

# A Sampling-based Framework For Parallel Data Mining

Shengnan Cong<sup>†</sup>   Jiawei Han<sup>†</sup>   Jay Hoeflinger<sup>§</sup>   David Padua<sup>†</sup>

<sup>†</sup> Department of Computer Science  
University of Illinois  
Urbana, IL 61801, USA

{cong, hanj, padua}@cs.uiuc.edu

<sup>§</sup> KAI Software Lab  
Intel Americas, Inc.  
Champaign, IL 61820, USA

jay.p.hoeflinger@intel.com

## ABSTRACT

The goal of data mining algorithm is to discover useful information embedded in large databases. *Frequent itemset* mining and *sequential pattern* mining are two important data mining problems with broad applications. Perhaps the most efficient way to solve these problems sequentially is to apply a pattern-growth algorithm, which is a divide-and-conquer algorithm [9, 10]. In this paper, we present a framework for parallel mining frequent itemsets and sequential patterns based on the divide-and-conquer strategy of pattern growth. Then, we discuss the load balancing problem and introduce a sampling technique, called *selective sampling*, to address this problem. We implemented parallel versions of both frequent itemsets and sequential pattern mining algorithms following our framework. The experimental results show that our parallel algorithms usually achieve excellent speedups.

## Categories and Subject Descriptors

D.1 [Programming Techniques]: Concurrent programming—*parallel programming*; H.2.8 [Database Management]: Database applications—*data mining*

## General Terms

Algorithms, Design, Experimentation, Performance

## Keywords

data mining, load balancing, parallel algorithms, sampling

## 1. INTRODUCTION

Data mining is a process to extract patterns from a large collections of data [13]. Due to the wide availability of large datasets and the desire of turning such datasets into useful information and knowledge, data mining has attracted a great deal of attention in recent years.

One important problem in data mining is the discovery of frequent itemsets in a transactional database [3].

The frequent itemset mining problem can be stated as follows: Let  $I = \{i_1, i_2, \dots, i_m\}$  be a set of items, and  $DB = \{T_1, T_2, \dots, T_n\}$  be a database where  $T_i (i \in [1..n])$ , called **transactions**, are subsets of items in  $I$  (i.e.  $T_i \in 2^I$ ). The **support** (or occurrence frequency) of an itemset  $A (A \in 2^I)$  is the fraction of transactions containing  $A$  in  $DB$ .  $A$  is a **frequent itemset** if  $A$ 's support is no less than a predefined support threshold. The **frequent itemsets mining problem** is to find *all frequent itemsets* in a database. Frequent itemset mining problem has a wide range of applications including correlation [6], cluster analysis [5] and associative classification [18].

Sequential pattern mining is another important problem in data mining [4]. Its objective is to discover frequent subsequences in a sequence database. Let  $I = \{i_1, i_2, \dots, i_n\}$  be a set of items. A **sequence**  $s$  is a tuple, denoted as  $\langle T_1, T_2, \dots, T_l \rangle$ , where the  $T_j (1 \leq j \leq l)$ , are called **events** or itemsets. Each event is a set denoted as  $(x_1, x_2, \dots, x_m)$  where  $x_k (1 \leq k \leq m)$  is an item. A **sequence database**  $S$  is a set of sequences. A sequence  $\alpha = \langle a_1, a_2, \dots, a_n \rangle$  is called a **subsequence** of another sequence  $\beta = \langle b_1, b_2, \dots, b_m \rangle$ , if there exist integers  $1 \leq j_1 \leq j_2 \leq \dots \leq j_n \leq m$  such that  $a_1 \subseteq b_{j_1}$ ,  $a_2 \subseteq b_{j_2}, \dots, a_n \subseteq b_{j_n}$ . If  $\alpha$  is a subsequence of  $\beta$ , we say that  $\beta$  contains  $\alpha$ . The **support** of a sequence  $\alpha$  in a sequence database  $S$  is the number of sequences in the database containing  $\alpha$  as a subsequence. The **sequential pattern mining problem** is to find *all* the subsequences whose support values are no less than a predefined support threshold. Sequential pattern mining can be applied to many real-world applications, such as the analysis of customer purchase patterns or web access patterns [8], the analysis of natural disasters or alarm sequences [27], and the analysis of disease treatments or DNA sequences [25].

The best algorithms for both frequent itemset mining problem and sequential pattern mining problem are based on pattern-growth [14], a divide-and-conquer algorithm that projects and partitions databases based on the currently identified frequent patterns and grow such patterns to longer ones using the projected databases.

In this paper, we propose a framework for parallel mining frequent itemset and sequential patterns. The partition into tasks is based on the divide-and-conquer strategy of the pattern-growth algorithm so that the inter-processor communication is minimized.

One problem we must handle is the possibility of an unbalanced workload because the size of the projected databases may vary greatly. We have observed that the mining time of the largest task may be tens or even hundreds of times

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

PPoPP'05, June 15–17, 2005, Chicago, Illinois, USA.  
Copyright 2005 ACM 1-59593-080-9/05/0006 ...\$5.00.

longer than the average mining time. The imbalance of the workload can significantly degrade performance.

To address the load balance problem, we developed *selective sampling*, a technique that estimates the relative mining time of projections and, in this way, enables the identification of large tasks and their decomposition and distribution across processors to achieve load balancing.

We implemented parallel algorithms for both frequent itemset mining problem and sequential pattern mining problem following the proposed framework. The implementation used MPI [19] and was evaluated on a Linux cluster with 64 processors. The experimental results show that our parallel algorithms achieve good speedups on various datasets.

To summarize, our key contributions are:

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

We use the datasets listed in Figure 1 to evaluate our implementations. Transactional datasets(Figure 1(a)) were used to evaluate frequent itemset mining. Sequence datasets(Figure 1(b)) were used to evaluate sequential pattern mining.

Dataset	#Transactions	#Items	Max Trans. Length
mushroom	8,124	23	23
connect	57,557	43	43
pumsb	49,046	7,116	74
pumsb_star	49,046	7,116	63
T40I10D100K	100,000	999	77
T50I5D500K	500,000	5,000	94

(a) Transactional datasets

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

(b) Sequence datasets

Figure 1: Dataset characteristics

The remainder of the paper is organized as follows. Section 2 describes the pattern-growth methods for frequent itemset mining and sequential pattern mining respectively. The framework for parallelization is introduced in Section 3. In Section 4, we attack the load balance problem and develop the sampling technique to achieve load balance for the framework. The experimental and performance results are presented in Section 5. Section 6 outlines the directions of future research and the related work is presented in Section 7. Section 8 summarizes our paper.

