Methods for parallel mining of frequent itemsets

Kessl, Robert
2012

Institute of Computer Science
Academy of Sciences of the Czech Republic

Methods for parallel mining of frequent itemsets

Robert Kessl

Technical report No. 1170

December 2012

Pod Vodárenskou věží 2, 182 07 Prague 8, phone: +420 266 051 111, fax: +420 286 585 789,
e-mail:ics@cs.cas.cz

Institute of Computer Science
Academy of Sciences of the Czech Republic

Methods for parallel mining of frequent itemsets[1]

Robert Kessl[2]

Technical report No. 1170

December 2012

Abstract:

One of the popular and important data mining algorithm is the algorithm for generation of so called frequent itemsets. We present a method for parallelization of an arbitrary algorithm for mining of frequent itemsets on a distributed memory computer. Our method staticaly load-balance the computation using probabilistic estimates of the load – however it always computes the set of frequent itemsets from the whole database. Our new method first samples the input database and then creates a sample of frequent itemsets using an arbitrary algorithm for mining of frequent itemsets and a reservoir sampling. The sample of frequent itemsets is then used for estimating the load. The sample of frequent itemsets is created either with a modified coverage algorithm or using the so called reservoir sampling algorithm. We experimentally evaluate the performance of our method on various datasets.

Keywords:
Data mining, parallel algorithms, frequent itemset mining, approximate counting

[1]This work is based on [19, 20, 21] with extended experimental results and more detailed description of the methods. This paper was supported from the Czech Science Foundation, grant number GA ČCR P202/10/1333.

[2]Institute of Computer Science, Academy of Sciences of Czech Republic Pod Vodarenskou Vezi 2, 182 07 Praha 8, Czech Republic, kesslr@cs.cas.cz

Methods for parallel mining of frequent itemsets*

Robert Kessl

Abstract

One of the popular and important data mining algorithm is the algorithm for generation of so called frequent itemsets. We present a method for parallelization of an arbitrary algorithm for mining of frequent itemsets on a distributed memory computer. Our method staticaly load-balance the computation using probabilistic estimates of the load – however it always computes the set of frequent itemsets from the whole database. Our new method first samples the input database and then creates a sample of frequent itemsets using an arbitrary algorithm for mining of frequent itemsets and a reservoir sampling. The sample of frequent itemsets is then used for estimating the load. The sample of frequent itemsets is created either with a modified coverage algorithm or using the so called reservoir sampling algorithm. We experimentally evaluate the performance of our method on various datasets.

Keywords: Data mining, parallel algorithms, frequent itemset mining, approximate counting

1 Introduction

Thanks to the automated data collection, companies collect huge amount of data. It is impossible to manually analyse such amounts of data. Therefore, automatic methods for analysis of the data are developed in *data mining*.

One of the important data mining tasks is the *mining of association rules* or *market basket analysis* [2]. The term market basket analysis comes from the first historical application. The market basket analysis comes from the need to analyse customer baskets of goods bought in a supermarket. The supermarket stores the list of items of the basket, called a *transaction*, into a database. The owner of the supermarket is interested in better shelfs organization and wants to analyse the behaviour of customers in the supermarket from the database of the transactions. The result of the process are so called *association rules*, i.e. rules $X \Rightarrow Y$ such that X, Y are sets of goods.

The association rules are mined in a two step process:

1. Mine all *frequent itemsets* (FIs in short): find all sets of items that occur in a fraction of transactions at least of size $min\_support^*$. The $min\_support^*$, called the relative minimal support, is a parameter of the computation.

2. Generate *association rules*: from the FIs generate all association rules with minimal confidence $min\_confidence$. An example of an association rule is {`bread`, `milk`} $\Rightarrow$ {`butter`} with confidence 15%, i.e. the `butter` occurs in 15% of transactions that also contains `bread` and `milk`.

Because the mining of FIs is computationally expensive, we can only mine some subsets of FIs, e.g. the mining of *maximal frequent itemsets* (MFIs in short) or the so called *closed itemsets*.

2 Mathematical foundation

First, we introduce the basic notion. Let $\mathcal{B} = \{b_i\}$ be a *base set* of items (items can be numbers, symbols, strings etc.). An arbitrary set of items $U \subseteq \mathcal{B}$ will be further called an *itemsets*. Further, we need to view the baseset $\mathcal{B}$ as an ordered set. The items are

*This work is based on [19, 20, 21] with extended experimental results and more detailed description of the methods. This paper was supported from the Czech Science Foundation, grant number GA ČR P202/10/1333.

therefore ordered using an arbitrary order $<$: $b_1 < b_2 < \ldots < b_n, n = |\mathcal{B}|$. Hence, we can view an itemset $U = \{b_{u_1}, b_{u_2}, \ldots, b_{u_{|U|}}\}$, $b_{u_1} < b_{u_2} < \ldots b_{u_{|U|}}$, as an ordered set denoted by $U = (b_{u_1}, b_{u_2}, \ldots, b_{u_{|U|}})$. If it is clear from context, we will not make difference between the set $\{b_{u_1}, b_{u_2}, \ldots, b_{u_{|U|}}\}$ and the ordered set $(b_{u_1}, b_{u_2}, \ldots, b_{u_{|U|}})$. We denote the ith smallest item of U ordered by the arbitrary order $<$ by $U[i] = b_{u_i}$. We denote the set of all itemsets, the powerset of $\mathcal{B}$, by $\mathcal{P}(\mathcal{B})$.

Let $U \subseteq \mathcal{B}$ be an itemset and $\mathtt{id} \in \mathbf{Z}$ a natural number, used as a unique identifier. We call the pair (id, U) a *transaction*. The $\mathtt{id}$ is called the *transaction id*. A subset W of a transaction $t = (id, U)$ will be further denoted by $W \dot{\subseteq} t$, i.e., W is a subset of t if and only if $W \subseteq U$. A superset V of a transaction will be denoted similarily, i.e., $t \dot{\subseteq} V$. Because U can be viewed as an ordered set, we can also view the transaction t as an ordered set and denote ith item of t by $t[i] = U[i]$. A *database* $\mathcal{D}$ on $\mathcal{B}$ (or database $\mathcal{D}$ if $\mathcal{B}$ is clear from context) is a sequence of transactions $t \dot{\subseteq} \mathcal{B}$, each transaction having a unique identifier in the whole $\mathcal{D}$. In our algorithms, we need to sample the database $\mathcal{D}$. A *database sample* is denoted by $\widetilde{\mathcal{D}}$.

We define the support of an itemset U in a database $\mathcal{D}$, denoted by $Supp(U, \mathcal{D})$, as the number of transactions containing U, but in some literature, the relative support is defined by $Supp^*(U) = Supp(U)/|\mathcal{D}|$.

An itemset is called *frequent itemset* (FI in short) iff $Supp(U, \mathcal{D}) \geq min\_support$ ($Supp^*(U, \mathcal{D}) \geq min\_support^*$), where $min\_support \in \mathbf{Z}$ ($min\_support^* \in \mathbf{R}$, $0 \leq min\_support^* \leq 1$) is a specified minimal support threshold. We will denote the set of all frequent itemsets, computed using $\mathcal{D}$, as $\mathcal{F}$. The set of all FIs, computed using $\widetilde{\mathcal{D}}$, is denoted by $\widetilde{\mathcal{F}}$. A sample of FIs, computed using $\widetilde{\mathcal{D}}$, is denoted by $\widetilde{\mathcal{F}}_s$. The sample $\widetilde{\mathcal{F}}_s$ is not necessarily subset of $\mathcal{F}$.

The basic property of frequent itemsets is the so called *monotonicity of support*:

Proposition 2.1 (Monotonicity of support)**.** *Let $U, V \subseteq \mathcal{B}$ be two itemsets such that $U \subsetneq V$ and $\mathcal{D}$ be a database. Then holds $Supp(U, \mathcal{D}) \geq Supp(V, \mathcal{D})$.*

We call the itemset U a *maximal frequent itemset* (or MFI in short) if $Supp(U, \mathcal{D}) \geq min\_support$, for any $V, U \subsetneq V$, $Supp(V, \mathcal{D}) < min\_support$. The set of all MFIs, computed using $\widetilde{\mathcal{D}}$, is denoted by $\widetilde{\mathcal{M}}$.

For the purpose of the description of our new parallel method, we denote the number of processors by P. The ith processor, $1 \leq i \leq P$, is denoted by p_i. At the start of the parallel algorithm, each processor p_i has a database partition D_i. Our parallel algorithms partitions the database at the beginning into P disjoint database partitions D_i, D_j such that $\bigcup_i D_i = \mathcal{D}$, $D_i \cap D_j = \emptyset, i \neq j$, and $|D_i| \approx |\mathcal{D}|/P$.

The multivariate hypergeometric distribution describes the following problem: let the number of colors be C and the number of balls colored with color i is M_i and the total number of balls is $N = \sum_i M_i$. Let X_i, $1 \leq i \leq C$, be a random variable representing the number of balls colored by the ith color. The sample of size n is drawn without replacement from the N balls. X_i of the n balls, such that $n = \sum_{i=1}^{C} X_i$, are colored by the ith color. Then the probability mass function is:

$$P(X_1 = k_1, \ldots, X_C = k_C) = \frac{\prod_{i=1}^{C} \binom{M_i}{k_i}}{\binom{N}{n}}.$$

2.1 The prefix-based equivalence classes

To decompose $\mathcal{P}(\mathcal{B})$ into disjoint sets, we need to order the items in $\mathcal{B}$. An equivalence relation partitions the ordered set $\mathcal{P}(\mathcal{B})$ into disjoint subsets called *prefix-based equivalence classes* (PBECs in short):

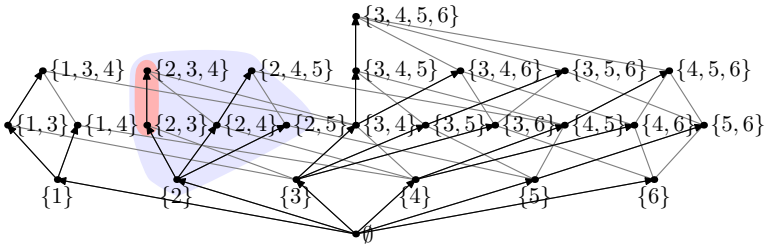
Definition 2.2 (prefix-based equivalence class)**.** *Let $U \subseteq \mathcal{B}, |U| = n$ be an itemset. We impose some order on the set $\mathcal{B}$ and hence view $U = (u_1, u_2, \ldots, u_n), u_i \in \mathcal{B}$ as an ordered set. A* prefix-based equivalence class *of U, denoted by $[U]_\ell$, is a set of all itemsets that have the same prefix of length ℓ, i.e., $[U]_\ell = \{W = (w_1, w_2, \ldots, w_m) | u_i = w_i, i \leq \ell, m = |W| \geq \ell, U, W \subseteq \mathcal{B}\}$*

To simplify the notation, we use $[W]$ for the prefix-based equivalence class $[W]_\ell$ iff $\ell = |W|$. Each $[W], W \subseteq \mathcal{B}$ is a sublattice of $(\mathcal{P}(\mathcal{B}), \subseteq)$.

