \langle(\_b)(df)cb\rangle, \langle(\_f)cbc\rangle\}$. Item b's projections within DB_a are: $P(b, DB_a, (a))= \{\langle(\_c)(ac)d(cf)\rangle, \langle(\_c)(ae)\rangle, \langle\rangle, \langle c\rangle\}$ and $P(\_b, DB_a, (a))= \{\langle(\_c)(ac)d(cf)\rangle, \langle(df)cb\rangle\}$.

Suppose S is a sequence and i is an item. We define an operator $\diamond$: $i \diamond S$ represents the concatenation of i to S with i as prefix[3]. Suppose (H) is the first event of S. We define $i \diamond S$ as follows:

- merge i into (H) as (iH), if (H) starts with " _ ".

- add i as a new event before (H), if (H) does not starts with " _ ".

Let $\{X_1, X_2, ..., X_n\}$ be a set of sequences, we have $i \diamond \{X_1, X_2, ..., X_n\} \equiv \{i \diamond X_1, i \diamond X_2, ..., i \diamond X_n\}$.

In PrefixSpan, the sequential patterns are mined by the concatenation of item i with the new patterns generated from mining i's projected datasets. The mining of a dataset DB_S can be formalized as function $F()$ below [4] :

```
function F($DB_S$, Q)
begin
    if ($DB_S$ is a set of empty sets) return NULL;
    else return $\bigcup_{i \in freq(DB_S)}((i \diamond F(P(i, DB_S, Q), Q \bowtie i)) \cup \{i\})$;
end
```

[3] i can be both i and $\_i$. Here we use i for illustration. Replace i with $\_i$ for the case of $\_i$.

[4] $freq(DB_S)$ represents the frequent items in DB_S. Q is the prefix pattern accumulated. If and only if DB_S is the whole dataset, Q is NULL.

In the implementation, the projected database can be represented in two forms: physical-projection and pseudo-projection [31]. The physical projection physically copies the suffixes of the sequences of the projection. The pseudo-projection uses pointers to record where the suffixes start in the sequences. We use pseudo-projection in our implementation.

The worst-case time complexity of PrefixSpan is $O((2N)^L)$, where N is the number of items in the dataset and L is the maximum length of all transactions. The constant 2 is introduced since each item can be added into a transaction through either intra-event or inter-event expansion. If using physical projection, the space complexity of PrefixSpan is $O((2N)^L)$, while if pseudo-projection is used, the space complexity can be reduced to the order of the size of the dataset.

2.2.3 Serial algorithms for closed-sequential-pattern mining

The set of closed-sequential-patterns is a subset of the set of all the sequential patterns, and the problem of mining the close-sequential-patterns is more complicated than mining all the sequential patterns.

There are two approaches for mining closed-sequential-patterns:

1. Find a candidate set of closed-sequential-pattern and for each newly discovered pattern, check the previous found ones to see whether this new one is closed w.r.t. the discovered patterns. The serial algorithm using this approach is *CloSpan* [40].

2. Greedily find the *final* set of closed-sequential-pattern. The algorithm for this approach is *BIDE* [37].

CloSpan follows a *candidate maintenance-and-test* paradigm over the set of already mined closed sequential pattern candidates. It uses this set to prune the search space and check if a newly found sequential-pattern is likely to be closed.

Since a large number of closed-sequential-patterns (or just candidates) would occupy a lot of memory storage and create a large space for searching new patterns, using CloSpan for mining long sequences or mining with very low support thresholds tends to be prohibitively expensive. Performance studies [37] have shown that BIDE is more efficient than CloSpan.

BIDE algorithm is actually an extension of the PrefixSpan algorithm. For each sequential pattern discovered, it check whether it is closed with a closure checking scheme against the dataset. It does not need to maintain the candidate pattern set as CloSpan does.

The description of BIDE algorithm is as following:

First, a scan of DB_S derives a list of the frequent 1-sequences. Then a second scan constructs the *projected databases* for the frequent 1-sequences. A projected database (or called projection) of a sequence i within DB_S, denoted as $P(i, DB_S)$, is a set of subsequences, which is made up of the sequences in DB_S containing i with the prefix before the 1st occurrence of i deleted.

The mining of DB_S with BIDE can be defined as function *F*() below. $freq(DB_S)$ represents the frequent items in DB_S. Function $Check(S)$ returns the sequences in S which can pass the BI-Directional Extension closure checking (closed patterns). $\diamond$ is the operator defined in Subsection2.2.2. The result of frequent closed sequential patterns is in set C.

```
function F($DB_S$, Q)
begin
    if ($DB_S$ is a set of empty sets) return NULL;
    else {
        $S = \bigcup_{i \in freq(DB_S)}((i \diamond F(P(i, DB_S, Q), Q \bowtie i)) \cup \{i\});$
        $C = C \cup Check(S);$
        return(S)
        }
end
```

BIDE algorithm mines only sequences with single-item events. That is, BIDE only considers inter-event expansion when expanding patterns.

Here is a simple example for illustration. Figure 2.8 is a sequence dataset for BIDE. A, B and C are single-item event[5]. With the support threshold as 2, the projected database for sequence AB is $\{C, CB, C, BCA\}$.

Sequence_id	Sequence
10	*C A A B C*
20	*A B C B*
30	*C A B C*
40	*A B B C A*

Figure 2.8: An example database for BIDE

After the projected databases are built, BIDE searches each projected database and enumerates the sequential patterns in the same way as PrefixSpan[29].

Upon getting a sequential pattern, BIDE applies a closure checking scheme, called *BI-Directional Extension*, to check whether the frequent sequential pattern is closed (See [37] for details). From the definition of the closed sequential pattern, we know that if an n-sequence, $S = e_1e_2...e_n$, is non-closed, there must exist at least one event, e', which can be used to extend sequence S to get a new sequence, S', which has the same support. The sequence S can be extended in three ways:

1. $S' = e_1e_2...e_ne'$ and $sup(S') = sup(S)$;

2. $\exists i(1 \leq i < n), S' = e_1e_2...e_ie'e_{i+1}...e_n$ and $sup(S') = sup(S)$;

3. $S' = e'e_1e_2...e_n$ and $sup(S') = sup(S)$;

The BI-Directional Extension checking scheme is to check whether any of the above three cases exists. If there is no such e' exists, then S must be a closed sequential pattern; otherwise S must be non-closed.

[5]We omit the () for simplicity, because every event only has one item for BIDE algorithm.

For example, consider the frequent subsequence AC : 4 in the database in Figure 2.8. Notice that item B appears in each of the sequence which contains AC and all appears as ABC. So AC : 4 is not closed. In contrast, study the subsequence ABC : 4, we cannot find any extension item for it, thus ABC : 4 is a frequent closed sequential pattern.

Similar to PrefixSpan, both the worst-case time and space complexities are $O(N^L)$ where N is the number of items in the dataset and L is the maximum length of all transactions. Since only inter-event expansion is allowed in BIDE, there is no constant 2 in the expression. In addition, if pseudo-projection is used as in our implementation, the space complexity can be reduced to the order of the size of the dataset.

Chapter 3

Parallelization Framework

Several efficient frequent pattern mining algorithms have been presented in the previous chapter. In this chapter, a generalized framework is introduced to provide parallel solutions for those frequent pattern mining problems. The parallelization framework targets distributed memory system.

3.1 Analysis of the serial algorithms

We build our parallelization framework on top of the efficient serial algorithms. As described in Section 2.2, if we ignore the difference in the format of the patterns to be discovered, the serial solutions for various frequent pattern mining problems belong to two categories:

One category mines the patterns level-wise, candidate generation-and-test approach. Example of this approach are the Apriori algorithm for frequent itemset mining and the GSP algorithm for sequential-pattern mining. They mine the patterns by conducting multiple database scans. Each scan identifies the frequent patterns with a certain length from the set of candidate patterns and then generates the candidates for the next pass from the frequent patterns discovered in the current pass.

The other category, such as the FP-growth, PrefixSpan and BIDE algorithms, solves the problem in a divide-and-conquer manner. In frequent itemset mining, the mining of the FP-tree for the whole database is divided into the mining of a series projected FP-trees corresponding to the frequent items in the database. Similarly, in sequential-pattern mining and closed-sequential pattern mining, the whole database mining is divided into the mining of projections of the frequent subsequences.

We considered these two categories of algorithms in terms of parallelizability and performance and decided to choose the algorithms from the second category for parallelization. There are two reasons for this decision:

1. According to our experiments, the performance of the first category is almost always worse than the second. The performance difference can be more than an order of magnitude in some cases. Many other studies [11, 30, 31] have also shown that usually the algorithms in the first category are not as efficient as those in the second category. The algorithms in the first category suffer from the cost due to the huge number of candidates patterns and the numerous scans of the dataset, especially when the minimum support threshold is low or the length of the patterns is long.

2. From the point of view for parallelization, the algorithms in the second category is more convenient to be parallelized than those in the first category. The candidate generation-and-test approach requires a global view of the frequent patterns discovered at each pass to generate the candidates for the next pass. This may introduce barriers and communication among the processors for parallelized algorithms. On the other hand, as we will see, the divide-and-conquer property of the algorithms in the second category can make the task partition more convenient without introducing much communication.

Hence, we use FP-growth, PrefixSpan and BIDE algorithm as the base sequential algorithms for the parallelization of mining frequent itemset, sequential-pattern and closed-sequential-pattern respectively.

3.2 Generalization of the serial algorithms

Although FP-growth, PrefixSpan and BIDE are aiming at different problems, they all solve the problems in a divide-and-conquer manner. Basically, the process of these algorithms can be generalized into three steps:

- Step 1: Identify the frequent items.
- Step 2: Project the whole database into sub-databases in related to each of the frequent items.
- Step 3: Mine the projected databases respectively.

In the third step, the projected databases are independent from each other. In FP-growth, because the frequent items are ordered in descending order of their frequency, the projected database of an item, i, only contains the items that are more frequent than i. For example, in Figure 2.4, the conditional database of p may only contains f, c, a, b or m, while the conditional database of m may contains f, c, a or b, but not p. In Prefix-Span and BIDE, because the items in the sequence are in the order of their appearance in the sequence, only the suffix of the first occurrence of an item i will be considered in i's projected database.

Consequently, the mining of the projected databases are independent jobs so that they can be distributed to different processors to be processed asynchronously.

3.3 Framework for parallelization

Corresponding to the three steps of the base sequential algorithms, we propose the following framework for parallel mining frequent patterns. Our target platform is a distributed memory system so the dataset is distributed across the processors and each process has $1/N$ of the dataset locally where N is the number of nodes in the system.

1. Count the occurrence of items (or 1-sequences) in parallel and perform a reduction to get the global counts. Select items, whose number of occurrences is no less than the support threshold.

2. Partition the frequent items (or 1-sequences) identified in step 1 and assign each subset to a different processor. Each processor builds the projection of the databases for the assigned items (or 1-sequences).

3. Each processor mines its projected database asynchronously in a pattern-growth manner without inter-processor communication.

In the first step, each processor counts the occurrence of items (or 1-sequences) in its local portion of the dataset in parallel and performs a reduction to get the global counts. Then the items (or 1-sequences) with numbers of occurrence above the threshold will be identified by each processor.

In the second step, we partition the frequent items (or 1-sequences) and assign them to the processors statically. Each processor then builds the projection for the assigned items. Since the projection is based on the whole dataset instead of just the local portion of the dataset, in our implementation, each processor broadcasts its local datasets to all the other processors. We found that it is more efficient to carry out the broadcast using a virtual ring structure where processor I only receives the package from Processor $((I-1)\ mod\ N)$ and only sends the package to Processor $((I+1)\ mod\ N)$. Thus, assume there are total N processors, the all-to-all broadcast is carried out in $(N-1)$ send-receive steps. The communication with our implementation collectively consume only no more than 1% of the mining time. An alternative implementation might be that each processor scans the local portion of the dataset and selectively sends the records required by each remote processor respectively.

Because the projected databases are independent, there is no inter-processor communication involved in step 3. Each processor just mines its own projected database using the original sequential algorithms (FP-growth, PrefixSpan or BIDE).

3.4 Challenges to the parallel framework

The framework presented in the previous section is a straightforward parallelization of the base sequential algorithms. The parallelization makes use of the divide-and-conquer property and partitions the task into independent subtasks so that the inter-processor communication is minimized.

However, minimizing the communication is not enough to obtain good parallel performance, another issue has to be addressed, which is *load balancing.*

We implemented a parallel frequent itemset mining algorithm and a parallel sequential-pattern mining algorithm with the framework proposed in Section 3.3. In the implementations, we use round-robin strategy to schedule the frequent items (or 1-sequences) ordered by frequency to the processors. We tested the performance of these two algorithms on a distributed memory system with up to 64 processors. The performance obtained was not satisfactory. For almost all the datasets we tested, the speedups go flat when the number of processors is above 16.

As an illustration, Figure 3.1 shows the speedup we obtained for one dataset *pumsb* to mine frequent itemset. The description of the datasets are in Figure 7.1(a). The shapes of the speedup curves are similar for the other datasets we tested.

Our study shows that it is the *imbalanced load* that mainly limits the scalability of the parallelization. The inherent task partition of the divide-and-conquer method may result in extremely imbalanced workload among the processors. In fact, the cost of processing the projected databases may vary greatly. From our experiments, the mining time of the largest task may be tens, or even hundreds, of times longer than the average mining time of all the other tasks.

Figure 3.2 shows the mining time distribution of the projected databases

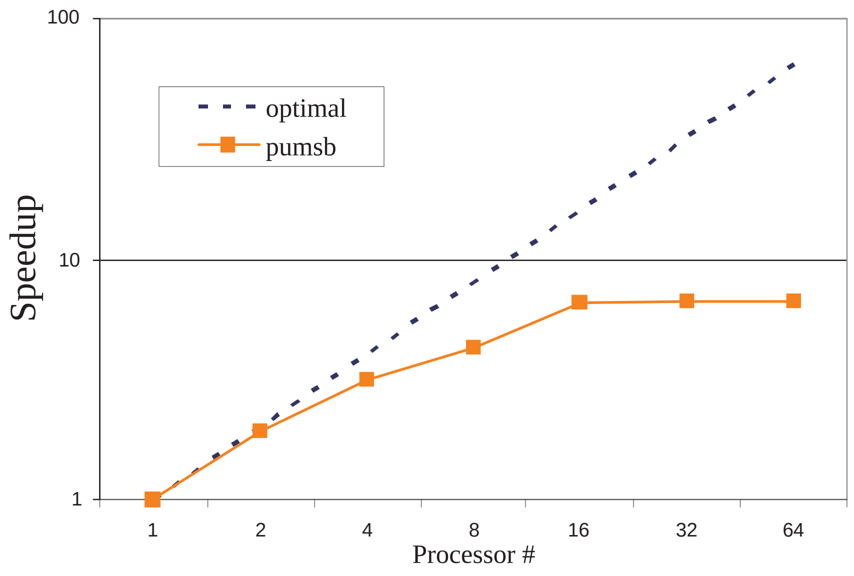
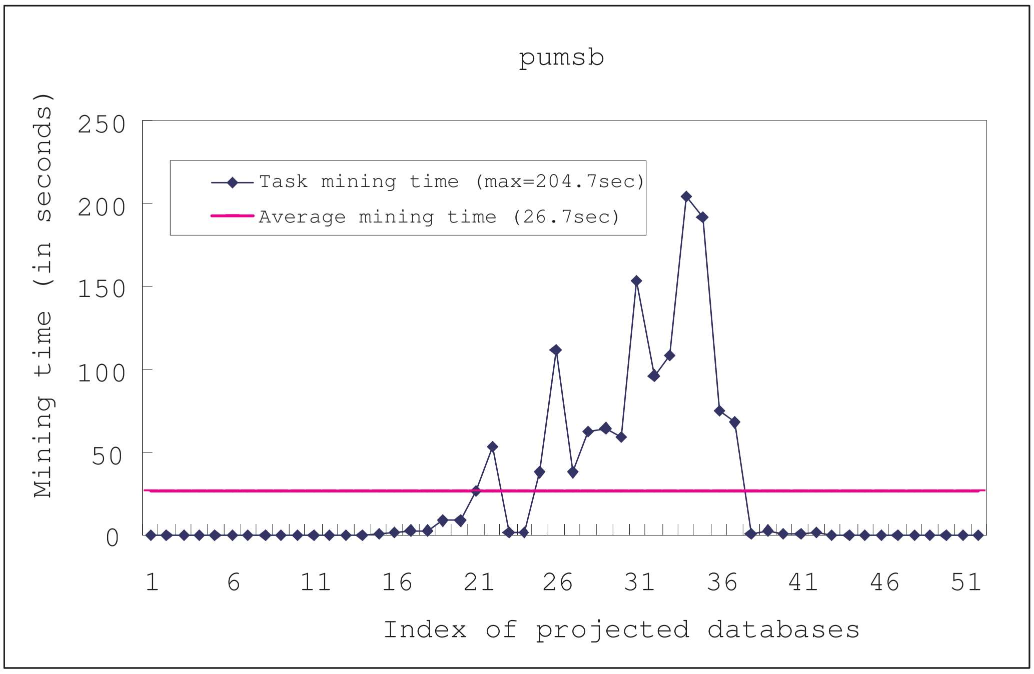


Figure 3.1: Preliminary speedup for dataset *pumsb*

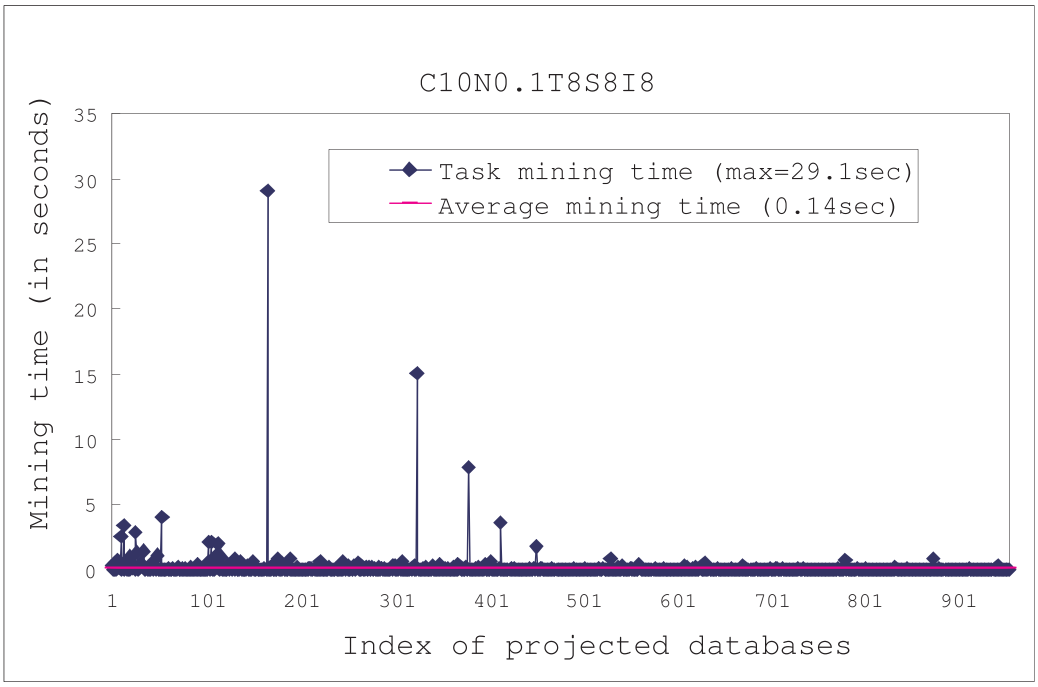
for two databases. Figure 3.2(a) is the transactional dataset *pumsb* (dataset characteristics in Figure 7.1(a)) for frequent itemset mining. Figure 3.2(b) is for the sequence dataset *C10N0.1T8S8I8* (dataset characteristics in Figure 7.1(b)) for sequential pattern mining. As it is shown in the graphs, the largest subtasks take around 14-20% of the overall mining time. Figure 3.3 shows the average and maximum mining time of the projected databases for all the datasets we tested[1].