## 2. THE PATTERN-GROWTH APPROACH

In this section, we introduce efficient sequential algorithms for frequent itemset mining and sequential pattern mining respectively. These two algorithms, although attacking different problems, share the same pattern-growth strategy.

### 2.1 Frequent Itemset Mining

FP-growth algorithm[15] uses a tree structure, called FP-tree, to store the information about frequent itemsets. Every branch (from the root) of the FP-tree represents a transaction pattern in the database. 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. 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.

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

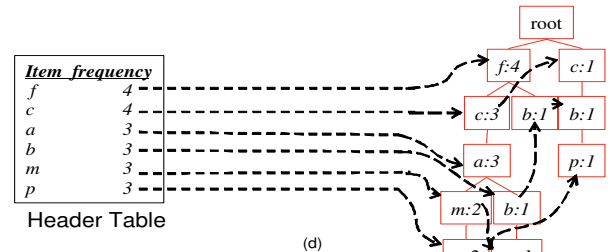
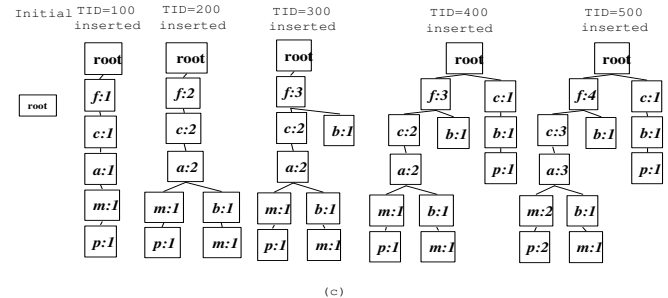


Figure 2: Example for FP-tree construction

Here is a simple example of the FP-tree representation. Let the transaction database  $DB$  be as shown in Figure 2(a) and the minimum support threshold be 3. First, a scan of  $DB$  derives a list of frequent single items with their frequency,  $((f : 4), (c : 4), (a : 3), (b : 3), (m : 3), (p : 3))$ . These items are ordered in descending order of their frequency. This ordering is important since the items will appear in the branches of the FP-tree following this order. Then, a second  $DB$  scan constructs the FP-tree. The root of a tree is first created. Then the frequent items in each transaction are first sorted in descending order of frequency as in Figure 2(b) and then inserted as a branch into the tree from the root with shared prefix path coalesced and counter of

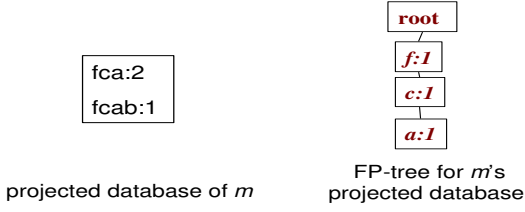


Figure 3: Example for tree projection

the node updated (Figure 2(c)). Whenever there is a new node created, the node is linked to the linked list of the item. The complete FP-tree for the example database is as shown in Figure 2(d).

After the FP-tree is built, FP-growth mines the tree in a divide-and-conquer way. For each item  $i$  in the header table of the FP-tree, examine  $i$ 's *projected database*. The projected database  $P_i$  of  $i$  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 of  $i$  in the tree are visited. Then tracing the branches from  $i$ 's nodes back to the root will form  $i$ 's projected database. The projected database is then represented in FP-tree form and mined recursively. Figure 3 illustrates the projected database and the FP-tree for the projection of item  $m$  in the example in Figure 2. Item  $b$  does not appear in the FP-tree for  $m$ 's projection because it is not *frequent* in  $m$ 's projected database. Suppose  $T$  is the FP-tree for the whole database and  $P(i, T)$  represents the projected database of  $i$  ( $i \in \text{Header}(T)$ ).  $\{X_1, X_2, \dots, X_n\}$  is a set of itemsets and  $i$  is an item. We define an operator  $\oplus$  as:

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

Generally, the frequent itemsets are achieved by the concatenation of item  $i$  with the new patterns generated from mining  $P(i, T)$ . The mining of the FP-tree,  $T$ , of a dataset can be defined as function  $F()$  below:

```

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

```

## 2.2 Sequential Pattern Mining

Following a philosophy similar to that of FP-growth, an algorithm, called *PrefixSpan* [21], was proposed. It has proved to be one of the most efficient algorithms for sequential pattern mining.

For sequential pattern mining, we assume that the items are totally ordered<sup>1</sup>, so that given an itemset  $T = (i_1, i_2, \dots, i_n)$ , we assume that  $i_1 < i_2 < \dots < i_n$ .

Given a sequence  $\alpha = \langle a_1, a_2, \dots, a_n \rangle$ , a sequence  $\beta = \langle b_1, b_2, \dots, b_m \rangle$  ( $m \leq n$ ) is called a **prefix** of  $\alpha$  if and only if  $a_i = b_i$  ( $i \leq m - 1, b_m \subseteq a_m$ ), and either  $a_m - b_m = \emptyset$  or all the frequent items in  $(a_m - b_m)$  are larger than those in  $b_m$  according to the total ordering. The sequence  $\gamma = \langle a_m - b_m, a_{m+1}, \dots, a_n \rangle$  is called the **suffix** of  $\alpha$  relative to prefix  $\beta$ .

<sup>1</sup>We use alphabetical order in our following examples.

Sequence_id	Sequence
10	$\langle (a)(abc)(ac)(d)(cf) \rangle$
20	$\langle (ad)(c)(bc)(ae) \rangle$
30	$\langle (ef)(ab)(df)(c)(b) \rangle$
40	$\langle (e)(g)(af)(c)(b)(c) \rangle$

(a)

Prefix	Projected database (suffixes)
$\langle (a) \rangle$	$\langle ()(abc)(ac)(d)(cf) \rangle, \langle (d)(c)(bc)(ae) \rangle, \langle (b)(df)(c)(b) \rangle, \langle (f)(c)(b)(c) \rangle$
$\langle (b) \rangle$	$\langle (c)(ac)(d)(cf) \rangle, \langle (c)(ae) \rangle, \langle (d)(f)(c)(b) \rangle, \langle ()(c) \rangle$
$\langle (c) \rangle$	$\langle ()(ac)(d)(cf) \rangle, \langle ()(bc)(ae) \rangle, \langle ()(b) \rangle, \langle ()(b)(c) \rangle$
$\langle (d) \rangle$	$\langle ()(cf) \rangle, \langle ()(c)(bc)(ae) \rangle, \langle (f)(c)(b) \rangle$
$\langle (e) \rangle$	$\langle (f)(ab)(df)(c)(b) \rangle, \langle (af)(c)(b)(c) \rangle$
$\langle (f) \rangle$	$\langle ()(ab)(df)(c)(b) \rangle, \langle ()(c)(b)(c) \rangle$

(b)

Figure 4: Example of Projection

Given a sequence  $\alpha$ , we define projection  $P(\alpha, DB)$  of a database  $DB$  as the suffixes of sequences in  $DB$  of which  $\alpha$  is the prefix.