Example 1: Ilustration of the mathematical notion

$\mathcal{D}$: DFS expansion tree and PBECs:

TID	Transaction
1	$\{1,2,3,4,6\}$
2	$\{3,5,6\}$
3	$\{1,3,4\}$
4	$\{1,2,6\}$
5	$\{1,3,4,5,6\}$
6	$\{1,2,3,4,5\}$
7	$\{2,3,4,5\}$
8	$\{2,3,4,5\}$
9	$\{3,4,5,6\}$
10	$\{2,4,5\}$
11	$\{1,2,4,5\}$
12	$\{2,3,4,5,6\}$
13	$\{3,4,5,6\}$
14	$\{4,5,6\}$
15	$\{1,3,4,5,6\}$



The picture shows the set $\mathcal{F}$ of the database $\mathcal{D}$ with $min\_support = 5$. The grey lines show the subset/superset relationship. The arrows show the DFS expansion tree.

- Prefix-based equivalence class $[(2)] \cap \mathcal{F} = \{\{2\},\{2,3\},\{2,4\},\{2,5\},\{2,3,4\},\{2,4,5\}\}$, marked in blue.
- Prefix-based equivalence class $[(2,3)] \cap \mathcal{F} = \{\{2,3\},\{2,3,4\}\}$ is a subclass of $[(2)]$, marked in red.
- The DFS expansion tree is higlighted using thicker lines with arrows. The extensions of the tree node $\{2\}$ is the set of nodes $\{3,4,5\}$, i.e., nodes $\{\{2,3\},\{2,4\},\{2,5\}\}$.
- The MFIs is the set $\mathcal{M} = \{\{1,3,4\},\{2,3,4\},\{2,4,5\},\{3,4,5,6\}\}$

Definition 2.3 (Extensions)**.** *Let $U \subseteq \mathcal{B}$ be an itemset. We impose some order $<$ on the set $\mathcal{B} = (b_1, b_2, \ldots, b_n)$ and view $U = (u_1, u_2, \ldots, u_m), u_i \in \mathcal{B}$ as an ordered set. The* extensions *of the prefix-based equivalence class $[U]$ is an ordered set $\Sigma \subseteq \mathcal{B}$ such that $U \cap \Sigma = \emptyset$ and for each $W \in [U]$ holds that $W \setminus U \subseteq \Sigma$. We denote the prefix-based equivalence class together with the extensions Σ by $[U|\Sigma]$.*

For example, let have $\mathcal{B} = \{1,2,3,4,5\}$, a prefix $U = \{1,2\}$, and the extensions $\Sigma = \{3,5\}$. Then $[U|\Sigma] = [\{1,2\}|\{3,5\}] = \{\{1,2,3\},\{1,2,5\},\{1,2,3,5\}\}$.

Proposition 2.4. *Let $U_i = \{b_i\}, b_i \in \mathcal{B}$ for all $i, 1 \leq i \leq |\mathcal{B}|$, and $\Sigma_i = \{b | b > b_i; b, b_i \in \mathcal{B}\}$ then $[U_i|\Sigma_i]$ are disjoint.*

Proof. The reason is obvious: each $W \in [U_i|\Sigma_i]$ contains b_i and *does not contain* $b < b_i$. □

Corollary 2.5. *Let $Q = \{(U_i, \Sigma_i)\}$ be a set such that $[U_i|\Sigma_i]$ are disjoint and $q = (V, \Sigma_V) \in Q$. Let $W_i = V \cup \{b_i\}, b_i \in \Sigma_V$ and $\Sigma_{W_i} = \{b | b_i < b; b_i, b \in \Sigma_V\}$ forms the PBECs $[W_i|\Sigma_{W_i}]$. Let have a new set of pairs $Q' = (Q \setminus \{q\}) \cup (\bigcup_i \{(W_i, \Sigma_{W_i})\})$. Then the pairs $(U'_i, \Sigma'_i) \in Q'$ forms disjoint PBECs $[U'_i|\Sigma'_i]$.*

We simplify the notation and omit the extensions if clear from context. Further, we need to partition $\mathcal{F}$ into n disjoint sets, denoted by $F_1, \ldots, F_n$, satisfying $F_i \cap F_j = \emptyset, i \neq j$, and $\bigcup_i F_i = \mathcal{F}$. This partitioning can be done using the PBECs. Let have PBECs $[U_l], (\bigcup_l [U_l]) \cup (\bigcup_l \mathcal{P}(U_l)) = \mathcal{F}, 1 \leq l \leq m$ and sets

of indexes of the PBECs $L_i \subseteq \{k|1 \leq k \leq m\}, 1 \leq i, j \leq n$ such that $L_i \cap L_j = \emptyset, i \neq j$, and $\sum_i |L_i| = m$ then $F_i = \bigcup_{l \in L_i}([U_l] \cap \mathcal{F})$.

3 Existing parallel algorithms

We consider basicaly two categories of parallel computers:

1. shared memory (SM in short) machines;
2. distributed memory (DM in short) machines.

Designing parallel algorithms for mining frequent itemsets on *shared memory machines* is relatively straightforward: the machine hardware supports easy parallelisation of the problem. All the processors have access to the shared memory. If we store the database in the shared memory and use a simple stack splitting algorithm with arbitrary distributed termination detection and dynamic load-balancing, the results must be very good. The reason is, that each processor has accesss to the whole database and to the datastructures created by other processors. To our best knowledge, the parallel algorithms for shared memory machines use the datastructures created by the other processors only for reading. Therefore the memory pages containing the data structures are read by the processors and there is no need for invalidation of the memory pages.

Parallel mining of FIs on DM machines is a hard task for couple reasons:

1. The databases are usually quite large and we want to have the database distributed among the processors so we utilize the main memory of all nodes. Frequent re-distribution of the database is out of question due to the size of the database.
2. The problem of parallel mining of FIs is highly irregular. For the same reasons as in 1 the dynamic load-balancing is out of question.

In this chapter, we will briefly describe existing parallel algorithms for mining of FIs. In Section 3.1, we show an example of a shared-memory parallel algorithm. Section 3.2 describes Apriori-based DM algorithm, Section 3.2.3 describes an asynchronous algorithm that does not need a sequential FI mining algorithm, Section 3.3 describes Eclat-based DM algorithms, and Section 3.4 describes FPGrowth-based DM parallel algorithms.

During the whole chapter, we denote a disjoint database partitions by $D_i, 1 \leq i \leq P$. D_i has always the size $|D_i| \approx |\mathcal{D}|/P$.

3.1 Example of a shared memory algorithm

An example of an algorithm that is designed for shared memory multiprocessors is the Multiple Local Frequent Pattern Tree algorithm (the MLFPT algorithm for short) [33]. The MLFPT algorithm, see Algorithm 1, is a parallelization of the FPGrowth algorithm. We omit the details of the FPGrowth algorithm in this section. The algorithm is summarized in Algorithm 1.

The reported speedup of this algorithm is quite good, e.g., 53.35 at 64 processors, 29.22 at 32 processors, and 7.53 at 8 processors with running time $\approx$ 25000 seconds on single processor. The experiments used databases of size 1M, 5M, 10M, 25M, and 50M transactions.

3.2 Apriori-based parallel DM algorithms

The first sequential FI mining algorithm was the Apriori algorithm. We omit the details of the sequential Apriori algorithm in this section.

There are many parallel algorithms based on the Apriori algorithm. The first algorithm was described by Agrawal et al. [1]. Agrawal proposed three parallel algorithms:

1. The Data Distribution algorithm.
2. The Count Distribution algorithm.
3. The Candidate Distribution algorithm.

Algorithm 1 The Multiple Local Frequent Pattern Trees algorithm

MLFPT(**In:** Database $\mathcal{D}$,
In: Integer *min_support*,
Out: Set $\mathcal{F}$)

1 **for** each processor p_i **do-in-parallel**
/* Parallel FPTree creation */
2 Loads i-th partition D_i of the database $\mathcal{D}$ into the main memory.
3 Count local support for each item $b \in \mathcal{B}$, on each processor.
4 Exchange local supports with other processors to compute global support for each $b \in \mathcal{B}$ (hence an all-to-all broadcast takes place).
5 Prune not frequent items, i.e., remove from $\mathcal{B}$ all items $b \in \mathcal{B}$ such that $Supp(\{b\}, \mathcal{D}) < min\_support$.
6 Create FP-Tree T_i from D_i
7 Barrier – processor p_i waits until all other processors has finished creation of the FP-Tree.
/* Asynchronous FI mining phase */
8 A modified FPGrowth algorithm is started: the modified algorithm is almost the same as the original FPGrowth algorithm but at the beginning it processes each FP-Tree T_i, creating a local FP-Tree that is used for further computations.
9 the computed FIs are put into the set $\mathcal{F}$
10 **end for**

Because Agrawal evaluated *the count distribution algorithm* as the best of these three algorithms, we will describe this algorithm, see Section 3.2.1. An improvement of the Apriori algorithm, the Fast Parallel Mining algorithm (the FPM algorithm in short) is described in Section 3.2.2.

3.2.1 The Count distribution algorithm

To describe the algorithm, we need to define the candidate itemset:

Definition 3.1 (candidate itemset on frequent itemset)**.** *Let k be an integer, U be an itemset of size k, $\mathcal{D}$ a database, and F_{k-1} the set of all frequent itemsets of size $k-1$. If each subset $W \subseteq U, |W| = k-1$ is frequent, $W \in F_k$, then U is called the* candidate itemset*. The set of all candidates of size k, denoted by C_k, is:*

$$C_k = \big\{U | U \subseteq \mathcal{B}, |U| = k, \text{ and for each } V \subsetneq U, |V| = k-1 \text{ follows that } V \in F_{k-1}\big\}.$$

Since the computation of the support is the most computationally expensive part, it computes the support for *candidate itemsets* in parallel. In the following text, we denote the set of all frequent itemsets of size k by F_k and the superset of all FIs, called candidate itemsets, of size k by C_k, i.e., $F_k \subseteq C_k \subseteq \mathcal{P}(\mathcal{B})$.