Since the granularity of the subtasks is not fine enough, we need to break the large subtasks into smaller ones. Simply partitioning all the subtasks into smaller ones will produce too many subtasks and introduce much overhead for the mining process. According to our experiments, for some datasets, the overhead introduced by such unnecessary task partitioning consumed all performance benefit from task partitioning and even resulted in 10-20% slow down. Therefore, we have to identify those projections which need to be further partitioned into smaller ones.

[1]The detailed characteristics of the datasets is in Figure 7.1 and the setting-ups of our experiments will be discussed in Chapter 7



(a)



(b)

Figure 3.2: Example of load imbalance

Dataset	Mushroom	Connect	Pumsb	Pumsb_star
Average (sec)	0.17	2.37	26.7	21.9
Maximum (sec)	0.78	23.5	204.6	899.6

Dataset	T30I0.2D1K	T40I10D100K	T50I5D500K
Average (sec)	10.86	0.48	0.56
Maximum (sec)	281.1	35.2	58.6

(a) Datasets for frequent itemset mining

Dataset	C10N0.1T8S8I8	C50N10T8S20I2.5	C100N5T2.5S10I1.25
Average (sec)	0.14	0.028	0.037
Maximum (sec)	29.1	7.47	7.84

Dataset	C200T2.5S10I1.25	C100N20T2.5S10I1.25
Average (sec)	0.038	0.047
Maximum (sec)	21.9	7.99

(b) Datasets for sequential -pattern mining

Figure 3.3: Mining time distributions

An alternative solution to static task partitioning is to use dynamic work stealing strategy. When a processor finishes its assignment, it will ask for sharing some work from the busy processors. Theoretically work stealing can dynamically balance the workload among processors. However, for parallel frequent-pattern mining, the additional communication costs and the sequential computation introduced by work stealing may consume the benefit from the balanced workload, since in order to allow an idle processor to join the busy ones, a busy processor must build a projection for the requesting processor and send the projection to it. In addition, it is usually more difficult to implement a work stealing strategy than a static method. Based on these considerations, we focused on using static task partitioning strategy to address the load balancing problem.

Chapter 4

Addressing The Load Balancing Problem

In order to achieve scalability for the parallelization framework we proposed, load balancing is an important issue that must be considered. As discussed in Section 3.4, it is of crucial importance to estimate relative mining times of projections. Such estimation needs to be conducted before the actual mining.

4.1 Static estimation

We first attempted to build a model to estimate the mining time of the projections from the static properties of the dataset. We studied the characteristics of the dataset, such as the number of items, the number of the transactions and the width of the transactions, to build the estimation model. It was not possible to find an accurate estimation formula that worked for all the datasets in our benchmarks.

We also examined the properties of the projections, such as the depth and the density of the FP-tree, in order to find correlations between these characteristics and the mining time. However, no correlation was found.

Figure 4.1 and Figure 4.2 shows the relation between projected FP-tree depth and the mining time of the projections with the dataset *connect* and *T40I10D100K* respectively (dataset characteristics in Figure 7.1(a)). The mining time curves use the left vertical scale while the FP-tree depth lines use the right vertical scale. As the graphs depict, the two curves of tree depth and the mining time do not match at all. It is extremely difficult, if not

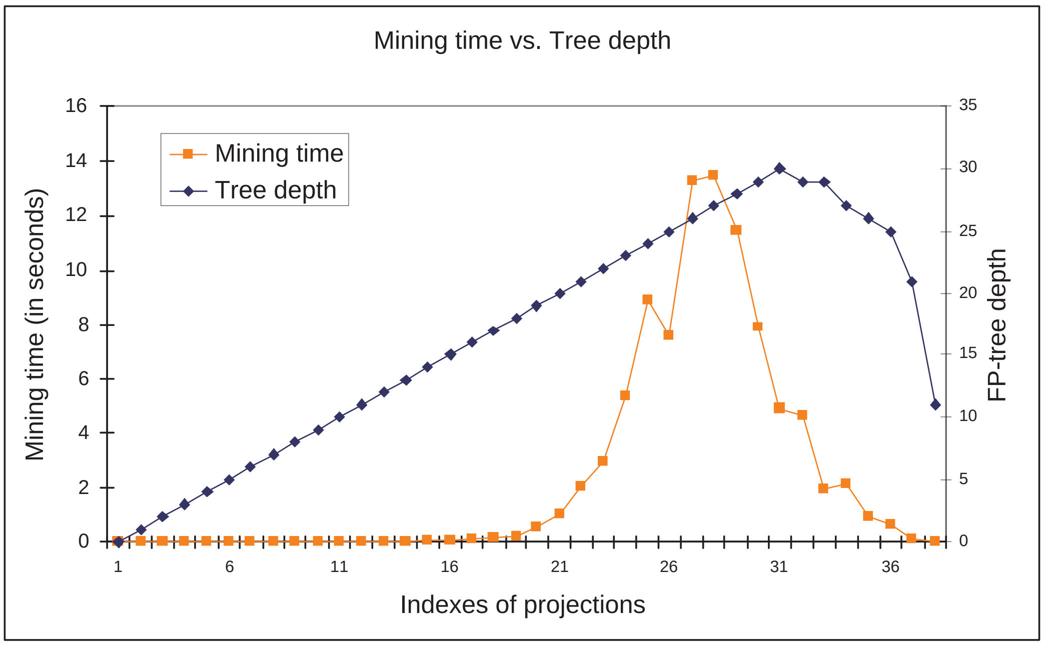


Figure 4.1: FP-tree depth and mining time for dataset *connect*

impossible, to predict correctly which projections are *large*[1] from the FP-tree depth.

4.2 Random Sampling

An alternative to using static formulas is to use run-time sampling to estimate the mining time dynamically. By mining a small sample of the original dataset and timing the mining time of the projected databases of the sample we may be able to give an estimation of which projected dataset takes longer to be mined. The support threshold keeps the same % value when mining the sample.

Accuracy and overhead are two important characteristics which we use to evaluate the sampling techniques.

Random sampling is a fairly straightforward heuristic. Random sampling is performed by randomly collecting a fraction of transactions or sequences in the dataset as a sample

[1]We call those items, whose projected databases need a long time to be mined, *large* items. Those, with short-mining-time projected databases, are called *small* items.

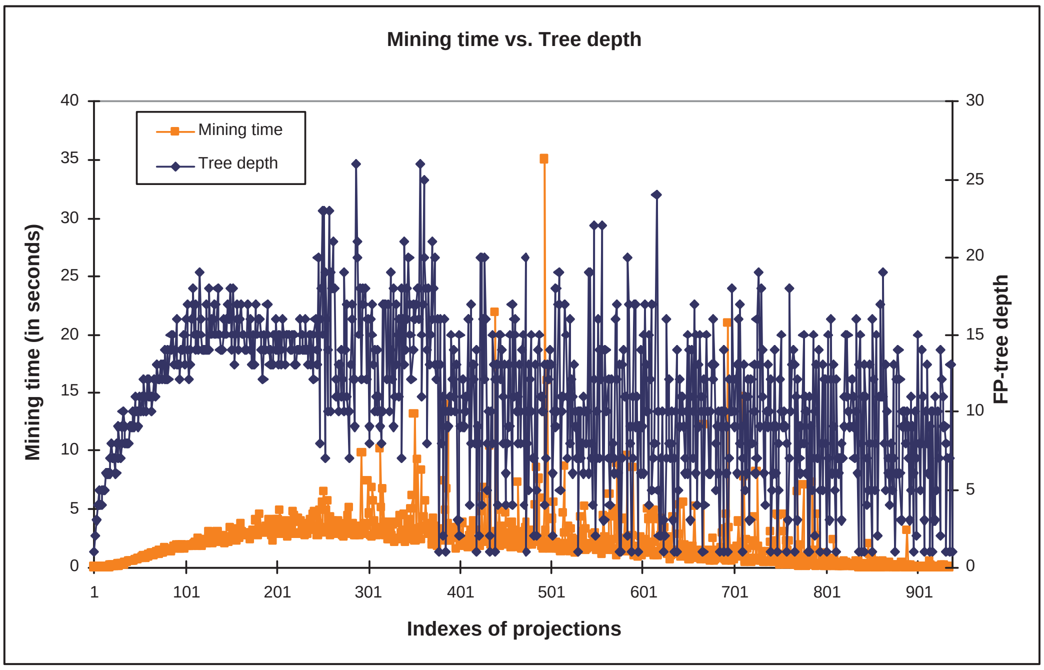


Figure 4.2: FP-tree depth and mining time for dataset *T40I10D100K*

and mining it with the support threshold using the same percentage value[2].

We did experiments with random sampling for frequent itemset mining, sequential-pattern mining and closed-sequential-pattern mining. We compared the mining time of each item's projected database in the sample with the corresponding mining time in the original dataset. Unfortunately, we found that the mining time curves of random sampling do not match well with the curves of the whole datasets. Consequently, random sampling is not able to provide us a good estimation of the mining time.

Figure 4.3 shows one of our experimental results on random sampling with the dataset *pumsb* (dataset characteristics in Figure 7.1(a)) for frequent itemset mining. The curves show the times for mining the projected database of each frequent item both for the whole dataset and for a 1% random sample of the dataset. The curve for the whole dataset uses the left vertical scale while the one for the sample uses the right vertical scale. Clearly,

[2]Note that although the relative value of support threshold keeps the same percentage, the absolute value of support threshold is reduced by the same fraction as the sample. If the absolute value of the support threshold is less than 1 when mining the random sample, we use 1 instead.

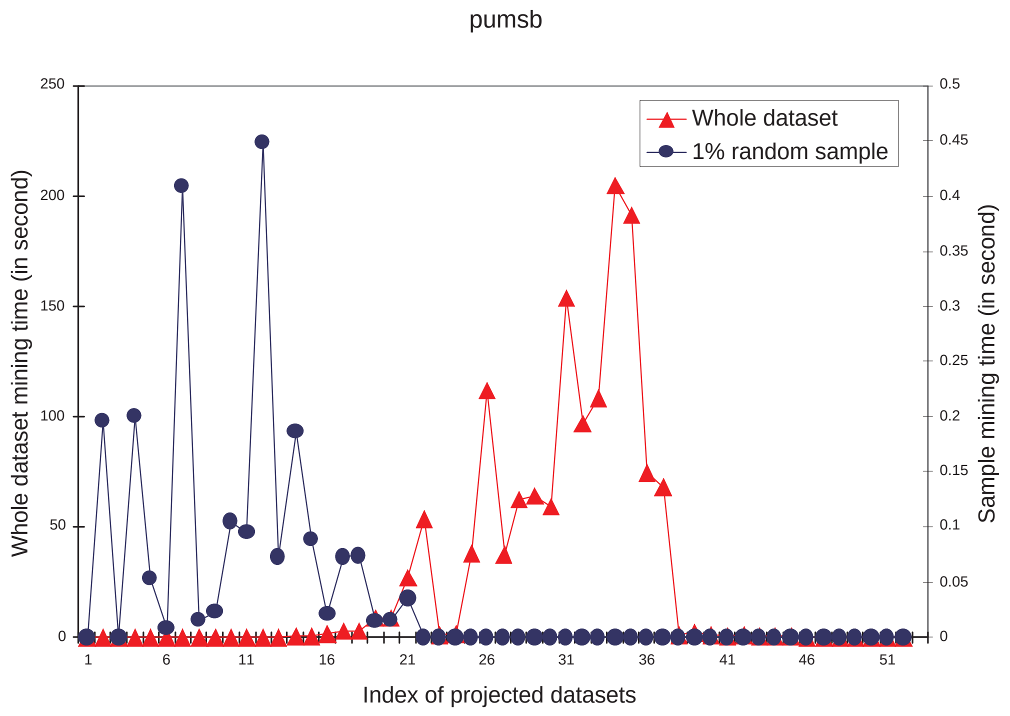


Figure 4.3: Random sampling for frequent itemset mining

there is no correlation between the mining time of a random sample and the time for the whole dataset. Therefore, through random sampling we are not able to determine which projections are the most time-consuming. In our experiments, only when we increased the sample size to about 30% did we obtain a relatively accurate estimation. But then the overhead of sampling exceeds more than 50% of the original mining time.

Things are similar with sequential pattern mining and closed-sequential pattern mining. Figure 4.4 is the experimental results of random sampling with the dataset *C10N0.1T8S8I8* (dataset characteristics in Figure 7.1(b))for sequential pattern mining. And Figure 4.5 is the results of random sampling for closed-sequential-pattern mining with dataset *C200S25N9* (dataset characteristics in Figure 7.1(c)). Clearly, there is no correlation between curves of the random sampling and the curves of the whole datasets.

The main reason why random sampling cannot provide an accurate estimation of the mining time is that the shape of the FP-tree of a random sample may be different from the

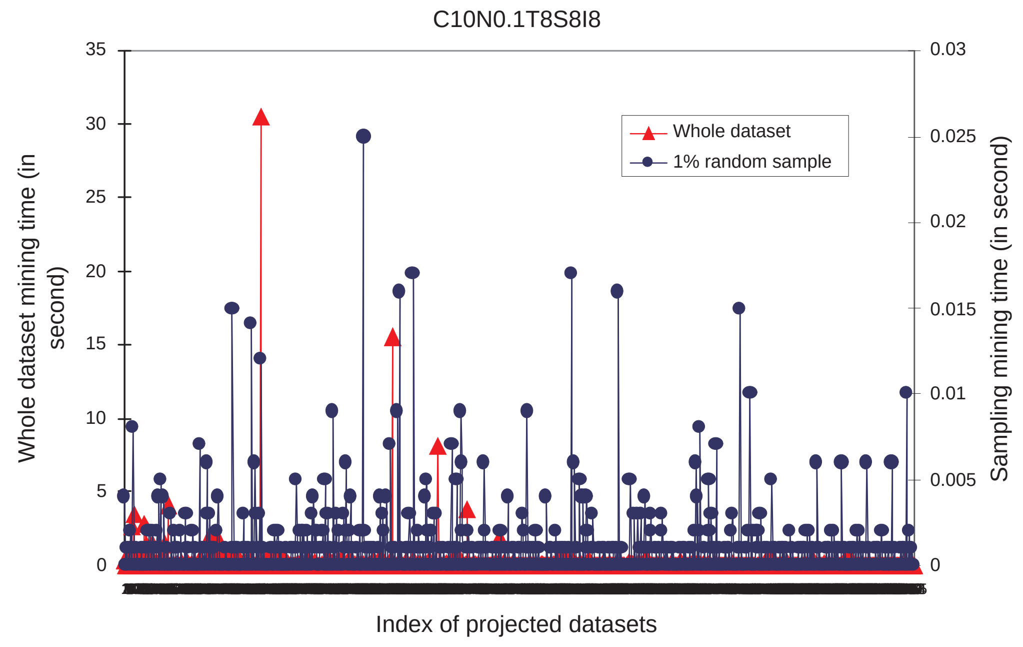


Figure 4.4: Random sampling for sequential pattern mining

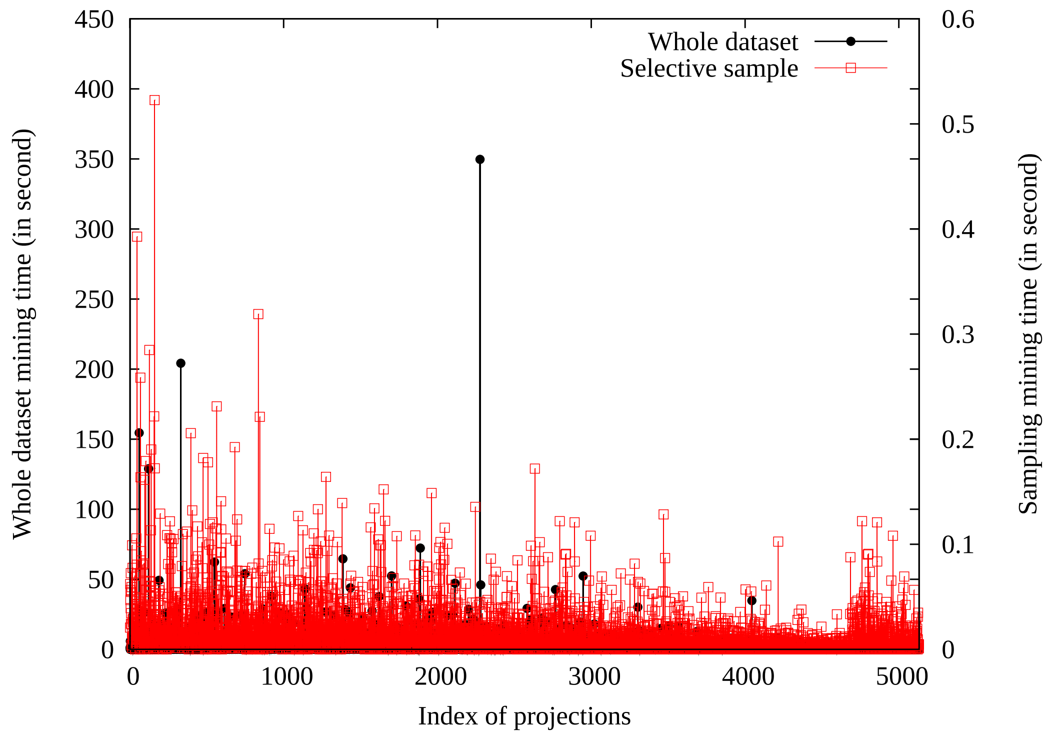


Figure 4.5: Random sampling for closed-sequential pattern mining

shape of the FP-tree of the whole dataset. The frequency order of the items may be changed when sampling since the dataset can be skewed and the items may not be evenly distributed in the dataset. Consequently, the relative projection mining times of the items may be quite different from that of the whole dataset.

4.3 Selective Sampling

Since a random sample is not able to give an accurate estimation of the projection mining time distribution of the whole dataset, we need to change the sampling strategy so that the sample can maintain the feature of the whole dataset while the mining time of the sample should be significantly reduced.

4.3.1 How selective sampling works

We devised a sampling technique, called *selective sampling*, which has proved to be quite accurate in identifying time-consuming projections.

Instead of randomly selecting a fraction of transactions or sequences from the dataset, selective sampling maintains the information of every transaction or sequences in the dataset. However, selective sampling only keeps the *"important"* items or subsequences of each transaction in the sample and remove the rest. Whether an item or a subsequence is "'important"' depends on its frequency and the location in the projections. The support threshold keeps the same % value when mining the selective sample.

Next we discuss in detail how selective sampling works in detail.

For frequent itemset mining, selective sampling discards a fraction t of the most frequent items from each transaction as well as those items whose frequencies are less than the support threshold.

For example, let$\langle(f:4),(c:4),(a:3),(b:3),(m:3),(p:3),(s:1)\rangle$ be the items in a dataset sorted in descending order of frequency. Let t be 33% and the support threshold be

3. Then f and c are the top 33% most frequent items. When applying selective sampling, for each transaction in the dataset, we discard c, f and the infrequent item s and preserve a, b, m and p if they appear, for each transaction.

For sequential pattern mining and closed-sequential pattern mining, selective sampling discards l items from the tail of each sequence, as well as the infrequent items of each sequence in the dataset. The number l is computed by multiplying a given fraction t by the average length of the sequences in the dataset.

For example, let $(\langle a\rangle : 4), (\langle b\rangle : 4), (\langle c\rangle : 4), (\langle d\rangle : 3), (\langle e\rangle : 3), (\langle f\rangle : 3), (\langle g\rangle : 1)$ be the items in the database. Let the support threshold be 4 and the average length of the sequences in the dataset be 4. Support t equals to 75% so that l is 3 $(4 * .75)$. Then a, b and c are frequent items because their support values are no less than the threshold. Given a sequence as $\langle a(abc)(ac)d(cf)db\rangle$, the selective sample of this sequence is $\langle a(abc)a\rangle$. The suffix $\langle(\_c)d(cf)db\rangle$ is discarded because it contains the last l frequent items of the sequence (d and f do not count because they are infrequent items.).

Selective sampling for sequential-pattern mining works in a similar way as for frequent-itemset mining. The items at the tail of each sequence are similar to the items closed to the root (most frequent items) of FP-tree in frequent itemset mining, because the projections in the sequential pattern mining and closed-sequential pattern mining are suffixes of the sequences in the datasets while in the frequent-itemset mining, the projections are prefix paths to the root.