Here is an example. Figure 4(a) is a sequence dataset and the support threshold is set to 2.  $\langle (a) \rangle:4$ ,  $\langle (b) \rangle:4$ ,  $\langle (c) \rangle:4$ ,  $\langle (d) \rangle:3$ ,  $\langle (e) \rangle:3$  and  $\langle (f) \rangle:3$  are frequent length-1 sequences in  $DB$  with their frequencies. Figure 4(b) illustrates the projections of these frequent-1 sequences of  $DB$ .

Given a set of suffixes  $T = \{\gamma_1, \gamma_2, \dots, \gamma_n\}$ , we can extend the prefix  $\alpha$  in two ways: assemble an item that occurs frequently in  $\gamma_i$  ( $1 \leq i \leq n$ ) to the last itemset of  $\alpha$ , or append an item, that occurs frequent in itemsets other than the first one in  $\gamma_i$  ( $1 \leq i \leq n$ ), to  $\alpha$  as a new itemset.

Given a sequence  $\alpha = \langle a_1, a_2, \dots, a_n \rangle$ , we define:

$$\alpha \bowtie (b) = \langle a_1, a_2, \dots, a_n \cup (b) \rangle$$

$$\alpha || (b) = \langle a_1, a_2, \dots, a_n, (b) \rangle$$

Using these definitions, we can now define the PrefixSpan algorithm to compute the set of all sequential patterns of a sequence database  $DB$  as calling  $F(\langle \rangle, DB)$ , where  $F$  is defined as follows.

```

function  $F(\alpha, PDB)$  /*PDB is a projection of prefix  $\alpha^*$ */
begin
  if ( $PDB$  is a set of empty sequences) return an empty set;
  else {
    Search in PDB for the set of items  $b$  that occurs frequently in an itemset of the sequences in  $PDB$  such that  $\alpha \bowtie (b)$  is a frequent prefix of  $PDB$ ;
    Search in PDB for set of items  $c$  that occurs frequently in itemsets other than the first one in  $PDB$  such that  $\alpha || (c)$  is a frequent prefix of  $PDB$ ;
    return  $\bigcup_b ((\alpha \bowtie (b)) \cup F(\alpha \bowtie (b), P(\alpha \bowtie (b), PDB))) \cup \bigcup_c ((\alpha || (c)) \cup F(\alpha || (c), P(\alpha || (c), PDB)))$ 
  }
end

```

## 3. PARALLELIZING FRAMEWORK

Although FP-growth and PrefixSpan aim at different problems, they both solve the problems 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 of projected FP-trees corresponding to the frequent items in the database. Similarly, in sequential pattern mining, the whole database mining is divided into the mining of projected databases of the frequent items. Basically, both FP-growth and PrefixSpan consist of three steps:

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

In both cases, the projected databases are independent of 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, the conditional database of  $p$  may only contain  $f$ ,  $c$ ,  $a$ ,  $b$  or  $m$ , while the conditional database of  $m$  may contain  $f$ ,  $c$ ,  $a$  or  $b$ , but not  $p$ . In PrefixSpan, 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.

Such divide-and-conquer property is convenient for task partitioning in parallel processing. Therefore, we propose the framework for parallel mining frequent itemsets and sequential patterns as follows:

1. Count the occurrence of items 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 and assign each subset to a different processor. Each processor computes the projection of the databases for the assigned items.
3. Each processor mines its projected database asynchronously in a pattern-growth manner without inter-processor communication.

Because the projected databases are independent, there is no inter-processor communication involved after the task scheduling. Each processor just mines its own projected database using sequential FP-growth or PrefixSpan. However, although the communication has been minimized, another issue has to be considered, which is load balancing. In fact, the inherent task partition of pattern-growth method may result in extremely imbalanced workload among the processors because 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 5 shows the mining time distribution of the projected databases for two databases. Figure 5(a) is the transactional dataset *pumsb* for frequent itemset mining. Figure 5(b) is for the sequence dataset *C10N0.1T8S8I8* for sequential pattern mining. Figure 6 shows the average and maximum mining time of the projected databases for all the datasets we tested.

This imbalanced workload greatly restricts the scalability of parallelism. To address this problem, we have to identify which projected databases take much longer time than the others and split them into smaller ones.

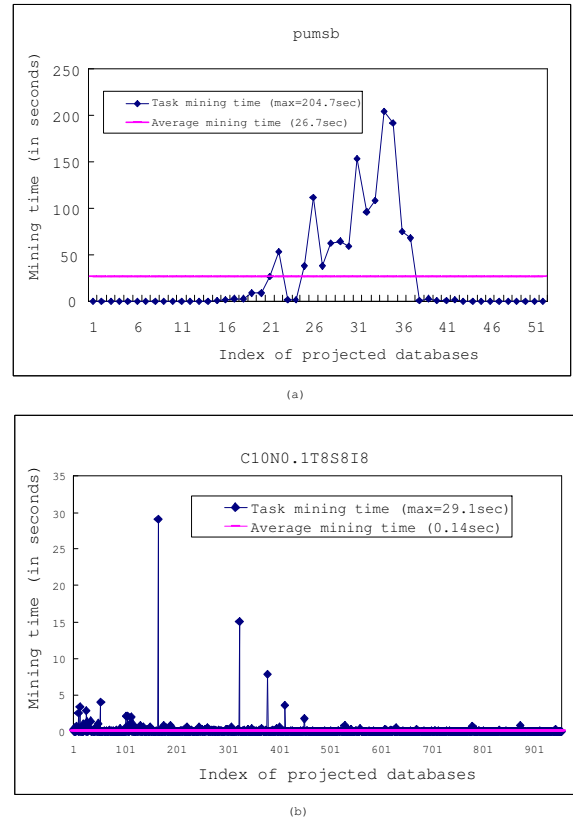


Figure 5: Example of load imbalance

## 4. SAMPLING TECHNIQUE

In order to identify which projected databases take longer mining time before task scheduling, we need an estimation of the relative mining time of the projected databases before the actual mining. Since different datasets have different characteristics, it is difficult, if not impossible, to find an accurate estimation formula for all datasets. An alternative to using formulas is to use run-time sampling. 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 for the whole database, according to the ratio of the mining time of the projected databases for the sample. 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. Accuracy and overhead are two important characteristics which we use to evaluate the sampling techniques.