In the description of the Count Distribution algorithm, summarized in Algoirthm 2, we use:

1. The Compute-Support procedure that computes the support of a set of itemsets from a database.
2. The Generate-Candidates function that generate candidates from a set of frequent itemsets.

The understanding of the details of the Compute-Support procedure and the Generate-Candidates function are not important in order to understand the details of the Count Distribution algorithm. Therefore, we omit the details in this Section, they can be found in [1].

First, each processor p_i loads its part of the database, creates initial set of candidate itemsets $C_1 = \big\{\{b\} | b \in \mathcal{B}\big\}$, and computes its support in the database part D_i. The support of candidates can be computed using the Compute-Support procedure. Since each processor knows $\mathcal{B}$, each processor has the same set of initial candidate itemsets. Then, the local supports of the initial candidates are broadcast, so each processor can compute the global support of the initial candidates. C_1 is pruned and each processor gets frequent itemsets of size 1, i.e., $F_1 = \big\{U | U \in C_1 \text{ and } Supp(U, \mathcal{D}) \geq min\_support\big\}$. Since each processor has the same initial set of candidates and knows the global supports, then each p_i also has to have the same frequent itemsets of size 1. Thus, the first step is correct. All frequent itemsets of size k will be further denoted by F_k.

In step k, processors create a set of candidates C_k of size k from the previous frequent itemsets F_{k-1} of size $k-1$. The set C_k can be computed using the GENERATE-CANDIDATES function. The candidates are generated by calling C_k =GENERATE-CANDIDATES(F_{k-1}). Since each processor p_i has the same set of frequent itemsets of size $k-1$, each processor generates the same set of candidates. Then each processor p_i computes the local support for these candidates within its database part D_i and broadcast the local supports to each other processor. Each processor updates local support, computing global support for all these candidates, and creates frequent itemsets of size k, i.e., $F_k = \{U|U \in C_k \text{ and } Supp(U,\mathcal{D}) \geq min\_support\}$. Since each processor has correct frequent itemsets of size $k-1$ at the beginning of step k, each processor has to have correct candidates C_k. Thus, after exchanging and updating local supports and pruning candidates, all processors have the correct frequent itemsets of size k. Note that only the support values of each $U \in C_k$ must be exchanged, because every processor has exactly the same set of candidates.

The COUNT-DISTRIBUTION algorithm is summarized in the Algorithm 2.

Algorithm 2 The APRIORI-COUNT-DISTRIBUTION algorithm

APRIORI-COUNT-DISTRIBUTION(**In:** Database $\mathcal{D}$, **In:** Integer $min\_support$, **Out:** Set $\mathcal{F}$)

1 Processor p_i loads the database part D_i.
2 $k \leftarrow 1$
3 **for** each $p_i, i = 1, \ldots, P$ **do-in-parallel**
4 **if** $k = 1$ **then**
5 p_i generates initial candidates $C_1 \leftarrow \big\{\{b_i\}|b_i \in \mathcal{B}\big\}$.
6 **else**
7 p_i generates candidates C_k from frequent itemsets F_{k-1}, by calling $C_k \leftarrow$ GENERATE-CANDIDATES(F_{k-1}).
8 **end if**
9 p_i counts the support for candidates C_k over local database fraction using the COMPUTE-SUPPORT procedure.
10 p_i broadcasts the local support of the itemsets in C_k to each other processor (all-to-all broadcast).
11 p_i prune candidates, creating $F_k = \{U|U \in C_k, Supp(U,\mathcal{D}) \geq min\_support\}$.
12 **if** the set of frequent itemsets F_k is empty **then**
13 return all generated frequent itemsets, i.e., return $\mathcal{F} = \bigcup_k F_k$ and terminate.
14 **end if**
15 $k \leftarrow k + 1$
16 **end for**

3.2.2 The Fast Parallel Mining algorithm (FPM)

Cheung [6, 7] proposed two prunning techniques for the Count distribution algorithm, see Algorithm 3. The prunning techniques leverage two important relationships between a partitioned database and frequent itemsets. Let $\mathcal{D}$ be a database partitioned into n disjoint parts D_i of size $|D_i| \approx |\mathcal{D}|/P$, processor p_i having database part D_i. Cheung observed that if an itemset U is frequent in a database $\mathcal{D}$, i.e., $Supp^*(U,\mathcal{D}) \geq min\_support^*$, then U must be frequent in at least one partition D_i, i.e., there exists i such that $Supp^*(U,D_i) \geq min\_support^*$. Note that we are using the *relative supports*, instead of the *absolute supports*. Cheung proposed two kind of optimizations: 1) distributed pruning; 2) global pruning.

1) Distributed pruning: uses an important relationship between frequent itemsets and the partitioned database: *every (globally) frequent itemset in the whole database $\mathcal{D}$ must also be (locally) frequent on some processors in the database part D_i.*

If an itemset U is globally frequent (i.e. $Supp^*(U,\mathcal{D}) \geq min\_support^*$) and locally frequent on some processor p_i (i.e. $Supp^*(U,D_i) \geq min\_support^*$), then U is called *gl-frequent*. We will use $GL_{k(i)}$ to denote the gl-frequent itemsets of size k at p_i. As in the Apriori Count-Distribution algorithm, we denote the set of all FIs of size k computed in step k by F_k. Note that $\forall i, 1 \leq i \leq P, GL_{k(i)} \subseteq F_k$.

Lemma 3.2. *[7] If an itemset U is globally frequent then there exists a processor p_i such that U and all*

its subsets are gl-frequent *at processor* p_i.

For the next theorem, we need a function that creates the set of candidates:

$$CG_{k(i)} = \big\{U|U \subseteq \mathcal{B}, |U| = k, \text{ and for each } V \subsetneq U, \\ |V| = k-1 \text{ follows that } V \in GL_{k-1(i)}\big\}.$$

$CG_{k(i)}$ can be computed from $GL_{k(i)}$ using the algorithm GENERATE-CANDIDATES by calling $CG_{k(i)} = \text{GENERATE-CANDIDATES}(GL_{k-1(i)})$, see [1].

It follows from Lemma 3.2 that if $U \in F_k$, then there exists a processor p_i, such that all its subsets of size $k-1$ are gl-frequent at processors p_i, i.e., they belong to $GL_{k-1(i)}$.

Theorem 3.3. *[7] For every $k > 1$, the set of all frequent itemsets of size k, F_k, is a subset of $F_k \subseteq CG_k = \bigcup_{i=1}^{n} CG_{k(i)}$, where $CG_{k(i)} = \{U|U \subseteq \mathcal{B}, |U| = k$, and for each $V \subsetneq U, |V| = k-1$ follows that $V \in GL_{k-1(i)}\}$.*

In [7] is shown that CG_k, which is a subset of the Apriori candidates, could be much smaller then the number of the Apriori candidates.

2) Global prunning: after the supports of all itemsets are exchanged among the processors, the local support counts $Supp(U, D_i)$ also available for all processors. Let $|U| = k$. At each partition D_i, the monotonicity principle holds for all itemsets, i.e., $Supp(U, D_i) \leq Supp(V, D_i)$ iff $V \subsetneq U$. Therefore the local support $Supp(U, D_i)$ is bounded by

$$maxsupp(U, D_i) = \min_V \big\{Supp(V, D_i)|V \subsetneq U, \text{ and } \\ |V| = |U| - 1\big\}$$

from above, i.e., $Supp(U, D_i) \leq maxsupp(U, D_i)$. Because the global support $Supp(U, \mathcal{D}) = \sum_{1 \leq i \leq P} Supp(U, D_i)$ is the sum of its local support counts at all the processors, the value:

$$\sum_{1 \leq i \leq P} maxsupp(U, D_i)$$

is an upper bound of $Supp(U, D_i)$. If $\sum_{1 \leq i \leq P} maxsupp(U, D_i) < min\_support^* \times |\mathcal{D}| = min\_support$, then U can be pruned away. For the FPM algorithm, see Algorithm 3.

Algorithm 3 The FPM algorithm (Fast Parallel Mining algorithm)

FPM(**In:** Database $\mathcal{D}$,
In: Set $\mathcal{B}$,
In: Integer $min\_support$,
Out: Set $\mathcal{F}$)

1 **for** each processor p_i **do-in-parallel**
2 Compute the candidate sets $CG_{(k)} = \bigcup_{i=1}^{P} \text{GENERATE-CANDIDATES}(GL_{k-1(i)})$. (distributed prunning)
3 Apply global prunning to prune the candidates in CG_k.
4 Scan partition D_i to find out the local support counts $Supp(U, D_i)$ for all remaining candidates $U \in CG_k$.
5 Exchange $\{Supp(U, D_i)\}$ with all other processors to find out the global support counts $Supp(U, \mathcal{D})$.
6 Compute $GL_{k(i)} = \{U \in CG_k | Supp^*(U, D) \geq min\_support^* \times |D|$ and $Supp^*(U, D_i) \geq min\_support^* \times |D_i|\}$ and exchange the result with other processors.
7 **end for**
8 **return** $\mathcal{F} \leftarrow \bigcup_{i=1}^{P} GL_{k(i)}$

3.2.3 The asynchronous parallel FI mining algorithm

Veloso [31] proposed another parallelization of the frequent itemset mining process. This algorithm is based on the fact that if we know MFIs, we are able to mine all itemsets under MFIs asynchronously.

Each processor p_i reads its portion of the database D_i and computes the local support for all items in D_i. By exchanging the local supports the processors gets the support of all items in $\mathcal{D}$.

The algorithm uses the fact that if an itemset is frequent, it must be frequent in at least one partition D_i. Every processor p_i then finds all MFIs in its local database partition D_i and broadcasts them, together with the support, to other processors. Every processor now can find the union of MFIs that means to compute the frequent maximum of the MFIs. Now the processors know the boundaries of $\mathcal{F}$ and can pro-

ceed in a top-down fashion to compute the support of all itemsets. At the end, the processors exchange counts of the itemsets and prunes infrequent itemsets. This algorithm is summarized in Algorithm 4.