4.3.2 Accuracy and overhead of selective sampling

We examine the effectiveness of selective sampling by comparing the mining time of the projected databases for the selective sample with the mining time for the whole dataset. The experimental results show that selective sampling can effectively identify the large items in the dataset.

For instance, in the case of frequent itemset mining, Figure 4.6 shows our experimental

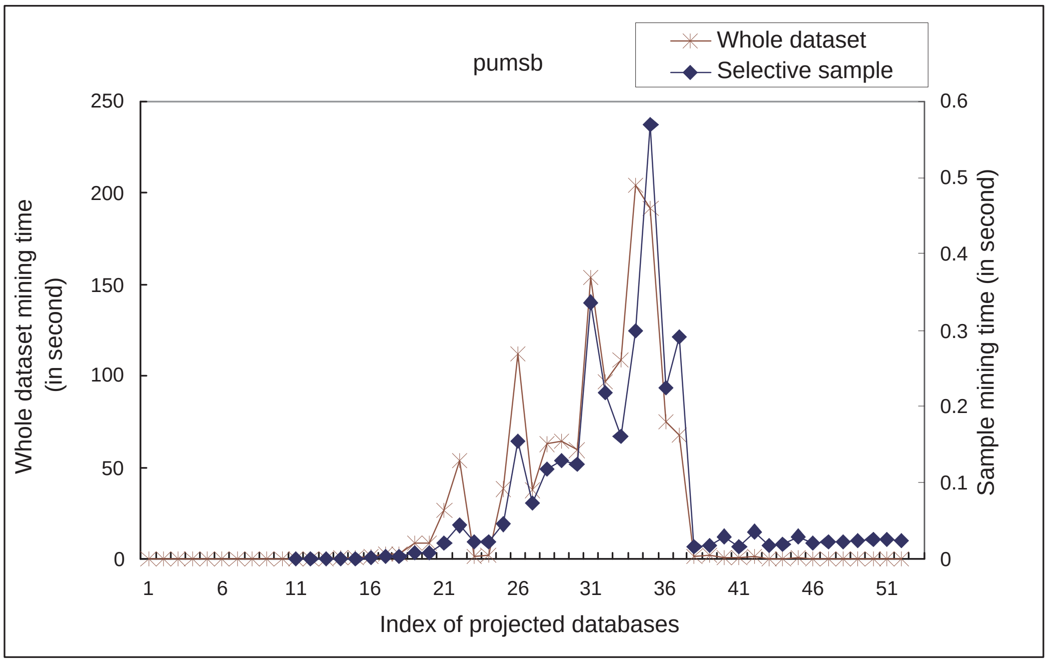


Figure 4.6: Accuracy of selective sampling for frequent itemset mining

results of selective sampling for the same dataset as Figure 4.3 with t set to 20%. The two curves match quite well. The mining time of item 1 to 10 are missing in the selective sample curve because these items are the top 20% most frequent items and are discarded during selective sampling. However, since the actual mining time of these top 20% items is ignorable, this does not affect balancing the load of each processor and it also proves the feasibility of our selective sampling strategy.

Figure 4.7 gives the results of selective sampling for sequential pattern mining with the dataset *C10N0.1T8S8I8*. The average sequence length of *C10N0.1T8S8I8* is 64 and we set t as 75% so that l is 48. We can see that the two curves nicely match to each other. That is to say, those large items in the mining of selective sample are also the large items in the mining of the whole dataset.

Similar results can be obtained for closed-sequential-pattern mining with selective sampling. Figure 4.8 is the results of selective sampling for closed-sequential pattern mining with the dataset *C200S25N9*.

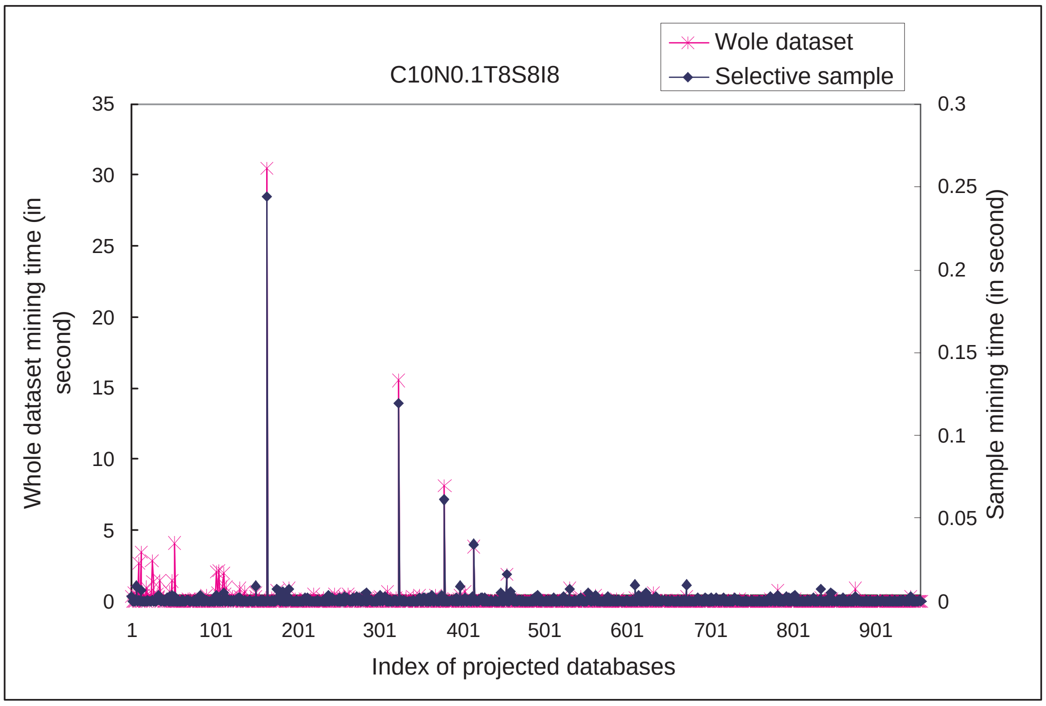


Figure 4.7: Accuracy of selective sampling for sequential pattern mining

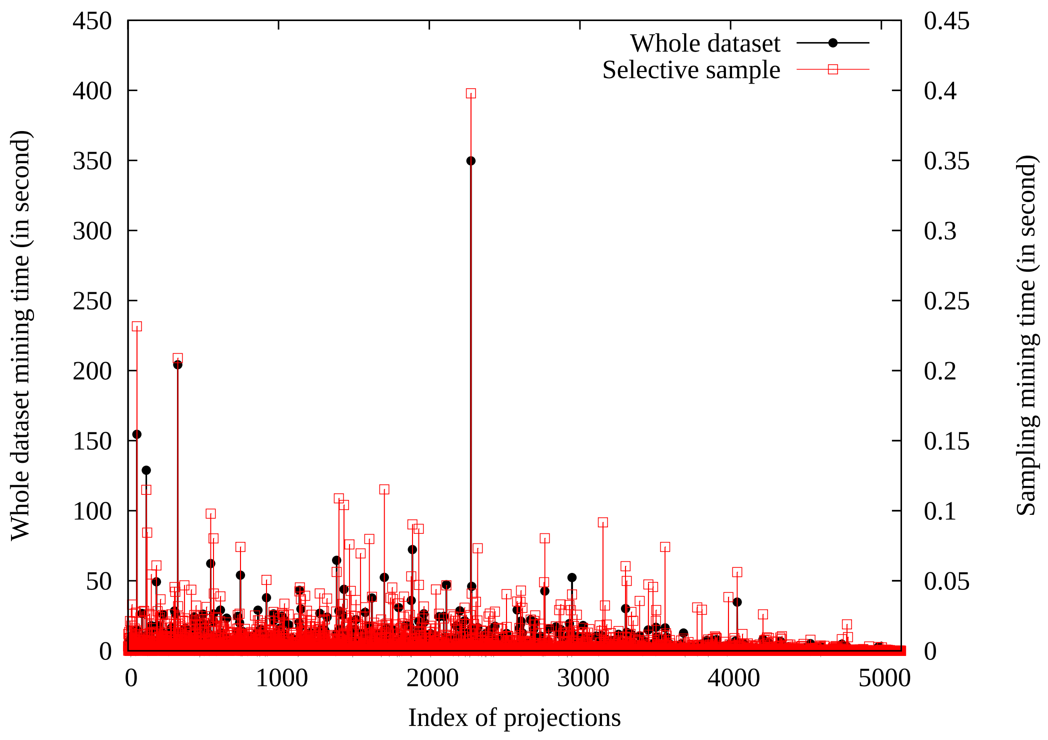


Figure 4.8: Accuracy of selective sampling for closed-sequential pattern mining

Figure 4.9 shows the matching graphs for all the benchmark datasets (dataset characteristics in Figure 7.1(a)) we used in frequent itemset mining. All of them are similar to Figure 4.6.

As you may expect, there is a trade-off between the accuracy and the overhead of selective sampling. The more information we discard from the original dataset, the less accuracy the sampling will obtain and the less overhead will be introduced by sampling.

Selective sampling discards the top t% most frequent items of the datasets in frequent itemset mining. We quantify the accuracy and overhead of selective sampling with different values of t. We computed the overhead of selective sampling as the mining time of selective sample versus the mining time of the whole dataset. We apply the following method to quantify the accuracy of selective sampling:

Suppose the total mining time of the whole dataset is T_{all}, the mining time of selective sample is T_{sample} and the number of processors is N. If we could evenly partition the work across N processors, then the mining time of each processor is $\frac{T_{all}}{N}$. So if the mining time of a projection is larger than $\frac{T_{all}}{N}$, it must be partitioned to achieve the optimal speedup. Suppose P is the set of projections whose mining time is larger than $\frac{T_{all}}{N}$. Each projection i in P has a weight W_i and $W_i = \frac{T_i}{T_{all}/N}$ where T_i is the actual mining time of projection i.

On the other hand, all the projections whose sampling mining time are larger than $\frac{T_{sample}}{2N}$ will be partitioned when mining the whole dataset. We use $\frac{T_{sample}}{2N}$ instead of $\frac{T_{sample}}{N}$ as the boundary to improve the accuracy of selective sampling. Suppose Q is the set of projections whose projection mining times are larger than $\frac{T_{sample}}{2N}$.

Then, the accuracy of selective sampling is defined as: $\frac{\sum W_j (j \in P \cap Q)}{\sum W_i (i \in P)}$

Figure 4.10 shows the accuracy and overhead of selective sampling for all the benchmark datasets we used with the values of t varying from 5% to 50%. As we can see from the graphs, for most of our benchmarks, we can obtain the balance between accuracy and overhead when when t's value is between 15% and 25%. We use 20% as the empirical value of t for frequent itemset mining. For sequential pattern mining and closed-sequential-pattern mining, the

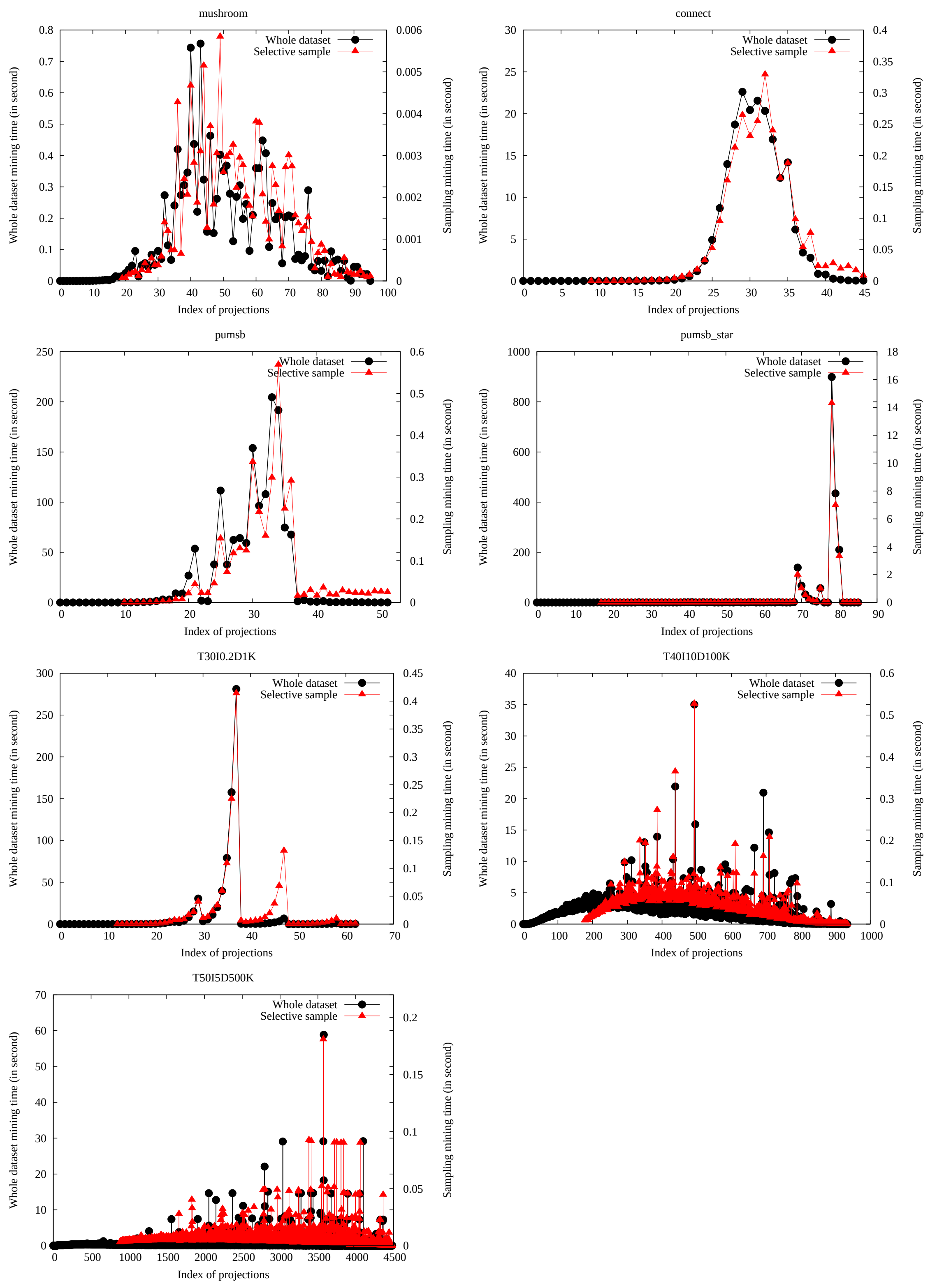


Figure 4.9: Accuracy of selective sampling

empirical value of t is 75%.

There is also a trade-off between the overhead introduced by selective sampling and the parallel performance improvement due to more balanced work load. We use the following formula to give an estimation of the parallel mining time T_{par}:

$T_{par} = \frac{\sum T_j(j \in P \cap Q)}{N} + max\{T_i\}(i \in P - Q) + \frac{\sum T_k(k \notin P)}{N}$

where N is the number of processors, P, Q and T use the same definitions as we discussed for accuracy. The projections in P are large projections which need to be partitioned and the projection in Q are the projections being actually partitioned according to sampling. Here, we assume that the projections in $P \cap Q$ are evenly distributed to the N processors, the projections not in P are also evenly distributed and there is no overhead introduced due to the projection partitioning. We use the value of T varying from 5% to 50% to tune the size of the sample and compute the value of T_{par} using $N = 64$. Figure 4.11 gives the comparison $\frac{T_{par}}{T_{all}}$ with $\frac{T_{sample}}{T_{all}}$ for all the datasets in in Figure 7.1(a) where T_{all} is the serial mining time of all the projections. As we can see from the graphs, 20% is a reasonable value for t to gain parallel performance and restrict the sampling overhead within a small range.

With t equals to 20%, the overhead of selective sampling in Figure 4.6 is only 0.71% of the sequential mining time of the whole dataset and the overhead of Figure 4.7, with t set to 75%, is only 0.52% of the sequential mining time while it still provides accurate information for the relative mining time estimation.

Figure 4.12 lists the percentage of mining time of selective sample versus the sequential mining time of all the datasets in Figure 7.1, with t being 20% for frequent itemset mining, 75% for sequential pattern mining and closed-sequential-pattern mining.

4.3.3 Why selective sampling works

To understand why the selective sampling works, we first study the case for frequent itemset mining.

Intuitively, selective sampling removes the most frequent items and the infrequent items