### 4.1 Random Sampling

Random sampling is a fairly straightforward heuristic. We did some experiments on random sampling, which randomly collects a fraction of transactions or sequences in the dataset as a sample and mines it with the support threshold reduced to the same fraction. However, comparing the mining time of each item's projected database in the sample with the corresponding mining time in the original dataset, we found that random sampling does not provide a good estimation. Figure 7 shows one of our experimental results on random sampling with the dataset *pumsb* for frequent itemset min-

Dataset	Mushroom	Connect	Pumsb	Pumsb_star
Average (sec)	0.17	2.37	26.7	21.9
Maximum (sec)	16.2	13.5	204.6	899.6

Dataset	T40I10D100K	T50I5D500K
Average (sec)	0.48	0.56
Maximum (sec)	44.2	58.6

(a) Transactional datasets

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) Sequence datasets

Figure 6: Mining time distributions

ing. The curves show the times for mining the projected database of each frequent item both in the whole dataset and in a 1% random sample of the dataset. The curve for the whole dataset follows the left vertical scale while the one for the sample follows the right vertical scale. Clearly, there is no correlation between the mining time of a random sample and the time for the whole dataset. Therefore, we cannot tell which items are large through random sampling, and, although mining the 1% random sample costs less than 1% of the original sequential mining time, it is not a feasible solution for relative mining time estimation. 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 is too large. Things are similar with sequential pattern mining. Figure 8 is the experimental results for random sampling with the dataset *C10N0.1T8S8I8* for sequential pattern mining.

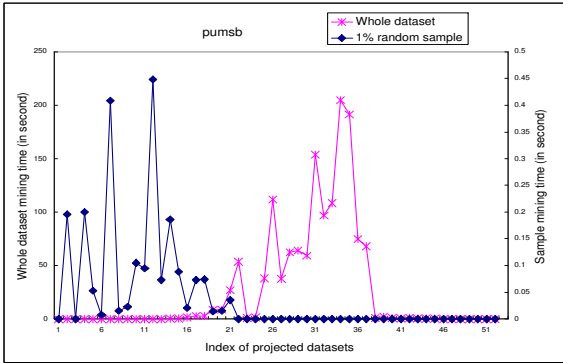


Figure 7: Random sampling for frequent itemset mining

## 4.2 Selective Sampling

### 4.2.1 How selective sampling works

We devised a sampling technique, called *selective sampling*, which has proved to be quite accurate in identifying the large items. Instead of randomly selecting a fraction of transactions or sequences from the dataset, selective sampling analyzes the information of every transaction or sequences in the dataset.

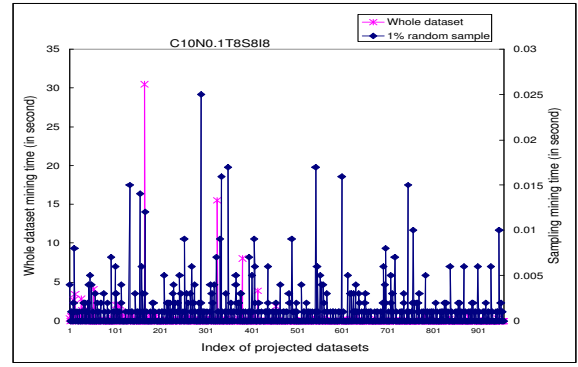


Figure 8: Random sampling for sequential pattern mining

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, 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  $\langle (a:4), (b:4), (c:4), (d:3), (e:3), (f:3) \text{ and } (g:1) \rangle$  be the items in the database. Let the support threshold be 4 and the average length of the sequences in the dataset be 4. Suppose  $t$  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 sequence  $\langle (a)(abc)(ac)(d)(cf)(d)(b) \rangle$ , its selective sample is  $\langle (a)(abc)(a) \rangle$ . The suffix  $\langle (c)(d)(cf)(d)(b) \rangle$  is discarded because it contains the last  $l$  frequent items of the sequence ( $d$  and  $f$  do not count because they are infrequent items.).

### 4.2.2 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 two curves match quite well in all datasets. For instance, in the case of frequent itemset mining, Figure 9 shows our experimental results of selective sampling for the same dataset as Figure 7 with  $t$  set to 20%. The mining time of items 1—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 little, this does not affect balancing the load of each processor and it also proves the feasibility of our selective sampling strategy. Figure 10 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. As we can see, the two curves are

quite close 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.

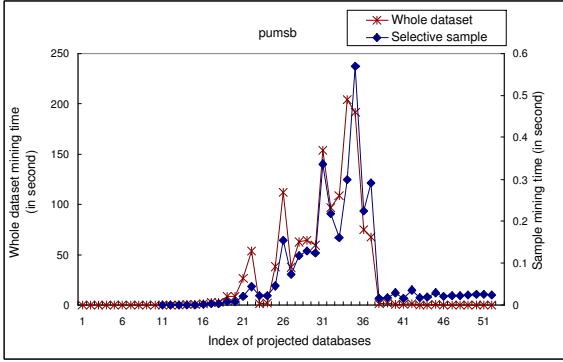


Figure 9: Accuracy of selective sampling for frequent itemset mining

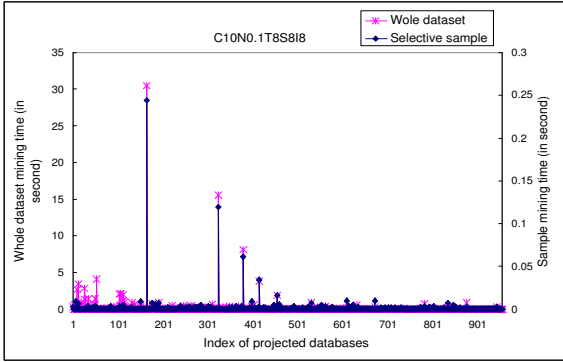


Figure 10: Accuracy of selective sampling for sequential pattern mining

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

According to our experiments, for frequent itemset mining, 20% is a reasonable value for  $t$  with the datasets we considered (75% for sequence pattern mining). For example, with  $t$  equals to 20%, the overhead of selective sampling in Figure 9 is only 0.71% of the sequential mining time of the whole dataset and the overhead of Figure 10, with  $t$  set to 75%, is only 0.52% of the sequential mining time while they both provide accurate information for the relative mining time estimation. Figure 11. presents the overhead values with  $t$  being 20% for frequent itemset mining and 75% for sequential pattern mining.