Algorithm 4 The ASYNCHRONOUS-FI-MINING algorithm

ASYNC-FI-MINING(**In:** Database $\mathcal{D}$,
In: Integer *min_support*,
Out: Set $\mathcal{F}$)

/* Phase 1: computation of MFIs */
for each processor p_i **do-in-parallel**
  Read its local database partition D_i.
  Computes all local MFIs, denoted by M_i.
end for
/* Phase 2 */
for each processor p_i **do-in-parallel**
  Broadcast of M_i (hence an all-to-all broadcast takes place).
  Compute $\bigcup_{1 \leq i \leq P} M_i$.
end for
/* Phase 3 (every node has $\bigcup_{1 \leq i \leq P} M_i$). */
for each processor p_i **do-in-parallel**
  Enumerates itemsets $U \subseteq m, m \in M_i$ in a top-down fashion.
end for
/* Phase 4 (reduction of results) */
for each processor p_i **do-in-parallel**
  Perform sum-reduction operation and removes itemsets $U, Supp(U) \leq min\_support$, i.e. processor p_i sends its frequent itemsets to p_{i+1} and the last processor removes all infrequent itemsets.
end for

The authors in [31] claims that the speedup range from 5 to 10 on 16 processors. Unfortunatelly, the paper [31] is missing a table of speedups, therefore we have estimated the speedup from graphs of the running time. Additionally, the problem is that in [31] there is no mention to the algorithm they use as a base for the computation of speedup, i.e., a sequential algorithm that is used for computation of the speedup of the method. If the used sequential algorithm is the Apriori algorithm then we have to argue that the Apriori algorithm itself is slow and the speedup could be much worse if the execution time of the parallel algorithm is compared to some other, quicker, sequential algorithm.

3.3 Eclat-based parallel algorithms

3.3.1 The bitonic scheduling

Zaki et. al. [35] proposed a parallelization of the Eclat algorithm [36], see Algorithm 5. The algorithm is similar to our method in the sense that it partitions $\mathcal{F}$ into prefix-based equivalence classes. However, it uses the *bitonic scheduling*, a heuristic for scheduling the prefix-based classes on the processors that is not able to capture the real size of each prefix-based equivalence class.

Algorithm 5 The PARALLEL-ECLAT algorithm

PARALLEL-ECLAT(**In:** Database $\mathcal{D}$,
In: Integer *min_support*,
Out: Set $\mathcal{F}$)

for each p_i **do-in-parallel**
  /* Initialization phase */
  Scan local database partition D_i.
  Compute local support for all itemsets of size 2,
  denoted by $C_2 = \{U | U \subseteq \mathcal{B}, |U| = 2\}$.
  Broadcast the local support of itemsets in C_2, creating global support of itemsets in C_2.
  /* Transformation Phase */
  Partition C_2 into equivalence classes
  Schedule the equivalence classes on all processors p_i
  Transform local database into vertical form
  Tidlists, needed by other process for computation of its assigned portion of the equivalence classes, are send to each other processor.
  /* Asynchronous phase */
  All processors computes frequent itemsets from the assigned equivalence classes.
  /* Final Reduction Phase */
  Aggregate results and output associations into $\mathcal{F}$
end for

The bitonic scheduling works this way: each equivalence class with n atoms is assigned a weight $\binom{n}{2}$, and the equivalence classes are assigned to processors p_i using a best-fit algorithm. The best-fit algorithm is in fact the same algorithm as the LPT-SCHEDULE, we use for assigning of the prefix-based equivalence classes, see Section 7.2 and Graham [13] for reference. The problem with this heuristic is that it does not capture the real size of the equivalence classes. This algorithm achieves speedups of $\approx$ 2.5–10.5 on 24 processors, $\approx$ 2–10 on 16 processors, $\approx$ 1.4–8 on 8 processors, and $\approx$ 3–3.5 on 4 processors. The experiments were performed on datasets generated using the IBM generator with average transaction size 10 and database size 800k, 1.6M, 3.2M, and 6.4M transactions.

3.4 FPGrowth-based parallel algorithms

The FPGrowth algorithm is an important sequential FI mining algorithm. In this section, we show two parallel algorithms based on the FPGrowth algorithm.

3.4.1 A trivial parallelization

A trivial distributed-memory parallelization of the FP-Growth algorithm is proposed in [26]. The parallelization uses dynamic load-balancing. The idea is that each processor creates its local FP-Tree, broadcast the local FP-Tree to other processors (resulting in global FP-Tree on every processor) and assign prefix-based equivalence classes to processors using a hash function. The problem is that the amount of assigned work is unpredictable and the resulting computational load highly unbalanced. The solution to the unbalanced computation is the use of dynamic load-balancing.

The dynamic load-balancing uses *minimal path-depth* threshold to estimate the granularity of a subtree. We define the *path-depth* as the maximal length of a path from the root to a list in an FP-Tree. Since the path-depth of the FP-Tree is non-increasing during the computation, the dynamic load-balancing works as follows: if a processor finishes its assigned work, it starts requesting work from other, busy, processors. The busy processors sends part of their assigned work to the requesting processor if and only if the path-depth is bigger than the *minimal path-depth* threshold.

The result of this approach is that the aggregate memory is not used efficiently. [26] reports speedup of $\approx$ 4–20 on 32 processors on a *single* dataset with 100K and maximal potentially frequent itemset size were set to 25, and 20. transactions. However, the speedup of 20 is achieved in only two experiments from five. In the rest of the experiments, the maximum speedup is $\approx$ 8 at 30 processors. The maximum execution time of the sequential algorithm was $\approx$ 900 seconds.

3.4.2 The Parallel-FPTree algorithm

The PARALLEL-FPTREE is proposed by Javed and Khokhar in [17], see Algorithm 6.

The problem with this approach is obvious: the computation must be unbalanced. However, in [17] present different results: an almost linear speedup. The reason for such results could be the very small running time of the algorithm (up to couple of seconds) and very small datasets (10000 transactions).

3.4.3 Map-reduce approach: the FPM algorithm

Li [22] presents an map-reduce approach for mining of top N frequent itemsets. The algorithm is a five step process:

1. *Sharding:* Dividing DB into P disjoint partitions (in [22] called shards), where P is the number of processors.

2. *Parallel Counting:* uses map-reduce [10] in the following way: a) the map process is fead with a partition(shard) and outputs $(b, 1), b \in \mathcal{B},$; b) reducer is fed with items and outputs its count.

3. *Grouping items:* in this step a K disjoint groups $G_i \subseteq \mathcal{B}$ of items are created.

Algorithm 6 The PARALLEL-FPTREE algorithm

PARALLEL-FPTREE(**In:** Database D,
In: Itemset $\mathcal{B}$,
In: Integer $min\_support^*$)

1 **for** each $p_i, 1 \leq i \leq P$ **do-in-parallel**
2 Each processor scans its assigned partition and computes the support for single items sets based on items in the local database.
3 The processors exchange the local supports and compute the global support for each itemset.
4 Each processor sorts the global support for the single itemsets and discards all the non-frequent items.
5 Each processor scans the assigned partition again and constructs a local FP-Tree.
6 The header table is partitioned into P disjoint sets and each processor is assigned to mine frequent patters for distinct set of item.
7 Since the partitioning in step 5 is static, each processor identifies the information from its local tree needed by other processors. The prefix paths of the single itemsets assigned to a processor in step 4 constitute the complete information nedded for the mining step. This is identified using a bottom up scan of the local FP-Tree.
8 The information in step 6 is communicated in $\log P$ rounds emplying a recursive merge of the tree strcuture over processors. For example, processor p_i communicates with processor $p^r_{P/2+1}\%P$ in round r where $1 \leq i \leq P$ and $0 \leq r \leq \log P$. At the end of each round, a processor simply unpacks the received information into its local FP-tree and prepares a new message for the next round of the merge. Performing merge this way prevents the sizes of the messages from frowing the merge progresses.
9 Each processor mines for frequent patterns in its assigned itemsets.
10 **end for**

4. *Parallel FPGrowth:* uses map-reduce in the following way: mapper knows the groups (created in previous step) and is fed with a transaction t. The transaction is then output as (group id, t). The reducer then takes as the input the pairs (group id, t) and starts the FP-Growth algorithm. The reducer maintains an array of top N itemsets.

5. *Aggregation:* as each processor has N itemsets, the output from the previous step is filtered so it results in top N itemsets (globally).

This authors of [22] claims that it has almost linear speedup for the number of processors up to 500: that is on P processors the speedup is P. The authors have very large datasets and are not able to compute do execute the FPGrowth algorithm on a single processor. Therefore, the authors uses a strange definition of speedup: *instead of using a standard definition of speedup* T_1/T_P, where T_1 is the execution time on single processor and T_P is the execution time on P processors. They assume that the speedup is 100 on 100 processors and they compute $T'_1 = T_{100} \cdot 100$ then they compute the speedup on P processors as usually T'_1/T_P. However, this is a problem, because the speedup will be different.

Let have an sequential algorithm whose speedup is linear: $S(P) = P \cdot C = T_1/T_P, 0 \leq C \leq 1$. We are not able to compute the parallel speedup of the algorithm and therefore we use $T'_1 = T_X \cdot X$, where T_X represents the parallel time for X processors. We denote such speedup by $S'(P) = P \cdot C' = T'_1/T_P = \frac{T_X \cdot X}{T_P}$. We know that $S(P_1)/S(P_2) = P_1/P_2$. Additionally, we know that $S(P) = T_1/T_P$ and therefore $S(P_1)/S(P_2) = \frac{T_1/T_{P_1}}{T_1/T_{P_2}} = \frac{T_{P_2}}{T_{P_1}}$. That is: $\frac{P_1}{P_2} = \frac{T_{P_2}}{T_{P_1}}$. We compute $C' = \frac{T_X \cdot X}{T_P \cdot P} = \frac{P \cdot X}{X \cdot P} = 1$. Therefore, using $T'_1 = T_X \cdot X$ instead of T_1 gives us always an ideal speedup $S'(P) = P$. A numerical example is shown in Table 1.