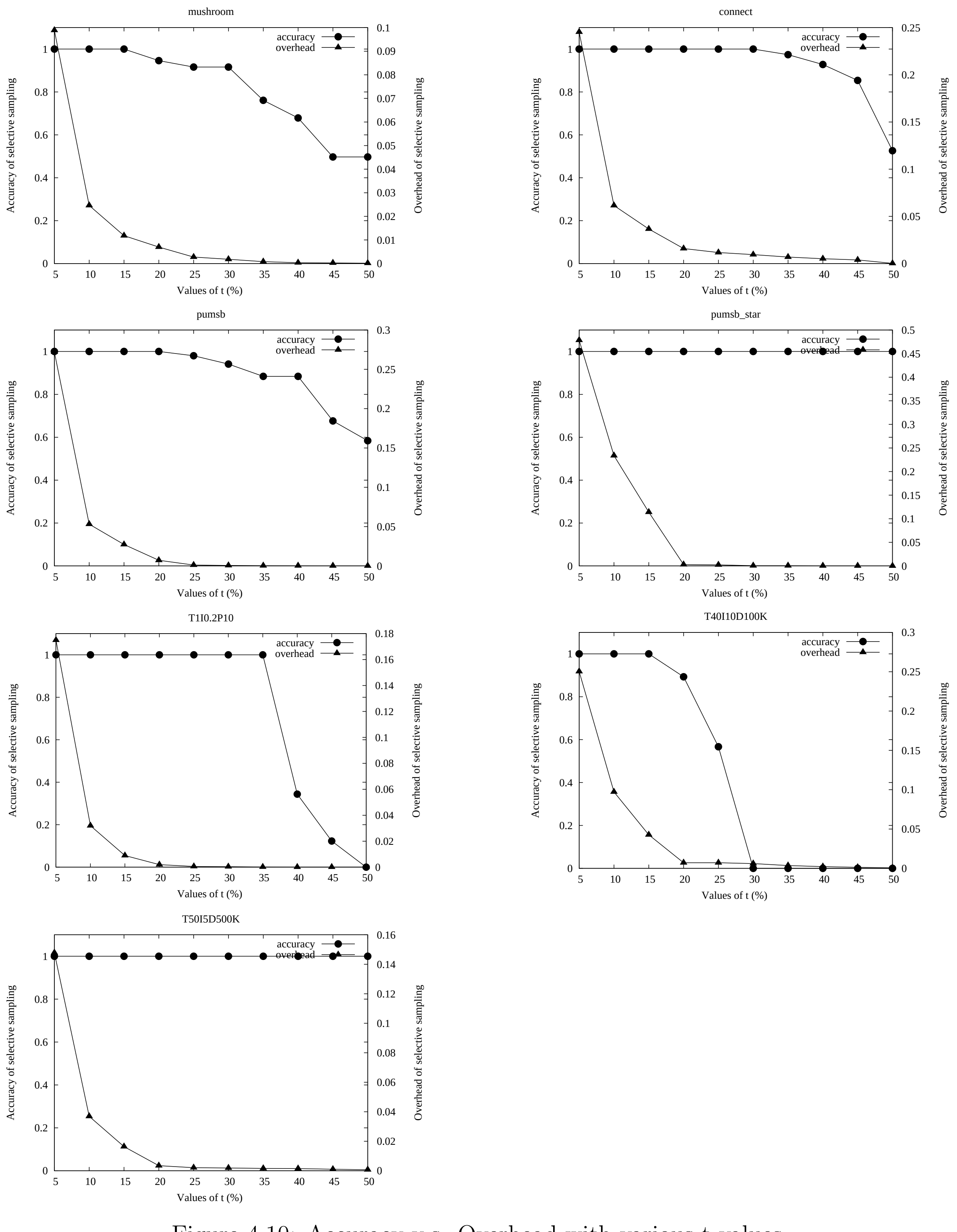


Figure 4.10: Accuracy v.s. Overhead with various t values

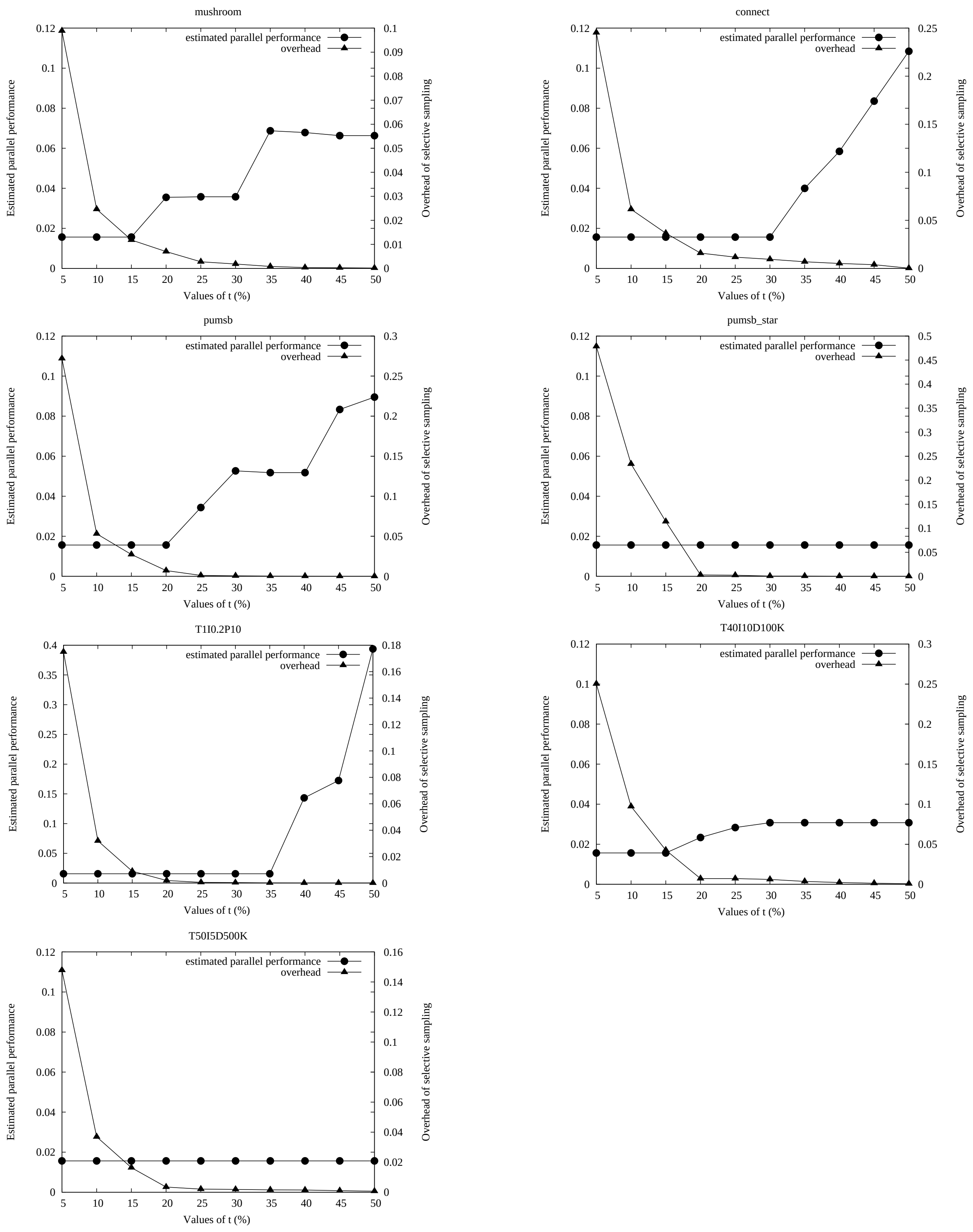


Figure 4.11: Parallel mining time v.s. Overhead with various t values

Dataset	Mushroom	Connect	Pumsb	Pumsb_star	T30I0.2D1K	T40I10D100K	T50I5D500K
Overhead	0.71%	1.80%	0.71%	0.29%	0.19%	0.72%	0.28%

(a) Datasets for frequent itemset mining

Dataset	C10N0.1T8S8I8	C50N10T8S20I2.5	C100N5T2.5S10I1.25	C200T2.5S10I1.25	C100N20T2.5S10I1.25
Overhead	0.51%	0.71%	0.60%	0.87%	0.53%

(b) Datasets for sequential-pattern mining

Dataset	C100S100N5	C100S50N10	C200S25N9	Gazelle
Overhead	1.32%	3.97%	0.53%	2.12%

(c) Datasets for closed -sequential-pattern mining

Figure 4.12: Overhead of selective sampling

in each transaction. The infrequent items are irrelevant to our *frequent* itemset mining so that we can safely remove them. Those most frequent items appear very frequently in the datasets so that they more likely appear in most of the transactions in the dataset. Therefore, removing them can greatly reduce the the mining time of the other items and the reduction may be done by a similar factor statistically.

In FP-growth algorithm, mining an item i's projection only mines frequent itemsets which contains i and those items which are more frequent than i. Considering the search space of frequent-itemset mining as Figure 4.13 shown, item A,B,C,D and E are in increasing order of their frequencies. Removing the most frequent item E will reduce the search space of the other items to half and therefore both the total mining time and each projection mining time are half, while the relative mining times of A, B, C and D keep the same. So if a few top frequent items are discarded, the projection mining times of the remained items will be reduced exponentially while their mining time will keep a similar distribution as they are with the whole dataset. Therefore, our selective sampling strategy can give an accurate estimation of the relative projection mining time while still keep a small overhead.

The reason for sequential pattern mining and closed-sequential-pattern mining is similar to that of frequent itemset mining. Referring to Figure 4.13, removing a certain item at the

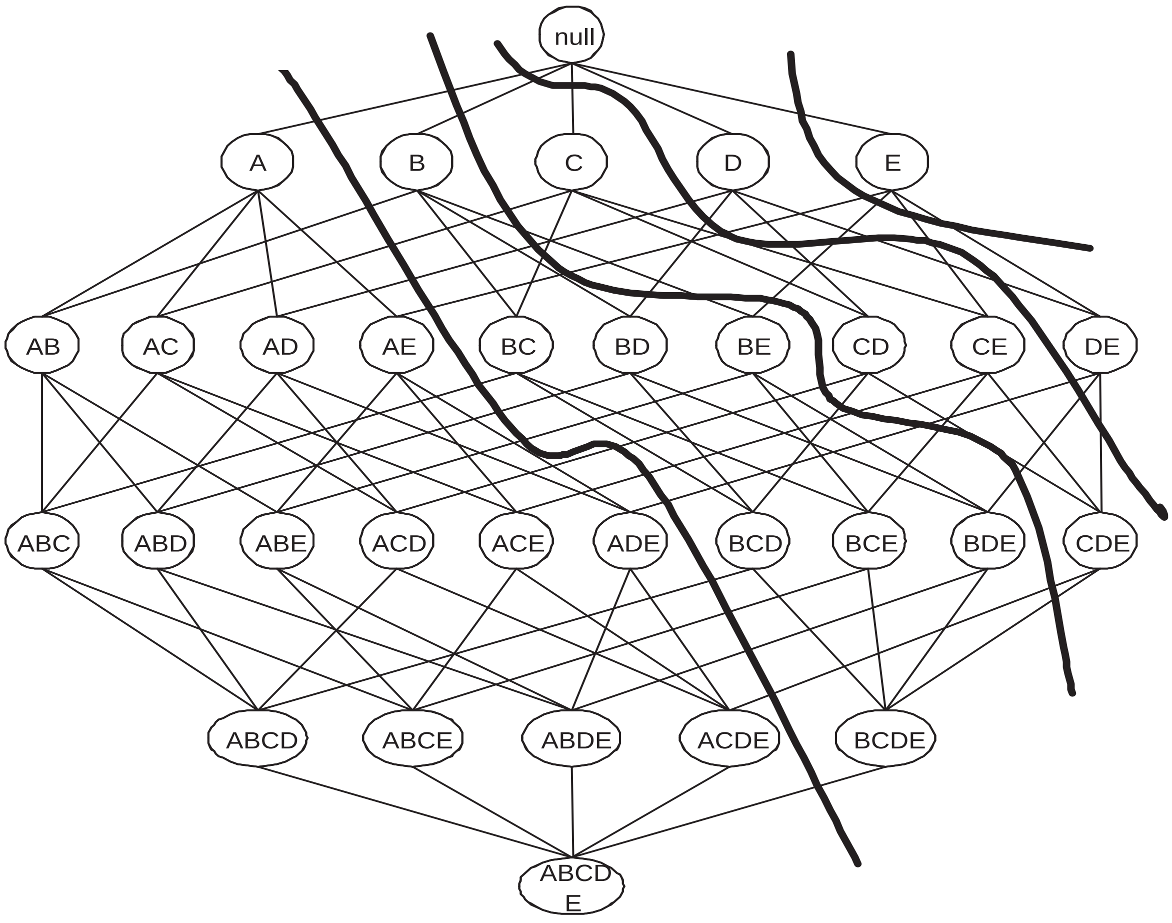


Figure 4.13: Search space of different items for frequent-itemset mining

tails of the sequences will reduce the search space of the remaining items to half, while still keep the relative mining times.

Chapter 5

Other Issues in Parallelization

In this chapter, we discuss some other issues that need to be addressed in applying selective sampling to the parallelization of frequent pattern mining.

5.1 Task partitioning

5.1.1 How to partition the large tasks

After the projections with long mining times are identified by selective sampling, they must be partitioned into smaller ones. The task partition is done following a *pattern-growth* way.

In frequent itemset mining, suppose $I = \langle i_1, i_2, ..., i_m \rangle$ are the ordered set of frequent items of the whole dataset listed in descending order of their frequency. (If there is a tie of frequency, the item with a smaller item identifier appears first.) The projection of item i_k $(0 \leq k \leq m)$ is partitioned into the projection of pair $(i_k i_p)$ where $0 \leq p < k$. In general, the projected database of an item i is split into the sub-databases of pairs which consist of the item i and another item which is more frequent than i (or appears before i in the ordered list of frequent items).

For example, let $\{(c:4),(f:4),(a:3),(b:3),(m:3),(p:3)\}$ be the frequent 1-items in frequent itemset mining and sorted in descending order in terms of frequency. The numbers after the column are the frequency of the corresponding items. If selective sampling estimates that the projection corresponding to m must be partitioned, then the mining of item m is partitioned into the mining of projections for (mb), (ma), (mf) and (mc).

In sequential pattern mining, we assume that items in an event appear in a fixed order[1]. Suppose $I = \{i_1, i_2, ..., i_m\}$ are the set of frequent items of the whole dataset listed in this fixed order. We must consider both inter-event and intra-event expansion when partitioning the projections. The projection of item $\langle i_k \rangle$ $(0 \leq k \leq m)$ is partitioned into the projection of intra-event expansion $\langle (i_k i_p) \rangle$ where $k \leq p \leq m$ and the projections of inter-event expansion $\langle i_k i_q \rangle$ where $0 \leq q \leq m$.

For example, let $\{a, c, d, f, g\}$ be the frequent 1-sequences in sequential pattern mining. We assume the items within an event appear in alphabetical order. If selective sampling estimates that the projection corresponding to d must be partitioned, then the mining of item d is partitioned into the mining of projections for $\langle (df) \rangle$, $\langle (dg) \rangle$, $\langle da \rangle$, $\langle dc \rangle$, $\langle dd \rangle$, $\langle df \rangle$ and $\langle dg \rangle$. The first two are the intra-event expansions of $\langle d \rangle$ and the rest are the inter-event expansions of $\langle d \rangle$.

In closed sequential pattern mining, we only need to consider inter-event expansion since the BIDE algorithm works for single-event sequence. Suppose $I = \{i_1, i_2, ..., i_m\}$ are the set of frequent items of the whole dataset. The projection of item $\langle i_k \rangle$ $(0 \leq k \leq m)$ is partitioned into the projection of $\langle i_k i_p \rangle$ where $0 \leq p \leq m$.

For example, let $\langle A, C, D, F, G \rangle$ be the frequent 1-sequences in closed-sequential-pattern mining. We assume the items within an event appear in alphabetical order. If selective sampling estimates that the projection corresponding to D must be partitioned, then the mining of item D is partitioned into the mining of projections for $\langle DA \rangle$, $\langle DC \rangle$, $\langle DD \rangle$, $\langle DF \rangle$ and $\langle DG \rangle$, which are all inter-event expansions of $\langle D \rangle$.

Following the method described above, the projections identified with long mining time are partitioned. Since our task partitioning follows the pattern-growth philosophy of the base sequential algorithms, the derived projections are independent and can be distributed to different processors to be mined in parallel.

[1]This order can be any total order among the items.

5.1.2 Necessity of multi-level task partitioning

With the partitioning strategy described previously, a projection of a length-1 pattern is partitioned into a series of projections of length-2 patterns. For most of the datasets, the subtasks derived from our task partitioning strategy discussed in Subsection 5.1.1 are fine enough to achieve good performance for the parallelization of frequent pattern mining (See performance in Section 7.2.

However, for some special datasets, such as *pumsb_star* and $T30I0.2D1K$[2] for frequent itemset mining, the above task decomposition is not enough to obtained a balanced workload among processors.

Figure 5.1(a) and Figure 5.1(b) show the mining time distribution of the projections of single items in datasets *pumsb_star* and $T30I0.2D1K$ respectively. As shown in the graph, the mining time distributions of the projections are extremely imbalanced. As a consequence, when applying the above partitioning strategy, some projections derived from the extremely large projections are still very large and needs to be further partitioned.

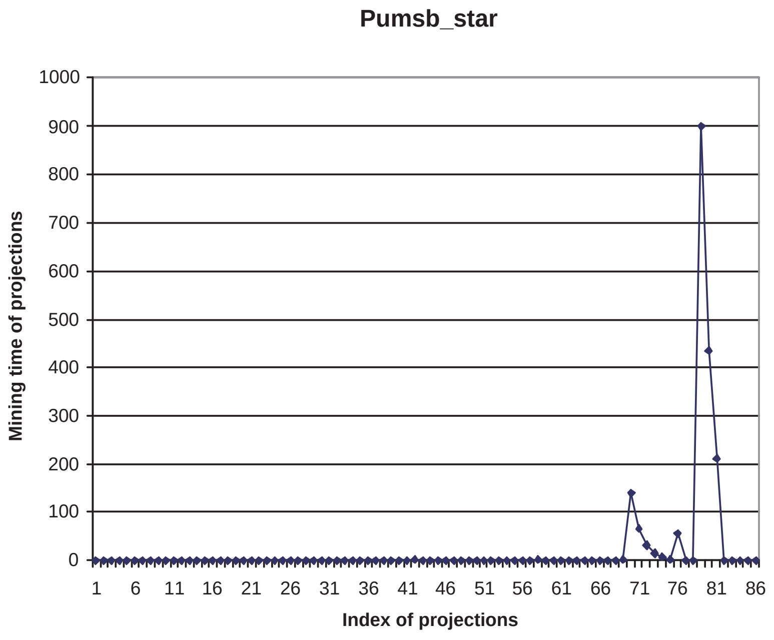
Therefore, we must extend our task partitioning strategy so that the extremely large projections of a length-1 pattern can be partitioned into the projections of not only length-2 patterns, but also length-l ($l > 2$) patterns. We call the extended task partitioning *multi-level task partitioning.* The projections of length-l pattern are called level-l projections.

As we discussed in Chapter 4, over-partitioning (unnecessary partitioning) can introduce overhead due to too small tasks, which can cause performance loss. This is a trade off on task granularity. The task partitioning algorithm must decide how many levels of partitioning is needed to produce small-enough subtasks but not introducing too much overhead.

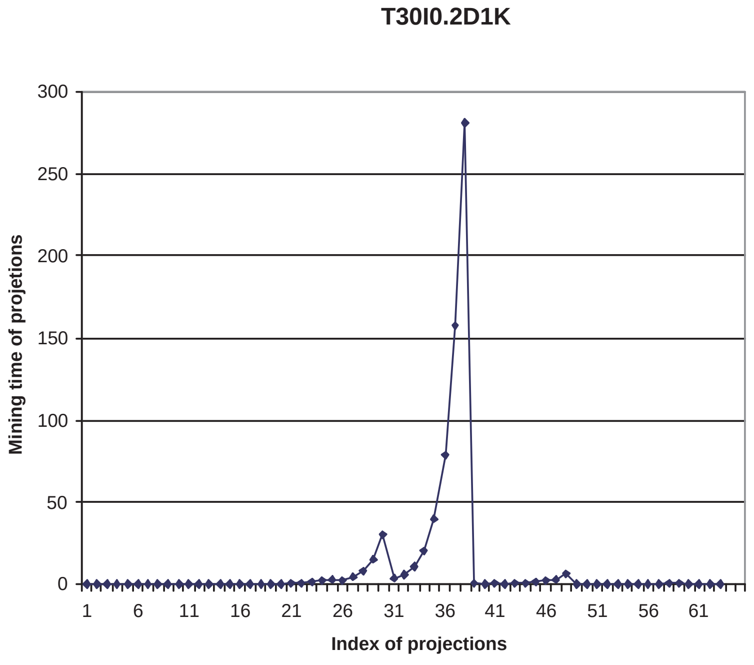
5.1.3 Multi-level task partitioning

We focus on frequent itemset mining to design the multi-level task partitioning strategy.

[2]The detailed characteristics of the dataset is described in Figure 7.1(a))



(a)



(b)

Figure 5.1: Projection mining time distribution of single items

In selective sampling, we record the mining times of the level-1 projections as an estimation of the relative mining time of them of the whole dataset. We could naively record the mining times of the projections of all the levels in selective sampling to obtain the estimation of the relative mining time of the projections of every level. However, since the number of projections increases exponentially with the number of levels, the overhead of selective sampling would become large. In our experiments, the overhead of selective sampling is more than doubled when we records the sampling mining time of projections for more than 2 levels.

There is an important property which helps us to design the multi-level task partitioning strategy. In frequent-itemset mining, the projection mining time of an itemset is no more than the projection mining time of its subset. This is because the frequency of an itemset is always no more than the frequency of its subset.

For example, the frequent itemsets containing AB must contain B, so the projection mining time of AB is no more than the projection mining time of B. Similarly, the projection mining time of AC is no more than the projection mining time of C, the projection mining time of ABC is no more than the projection mining time of AC and so on.

We designed a multi-level task partitioning strategy as following:

Let Q be the items whose sampling projection mining times are greater than $\frac{T_{Sample}}{N}$, where T_{Sample} is the total sampling mining time and N is the number of processors. The level-1 projection of item i ($i \notin Q$) is not partitioned. The itemsets corresponding to all the subsets of Q will be partitioned. For the items in Q, let their level-1 projections be partitioned into the projections of patterns in L. The patterns in L are generated as $S \cup \{\forall i\}$, where S ($S \neq \emptyset$) is an element of $2^{|Q|}$ and i is an item $\notin Q$ and is more frequent than any items in S.

For example, as in Figure 5.2 shown, suppose the sampling projection mining time of item A, B and C are greater than $\frac{T_{Sample}}{N}$, the following itemsets are then partitioned: A, B, C, AB, AC, BC, ABC and the derived pattens from the partitioning are in grey.