#### 4.2.3 Why selective sampling works

Let us now discuss the reason why selective sampling works.

We first discuss the reason why it works for frequent itemset mining. In the FP-growth algorithm, the most frequent items are close to the root of the FP-tree because the items in a transaction are inserted into the FP-tree in descending order of their frequency. The most frequent items appear in

Dataset	Mushroom	Connect	Pumsb	Pumsb_star	T40110D100K	T5015D500K
Overhead	0.71%	1.80%	0.71%	2.90%	2.05%	0.28%

(a) Overhead of transactional datasets

Dataset	C10N0.1T8S8I8	C50N10T8S20I2.5	C100N5T2.5S10I1.25
Overhead	0.51%	0.71%	0.60%

Dataset	C200T2.5S10I1.25	C100N20T2.5S10I1.25
Overhead	0.87%	0.53%

(b) Overhead of sequence datasets

Figure 11: Overhead of selective sampling

most of the branches from the root so that removing them will make the FP-tree shallower and the mining time of the FP-tree is reduced.

On the other hand, every subset of a frequent itemset must be frequent[3]. According to our experiments, the mining time of a projection is proportional to the number of frequent itemsets in the mining output of this projection. Thus, if there is a length- $L$  frequent itemset in the output of mining  $i$ 's projection, there must be additional  $2^L$  frequent itemsets in this output (every subset of the length- $L$  frequent itemset is a frequent itemset.). Therefore, we can expect that if there are no long frequent itemsets in the output of mining  $i$ 's projection, the total number of frequent itemsets in the output of mining  $i$ 's projection would not be large. And consequently the mining time of  $i$ 's projection would not be long. In FP-growth, only the prefix paths of the FP-tree are examined for the nodes of an item. The most frequent items have short prefix paths and therefore do not have long frequent itemsets in their mining outputs. So the mining time of their projections is short and removing them will not hinder the identification of *large* projections.

The reason for sequential pattern mining is similar to that of frequent itemset mining. When doing projection for a sequential pattern, only the suffixes (with the pattern as prefix) will be collected. Comparable to the items near the root of the FP-tree, the items in the tail of a sequence are in the projections of all items appearing before them in the sequence. So by removing the items at the tails of the sequences, the sequences in the projections become shorter and therefore, the mining time of the dataset is reduced. At the same time, the items at the tail of the sequences have short suffixes so that no long pattern exists in the output of mining their projections. Thus, we can safely remove them while the *large* items still remain in the sample.

#### 4.2.4 Applying selective sampling to the framework

We perform selective sampling to identify the large items and divide their projected databases into smaller sub-databases. The partition proceeds in a pattern-growth way.

For example, in the frequent itemset mining problem, let  $\langle (f : 4), (c : 4), (a : 3), (b : 3), (m : 3), (p : 3) \rangle$  be the frequent single items sorted in descending order by their frequency. Suppose  $m$  is a large item, the mining task of item  $m$  is divided into the mining subtasks of  $(mb)$ ,  $(ma)$ ,  $(mc)$  and  $(mf)$  respectively. Generally, the projected database of  $i$  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).

These pairs are called frequent-2 itemsets. Similarly, in sequential pattern mining, suppose the frequent items are  $\langle\langle a \rangle : 4\rangle, \langle\langle b \rangle : 4\rangle$  and  $\langle\langle c \rangle : 3\rangle$  and  $b, c$  are frequent items in  $a$ 's projected database, the projected database of  $\langle a \rangle$  can be divided into a series of sub-databases with prefix  $\langle\langle a \rangle(a)\rangle, \langle\langle a \rangle(b)\rangle, \langle\langle a \rangle(c)\rangle, \langle\langle ab \rangle\rangle$  and  $\langle\langle ac \rangle\rangle$ . Generally, the projected database of  $i$  is split into the sub-databases of inter or intra pairs of  $i$  with all the frequent items in  $i$ 's projected database.

By breaking the large tasks into smaller ones, the size of the tasks tends to be closer and more tasks are available for scheduling and, as a result, load balancing improves.

We use frequent itemset mining problem to illustrate how selective sampling is applied in our framework. Selective sampling for sequential pattern mining works in a similar way.

We assume that there are total  $M$  frequent items in the whole dataset. And the number of processors in the system is  $P$ . Selective sampling performs in the following four steps.

1. Sample tree construction

One of the processors, say processor 0, does selective sampling by scanning the whole dataset, discarding the top 20% most frequent items and the non-frequent items in each transaction. The items left in the selective sample are sorted in descending order of frequency and inserted into an FP-tree structure, named  $T$ .

2. Sample tree mining

Processor 0 mines  $T$  using the FP-growth algorithm. However, instead of outputting the frequent itemsets found, processor 0 records the mining time of each frequent items in the header table of  $T$ . Assume there are  $S$  items in  $T$ 's header table. The mining time of item  $i$  for the sample is  $L_i (i \in [1..S])$ .

3. Task partition

If there is an item  $i (i \in [1..S])$  such that  $L_i > \frac{1}{4} \frac{\sum L_i}{P}$ , task  $i$  will be partitioned. That is, if the mining time of  $i$  is larger than  $\frac{1}{4}$  of the average mining time of each processor, which is  $\frac{\sum L_i}{P}$  if the items are distributed to all  $P$  processors,  $i$  will be treated as a *large* item.

4. Task scheduling

Task scheduling consists of two parts. Part I is to schedule the projections corresponding to the frequent items which appear in the sample and Part II is to schedule the projection of items which are the top 20% most frequent items found in Step 1 and have been discarded during the sampling. For Part I, since we have identified the *large* projections with selective sampling, these large projections are further partitioned into subtasks and then scheduled evenly in a round-robin way. For the *small* projections, we have recorded their sample mining time so we schedule them using a bin-packing heuristic. For Part II, since the projected databases take little time to be mined, they are scheduled evenly with round-robin.