To conclude: the authors of [22] claims to have an almost linear speedup with a huge database. As shown, their claims may not be true and the results can be very different. The question is, why the authors did not use smaller dataset that could be processed on a single machine and then compute the speedup for $P < 100$. Additionally, we tried to estimate the execution time on a single processor:

An ideal algorithm		
Processors P	Parallel time T_P [seconds]	Speedup
1	1000	
100	10	100
500	2	500
1000	1	1000

Algorithm with speedup $P/2$ on P processors		
Processors P	Parallel time T_P [seconds]	Speedup
1	1000	
100	20	50
500	4	250
1000	2	500

Algorithm with speedup $P/2$ on P processors, using $T_1' = T_{100} \cdot 100$		
Processors P	Parallel time [seconds] T_P	Speedup
1	$2000 = 20 \cdot 100$(!!!)	
100	20	100
500	4	500
1000	2	1000

Table 1: A numerical example of a perfect parallelization and the approach used in [22] for computing speedup.

$T_1 \approx 33270$. We realized that the speedup on 500 processors is ≈ 5.3. However, extrapolation is not a good method of computation of T_1.

3.4.4 A sampling based framework for parallel data mining

[9] presents a parallel version of the FPGrowth algorithm with approach similar to ours:

1. Sample tree construction: scan the whole database constructing the sample database by discarding 20% of the most frequent items and the non-frequent items. From the sampled database is created the sample trie T.

2. Sample tree mining: processor 0 mines the trie T, but instead of outputing the FIs it measures the time needed for processing each item. We denote the execution time for item b_i, i.e., for PBEC $[\{b_i\}]$ by T_i.

3. Task partition: PBECs $[\{b_i\}]$ whose $T_i > \frac{1}{4}\frac{\sum_i T_i}{P}$ is further partitioned.

4. Task scheduling:

The authors of [9] designed a *selective sampling* method that works as follows: from the whole set of items of the input database $\mathcal{D}$, $\mathcal{B}$, is removed a fraction f (reported to be 20%) of the most frequent itemsets resulting in the set $\mathcal{B}'$; then is created a database $\mathcal{D}' = \{t' | t \in \mathcal{D}, t' = t \cap \mathcal{B}'\}$, called selective sample. The FPGrowth algorithm is then executed on the database $\mathcal{D}'$. The execution times of the FPGrowth algorithm on the whole database $\mathcal{D}$ then correlates with the execution times of $\mathcal{D}'$. The execution times are then used for scheduling the PBECs on the processors.

The reported speedups are approximatelly < 32 on 64 processors, < 8 on 16 processors, < 6 on 8 processors. In one case the speedup is very bad, i.e., ≈ 4 on 32 processors, 1.5 on 32 processors, and 1.5 on 8 processors.

In this approach, we suspect one possible problem: the problem of estimating the number of FIs in particular PBEC is #P-hard. That is: it may be possible that this heuristic may not well represent the execution time on the whole database. Additionally: what happens when the support is changed ? The authors of [9] have made one experiment for the following small databases: `pumsb`, `pumsb_star`, `connect`, `mushroom`, `T40I10D100K` with unknown value of minimal support. The unreported value of support in [9] is, from our point of view, a very big problem.

3.4.5 The DFP algorithm

In [4] is presented the Distributed FPGrowth (DFP in short) algorithm. The authors use various parallel optimization techniques:

1. *Minimization of communication costs:* the processors are organized in a ring, each processor creates a local tree, serialize the tree using a depth first traversal into an array of integers.

This reduces the cost from 48 bytes per node to 8 bytes per node. The tree is then send to the processors right neighborhood. It is easy to deserialize and merge the tree at once into an existing tree.

2. *Prunning redundant data:* processor p_i is assigned with items $W \subseteq \mathcal{B}$ receives from all other processors only the paths from every $b \in W$ to the root. This should reduce the communication costs.

3. *Partitioning of the mining process:* the authors use the same strategy as in [9].

4. *Memory optimization:* [4] presents various memory-optimization techniques that are used when the trie does not fit in main memory.

Unfortunately, the authors of [4] do not measure the speedup. The claimed reason is that the data do not fit in main memory. However, the question is: *why the researches did not use smaller (sampled) database* to show the speedup and then measurements on large datasets ? One of the possible answer is that the speedup may not very good. In the paper are provided some running times for 8 and 48 machines. If we take these numbers and try to *estimate the running time on a single processor*, we conclude that the speedup could be ≈ 4 on 8 processors and ≈ 9 on 48 processors.

4 The ParDCI algorithm

In this section, we present the ParDCI algorithm presented in [25]. This algorithm uses hybrid approach: first it employs the same breadth-first search as the Apriori algorithm and then it switches to depth-first search algorithm, a variant of the Eclat algorithm.

The parallel version of this algorithm is designed for cluster of shared-memory workstations, i.e., a cluster of workstations where each workstation is a shared-memory multiprocessor. The ParDCI algorithm works as follows:

1. Breadth-first search phase: it uses the candidate distribution [1] approach on inter-[1] and intra-node[2] levels. When the ParDCI realize that the vertical represetation fits into main memory it switches to the depth-first search.

2. The ParDCI splits the candidates into l partitions based on common prefixes: all candidates that shares the same prefix are assigned to the same section. Then the l partitions are assigned to processors based on the number of candidates in each section. At the the intra-node level the sections are assigned using dynamic load-balancing.

It has been shown that the sequential DCI algorithm uses more memory then its depth-first search counterparts, see [12], the memory increases exponentially with lower values of $min\_support$. However, the speedup of the algorithm is ≈ 5 on 6 processors for $min\_support^* = 0.015$ and ≈ 3.5 on 6 processors for $min\_support^* = 0.05$.

One problem is that the PBECs are scheduled based on the number of extensions. This approach seems to be worse then the bitonic scheduling (that does not works well). Another problems with this algorithm is the fact that the dynamic load-balancing on shared memory machines gives almost optimal speedup. Therefore, the algorithm is in fact executed on 3 processors with speed two-times the original processor. Another question is: why the authors did not setup the experiments in such a way that they show how good is the speedup without intra-node dynamic load-balancing. Therefore, the results can be much worse on larger number of processors.

4.1 Summary and conclusion

We have described parallel algorithms based on the Apriori, the FPGrowth and the Eclat algorithm. The biggest problem of the Apriori algorithm is its slowness and memory consumption. Therefore, parallelization of the Apriori algorithm is not practical. The biggest advantage of the parallel Apriori algorithms is that they use the aggregate memory of the

[1] among workstations
[2] within one workstation

cluster efficiently. That is: every processor from P processors has a database partition of size $|\mathcal{D}|/P$. The parallel Apriori algorithms usually works in iterations that correspond to the sequential Apriori iterations, except that they are done in parallel. The authors claims that static load-balancing is used. We must argue that the load is not statically balanced at all: parallel execution of the sequential iterations should not be considered as static load-balancing .

The parallelizations of the Eclat and the FP-Growth algorithms use an estimate of the sizes of the prefix-based classes. However, the estimates are very simple and do not capture the real amount of work assigned to the processors. The authors of these algorithms are basicaly overlooking the fact that counting the number of FIs in single PBEC is #P-hard[15]. Dynamic load-balancing on distributed-memory parallel computers also does not work. The reason is that the computation is quite fast and exchanging large portions of the database among processors can be quite time-consuming.

Parallelizations of other algorithms than the Apriori algorithm do not achieve good speedups. But, the Apriori itself is quite slow.

The best solution should:

1. distribute the computation: computation time of each processors should be approximately the same.

2. distribute the data: the database should be distributed among the processors so that processor p_i has database partition of size $|\mathcal{D}|/P$.

The algorithms that use the simple static load-balancing have the potential to have the database distributed, so each processor p_i has $|\mathcal{D}|/P$ number of transactions.

5 Approximate counting by sampling

Our method for parallel mining of FIs is based on efficiently estimating the number of FIs in a given prefix-based equivalence class (PBEC in short). To estimate the relative number of FIs in a PBEC, we do not need to count the relative number of FIs exactly. We can estimate the relative sizes of FIs in PBECs with a sampling algorithm that approximately counts the relative number of FIs in a PBEC. Further, when talking about the relative (absolute) size of a PBEC, we always mean the relative (absolute) number of FIs in the PBEC.

This chapter is organized as follows: first, in Section 5.1 we show how to estimate support of an itemset from a database sample. In Section 5.2 we show the two methods for estimating the size of a PBEC. In Section 5.3 we discuss how to choose the sampling parameters and the effect on the estimating of the size of the relative number of FIs in a union of PBECs.

5.1 Estimating the support of an itemset from a database sample

The time complexity of the detection whether an itemset U is frequent or not is in fact the complexity of computing the *relative support* $Supp^*(U, \mathcal{D})$ in the input database $\mathcal{D}$. If we know the approximate relative support of U, we can detect whether U is frequent or not with certain probability. We can estimate the relative support $Supp^*(U, \mathcal{D})$ from a database sample $\widetilde{\mathcal{D}}$, i.e., we can use $Supp^*(U, \widetilde{\mathcal{D}})$ instead of $Supp^*(U, \mathcal{D})$. The approach of estimating the relative support of U was described by Toivonen [30].

We define the error of the estimate of $Supp^*(U, \mathcal{D})$ from a database sample $\widetilde{\mathcal{D}}$ by $err_{supp}(U, \widetilde{\mathcal{D}}) = |Supp^*(U, \mathcal{D}) - Supp^*(U, \widetilde{\mathcal{D}})|$. The estimation error can be analyzed using the Chernoff bound without making other assumptions about the database. The error analysis then holds for a database of arbitrary size and properties.

Theorem 5.1. *[30] Given an itemset $U \subseteq \mathcal{B}$, two real numbers $\epsilon_{\widetilde{\mathcal{D}}}, \delta_{\widetilde{\mathcal{D}}}, 0 \leq \epsilon_{\widetilde{\mathcal{D}}}, \delta_{\widetilde{\mathcal{D}}} \leq 1$, and a random sample $\widetilde{\mathcal{D}}$ drawn from database $\mathcal{D}$ of size*

$$|\widetilde{\mathcal{D}}| \geq \frac{1}{2\epsilon^2_{\widetilde{\mathcal{D}}}} \ln \frac{2}{\delta_{\widetilde{\mathcal{D}}}},$$

then the probability that $err_{supp}(U, \widetilde{\mathcal{D}}) > \epsilon_{\widetilde{\mathcal{D}}}$ is at most $\delta_{\widetilde{\mathcal{D}}}$.