With the above partitioning strategy, the projection mining times of the derived patterns

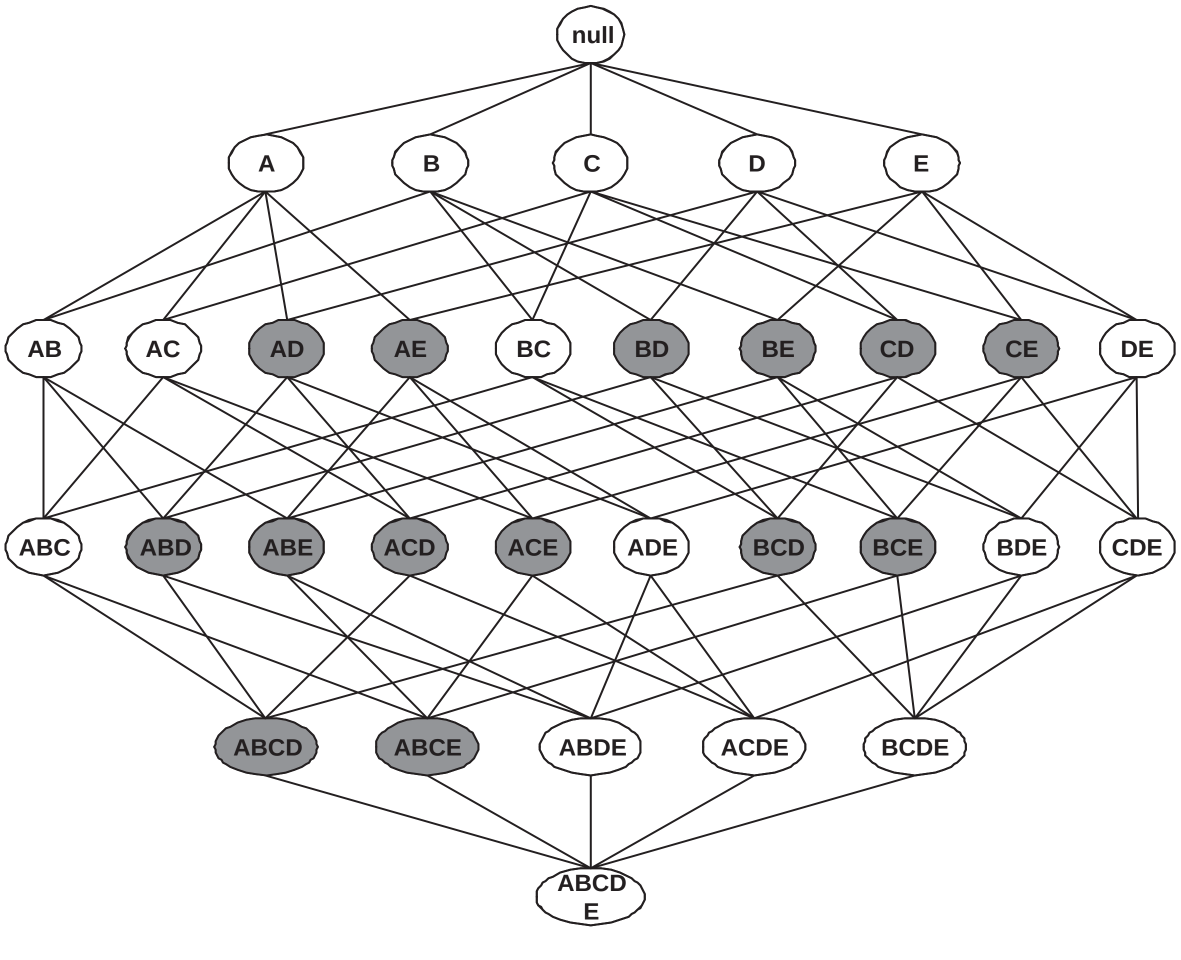


Figure 5.2: Search space of different items for frequent-itemset mining

are less than $\frac{T_{sample}}{N}$. This is because each of the derived patterns is a superset of an itemset with projection mining time less than $\frac{T_{sample}}{N}$. According to the property discussed above, the projection mining times of the derived patterns are all less than $\frac{T_{sample}}{N}$. Suppose sampling can give an accurate estimation, these projection mining time on the whole dataset are less than $\frac{T_{all}}{N}$.

For example, in Figure 5.2, since the sampling projection mining time of D and E are less than $\frac{T_{sample}}{N}$, the itemsets corresponding to the grey nodes are all supersets of either D or E. So their projection mining time are less than $\frac{T_{sample}}{N}$.

Consequently, the large 1-level projections are partitioned into fine enough sub-projections with our multi-level task partitioning strategy. As we will see in Chapter 7, this algorithm can greatly improve the performance of the datasets which require multi-level task partitioning.

5.2 Scheduling

After the projections with long mining time are partitioned, the projections/sub-projections are to be assigned to the processors. Basically there are three types of tasks to be scheduled:

1. The sub-projections derived from partitioning.
2. The projections without partitioning.
3. The projections not appearing in sample.

We applied simple strategies to schedule these three types of tasks. For the first and third categories, we use round-robin to schedule them. For the second category, since we have had an estimation of their mining time distribution from sampling, we use a bin-packing strategy to schedule these tasks according to the estimation. By applying the above scheme, we are able to statically distribute the tasks to the processors.

5.3 Parallel sampling

Although the average overhead of selective sampling is only 1% of the sequential mining time on average for Par-FP, Par-Span and Par-CSP, such overhead will become more critical when the number of processors grows large.

According to Amdahl's Law, the best possible speedup with 1% serial component on 64 nodes is restricted to 39. Therefore, in order to get better speedups on a large number of processors (≥ 128), we need to further reduce the overhead of sampling.

One promising solution is to parallelize the selective sampling. Since the process of mining the selective sample is analog to the sequential mining of the whole dataset, we may parallelize the mining of the sample in a similar way so that the overhead of sampling can be reduced.

Chapter 6

Parallel Frequent Pattern Mining Algorithms

Selective sampling is a very useful and important technique for balancing the work load of the parallelization framework proposed in Chapter 3. We designed three parallel algorithms, targeting frequent itemset mining, sequential pattern mining and closed-sequential-pattern mining respectively based on the proposed framework with the selective sampling technique. We introduce the detail of these algorithms in the following sections.

6.1 Parallel frequent itemset mining algorithm

We designed a parallel frequent itemset mining algorithm, called *Par-FP*, based on FP-growth algorithm. Algorithm 1 is the Par-FP algorithm which is presented in SPMD form.

Algorithm 1 Par-FP(I, DB_I, min_sup, FP_I)

```
Input: $I$ is the processor ID, $DB_I$ is a portion of the dataset assigned to processor $I$, $min\_sup$ is the minimum support threshold
Output: $FP_I$ is a portion of frequent itemsets
 1: $C_I = number\_of\_items(DB_I)$;
 2: GLOBAL_COUNTS=all_to_all_sum($C_I$);F1=frequent_1-itemsets(GLOBAL_COUNTS);
 3: if ($I == 0$) then
 4:     $S\_RESULT = selective\_sampling(F1, I, DB_I, min\_sup)$;
 5:     $F2 = partition(F1, S\_RESULT)$; // Partition the most time consuming subtasks and
        assign the new set of subtasks to $F2$.
 6:     $LTASK = schedule(F2, F1)$
 7:     send the elements in $LTASK$ to the corresponding processor;
 8: else
 9:     receive the task assignments from Processor 0.
10: end if
11: build FP-tree for assigned elements;
12: apply FP-Growth algorithm to mine the local FP-tree and output to $FP_I$;
```

We assume that the dataset is partitioned into N subsets (N is the total number of processors) and the subset assigned to processor P_I is denoted as DB_I. In line 1, each processor counts the single items for the part of the dataset assigned to it. Then an all-to-all reduction is performed to compute the global counts and the global numbers are stored in variable GLOBAL_COUNTS on each processor. Those items whose global numbers of occurrence are above the support threshold are identified and stored in variable $F1$.

Next, one of the processor (we use Processor 0 in our illustration) performs selective sampling (in line 4). The details of the function *selective_sampling* is as follows:

Each processor scans the local portion of the dataset, discarding the top 20% most frequent items in $F1$ and the non-frequent items in each transaction, and then sends the portion of the seletive sample to Processor 0. Processor 0 receives the portions of the sample and builds an FP-tree for the selective sample in its memory. The items in the transactions of the selective sample are sorted in descending order of frequency and inserted into an FP-tree structure, named T. T is constructed in the same way as FP-growth algorithm states. Then Processor 0 mines T using serial FP-growth algorithm. However, instead of outputting the frequent itemsets found, P returns the mining time of each frequent items in T. Assume there are S frequent items in T. The mining time of item i for the sample is $L_i(i \in [1..S])$ and stored in S_RESULT.

Based on the result stored in S_RESULT, Processor 0 partitions the most time consuming subtasks by calling function $partition()$ in line 5. We implemented two strategy for function $partition()$ in Par-FP. One partitions the large tasks only one-level and the other uses the multi-level partitioning heuristic discussed in Section 5.1 to decompose the large tasks in multi-levels. We will compare performance of both of the strategies later to show the effectiveness of multi-level task partitioning.

The workload are distributed to the processors by calling function $schedule()$ in line 6. We use the static task scheduling strategy discussed in Section 5.2 to distribute the tasks.

The results of task scheduling is stored in $LTASK$. Processor 0 then sends the assigned subtasks in $LTASK$ to the corresponding processor (line 7). After having received its assignment (line 9), every processor builds the local FP-tree with the transactions containing the assigned elements (line 11). For example, if a processor is assigned with a and b, the FP-tree constructed by this processor only contains the transactions containing a or b and those items which are less frequent than a and b are removed when inserting into the FP-tree. In this way, the FP-tree on each processor can be much smaller than the FP-tree for the whole dataset. It makes our algorithm be capable of handling larger datasets than the sequential FP-growth algorithm with the memory size limitation.

The FP-tree construction requires each processor to access the whole dataset. Since the dataset is distributed across the nodes, each portion needs to be broadcast to all the processors. In our implementation, we found that it is more efficient to carry out the broadcast using a virtual ring structure where processor I only receives the package from Processor $((I-1)\ mod\ N)$ and only sends the package to Processor $((I+1)\ mod\ N)$. Thus, assume there are total N processors, the all-to-all broadcast is carried out in $(N-1)$ non-blocking send-receive steps which collectively consume no more than 1% of the mining time. We could let each processor selectively sends the records required by each remote processor respectively. However, since the communication cost with our implementation is insignificant, using the alternative method will not bring much improvement to the overall performance.

At last, in line 12, each processor mines the FP-tree in its local memory independently without any communication with the other processors. The frequent itemsets discovered by each processor are output by the processors asynchronously to different files FP_I. The overall frequent itemsets of the dataset are just the concatenation of all these files.

6.2 Parallel sequential-pattern mining algorithm

Our parallel sequential-pattern mining algorithm is called *Par-Span* (illustrated in Algorithm 2).

Algorithm 2 Par-Span(I, DB_I, min_sup, SP_I)

```
Input: $I$ is the processor ID, $DB_I$ is a portion of the dataset assigned to processor $I$, $min\_sup$ is the minimum support threshold
Output: $SP_I$ is a portion of sequential-patterns
1: $C_I = number\_of\_1\text{-}sequences(DB_I)$;
2: GLOBAL_COUNTS=all_to_all_sum($C_I$);F1=frequent_1-sequences(GLOBAL_COUNTS);
3: if ($I == 0$) then
4:     $S\_RESULT = selective\_sampling(F1, I, DB_I, min\_sup)$;
5:     $F2 = partition(F1, S\_RESULT)$; // Partition the most time consuming subtasks and
       assign the new set of subtasks to $F2$.
6:     $LTASK = schedule(F2, F1)$
7:     send the elements in $LTASK$ to the corresponding processor;
8: else
9:     receive the task assignments from Processor 0.
10: end if
11: build projection for assigned elements;
12: apply Prefix-Span algorithm to mine the projections for the assigned elements and output
    to $SP_I$;
```

Par-Span is based on the sequential Prefix-Span algorithm. The whole flow of Par-span algorithm is similar to that of Par-FP.

First, each processor counts the number of occurrence of each 1-sequences(line 1) and then followed by a global reduction to obtain the global counts. The frequent ones are identified (line 2). Then, one of the processor performs selective sampling to identify the large subtasks (line 3). After that, the large subtasks are partitioned to smaller ones. We only implemented one-level task partitioning for Par-Span algorithm.We applied strategy discussed in Section 5.2 to schedule the subtasks(line 4). Then each processor builds the projection for the assigned tasks and then mines the projections independently using the conventional Prefix-Span algorithm.

6.3 Parallel closed-sequential-pattern mining algorithm

In this section, we describe *Par-CSP*, the parallel algorithm to mine closed sequential-patterns. Algorithm 3 is the Par-CSP algorithm which is presented in SPMD form.

```
Algorithm 3 Par-CSP(I, $DB_I$, $min\_sup$, $CSP_I$)
Input: $I$ is the processor ID, $DB_I$ is a portion of the dataset assigned to processor $I$,
$min\_sup$ is the minimum support threshold
Output: $CSP_I$ is a portion of sequential-patterns
 1: $C_I = number\_of\_1\text{-}sequences(DB_I)$;
 2: GLOBAL_COUNTS=all_to_all_sum($C_I$);F1=frequent_1−sequences(GLOBAL_COUNTS);
 3: $PSP = pseudo\_projection(F1, DB_I)$;
 4: $GLOBAL\_PSP = all\_to\_all\_broadcast(PSP)$;
 5: $S\_RESULT = selective\_sampling(F1, I, DB_I, min\_sup)$;
 6: $F2 = partition(F1, S\_RESULT)$; // Partition the most time consuming projections
    and assign the new set of projections to $F2$.
 7: if ($I == 0$) then
 8:    accept requests from slave nodes and reply to each request with a different identifier
       from set $F2$ until all projections have been assigned;
 9: else
10:    send request for a projection identifier to the master node;
11:    stop if all projections have been assigned;
12:    apply BIDE algorithm to element of $GLOBAL\_PSP$ assigned by the master processor;
13:    accumulate the closed sequential-patterns into $CSP_I$ and go back to send request
       operation;
14: end if
```

Each processor first counts the number of occurrence of each 1-sequences(line 1) and then followed by a global reduction to obtain the global counts. The frequent 1-sequences are identified (line 2). Each processor then builds pseudo projections for for all the frequent 1-sequences based on the local portion of datasets (line 3) and broadcast the projections built to all the other processors(line 4). Note that we used a different strategy here from the previous two algorithms. Each processor has the projections for all the 1-sequences. Since the projections are pseudo projection, each processors basically has the whole dataset and the pointers for each projections in memory. Then the processors perform selective sampling in parallel. Each processor mines the selective sample for a portion of the frequent 1-sequences and records the mining time(line 5). Then the large subtasks are partitioned. A dynamic scheduling is applied in Par-CSP algorithm[1]. The indexes of subtasks are kept in a task queue on the node 0 and assigned to the other nodes. The processors other than node 0, are initially assigned one index each. After a processor completes the mining of a subtask, it sends a request to node 0 for another. Node 0 replies with the index of the next subtask in the queue and removes it from the queue. This process continues until the queue of subtasks is empty.

The requests and replies to and from node 0 are short messages and, therefore, the communication time is usually negligible relative to the mining time. The subtasks estimated to take longer time in sampling are to be scheduled earlier. Those very small subtasks are scheduled in chunks to avoid communication contention.

[1]The scheduling strategy discussed in Section 5.2 is used when the number of processors is less than 8.

Chapter 7

Experimental Analysis

In this chapter, we evaluate the performance of our parallel frequent pattern mining algorithms and performance optimization techniques by conducting a series of experiments on various datasets.

7.1 Experiment Setup

Our experiments were performed on two Linux clusters. On the first cluster, A, each node has a 1GHz Pentium III processor and 1GB main memory. On the second cluster, B, each node has dual 1.3 GHz Intel Itanium2 processors and 4GB main memory. The network interconnections on both clusters are Gigabit Ethernet Myrinet 2000. We used MPICH-GM to implement our parallel algorithms. MPICH-GM is a portable implementation of MPI that runs over Myrinet. The operating system is Linux 7.2 and the compiler is GNU g++ 2.96. On both cluster, global disks are used and the processors may do concurrent I/O operations.

A comprehensive performance study has been conducted in our experiments on 16 databases representing both synthetic and real world datasets. The main characteristics of these datasets and the support threshold values used in our experiments are shown in Figure 7.1. There are seven transactional datasets for frequent itemset mining, five sequence datasets for sequential pattern mining and four sequence datasets in closed-sequential pattern mining[1].

Among the seven transactional datasets, four of them are realistic datasets downloaded

[1]The sequence datasets for closed-sequential pattern requires 1-item events, which is limited by the sequential BIDE algorithm.

Dataset	#Transactions	#Items	Support threshold	Ave Trans. Length	Max Trans. Length
mushroom	8,124	120	1%	23	23
connect	57,557	129	30%	43	43
pumsb	49,046	7,116	50%	43	74
pumsb_star	49,046	7,116	20%	43	63
T30I0.2D1K	1,000	200	5%	30	54
T40I10D100K	100,000	999	0.03%	44	77
T50I5D500K	500,000	5,000	0.05%	55	94

(a) Datasets for frequent itemset mining

Dataset	#Sequences	#Items	Support threshold	Ave. #Events/sequence	Ave #Items/Event
C10N0.1T8S8I8	10,000	1,000	0.25%	8	8
C50N10T8S20I2.5	50,000	10,000	1%	20	8
C100N5T2.5S10I1.25	100,000	5,000	0.02%	10	2.5
C200N10T2.5S10I1.25	200,000	10,000	0.01%	10	2.5
C100N20T2.5S10I1.25	100,000	20,000	0.01%	10	2.5

(b) Datasets for sequential-pattern mining

Dataset	#seq.	#items	Support threshold	Ave.seq.len.	Max.seq.len.
C100S50N10	100,000	6,044	0.01%	31	56
C100S100N5	100,000	4,162	0.01%	62	101
C200S25N9	178,742	5,661	0.005%	16	39
Gazelle	2,937	1,423	0.2%	29	1,443

(c) Datasets for closed-sequential-pattern mining

Figure 7.1: Dataset characteristics

from the FIMI repository[1] and the rests are synthetic datasets generated by the IBM dataset generator [19]. Figure 7.2 gives the frequencies of the items in each dataset for frequent-itemset mining.

Most of the sequence datasets are synthetic datasets. Only one, named *Gazelle*, is a real dataset. Gazelle comes from click-stream data provided by Blue Martini company. We consider different products as different items and the page views as events. We treat 10 consecutive Web click-stream as a sequence from one customer[2].

7.2 Execution time and speedups

We first examine the parallel performance of the three algorithms by comparing the mining time of the parallel algorithms against the corresponding serial algorithms. For frequent-itemset mining, our serial implementation is based on the implementation provided by the inventer of the FP-growth algorithm. And our serial implementations for sequential-pattern mining and closed-sequential pattern mining are provided by the authors of PrefixSpan algorithm and BIDE algorithm respectively. All these implementations are among the best serial implementations.

7.2.1 Par-FP

We tested Par-FP on different numbers of processors, including 2, 4, 8, 16, 32 and 64, on the two clusters. Figure 7.3 shows the speedups for each datasets. In the graphs, *clusterA* refers to the cluster with Pentium III processors while *clusterB* is the cluster with Itanium2 processors. In the experiments of this section, we only applied one-level task partitioning after sampling. The results for multi-level task partitioning will be shown in Section 7.3.