For performance enhancement, we apply one optimization before applying selective sampling. If  $\frac{M}{P} > 10$  where  $M$  is the number of frequent items, we will use round-robin strategy instead of selective sampling to partition the frequent items. This is because when  $\frac{M}{P} > 10$ , each processor has more than 10 tasks on average, and thus the number of tasks

is large enough so that it tends to balance the load. Therefore, it becomes unnecessary to do the sampling-based task partition. This optimization can improve the performance when  $P$  is small or  $M$  is very large.

## 5. EXPERIMENTAL RESULTS

### 5.1 Experimental Setups

All of our experiments were performed on a Linux cluster consisting of 64 nodes. Each node has a 1GHz Pentium III processor and 1GB main memory. We used MPICH-GM 1.2.4.8a in the implementation of our parallel algorithms. MPICH-GM is a portable implementation of MPI that runs over Myrinet. The operating system is Redhat Linux 7.2 and we used the GNU g++ 2.96 compiler.

We used eleven databases (Figure 1) in the performance study, six transactional datasets for frequent itemset mining and five sequence datasets for sequential pattern mining. Four of the transactional datasets are realistic datasets downloaded from the FIMI repository[1]. The rests are synthetic datasets generated by the IBM dataset generator[16].

### 5.2 Speedups of Parallel Algorithms

We implemented a parallel frequent itemset mining algorithm and a parallel sequential pattern mining algorithm based on FP-growth and PrefixSpan respectively. Our parallel implementation follows the framework proposed in Section 3 and applies the selective sampling technique discussed in Section 4. We named our parallel frequent itemset mining algorithm **Par-FP**, and the parallel sequential pattern mining algorithm **Par-ASP**. We measured speedups to examine the scalability of our parallel algorithms. For each dataset we executed our algorithms on 2, 4, 8, 16, 32 and 64 processors and compared the mining time to that of the sequential FP-growth algorithm for frequent pattern mining and with PrefixSpan algorithm for sequential pattern mining. The speedups of Par-FP and Par-ASP are shown in Figure 12 and Figure 13 respectively.

From the graphs, we can see that both Par-FP and Par-ASP have obtained good speedups on most of the datasets we tested. The speedups are scalable up to 64 processors on our 64-processor system. For frequent itemset mining, Par-FP shows better speedups on synthetic datasets than real ones. This is because synthetic datasets have more frequent items and, after the large projected databases are partitioned, the sub-databases derived are of similar size. However, in real datasets, the number of frequent items is small and even when the large tasks are partitioned into smaller subtasks, the size of the subtasks may still be larger, or even much larger as in *pumb\_star*, than those small tasks, which is also the reason for the flattening speedup of *C10N0.1T8S8I8* for Par-ASP.

In order to illustrate the effectiveness of the selective sampling technique, we compared the performance of Par-FP and Par-ASP to the straight parallel implementations with selective sampling turned off on 64 processors. Figure 14 and Figure 15 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 efficiency of the parallel implementation in all the cases we tested. For most of the cases, the speedups can be enhanced by more than 50%. Selective sampling is able to greatly improve the scalability of the parallelization.