Proof. See [30] for proof. □

5.2 Estimating the relative size of a PBEC

To differentiate between the MFIs $\mathcal{M}$, the FIs $\mathcal{F}$, and other stuff computed using the $\mathcal{D}$ from the stuff computed using $\widetilde{\mathcal{D}}$, we use the symbol $\sim$ over the same symbols: all frequent itemsets $\widetilde{\mathcal{F}}$, all maximal frequent itemsets $\widetilde{\mathcal{M}}$, and the sample of FIs $\widetilde{\mathcal{F}}_s$.

In our parallel method for mining FIs, we need to estimate the relative size of a PBEC. This can be estimated using a sample of FIs $\widetilde{\mathcal{F}}_s \subseteq \widetilde{\mathcal{F}}$ computed using $\widetilde{\mathcal{D}}$. There are two ways for creating $\widetilde{\mathcal{F}}_s$: 1) compute $\widetilde{\mathcal{M}}$ and get $\widetilde{\mathcal{F}}_s$ using the *modified coverage algorithm*; 2) Compute $\widetilde{\mathcal{F}}$ and get $\widetilde{\mathcal{F}}_s$ using the *reservoir sampling*. These two algorithms are presented in the next two sections.

5.2.1 The coverage algorithm and its modification

Let us have the MFIs $\widetilde{\mathcal{M}}$, computed from $\widetilde{\mathcal{D}}$. The set of all MFIs $\widetilde{\mathcal{M}}$ is the upper bound on the set $\widetilde{\mathcal{F}}$, i.e., $\widetilde{\mathcal{F}} = \bigcup_{m \in \widetilde{\mathcal{M}}} \mathcal{P}(m)$. To create a sample $\widetilde{\mathcal{F}}_s \subseteq \widetilde{\mathcal{F}}$ of independently and identically distributed (i.i.d.) elements chosen from the uniform distribution, we can use the *coverage algorithm* [23] that uses $\widetilde{\mathcal{M}}$ for creation of the sample. To make the sampling in our parallel method for mining of FIs faster, we have modified the algorithm, so it does not create sample from *uniform* distribution, it creates only *independently* distributed sample. The coverage algorithm (or its modification) produces only the sample $\widetilde{\mathcal{F}}_s$.

The coverage algorithm estimates the relative size of a set $F \subseteq \widetilde{\mathcal{F}}$. The basic idea is to sample the set $\widetilde{\mathcal{F}}$ by selecting the set $m \in \widetilde{\mathcal{M}}$ and create a sample U as a subset $U \subseteq m$. Unfortunately, this process samples the set $\mathcal{S} = \biguplus_{m \in \widetilde{\mathcal{M}}} \mathcal{P}(m)$, where $\biguplus$ denotes the multiset sum(union), i.e., $\mathcal{S}$ contains all subsets of $m \in \widetilde{\mathcal{M}}$ multiple times and $|\mathcal{S}| = \sum_{m \in \widetilde{\mathcal{M}}} |\mathcal{P}(m)|$. To make the sample uniformly distributed, we should sample $\widetilde{\mathcal{F}} = \bigcup_{m \in \widetilde{\mathcal{M}}} \mathcal{P}(m)$. Assuring that we sample $\widetilde{\mathcal{F}}$ is very time-consuming in the case of large $|\widetilde{\mathcal{M}}|$, see [23]. We give up sampling of $\widetilde{\mathcal{F}}$ and sample $\mathcal{S}$, not making identically distributed sample.

In the case of the *coverage algorithm*, we can analyse the dependency of the error $\epsilon_{\widetilde{\mathcal{F}}_s}$ and the probability of the error $\delta_{\widetilde{\mathcal{F}}_s}$ of the estimated size of a PBEC on the number of samples $|\widetilde{\mathcal{F}}_s|$:

Theorem 5.2 (estimation error of the size of a subset $F \subseteq \widetilde{\mathcal{F}}$)**.** *[23] Let $\widetilde{\mathcal{M}}$ be the set of MFIs such that $\widetilde{\mathcal{F}} = \bigcup_{m_i \in \widetilde{\mathcal{M}}} \mathcal{P}(m_i)$, $F \subseteq \widetilde{\mathcal{F}}$, $\rho = |F|/|\widetilde{\mathcal{F}}|$, two real numbers $\epsilon_{\widetilde{\mathcal{F}}_s}, \delta_{\widetilde{\mathcal{F}}_s}$ such that $0 \leq \epsilon_{\widetilde{\mathcal{F}}_s}, \delta_{\widetilde{\mathcal{F}}_s} \leq 1$, and $\widetilde{\mathcal{F}}_s$ is the* independently and identically distributed *sample of $\widetilde{\mathcal{F}}$. Then the estimate $\frac{|F \cap \widetilde{\mathcal{F}}_s|}{|\widetilde{\mathcal{F}}_s|}$ is an estimation of $|F|/|\widetilde{\mathcal{F}}|$ with error at most $\epsilon_{\widetilde{\mathcal{F}}_s}$ with probability at least $1 - \delta_{\widetilde{\mathcal{F}}_s}$ provided*

$$|\widetilde{\mathcal{F}}_s| \geq \frac{4}{\epsilon^2_{\widetilde{\mathcal{F}}_s} \rho} \ln \frac{2}{\delta_{\widetilde{\mathcal{F}}_s}}.$$

Proof. The proof of the theorem is again based on the Chernoff bounds. We know that $P\left[|F \cap \widetilde{\mathcal{F}}_s| \geq (1 + \epsilon_{\widetilde{\mathcal{F}}_s})\rho|\widetilde{\mathcal{F}}_s|\right] \leq e^{-|\widetilde{\mathcal{F}}_s|\rho\epsilon^2_{\widetilde{\mathcal{F}}_s}/4}$ and similarily for the lower bound: $P\left[|F \cap \widetilde{\mathcal{F}}_s| \leq (1 - \epsilon_{\widetilde{\mathcal{F}}_s})\rho|\widetilde{\mathcal{F}}_s|\right] \leq e^{-|\widetilde{\mathcal{F}}_s|\rho\epsilon^2_{\widetilde{\mathcal{F}}_s}/4}$. Therefore:

$$P\left[(1 - \epsilon_{\widetilde{\mathcal{F}}_s})|\widetilde{\mathcal{F}}_s|\rho \leq |F \cap \widetilde{\mathcal{F}}_s| \leq (1 + \epsilon_{\widetilde{\mathcal{F}}_s})|\widetilde{\mathcal{F}}_s|\rho\right] \geq 1 - 2e^{-|\widetilde{\mathcal{F}}_s|\rho\epsilon^2_{\widetilde{\mathcal{F}}_s}/4} = 1 - \delta_{\widetilde{\mathcal{F}}_s}$$

□

In our case, we do not have uniform sample of $\widetilde{\mathcal{F}}$ and therefore the bounds cannot be used. But the estimates of the size of a PBEC made using the sample are sufficient for our purpose.

5.2.2 The reservoir sampling algorithm

In this section, we show the *reservoir sampling algorithm* that creates an uniformly but not independently distributed sample $\widetilde{\mathcal{F}}_s$ of $\widetilde{\mathcal{F}}$ on the contrary of the previous section.

Vitter [32] formulates the problem of *reservoir sampling* as follows: given a stream of records, $\widetilde{\mathcal{F}}$, the

task is to create a sample $\widetilde{\mathcal{F}}_s$ of size n *without replacement* from the stream without any prior knowledge of $|\widetilde{\mathcal{F}}|$. This solves our problem of making a uniform sample $\widetilde{\mathcal{F}}_s \subseteq \widetilde{\mathcal{F}}$. The Vitter's [32] algorithm runs in $\mathcal{O}(|\widetilde{\mathcal{F}}_s|(1+\log\frac{|\widetilde{\mathcal{F}}|}{|\widetilde{\mathcal{F}}_s|}))$.

In Theorem 5.2 we analysed the error of the approximation of the relative size of an arbitrary set using an i.i.d. sample using the Chernoff bounds. In the case of the reservoir sampling, we have to use the bounds for the *hypergeometric distribution*:

Theorem 5.3 (Estimation error of the size of a subset $F \subseteq \widetilde{\mathcal{F}}$)**.** *Let $F \subseteq \widetilde{\mathcal{F}}$ be a set of itemsets. The relative size of F, $\frac{|F|}{|\widetilde{\mathcal{F}}|}$, is estimated with error $\epsilon_{\widetilde{\mathcal{F}}_s}, 0 \leq \epsilon_{\widetilde{\mathcal{F}}_s} \leq 1$, with probability $\delta_{\widetilde{\mathcal{F}}_s}, 0 \leq \delta_{\widetilde{\mathcal{F}}_s} \leq 1$, from a hypergeometrically distributed sample $\widetilde{\mathcal{F}}_s \subseteq \widetilde{\mathcal{F}}$ with parameters $N = |\widetilde{\mathcal{F}}|, M = |F|$ of size:*

$$|\widetilde{\mathcal{F}}_s| \geq -\frac{\log(\delta_{\widetilde{\mathcal{F}}_s}/2)}{D(\rho+\epsilon_{\widetilde{\mathcal{F}}_s}||\rho)}$$

where $D(x||y)$ is the Kullback-Leibler divergence of two hypergeometrically distributed variables with parameters x, y and $\rho = |F|/|\widetilde{\mathcal{F}}|$.

The expected value of the size $|F \cap \widetilde{\mathcal{F}}_s|$ is $\mathrm{E}[|F \cap \widetilde{\mathcal{F}}_s|] = |\widetilde{\mathcal{F}}_s| \cdot \frac{|F|}{|\widetilde{\mathcal{F}}|}$.