[2]Because the average length of one web click stream is only 3, without the merging, the sequential mining time is so short(less than 120 seconds with the absolute support threshold as 1) that it is not able to clearly show the efficiency of parallelism. Such merging is also reasonable and accords with the people's web browse manner because a person usually access the web with intervals during a period of time

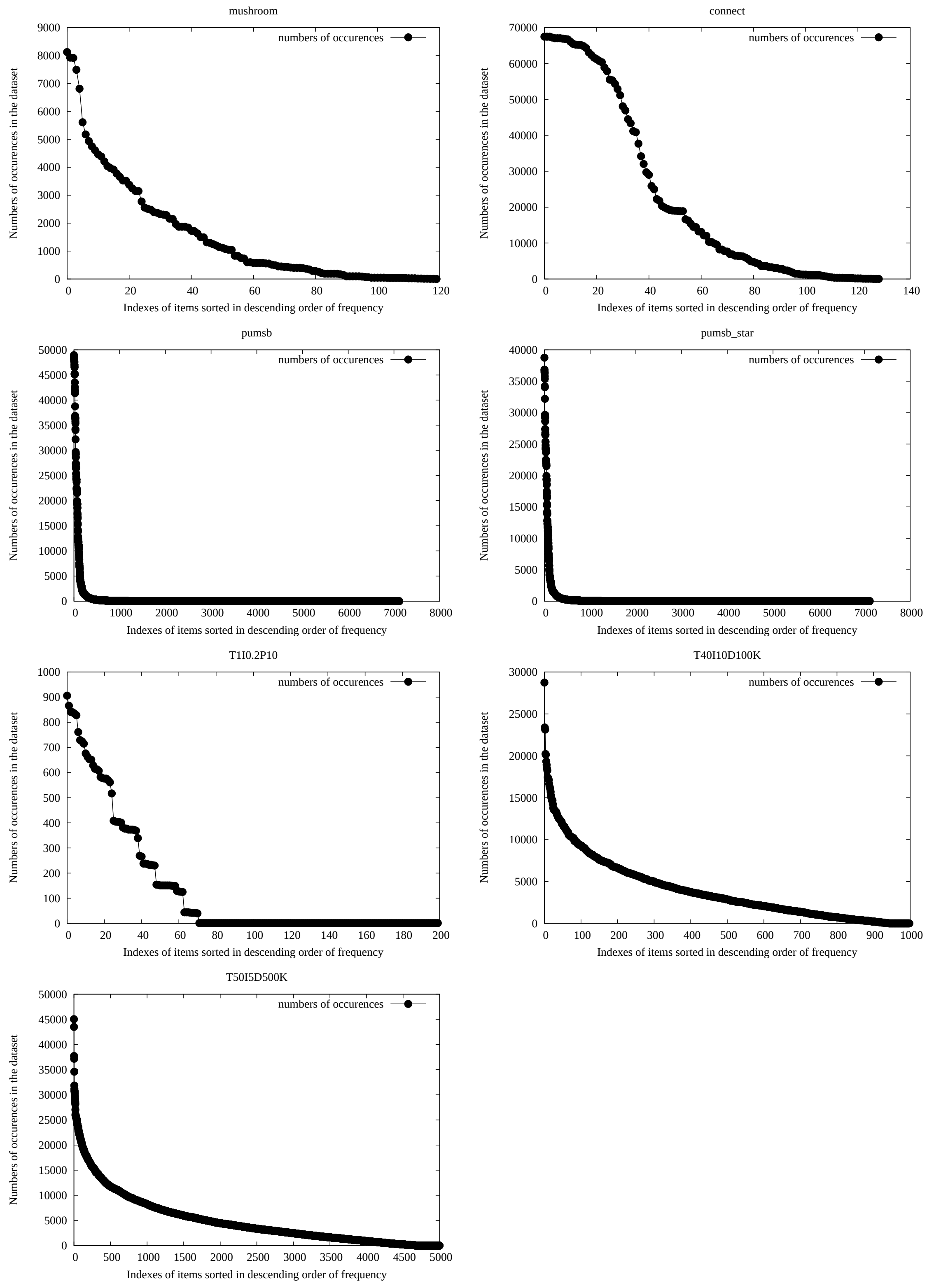


Figure 7.2: Frequency histogram of items in the datasets for frequent-itemset mining

Since the speedups of Par-FP on the two clusters are similar, we use either of the cluster in the following experiments according to the availability of the cluster.

As the charts indicate, Par-FP can achieve good speedups for most of the datasets on both of the clusters. The speedups are scalable to 64 processors. Only the performance of one of the dataset, *pumsb_star*, is not satisfactory. As shown in the next section, the speedup on *pumsb_star* can be greatly improved by using multi-level partitioning.

The parallel performance analysis for Par-FP with 64 processors on cluster B is shown in Figure 7.4. The first row lists the serial execution time of these datasets and the second row is the parallel execution time of Par-FP on 64 processors with one-level-task partitioning. The parallel execution time basically consists of four components:

- Identify the frequent-1 items followed by a reduction
- Selective sampling
- Build projections(FP-tree structure) for the assigned tasks/subtasks
- Mine the projection

Row 3-6 are the detailed execution times of the above four steps for each datasets. Row 7 lists the maximal mining time of all the tasks/subtasks and row 8 is the total number of tasks/subtasks. The times in the tale are all in seconds.

There are several reasons which cause the performance gap between the optimal speedup and the speedups.

First, as the table shown, the maximal subtask mining time may still be large after one-level task partitioning. For datasets *pumsb_star* and $T30I0.2D1K$, the maximal subtasks take the majority portion of the parallel mining time and therefore this is the main reason for the poor performance of these two datasets. For the other datasets, the maximal subtasks on average take 18% of the parallel mining time which may introduce unbalanced work load and harm the performance.

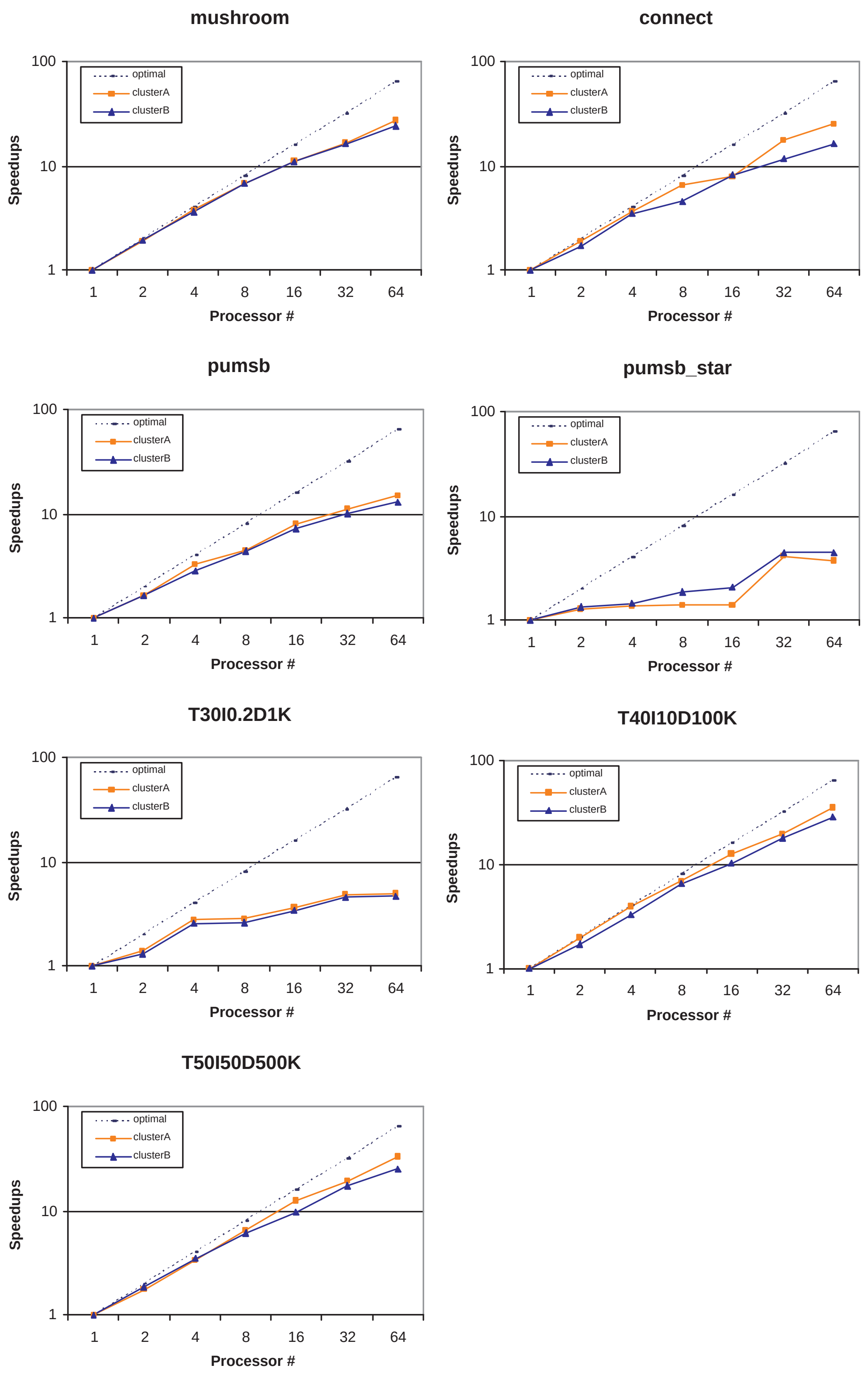


Figure 7.3: Par-FP speedups

	mushroom	connect	pumsb	pumsb_star	T30I0.2D1K	T40I10D100K	T50I5D500K
Serial execution time	19.8	220.3	1388.4	1950.8	728.3	2918.3	4003.4
Parallel execution time on 64 processors	0.83	13.8	106.8	527.2	161.8	102.1	160.2
Find frequent-1 items	0.008	0.03	0.04	0.03	0.002	0.04	0.28
Selective sampling	0.14	3.9	9.8	5.7	1.3	20.6	11.2
Build projections	0.08	0.88	1.78	1.25	0.06	1.61	8.9
Mining projections	0.602	8.09	95.2	520.2	160.4	79.85	139.8
Maximal subtask	0.23	0.77	22.4	428.1	153.6	21.9	24.3
Number of subtasks	534	842	458	480	307	1423	8021

Figure 7.4: Parallel performance analysis for Par-FP on 64 processors

Second, although the overhead of selective sampling is no more than 1% of the serial execution time, such overhead become non-neglectable when coming to 64 processors according to Amdahl's Law. For example, in datasets *connect* and $T40I10D100K$, the selective sampling overhead takes more than 20% of the parallel execution time.

Furthermore, our simple static scheduling strategy discussed in Section5.2 may also lead to unbalanced work load, especially when the derived subtasks are large and the total number of subtasks is small, such as the case for dataset *mushroom*.

7.2.2 Par-Span

We then test Par-Span on the Pentium III cluster with 2, 4, 8, 16, 32 and 64-processor configurations. Figure 7.5 shows the speedups obtained with the datasets in Figure 7.1(b) and Figure 7.6 gives the parallel performance analysis on 64 processors.

From the graphs, we can see that Par-Span has achieved good speedups for most of the cases with the dataset on up to 64 processors. However, from Figure 7.6, we noticed that load balancing and sampling overhead are still the main reason to cause the gap between optimal speedup and Par-Span speedups. Similar to *pumsb_star* and $T30I0.2D1K$ in frequent-itemset mining, the speedup of Par-Span on dataset $C10N0.1T8S8I8$ goes flat after 8 processors. As we can see from Figure 7.6, the maximal subtasks is around 65% of

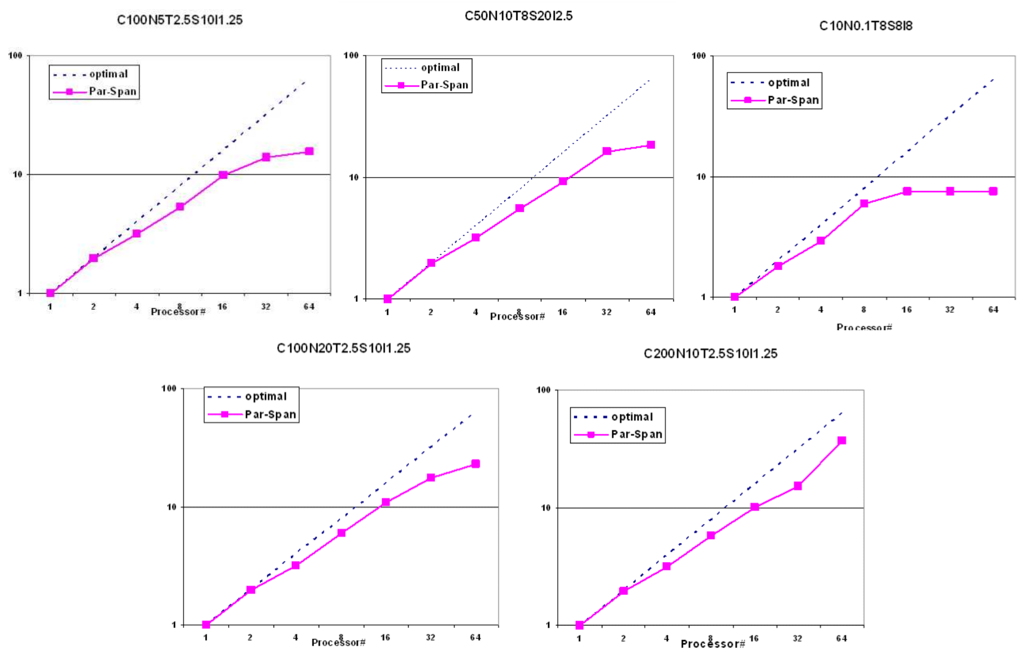


Figure 7.5: Par-Span speedups

the parallel mining time for $C10N0.1T8S8I8$. In this case, multi-level task partitioning is necessary to improve the performance on large number of processors.

7.2.3 Par-CSP

At last, we test the performance for Par-CSP on the Pentium III cluster. The speedup is shown in Figure 7.7. The sequential execution time of the four datasets are respectively: $C100S100N5$ (2322.1sec), $C100S50N10$ (208.9sec), $C200S25N9$ (8630.5sec) and *Gazelle* (142.8sec). As shown in Figure 7.7, Par-CSP is scalable up to 64 processors for most of the datasets. Among the four datasets, the performance of dataset *Gazelle*) drops obviously when the number of processors are large. From our analysis, this is because of load balancing. Selective sampling failed in identifying a large projection whose projection mining time is 7.3 seconds (5% of serial execution time).

	C10N0.1T8S8I8	C50N10T8S20I2.5	C100N5T2.5S10I1.25	C200T2.5S10I1.25	C100N20T2.5S10I1.25
Serial execution time	136.9	254.8	183.9	377.1	928.5
Parallel execution time on 64 processors	18.3	13.8	11.8	10.2	40.5
Find frequent-1 items	0.006	0.008	0.007	0.009	0.007
Selective sampling	0.7	1.8	1.1	3.2	4.9
Build projections	0.03	0.03	0.04	0.05	0.04
Mining projections	17.6	11.9	10.7	6.9	35.6
Maximal subtask	11.8	2.06	1.57	1.12	2.47
Number of subtasks	4775	28530	27112	89694	59613

Figure 7.6: Parallel performance analysis for Par-Span on 64 processors

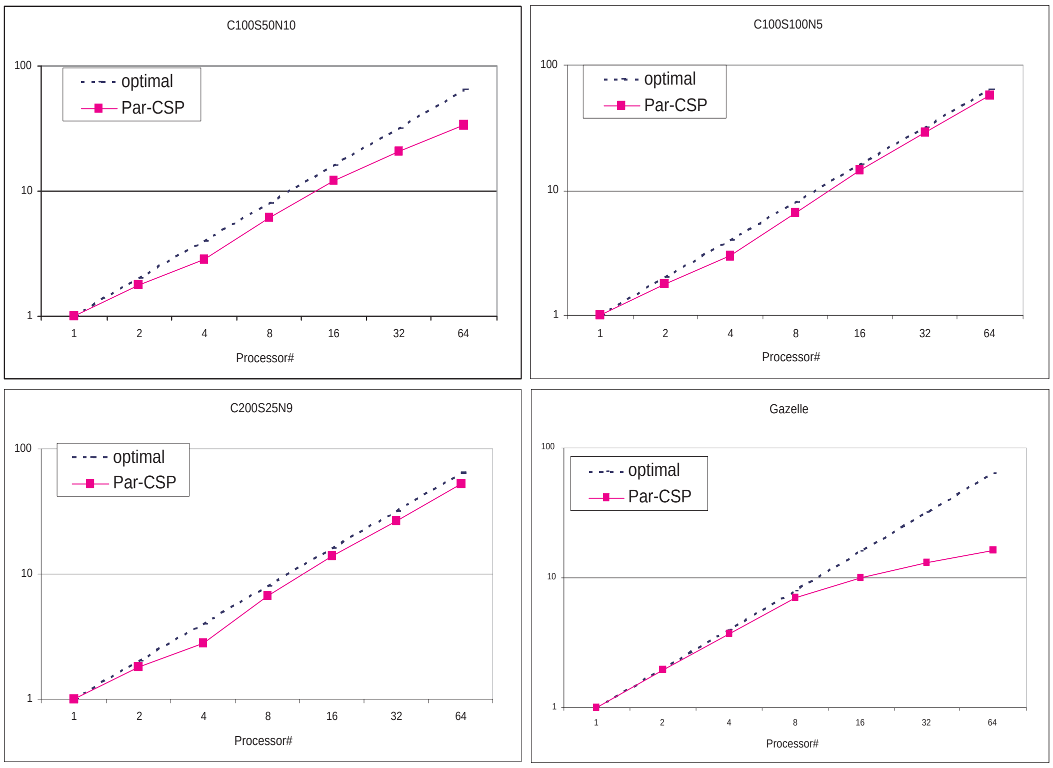


Figure 7.7: Par-CSP speedups

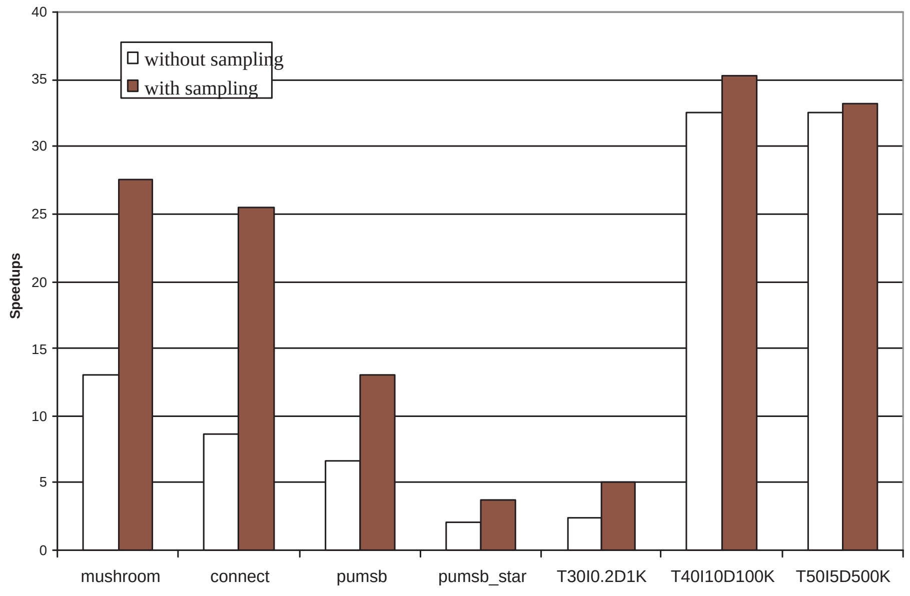


Figure 7.8: Effectiveness of selective-sampling on Par-FP with 64 processors

7.3 Effectiveness of selective sampling

7.3.1 One-level task partitioning

In order to evaluate the effectiveness of the selective sampling technique, we compared the performance of Par-FP and Par-Span to the straight parallel implementations without selective sampling on 64 processors. Figure 7.8 and Figure 7.9 show the difference of speedups for frequent itemset mining and sequential pattern mining respectively.

As we can see from the graphs, selective sampling can improve the performance of the parallel algorithms in all the cases we tested. For most of the cases, the speedups can be improved by more than 50%.

For Par-CSP, we compared the performance with selective sampling enabled and disabled for dataset $C200S25N9$ on 4, 8, 16, 32 and 64 processors respectively (see the results in Figure 7.10). It is noticed that when the number of processors is small, the sampling technique

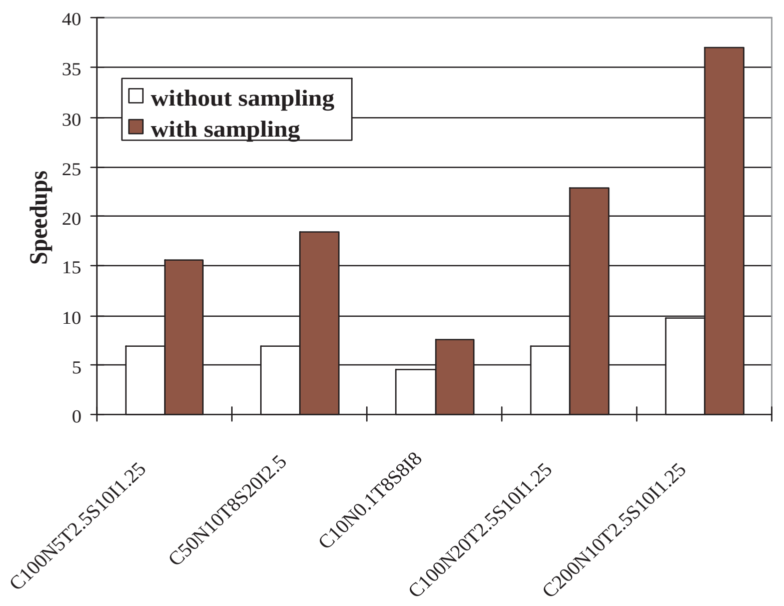


Figure 7.9: Effectiveness of selective-sampling on Par-Span with 64 processors

does not show much advantage. This is because when there are only a few processors, the number of subtasks assigned to each processor is large enough so that it tends to balance the load. However, when the number of processors becomes larger, the sampling technique greatly improves performance. For 64 processors, the performance of Par-CSP is improved by more than 50%.