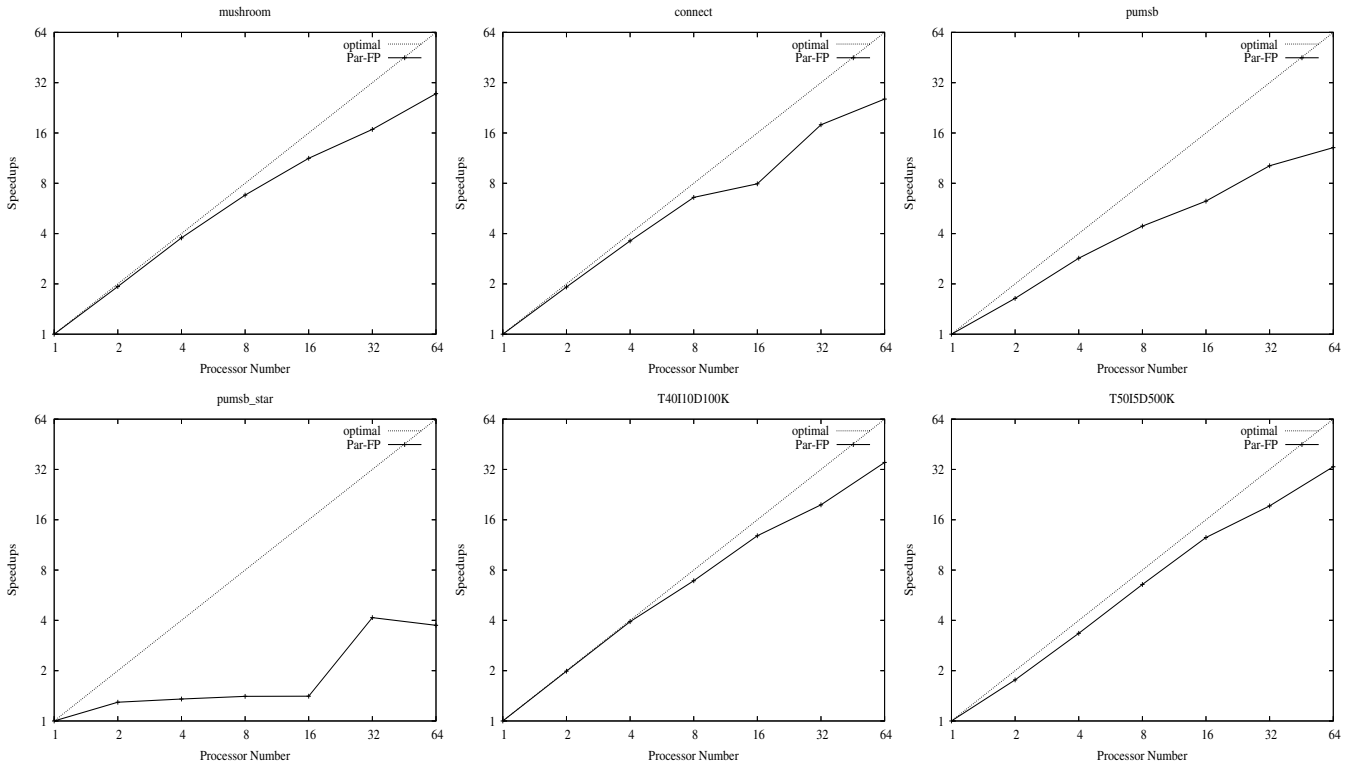


Figure 12: Speedups of Par-FP algorithms

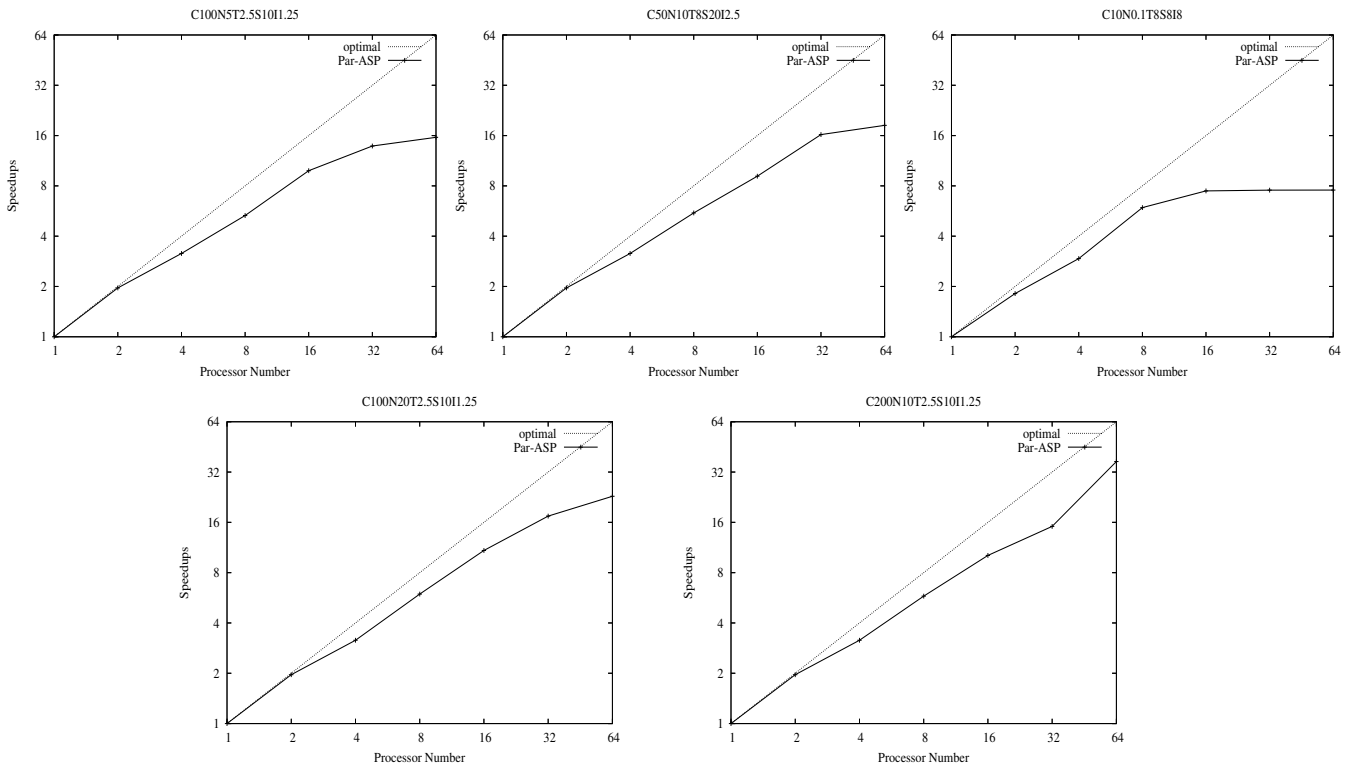


Figure 13: Speedups of Par-ASP algorithms

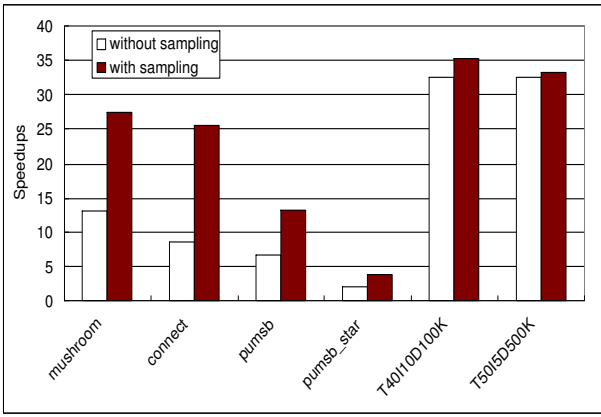


Figure 14: Effectiveness of selective-sampling on Par-FP

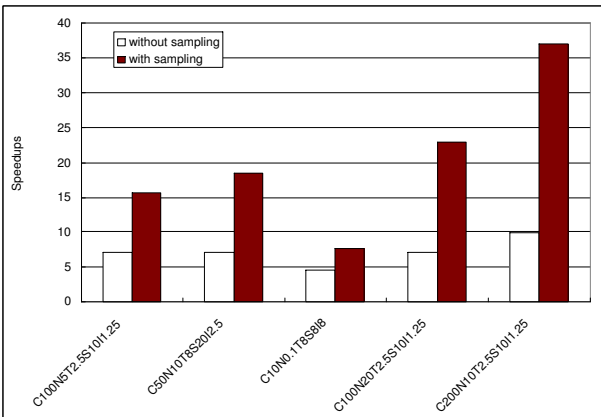


Figure 15: Effectiveness of selective-sampling on Par-ASP

## 6. FUTURE WORK

There are several optimizations which can further improve the performance and scalability of the parallel algorithms built with our framework.

First, as is shown in Figure 12 and Figure 13, the speedups of datasets *pumsb\_star* and *C10N0.1785818* are not satisfactory. This is because of load imbalance. The mining time of some large items is so large that the subtasks derived from them are still much bigger than the small tasks. To solve this problem, the large subtasks need to be further divided into smaller ones, which is to conduct multi-level task partition. Then we need to estimate which subtasks need to be further divided and how many levels are necessary. The selective sampling technique can be extended to fulfill this duty. In addition to just recording the mining time corresponding to the frequent-1 items during the sample mining, we may also record the mining time of their subtasks. From our experiments, the mining time curves of subtasks by selective sampling still match quite well with the corresponding curves of the whole dataset. So we can recursively use the curves obtained from sampling to identify the large subtasks.

Second, although the average overhead of selective sampling is only 1.1% of the sequential mining time on average for both Par-FP and Par-Span, such overhead will become

more critical when the number of processors grows large. According to Amdahl's Law, the best possible speedup with 1.1% serial component on 64 nodes is restricted to 37.8. Therefore, in order to get better speedups, we need to 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 may be reduced.