Proof. The proof is based on bounds provided in [29] which are a summarization of [8] and the fact that $D(p+\epsilon||p) > D(p-\epsilon||p)$. □

5.3 Estimating the size of a union of PBECs

It can be expected that the relative sizes of PBECs computed using $\widetilde{\mathcal{D}}$ will be similar to the relative sizes computed using $\mathcal{D}$. The following theorem bounds the difference between the size of union of PBECs computed from $\mathcal{D}$ and $\widetilde{\mathcal{D}}$:

Theorem 5.4 (bounds on the size of a set of FIs from a given PBEC)**.** *Let $V_i \subseteq \mathcal{B}, 1 \leq i \leq n, [V_i] \cap [V_j] = \emptyset, i \neq j$. We use two sets of itemsets:*

1. $A = \{U | Supp^*(U,\mathcal{D}) < min\_support^* \text{ and } Supp^*(U,\widetilde{\mathcal{D}}) \geq min\_support^*\}$, *i.e., the collection of itemsets U infrequent in $\mathcal{D}$ and frequent in $\widetilde{\mathcal{D}}$ – wrongly added FIs to $\widetilde{\mathcal{F}}$.*
2. $B = \{U | Supp^*(U,\mathcal{D}) \geq min\_support^* \text{ and } Supp^*(U,\widetilde{\mathcal{D}}) < min\_support^*\}$, *i.e., the collection of itemsets U frequent in $\mathcal{D}$ and infrequent in $\widetilde{\mathcal{D}}$ – wrongly removed FIs from $\widetilde{\mathcal{F}}$.*

The relative size of A is denoted by $a = \frac{|A|}{|\mathcal{F}|}$ and the relative size of B is denoted by $b = \frac{|B|}{|\mathcal{F}|}$. Then for two sets of itemsets $C = \bigcup_i [V_i] \cap \mathcal{F}$ and $\widetilde{C} = \bigcup_i [V_i] \cap \widetilde{\mathcal{F}}$, we have:

$$\frac{|\widetilde{C}|}{|\widetilde{\mathcal{F}}|}(1+a-b) - a \leq \frac{|C|}{|\mathcal{F}|} \leq \frac{|\widetilde{C}|}{|\widetilde{\mathcal{F}}|} \cdot (1+a-b) + b$$

Proof. From the assumptions follows: $|\widetilde{\mathcal{F}}| = |\mathcal{F}|(1+a-b)$. Therefore: $\frac{|\widetilde{\mathcal{F}}|}{(1+a-b)} = |\mathcal{F}|$. We know that the fraction a of FIs is not frequent in $\mathcal{D}$ but is frequent in $\widetilde{\mathcal{D}}$ are present in $\widetilde{\mathcal{F}}$. Therefore, we can compute the lower bound of the relative size of C: $|\widetilde{C}| \leq |C| + a \cdot |\mathcal{F}|$, i.e., $\frac{|\widetilde{C}|}{|\mathcal{F}|} \leq \frac{|C|}{|\mathcal{F}|} + a$. Then using the fact that $|\mathcal{F}| = \frac{|\widetilde{\mathcal{F}}|}{(1+a-b)}$ we have: $\frac{|\widetilde{C}|}{|\widetilde{\mathcal{F}}|}(1+a-b) - a \leq \frac{|C|}{|\mathcal{F}|}$.

We compute the upper bound of $\frac{|C|}{|\mathcal{F}|}$ using similar computations as for the lower bound. The fraction b of FIs $\mathcal{F}$ was not frequent in $\widetilde{\mathcal{D}}$ and frequent in $\mathcal{D}$ and therefore the lower bound of the size $|\widetilde{C}|$ is $|C| - b \cdot |\mathcal{F}| \leq |\widetilde{C}|$ and therefore $\frac{|C|}{|\mathcal{F}|} \leq \frac{|\widetilde{C}|}{|\widetilde{\mathcal{F}}|} \cdot (1+a-b) + b$ □

Corollary 5.5. *If the size of $\frac{|\widetilde{C}|}{|\widetilde{\mathcal{F}}|}$ is estimated with error $\epsilon_{\widetilde{\mathcal{F}}_s}, 0 \leq \epsilon_{\widetilde{\mathcal{F}}_s} \leq 1$, with probability $0 \leq \delta_{\widetilde{\mathcal{F}}_s} \leq 1$ then:*

$$\frac{|\widetilde{C}|}{|\widetilde{\mathcal{F}}|}(1-\epsilon_{\widetilde{\mathcal{F}}_s})(1+a-b) - a \leq \frac{|C|}{|\mathcal{F}|}$$

and

$$\frac{|C|}{|\mathcal{F}|} \leq \frac{|\widetilde{C}|}{|\widetilde{\mathcal{F}}|}(1-\epsilon_{\widetilde{\mathcal{F}}_s})(1+a-b) + b$$

with probability $\delta_{\widetilde{\mathcal{F}}_s}$.

Set C can be viewed as a partition processed by a single processor. We estimate the relative size of $|C|/|\mathcal{F}|$ from $\widetilde{\mathcal{F}}_s$ and we are able to bound the error made while estimating the size of a partition. Unfortunately, the bounds are not very tight and making tighter bounds is hard.

Let $U_i \subseteq \mathcal{B}, 1 \leq i \leq n$, be prefixes and $[U_i]$ corresponding PBECs. We are creating the PBECs by recursive splitting, see Propositions 2.4 and 2.5, and estimating the size using the sample, i.e., $|[U_i] \cap \widetilde{\mathcal{F}}|/|\widetilde{\mathcal{F}}| \approx |[U_i] \cap \widetilde{\mathcal{F}}_s|/|\widetilde{\mathcal{F}}_s|$. Let $L \subseteq [1, n]$ be the set of indexes of the PBECs. The set of indexes is chosen in such a way that $|\bigcup_{i \in L}[U_i] \cap \widetilde{\mathcal{F}}_s|/|\widetilde{\mathcal{F}}_s| \approx 1/P$. That is: the set L is *dependent* on the created sample $\widetilde{\mathcal{F}}_s$. Therefore, we are not able to use the Chernoff bounds (or the estimates using the Kullback-Leibler divergence) with the same sample $\widetilde{\mathcal{F}}_s$ (used for creation of PBECs) for estimation of the relative size of $F = \bigcup_{j \in L}[U_j] \cap \widetilde{\mathcal{F}}$ because the sets $[U_j]$, the set $\mathcal{F}$ and the sample $\widetilde{\mathcal{F}}_s$ are not independent. Instead, we must choose $\epsilon_{\widetilde{\mathcal{F}}_s}$ such that $\epsilon_{\widetilde{\mathcal{F}}_s} \cdot |L|$ is small enough. In Chapter 8, we experimentaly show the error and its probability made by a particular choice of the number of samples.

6 Approximate parallel mining of MFIs

In our method described in [21], we need to compute an approximation of MFIs $\widetilde{\mathcal{M}}$. In our previous paper [21], we have proposed that we can execute an arbitrary algorithm for mining of MFIs in parallel and compute a set M, such that $\widetilde{\mathcal{M}} \subseteq M$, instead of $\widetilde{\mathcal{M}}$. In this section, we show an important property of $|M|$.

Because, we have P processors at our disposal, we could execute an arbitrary algorithm for mining of MFIs in parallel. Unfortunately, parallel mining of MFIs using a DFS algorithm, is a hard task. We can relax the requirement of computing $\widetilde{\mathcal{M}}$ to a requirement of computing the set M such that $\widetilde{\mathcal{M}} \subseteq M \subseteq \widetilde{\mathcal{F}}$. We define a *candidate* on an MFI as follows:

Definition 6.1 (candidate itemset on MFI)**.** *Let $U \subseteq \mathcal{B}$ be a frequent itemset and Σ the extensions used by a DFS MFI algorithm for extending U. We call U a candidate itemset (or candidate in short) on an MFI if for each $b \in \Sigma$ the itemset $U \cup \{b\}$ is not frequent, i.e., $Supp(U \cup \{b\}) < min\_support$.*

The candidates are the leafs of the DFS algorithm for mining of MFIs.

Definition 6.2 (longest subset of a MFI in a PBEC)**.** *Let W be a maximal frequent itemset, $b \in \mathcal{B}$ an item, the set $\Sigma = \{b' \in \mathcal{B} : b < b'\}$, and $[\{b\}|\Sigma]$ the PBEC. We call the set $U = W \cap (\Sigma \cup \{b\})$* the longest subset of W in the PBEC $[\{b\}|\Sigma]$.

For example, let $U = \{1\}$ be a prefix and $\Sigma = \{2, 3, 5\}$ its extensions. For the MFI $m = \{1, 3, 4, 5\}$ the longest subset of m in $[U|\Sigma]$ is the set $\{1, 3, 5\}$.

The longest subset of a MFI in a PBEC can be a candidate set, but there exists longest subsets that are not candidates. We say that *W is a candidate on the MFI $U, W \subsetneq U$ in a PBEC*, if it is a *candidate* and a longest subset of U in the PBEC, i.e., it is a leaf of a DFS tree and it is a longest subset.

A MFI $W = (b_{w_1}, \ldots, b_{w_{|W|}}), b_{w_1} < \ldots < b_{w_{|W|}}$ is discovered by a DFS MFI algorithm by expanding first $[(b_{w_1})]$, then $[(b_{w_1}, b_{w_2})]$, etc. To our best knowledge, all MFIs DFS mining algorithms works as follows: the algorithm initializes $\widetilde{\mathcal{M}} \leftarrow \emptyset$ and starts a DFS search in the lattice of all FIs, skipping some FIs. The PBECs are expanded in the order of b_i, i.e., $[(b_1)|(b_2, b_3, \ldots, b_{|\mathcal{B}|})]$ is processed first, then $[(b_2)|(b_3, \ldots, b_{|\mathcal{B}|})]$ is processed, etc. Therefore, if $U = (b_{u_1}, \ldots, b_{u_{|U|}})$ is an MFI and $W \subseteq U$ a candidate itemset. Candidate itemsets W are discovered *after* discovering U. If the algorithm finds a candidate itemset W, it looks into $\widetilde{\mathcal{M}}$ and if $\widetilde{\mathcal{M}}$ *contains a superset* of W, the algorithm skips W (*not* storing W in $\widetilde{\mathcal{M}}$). If $\widetilde{\mathcal{M}}$ *does not contains* a superset of W, it is an MFI and is stored into $\widetilde{\mathcal{M}}$.

An important fact is that an MFI $U \in \widetilde{\mathcal{M}}$ is always discovered by a DFS algorithm for mining MFIs *before* discovering any of its candidates. This follows from the fact that the items in the baseset $\mathcal{B}$ are ordered and a DFS algorithm for mining of MFIs processes $[(b_i)]$ and its extensions in the order of the items in $\mathcal{B}$.