7.3.2 Multi-level task partitioning

We then check the effectiveness of the selective multi-level task partitioning discussed in Section 5.1. Figure 7.11 shows the comparison of speedups for Par-FP for all the datasets in Figure 7.1(a). As the figure shown, multi-level task partitioning can greatly improve the scalability for the dataset *pumsb_star* and $T30I0.2D1K$. For the other datasets, since the mining time variation between the subtasks is not as great as *pumsb_star* and $T30I0.2D1K$, the improvement on the speedups is not so significant.

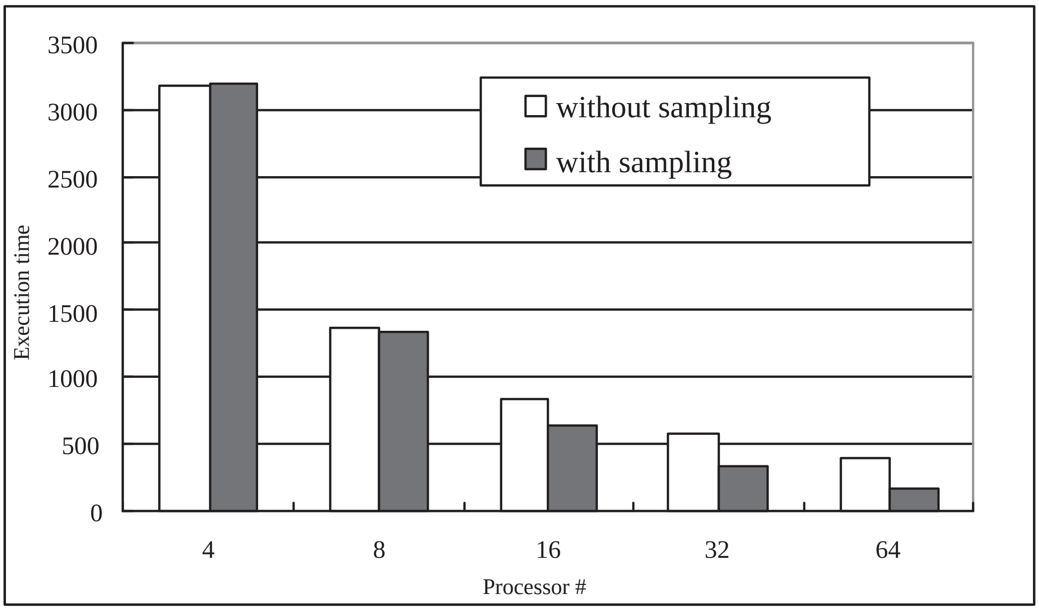


Figure 7.10: Effect of selective sampling on Par-CSP with various numbers of processors

7.4 Sensitivity analysis

A dataset has various parameters, such as number of items, number of transactions and so on. It would be interesting to study whether the parallel algorithms can maintain good performance when these parameters change. The datasets used in this section are all generated by the IBM dataset generator [19]. We studied the performance in regard to four important parameters of a dataset: the number of items, the number of transactions, the width of transactions and the support threshold. We use Par-FP to examine the performance with different settings of those parameters. We applied multi-level task partitioning when executing the following experiments.

7.4.1 Numbers of items

We generated four datasets with the number of items ranging from 1,000 to 1,000,000. The key parameters of the datasets are listed in Figure 7.12. The speedups are shown in Figure 7.13 and the performance analysis is in Figure 7.14. As we can see from the speedup

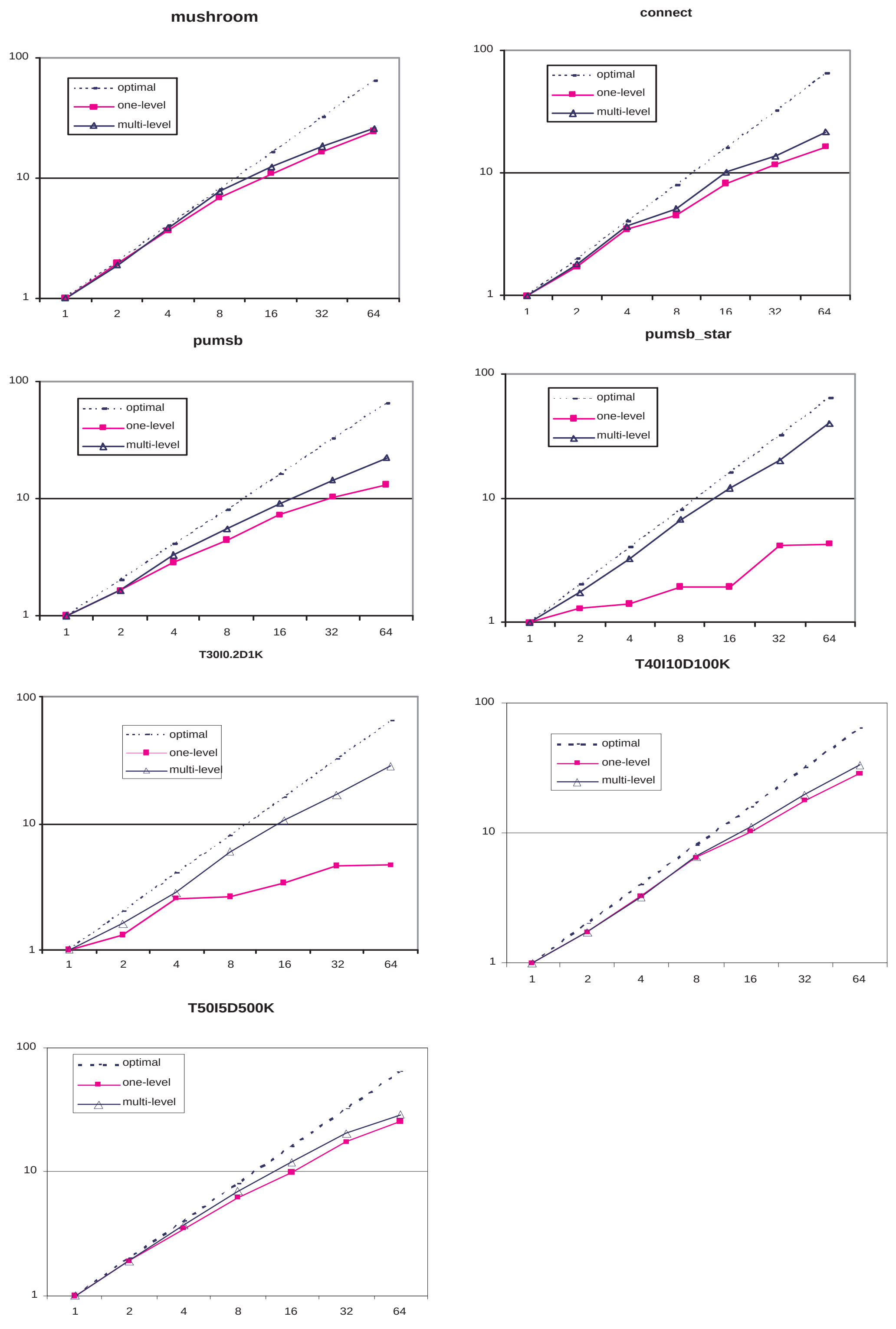


Figure 7.11: Effectiveness of multi-level task partitioning

Datasets	#Items	#Transactions	Ave Width of Trans	Support Threshold
I1	1000	100000	30	0.01%
I10	10000	100000	30	0.01%
I100	100000	100000	30	0.01%
I1000	1000000	100000	30	0.01%

Figure 7.12: Dataset parameters for tests of various items

graph, our algorithm shows good scalability with all the four datasets. The more items in the datasets, the more subtasks to be mined, and therefore the work load tends to be more even with our scheduling method. This may give the reason why the speedup of dataset $I1$ is worse than the other datasets.

7.4.2 Width of transactions

Four datasets are generated by setting the width of transactions from 10 to 90. Figure 7.15 lists the basic parameters of the datasets and Figure 7.16 shows the speedups on these datasets. We analyze the parallel performance on 64 processors in Figure 7.17. Our algorithm has achieved good speedups for all the five datasets up to 64 processors.

From our experiments, we noticed that the mining time of the dataset increases greatly by increasing the width of transactions while keeping all the other dataset parameters unchanged. The sequential mining time of the dataset varied from a few seconds with the transaction width as 10 to more than fourteen hours with the transaction width as 90. This further proves why the overhead of our selective sampling technique can be small. By removing the most frequent items, the width of each transaction are shorten so that the mining time of the sample can be greatly reduced.

7.4.3 Numbers of transactions

We generated seven datasets with the number of transactions set from 10,000 to 10,000,000. The dataset parameters are listed in Figure 7.18 and the performance results are shown in

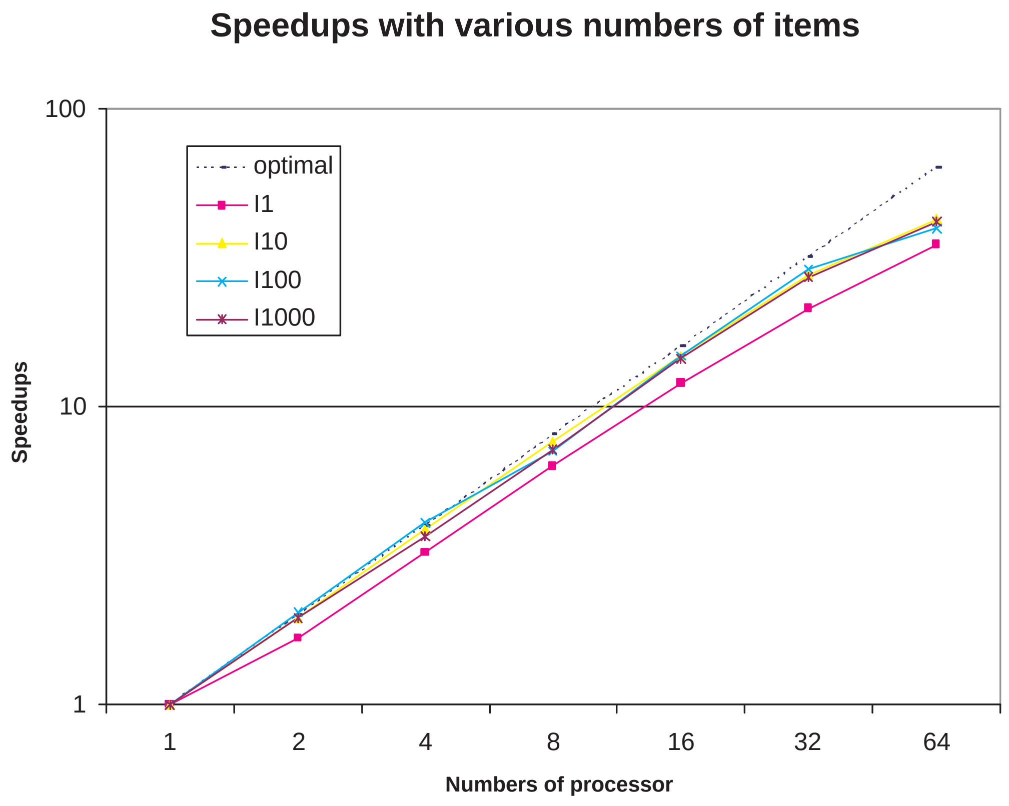


Figure 7.13: Performance of datasets with various number of items

	I1	I10	I100	I1000
Serial execution time	373.6	437.3	538.7	1212.7
Parallel execution time on 64 processors	11.1	10.2	13.5	28.9
Find frequent-1 items	0.03	0.04	0.03	0.04
Selective sampling	1.61	2.37	2.83	3.87
Build projections	0.05	0.08	0.39	1.05
Mining projections	9.41	7.71	10.25	23.94
Maximal subtask	0.48	0.33	0.49	0.27
Number of subtasks	2433	13960	152314	1025682

Figure 7.14: Parallel performance analysis with datasets of various numbers of items

Datasets	#Items	#Transactions	Ave Width of Trans	Support Threshold
W10	1000	100000	10	0.1%
W30	1000	100000	30	0.1%
W50	1000	100000	50	0.1%
W70	1000	100000	70	0.1%
W90	1000	100000	90	0.1%

Figure 7.15: Dataset parameters for tests of various width of transactions

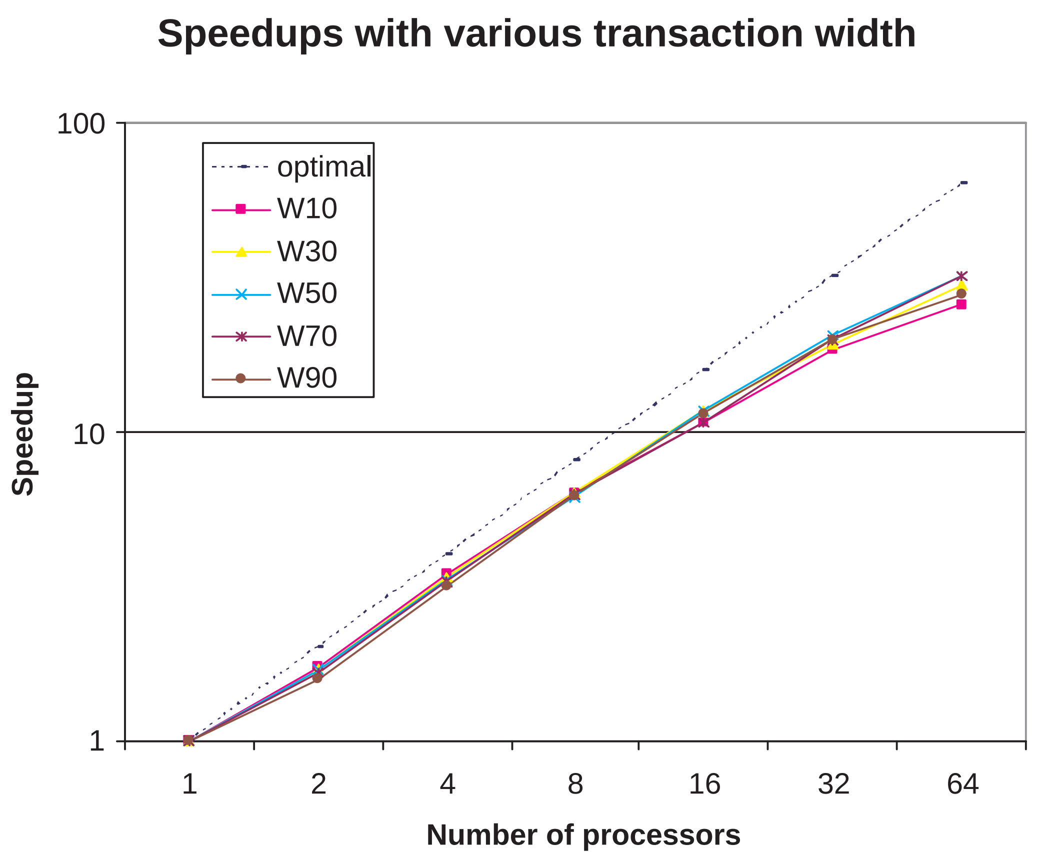


Figure 7.16: Performance of datasets with various width of transactions

	W10	W30	W50	W70	W90
Serial execution time	12.3	42.3	470.9	4814.5	51776.1
Parallel execution time on 64 processors	0.48	1.41	14.7	150.5	1849.1
Find frequent-1 items	0.006	0.03	0.14	0.18	0.22
Selective sampling	0.05	0.11	1.88	39.94	224.9
Build projections	0.01	0.04	0.49	1.42	2.79
Mining projections	0.41	1.23	12.2	108.9	1621.19
Maximal subtask	0.02	0.05	0.22	4.44	38.2
Number of subtasks	1921	2829	4119	6002	7855

Figure 7.17: Parallel performance analysis with datasets of various width of transactions

Datasets	#Items	#Transactions	Ave Width of Trans	Support Threshold
T10	1000	10000	30	0.1%
T100	1000	100000	30	0.1%
T1000	1000	1000000	30	0.1%
T3000	1000	3000000	30	0.1%
T3500	1000	3500000	30	0.1%
T5000	1000	5000000	30	0.1%
T10000	1000	10000000	30	0.1%

Figure 7.18: Dataset parameters for tests of various numbers of transactions

Figure 7.19. We gives the parallel performance analysis in Figure 7.20.

Generally, from the speedup graph, we can see that the performance of our algorithm is very scalable with the number of processors increasing. However, with the numbers of transactions increasing, the FP-tree structure of the dataset is enlarged correspondingly. For the dataset with 3000K transactions, the size of the FP-tree for the sequential algorithm exceeds the memory available on the processor so that the sequential program fails. While our Par-FP algorithm can still work on 2 and more processors. This is because in Par-FP the FP-tree on each processor is for the assigned items of each processor and the size of the tree is smaller than the whole FP-tree in the sequential algorithm, and the more processor,

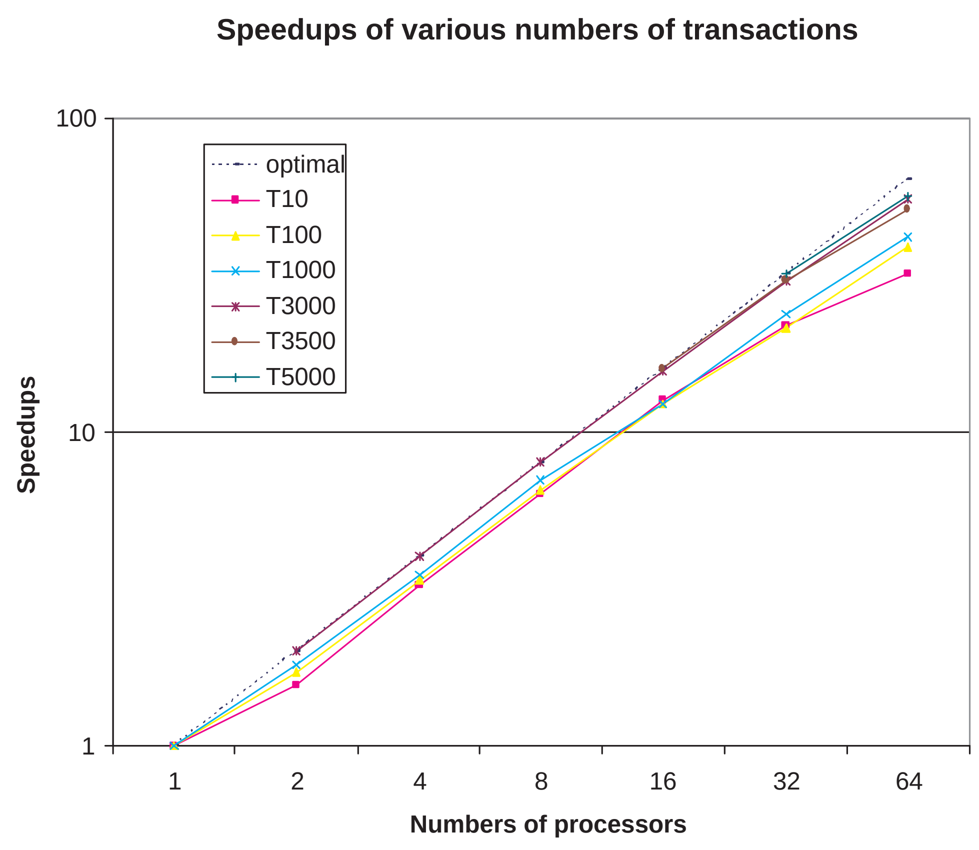


Figure 7.19: Performance of datasets with various number of transactions

	T10	T100	T1000	T3000	T3500	T5000
Serial execution time	19.5	42.3	518.6	-	-	-
Parallel execution time on 64 processors	0.78	1.41	15.8	74.9	90.2	116.7
Find frequent 1 items	0.002	0.03	0.28	0.45	0.53	1.12
Selective sampling	0.09	0.11	3.02	8.94	9.27	12.3
Build projections	0.03	0.04	2.19	7.24	8.33	10.5
Mining projections	0.658	1.23	10.3	58.3	72.1	92.8
Maximal subtask	0.03	0.05	0.54	1.88	2.08	2.76
Number of subtasks	1877	2829	2941	3550	3733	5211

Figure 7.20: Parallel performance analysis with datasets of various numbers of transactions

the smaller FP-tree each processor has. For dataset T3500, Par-FP runs successfully with 16 and more processors. For dataset T5000, Par-FP can run on 32 and more processors. For dataset T10000, Par-FP fails on 64 processors due to memory limitation. However, if more processors are available, we believe that Par-FP can mine T10000 successfully because when increasing the number of processors, the memory requirement of each processor is decreasing. This proves that our parallel algorithm is much more scalable than the sequential FP-growth algorithm in terms of the size of the dataset.