Furthermore, 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.

## 7. RELATED WORK

The parallel frequent itemset mining algorithms published are mostly based on candidate generation-and-test approach [2, 12, 20, 30, 7, 23]. Despite the various heuristics used to improve the efficiency, they still suffer from costly database scans, communications and synchronizations. This greatly limit the efficiency of these parallel algorithms<sup>2</sup>.

As far as we know, there are three parallel frequent itemset mining algorithms based on pattern-growth methods [28, 17, 22]. The algorithm presented in [28] 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 [17] the FP-growth algorithm is parallelized for a distributed memory system and reports the speedups only of 8 and fewer processors. Both [28] and [17] did not address the load balancing problem. In [22], 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 [26], 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.

We are aware of three papers on the parallelization of sequential pattern mining [24, 29, 11]. The method in [24] 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 [11], the author presents a parallel tree-projection-based algorithm and proposed both static and dynamic strategies for task assignment. The tree-projection based algorithm is also based on candidate generation-and-test approach, which is often less efficient than the pattern-growth approach used in our Par-ASP [21]. The static task assignment strategy presented is based on bin-packing and bipartite graph partitioning. Although it works well with some datasets, it may lead to unbalanced

<sup>2</sup>In [23], the number of bytes transmitted was claimed to be reduced, but no data of performance was reported

work distribution in some cases. The dynamic task assignment proposed uses a protocol to balance the workload of the processors, which may introduce large inter-processor communications. The method proposed in [29] targets shared memory systems. As the database is shared among processors, a dynamic strategy followed a *work-pool* model and fine-grained synchronizations can be used to balance the work load. This method is not always effective in a distributed memory system.

## 8. CONCLUSIONS

In this paper, we introduce a framework for parallel mining frequent itemset and sequential patterns, based on the divide-and-conquer property of the pattern growth algorithm. However, such a divide-and-conquer property, though minimizing inter-processor communication, causes load balancing problems, which restricts the scalability of parallelization. We present a sampling technique, called *selective sampling*, to address this problem. We implemented parallel frequent itemsets and sequential pattern mining algorithms with our framework. The experimental results have shown that our parallel algorithms have achieved good speedups on various datasets on at least 64 processors.

## 9. ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation(NGS program) under Grant No. 0103610. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

## 10. 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] R. Agrawal and R. 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.
- [4] R. Agrawal and R. Srikant. Mining sequential patterns. In *Eleventh International Conference on Data Engineering*, pages 3–14, Taipei, Taiwan, 1995.
- [5] F. Beil, M. Ester, and X. 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.
- [6] S. Brin, R. Motwani, and C. 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.
- [7] D. W. Cheung, J. Han, V. T. Ng, A. W. Fu, and Y. Fu. A fast distributed algorithm for mining association rules. In *Proceedings of the fourth international conference on Parallel and distributed information systems*, pages 31–43. IEEE Computer Society, 1996.
- [8] M. N. Garofalakis, R. Rastogi, and K. Shim. SPIRIT: Sequential pattern mining with regular expression constraints. In *The VLDB Journal*, pages 223–234, 1999.
- [9] B. Goethals and M. J. Zaki. Advances in frequent itemset mining implementations: report on fimi’03. *SIGKDD Explorations*, 6(1):109–117, 2004.
- [10] G. Grahne and J. Zhu. Efficiently using prefix-trees in mining frequent itemsets. In *FIMI*, 2003.
- [11] V. Guralnik and G. Karypis. Parallel tree-projection-based sequence mining algorithms. *Parallel Comput.*, 30(4):443–472, 2004.
- [12] E.-H. Han, G. Karypis, and V. 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.
- [13] J. Han and M. Kamber. *Data Mining: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., 2001.
- [14] J. Han and J. Pei. Mining frequent patterns by pattern-growth: methodology and implications. *SIGKDD Explor. Newsl.*, 2(2):14–20, 2000.
- [15] J. Han, J. Pei, and Y. 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.
- [16] IBM dataset generator for associations and sequential patterns. <http://www.almaden.ibm.com/software/quest/Resources>.
- [17] A. Javed and A. A. Khokhar. Frequent pattern mining on message passing multiprocessor systems. *Distributed and Parallel Databases*, 16(3):321–334, 2004.
- [18] B. Liu, W. Hsu, and Y. Ma. Integrating classification and association rule mining. In *Knowledge Discovery and Data Mining*, pages 80–86, 1998.
- [19] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard.*, 5 1994.
- [20] J. S. Park, M.-S. Chen, and P. 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.
- [21] J. Pei, J. Han, B. Mortazavi-Asl, J. Wang, H. Pinto, Q. Chen, U. Dayal, and M.-C. Hsu. Mining sequential patterns by pattern-growth: The prefixspan approach. *IEEE Transactions on Knowledge and data engineering*, 16(10), 2004.
- [22] I. Pramudiono and M. Kitsuregawa. Tree structure based parallel frequent pattern mining on pc cluster. In *DEXA*, pages 537–547, 2003.
- [23] A. Schuster and R. 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.
- [24] T. Shintani and M. Kitsuregawa. Mining algorithms for sequential patterns in parallel: Hash based approach. In *PAKDD*, pages 283–294, 1998.
- [25] K. Wang, Y. Xu, and J. X. Yu. Scalable sequential pattern mining for biological sequences. In *Proceedings*

- of the *Thirteenth ACM conference on Information and knowledge management*, pages 178–187. ACM Press, 2004.
- [26] R. Wol, A. Schuster, and D. Trock. A high-performance distributed algorithm for mining association rules. In *ICDM*, 2003.
- [27] P.-H. Wu, W.-C. Peng, and M.-S. Chen. Mining sequential alarm patterns in a telecommunication database. In *Proceedings of the VLDB 2001 International Workshop on Databases in Telecommunications II*, pages 37–51. Springer-Verlag, 2001.
- [28] O. R. Zaiane, M. El-Hajj, and P. Lu. Fast parallel association rule mining without candidacy generation. In *ICDM*, pages 665–668, 2001.
- [29] M. J. Zaki. Parallel sequence mining on shared-memory machines. *Journal of Parallel and Distributed Computing*, 61(3):401–426, 2001.
- [30] M. J. Zaki, S. Parthasarathy, M. Ogihara, and W. Li. Parallel algorithms for discovery of association rules. *Data Min. Knowl. Discov.*, 1(4):343–373, 1997.