In Figure 7.18, the speedups for dataset T10, T100, T1000 are obtained by comparing the mining time of Par-FP with the mining time of the sequential FP-growth algorithm. For dataset T3000, T3500 and T5000, because the sequential algorithm fails, we compute the speedup with the performance of Par-FP on the least numbers of processors where Par-FP can run successfully. For example, dataset T3000 can run on 2 and more processors. We assume the speedup on 2 processors is 2 and use the mining time on 2 processors as the base to be compared with. The speedup on n processor is then computed by comparing the performance on n processor with the performance on 2 processors and then timed by 2.

7.4.4 Support threshold values

We studied the performance of Par-FP on different settings of minimum support threshold. Total four datasets are tested and their settings are listed in Figure 7.21, where the support threshold ranges from 0.0005% to 1%. The performance and the analysis are shown in Figure 7.22 and Figure 7.23. From the speedup graph, a better speedup is obtained when a smaller support threshold is applied. From Figure 7.23, we think the reason is probably because with lower suppor threshold, the mining time is longer so that the maximal subtask is relatively smaller. In addition, more subtasks are available to be scheduled with lower suppor threshold which may improve balance of the workload.

Datasets	#Items	#Transactions	Ave Width of Trans	Support Threshold
S1	1000	100000	30	1%
S2	1000	100000	30	0.1%
S3	1000	100000	30	0.01%
S4	1000	100000	30	0.005%

Figure 7.21: Dataset parameters for tests of various support threshold values

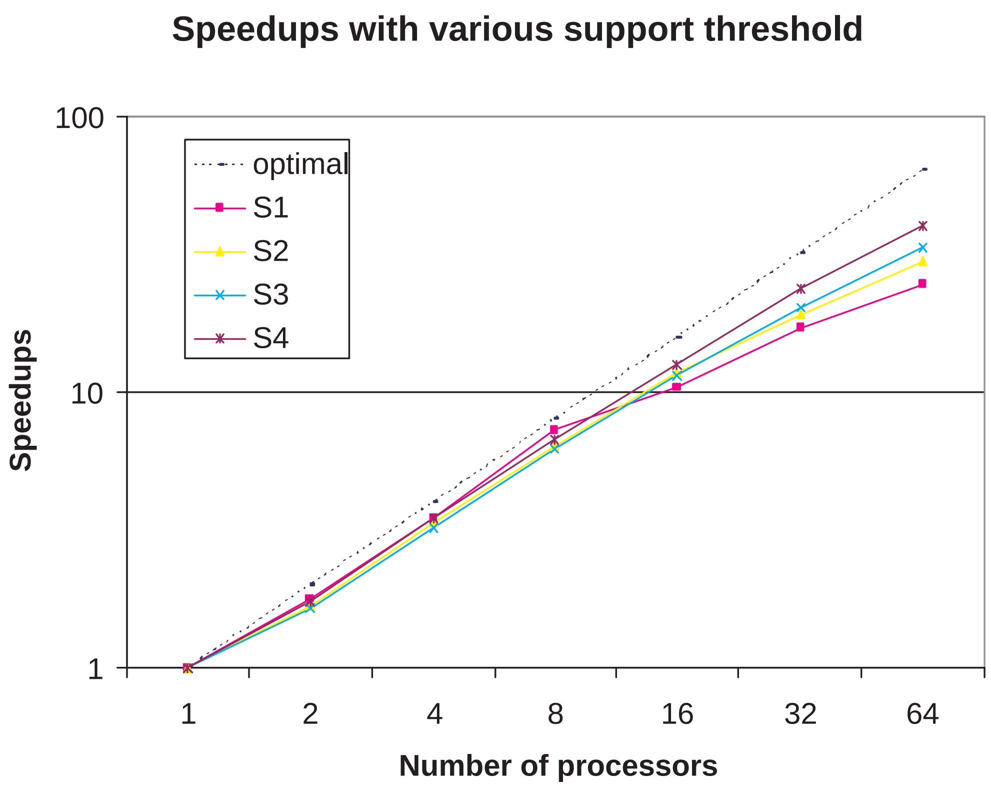


Figure 7.22: Performance of datasets with various values of minimum support threshold

	S1	S2	S3	S4
Serial execution time	9.18	42.3	373.6	1381.9
Parallel execution time on 64 processors	0.38	1.41	11.1	34.4
Find frequent-1 items	0.03	0.03	0.03	0.03
Selective sampling	0.09	0.11	1.61	6.57
Build projections	0.04	0.04	0.05	0.08
Mining projections	0.22	1.23	9.41	27.72
Maximal subtask	0.01	0.05	0.48	0.67
Number of subtasks	1685	2829	2433	5118

Figure 7.23: Parallel performance analysis with datasets of various support threshold values

Chapter 8

Related Work

8.1 Parallel frequent itemset mining algorithms

For frequent itemset mining, the published parallel frequent pattern mining algorithms are mostly based on candidate generation-and-test approach [2, 15, 27, 44]. However, despite the various heuristics used to improve the efficiency, they still suffer from the costly database scans, which are needed in all candidate generation-and-test approaches. This greatly limit the efficiency of these parallel algorithms.

To reduce the inter-processor communication during mining, Cheung et al. presented a parallel frequent itemset mining algorithm *FDM* [10]. FDM based on the property that a globally frequent itemset has to be within at least one locally frequent itemset. This algorithm requires two passes for mining the frequent itemsets, one local and the other global. After the local mining, each processor broadcasts its locally large itemsets to all the other processors for global mining. Although the *FDM* algorithm avoids communication during local mining, the number of local frequent sets that are sent between local and global mining is large and greatly degrades the performance of this algorithm.

Schuster et al. raised algorithms to reduce the amount of communication of FDM in [33]. However, although the number of bytes transmitted has been greatly reduced, the performance has not been improved as shown in [38], because of the increased number of messages sent among the processors.

As far as we know, there are three parallel frequent itemset mining algorithms based on pattern-growth methods [41, 20, 32].

The algorithm presented in [41] targets a shared-memory system. Although it is possible to apply the algorithm to a distributed-memory system by replicating the shared read-only FP-tree, it will not efficiently use the aggregate memory of the system and will suffer of the same memory constraint of the sequential algorithm.

In [20] the FP-growth algorithm is parallelized for a distributed memory system and reports the speedups only of 8 and fewer processors. Both [41] and [20] did not address the load balancing problem.

In [32], the load balancing problem is handled using a granularity control mechanism. The effectiveness of such mechanism depends on the optimal value of a parameter. However, the paper does not show an effective way to obtain such value at run time.

In [38], the authors proposed a parallel itemset mining algorithm, named D-Sampling, based on mining a random sample of the datasets. However, different from our work, the sampling here is not used to estimate the relative mining time, but to trade off between accuracy and efficiency. D-sampling algorithm mines all frequent itemsets within a predefined error, based on the mining results of the random sampling. feq

8.2 Parallel sequential-pattern mining algorithms

Although there have been numerous studies on sequential-pattern mining, the study on parallel sequential-pattern mining is still limited and is only confined to mining the complete set of sequential patterns.

The method in [34] is based on a candidate generation-and-test approach. The scheme involves exchanging the remote database partitions during each iteration, resulting in high communication cost and synchronization overhead.

In [42], Zaki presents a parallel sequential-pattern mining algorithm, called *p*SPADE, for discovering the set of *all* frequent subsequences. pSPADE adopts task parallelism to decompose the search space. To achieve load balancing, pSPADE utilizes a strategy, called *r*ecursive dynamic load balancing, which is a protocol to allow an idle processor to join the busy ones. Different from the Par-CSP algorithm proposed in this paper, *p*SPADE is targeting a shared-memory system. In a shared memory system, all the processors can access the same global memory space, which makes the proposed recursive dynamic load balancing strategy easy to be implemented. However, applying such a strategy in a distributed memory system, which is typical in a computer cluster environment, is too expensive to be practical.

Recently, Guralnik and Karypis [13] presented some parallel sequential-pattern mining algorithms toward a distributed-memory system for mining *the complete set* of sequential-patterns. These parallel algorithms are based on a tree-projection-based sequential algorithm, which is intrinsically similar to the PrefixSpan algorithm [29]. To partition the task, these algorithms use different strategies, such as bin-packing-based task distribution and bipartite graph partitioning-based task distribution. To attack the load balancing problem, the authors proposed a dynamic load-balancing strategy which allows an idle processor to join the busy ones. This strategy involves much more inter-processor communication than our selective sampling approach and the interruption of the busy processors may cause more overhead during mining.

All these three parallel formulations still retain the computation efficiency of the underlying serial algorithm to mine the complete set of sequential-patterns. However, mining the complete set of sequential-patterns is usually less efficient than mining the closed sequential-patterns, especially in mining long patterns and with low support threshold, when parallel processing is in greater demand.

Chapter 9

Conclusion

Data mining is an important application for parallel processing. In this thesis, we focus on frequent pattern mining, which is one of the fundamental tasks in data mining.

We propose a framework for parallel mining frequent patterns on a distributed memory system. This framework can be applied in parallel mining frequent itemsets, sequential patterns and closed-sequential patterns. The application domain of our framework may be extended to other interesting data mining issues, such as the discovery of correlation or mining subgraphs in a database.

We present a sampling technique, called *selective sampling*, to address the load balance problem of the parallelization. Selective sampling allows us to predict the relative time required to mine the projections and in this way enable us to identify large tasks, decompose them and evenly distributed them to the processors to achieve load balancing.

Based on the parallel framework and the selective sampling technique, we designed three algorithms Par-FP, Par-Span and Par CSP for parallel mining frequent itemsets, sequential patterns and closed-sequential patterns respectively.

We implemented the three parallel algorithms on a distributed memory system. A comprehensive performance study has been conducted in our experiments on both synthetic and real world datasets. The experimental results have shown that our parallel algorithms have achieved good speedups on various datasets and the speedups are scalable up to 64 processors on our 64-processor system.

References

[1] *http://fimi.cs.helsinki.fi/fimi03/.*

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

[3] Rakesh Agrawal, Hiekki Mannila, Ramakrishnan Srikant, Hannu Toivonen, and A. Inkeri Verkamo. Fast discovery of association rules. pages 307–328, 1996.

[4] Rakesh Agrawal and Ramakrishnan Srikant. Fast algorithms for mining association rules. In *Proc. 20th Int. Conf. Very Large Data Bases, VLDB*, pages 487–499. Morgan Kaufmann, 12–15 1994.

[5] Rakesh Agrawal and Ramakrishnan Srikant. Mining sequential patterns. In *Eleventh International Conference on Data Engineering*, pages 3–14, Taipei, Taiwan, 1995.

[6] Jay Ayres, Jason Flannick, Johannes Gehrke, and Tomi Yiu. Sequential pattern mining using a bitmap representation. In *KDD'02*, pages 429–435. ACM Press, 2002.

[7] Florian Beil, Martin Ester, and Xiaowei Xu. Frequent term-based text clustering. In *Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 436–442, 2002.

[8] Sergey Brin, Rajeev Motwani, and Craig Silverstein. Beyond market baskets: Generalizing association rules to correlations. In *SIGMOD 1997, Proceedings ACM SIGMOD International Conference on Management of Data, May 13-15, 1997, Tucson, Arizona, USA*, pages 265–276. ACM Press, 1997.

[9] R. Kruse C. Borgelt. Induction of association rules: Apriori implementation. In *Proceedings of the 15th Conference on Computational Statistics*, 2002.

[10] David W. Cheung, Jiawei Han, Vincent T. Ng, Ada W. Fu, and Yongjian Fu. A fast distributed algorithm for mining association rules. In *Proceedings of the fourth international conference on on Parallel and distributed information systems*, pages 31–43. IEEE Computer Society, 1996.

[11] Bart Goethals and Mohammed Javeed Zaki, editors. *FIMI '03, Frequent Itemset Mining Implementations, Proceedings of the ICDM 2003 Workshop on Frequent Itemset Mining Implementations, 19 December 2003, Melbourne, Florida, USA*, volume 90 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2003.

[12] Gösta Grahne and Jianfei Zhu. Efficiently using prefix-trees in mining frequent itemsets. In *FIMI*, 2003.

[13] Valerie Guralnik and George Karypis. Parallel tree-projection-based sequence mining algorithms. *Parallel Comput.*, 30(4):443–472, 2004.

[14] P. Ronkainen K. Hatonen H. Mannila H. Toivonen, M. Klemettinen. Pruning and grouping discovered association rules. ECML-95 Workshop on Statistics, Machine Learning, and Knowledge Discovery in Databases, April 1995.

[15] Eui-Hong Han, George Karypis, and Vipin Kumar. Scalable parallel data mining for association rules. In *SIGMOD '97: Proceedings of the 1997 ACM SIGMOD international conference on Management of data*, pages 277–288, 1997.

[16] J. Han and M. Kamber. *Data Mining: Concepts and Techniques.* Morgan Kaufmann Publishers Inc., 2001.

[17] Jiawei Han, Jian Pei, Behzad Mortazavi-Asl, Qiming Chen, Umeshwar Dayal, and Mei-Chun Hsu. Freespan: frequent pattern-projected sequential pattern mining. In *KDD'00*, pages 355–359. ACM Press, 2000.

[18] Jiawei Han, Jian Pei, and Yiwen Yin. Mining frequent patterns without candidate generation. In *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*, pages 1–12. ACM Press, 2000.

[19] IBM datset generator for associations and sequential patterns. *http://www.almaden.ibm.com/software/quest/Resources.*

[20] Asif Javed and Ashfaq A. Khokhar. Frequent pattern mining on message passing multiprocessor systems. *Distributed and Parallel Databases*, 16(3):321–334, 2004.

[21] Bing Liu, Wynne Hsu, and Yiming Ma. Integrating classification and association rule mining. In *Knowledge Discovery and Data Mining*, pages 80–86, 1998.

[22] Heikki Mannila, Hannu Toivonen, and A. Inkeri Verkamo. Efficient algorithms for discovering association rules. In Usama M. Fayyad and Ramasamy Uthurusamy, editors, *AAAI Workshop on Knowledge Discovery in Databases (KDD-94)*, pages 181–192, Seattle, Washington, 1994. AAAI Press.

[23] Heikki Mannila, Hannu Toivonen, and A. Inkeri Verkamo. Discovery of frequent episodes in event sequences. volume 1, pages 259–289, 1997.

[24] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard.*, 5 1994.

[25] S. Orlando, P. Palmerini, R. Perego, and F. Silvestri. Adaptive and resource-aware mining of frequent sets. In *ICDM '02: Proceedings of the 2002 IEEE International Conference on Data Mining (ICDM'02)*, page 338, Washington, DC, USA, 2002. IEEE Computer Society.

[26] Salvatore Orlando, Paolo Palmerini, and Raffaele Perego. Enhancing the apriori algorithm for frequent set counting. In *DaWaK '01: Proceedings of the Third International Conference on Data Warehousing and Knowledge Discovery*, pages 71–82, London, UK, 2001. Springer-Verlag.

[27] Jong Soo Park, Ming-Syan Chen, and Philip S. Yu. Efficient parallel data mining for association rules. In *Proceedings of the fourth international conference on Information and knowledge management*, pages 31–36. ACM Press, 1995.

[28] Nicolas Pasquier, Yves Bastide, Rafik Taouil, and Lotfi Lakhal. Discovering frequent closed itemsets for association rules. In *ICDT '99: Proceeding of the 7th International Conference on Database Theory*, pages 398–416, London, UK, 1999. Springer-Verlag.

[29] J. Pei, J. Han, B. Mortazavi-Asl, H. Pinto, Q. Chen, U. Dayal, and M.-C. Hsu. PrefixSpan mining sequential patterns efficiently by prefix projected pattern growth. In *ICDE'01*, pages 215–226.

[30] Jian Pei. *Pattern-growth Methods for Frequent Pattern Mining.* PhD thesis, Computering Science, Simon Fraser University, May 2001.

[31] Jian Pei, Jiawei Han, Behzad Mortazavi-Asl, Jianyong Wang, Helen Pinto, Qiming Chen, Umeshwar Dayal, and Mei-Chun Hsu. Mining sequential patterns by pattern-growth: The prefixspan approach. *IEEE Transactions on Knowledge and data enggineering*, 16(10), 2004.

[32] Iko Pramudiono and Masaru Kitsuregawa. Tree structure based parallel frequent pattern mining on pc cluster. In *DEXA*, pages 537–547, 2003.

[33] Assaf Schuster and Ran Wolff. Communication-efficient distributed mining of association rules. In *SIGMOD '01: Proceedings of the 2001 ACM SIGMOD international conference on Management of data*, pages 473–484. ACM Press, 2001.

[34] Takahiko Shintani and Masaru Kitsuregawa. Mining algorithms for sequential patterns in parallel: Hash based approach. In *PAKDD*, pages 283–294, 1998.

[35] R. Srikant. *Fast algorithms for mining association rules and sequential patterns.* PhD thesis, University of Wisconsin, Madison, 1996.

[36] Ramakrishnan Srikant and Rakesh Agrawal. Mining sequential patterns: Generalizations and performance improvements. In Peter M. G. Apers, Mokrane Bouzeghoub, and Georges Gardarin, editors, *Proc. 5th Int. Conf. Extending Database Technology, EDBT*, volume 1057, pages 3–17. Springer-Verlag, 25–29 1996.

[37] J. Wang and J. Han. BIDE efficient mining of frequent closed sequences. In *ICDE'04*, pages 79–91.

[38] R. Wol, A. Schuster, and D. Trock. A high-performance distributed algorithm for mining association rules. In *ICDM*, 2003.

[39] X. Yan and J. Han. gspan: Graph-based substructure pattern mining, 2002.

[40] X. Yan, J. Han, and R. Afshar. Clospan: Mining closed sequential patterns in large datasets. In *SDM'03*, 2003.

[41] Osmar R. Zaiane, Mohammad El-Hajj, and Paul Lu. Fast parallel association rule mining without candidacy generation. In *ICDM*, pages 665–668, 2001.

[42] Mohammed J. Zaki. Parallel sequence mining on shared-memory machines. *Journal of Parallel and Distributed Computing*, 61(3):401–426, 2001.

[43] Mohammed J. Zaki. Spade: An efficient algorithm for mining frequent sequences. *Machine Learning*, 42(1-2):31–60, 2001.

[44] Mohammed J. Zaki, Srinivasan Parthasarathy, Mitsunori Ogihara, and Wei Li. Parallel algorithms for discovery of association rules. *Data Min. Knowl. Discov.*, 1(4):343–373, 1997.

[45] Mohammed Javeed Zaki, Mitsunori Ogihara, Srinivasan Parthasarathy, and Wei Li. Parallel data mining for association rules on shared-memory multiprocessors. Technical Report TR618, 1996.

Vita

Shengnan Cong was born in Xi'an, China. She received her Bachelor of Engineering degree in Computer Science from Tsinghua University, Beijing, China in July 2000. In August 2000, she was admitted to the Department of Computer Science in the University of Illinois at Urbana-Champaign. She received her Master of Science degree in Computer Science in October 2003. For her Master thesis, Shengnan participated in the design and implementation of a hybrid programming model for the distributed memory system. In the summer of 2003, Shengnan worked as an intern at KAI software lab at Intel Inc., working on a performance model of Intel cluster OpenMP libruary. Shengnan's general research interests include parallel computing, data mining, compilers and distributed systems. After graduation, Shengnan will join Intel Corporation.