Schema of parallel computation of M: because $\widetilde{\mathcal{D}}$ is much smaller than the whole database $\mathcal{D}$, the processors have a copy of $\widetilde{\mathcal{D}}$ and knows the items that are frequent in the database $\mathcal{D}$. All processors partition the base set $\mathcal{B}$ to P blocks of size $\approx |\mathcal{B}|/P$. Processor p_i runs a sequential DFS MFI algorithm in the i-th part of $\mathcal{B}$, where the items b_j are interpreted as 1-prefixes, i.e., prefix-based equivalence classes $[(b_j)]$. When a processor finishes its assigned items, it asks other processors for work. The computation is terminated using the Dijkstra's token termination detection algorithm. The output of the algorithm is a superset of all MFIs. This approach computes the set M such that $\widetilde{\mathcal{M}} \subseteq M$.

We demonstrate the parallel execution (the parallel processing of assigned PBECs) of a sequential DFS algorithm for mining of MFIs on the following example (for simplicity without dynamic load balancing): because the computation is distributed, the algorithm is unable to check the candidate against all already computed MFIs which results in a superset of all MFIs. Let $B = \{1, 2, 3, 4, 5, 6\}$ and $P = 3$ and assume that the prefix-based equivalence classes $[(1)|(2, 3, 4, 5, 6)], [(2)|(3, 4, 5, 6)]$ were assigned to p_1; the prefix-based equivalence classes $[(3)|(4, 5, 6)], [(4)|(5, 6)]$ were assigned to p_2; and the classes $[(5)|(6)], [(6)|\emptyset]$ to p_3. The MFIs $\{\{1, 3, 4\}, \{2, 3, 4\}, \{2, 4, 5\}\}$ are correctly computed by p_1. The processor p_2 correctly computes the MFI $\{3, 4, 5, 6\}$, but processor p_3 computes also the itemset $\{5, 6\}$ as an MFI. The reason is that p_3 does not know that the MFI $\{3, 4, 5, 6\}$ was already computed by processor p_2.

Lemma 6.3. *Let $W = (b_{w_1}, \ldots, b_{w_{|W|}})$ be an MFI, $b \in W$ any of its element. There exists* at most one candidate *on the MFI W in the PBEC $[(b)]$. If such candidate exists then it is the longest subset $S_W = \{b' | b' \in W, b \leq b'\}$ of the MFI W in the PBEC $[(b)]$.*

Proof. In each PBEC $[(b)]$ all frequent sets $X \in [(b)]$ such that $X \subseteq W$ are always subsets of S_W. Consider sets $X \subsetneq S_W$: X cannot be a candidate because there exists an item $b \in S_W$ such that $X \cup \{b\}$ is frequent, due to the monotonicity of the support, see Theorem 2.1. Note that S_W is a candidate if and only if there is no frequent itemset in $[(b)]$, which is a proper superset of S_W. □

The number of candidates of the MFI W depends on all other mined MFIs and subset/superset relations of the longest subsets of all MFIs. The following theorem is a direct consequence of Lemma 6.3:

Theorem 6.4. *Let have a baseset $\mathcal{B}$ and $1 < P < |\mathcal{B}|$ processors $p_1, \ldots, p_P$, a database $\widetilde{\mathcal{D}}$, M_i is a set of itemsets computed by p_i, and $M = \bigcup_{1 \leq i \leq P} M_i$. Let W be the* longest *MFI, i.e., for all $U, W \in \widetilde{\mathcal{M}}$ holds that $|U| \leq |W|$. An arbitrary DFS algorithm for mining MFIs that is executed in parallel computes (as we have described in our schema) a set of itemsets M, such that $\widetilde{\mathcal{M}} \subseteq M$, of size:*

$$|\widetilde{\mathcal{M}}| < |M| = \left| \bigcup_{1 \leq i \leq P} M_i \right| \leq |W| \cdot |\widetilde{\mathcal{M}}|.$$

Proof. The proof of this theorem follows from the Lemma 6.3 and the fact that for each MFI U there are at most $|U|$ PBECs that contains some subsets of U, i.e., in the worst case the dynamic load-balancing causes that the described scheme discovers all candidates on a single MFI. □

If we do not use dynamic load-balancing and assign the items statically (each processor processig $|\mathcal{B}|/P$ PBECs), for an MFI $U = (b_{u_1}, \ldots, b_{u_{|U|}})$ each p_i computes the candidate on the MFI U, if it exists, in each of its assigned PBECs with prefix of size 1. Denote the longest MFI by W, as in the previous theorem. If we statically assign the PBECs to each processor and do not use dynamic load-balancing, the upper bound on $|M|$ is $|M| < P \cdot |\widetilde{\mathcal{M}}|$. The two bounds can be combined: $|M| < \min(P, |W|) \cdot |\widetilde{\mathcal{M}}|$.

7 Proposal of a new DM parallel method

In this section, we present our two methods [20], and [21] and a *new method* that is based on the reservoir

sampling [19]. These methods provides parallelizations of the DFS (or BFS) sequential frequent itemsets mining algorithm. The method has the following advantages over current existing algorithms: 1) it is universal, i.e., it is possible to parallelize any existing sequential algorithm for mining of FIs; 2) The computation is balanced staticaly: for a very large databases the dynamic load-balancing is too expensive. An important property of our method is that it distributes the computed FIs among processors.

Our new method is called *Parallel Frequent Itemset MIning* (Parallel-FIMI in short) and has three variants: Parallel-FIMI-Seq [20], Parallel-FIMI-Par [21], and a new variant Parallel-FIMI-Reservoir. This method works for any number of processors $P \ll |\mathcal{B}|$. The basic idea is to partition all FIs into P disjoint sets F_i, using PBECs, of relative size $\frac{|F_i|}{|\mathcal{F}|} \approx \frac{1}{P}$. Each processor p_i then processes partition F_i.

The input and the parameters of the whole method are the following: 1) Minimal support: the real number *min_support**; 2) The sampling parameters: real numbers $0 \leq \epsilon_{\widetilde{\mathcal{D}}}, \delta_{\widetilde{\mathcal{D}}}, \epsilon_{\widetilde{\mathcal{F}}_s}, \delta_{\widetilde{\mathcal{F}}_s} \leq 1$; 3) the relative size of a smallest PBEC: the parameter $\rho, 0 \leq \rho \leq 1$; 4) partition parameter: real number $\alpha, 0 \leq \alpha \leq 1$; 5) database parts $D_i, 1 \leq i \leq P$. Without loss of generality, we expect that each $b_i \in \mathcal{B}$ is frequent.

The whole method consists of four phases: Phases 1 and 2 prepare the PBECs and its assignment to the processors for Phase 4, i.e., the static load-balancing is created in Phases 1 and 2. In the Phase 3, we redistribute the database partitions so each processor can proceeds independently with the assigned PBECs. In the Phase 4, we execute an arbitrary algorithm for mining of FIs and the processors computes the FIs in it assigned PBECs.

7.1 Detailed description of Phase 1

In Phase 1, we create a sample $\widetilde{\mathcal{F}}_s$ of all frequent itemsets. The input of this phase, for processor p_i, are the database partitions D_i such that $D_i \cap D_j = \emptyset, i \neq j$, $|D_i| \approx |\mathcal{D}|/P$, the relative minimal support *min_support**, and the real numbers $0 \leq \epsilon_{\widetilde{\mathcal{D}}}, \epsilon_{\widetilde{\mathcal{F}}_s}, \delta_{\widetilde{\mathcal{D}}}, \delta_{\widetilde{\mathcal{F}}_s} \leq 1$. The output of this phase is the sample of FIs $\widetilde{\mathcal{F}}_s$ and the database sample $\widetilde{\mathcal{D}}$. We propose three methods for creation of $\widetilde{\mathcal{F}}_s$. The input and the output is the same for all of the three proposed variants of Phase 1. For the details on sampling, see the Sections 5.1 and 5.2.

We propose three variants of the first phase:

1. Compute the boundary M of $\widetilde{\mathcal{F}}$, see [21, 20]:
   (a) Sequentially: the boundary in this case is the set $M = \widetilde{\mathcal{M}}$, the Parallel-FIMI-Seq method, [20].
   (b) In parallel: the boundary in this case is a set M, such that $\widetilde{\mathcal{M}} \subsetneq M \subsetneq \widetilde{\mathcal{F}}$, the Parallel-FIMI-Par method, [21].

   And finally create the sample $\widetilde{\mathcal{F}}_s$ using the *modified coverage algorithm*.
2. Create the sample $\widetilde{\mathcal{F}}_s$ by putting together an arbitrary sequential algorithm for mining of FIs and the so called *reservoir sampling*, the Parallel-FIMI-Reservoir method, see [19].

7.1.1 The modified coverage algorithm based sampling

In this section, we show two variants of Phase 1 based on our modification of the *coverage algorithm*, see [20], [21]. Additionally, we put together the fragments of the algorithms shown in previous chapters.

(a) $\widetilde{\mathcal{M}}$ is computed sequentially[20]: the $\widetilde{\mathcal{M}}$ is computed on processor p_1 using an arbitrary algorithm for mining of MFIs. The sampling is performed sequentially by processor p_1 using the *modified coverage algorithm*. We omit the detailed description of this version of Phase 1 as it is a simple execution of an arbitrary sequential algorithm for mining of MFIs. For the purpose of showing the pseudocode of Parallel-FIMI-Seq.

(b) The set $M, \widetilde{\mathcal{M}} \subseteq M \subsetneq \widetilde{\mathcal{F}}$ ($\widetilde{\mathcal{M}}$ plus some additional frequent itemsets) is computed in parallel[21]: we have already described the parallel execution of a DFS algorithm for mining of MFIs in Section 6. Therefore, we only describe one important detail: creation of the sample $\widetilde{\mathcal{F}}_s$ from the candidates on the MFIs in parallel.

The computed sets are distributed among the processors and the number of these sets can be large. Therefore, we perform the sampling in parallel. We denote the set of itemsets computed by p_i by M_i. *The parallel sampling of* $\widetilde{\mathcal{F}}$ *using* $M = \bigcup_{1 \leq i \leq P} M_i$ is performed in the following way: every processor p_i broadcasts the sum $s_i = \sum_{m \in M_i} |\mathcal{P}(m)|$ of sizes of powersets of its local MFIs (hence, an all-to-all broadcast takes place), creates a fraction of sample $\widetilde{\mathcal{F}}_s$ of size $|\widetilde{\mathcal{F}}_s| \cdot \frac{s_i}{\sum_{1 \leq j \leq P} s_j}$, and finally sends them to p_1. We should pick the number of samples chosen by each processor from a multivariate binomial distribution with parameters $p_i = \frac{s_i}{\sum_{1 \leq j \leq P} s_j}$ and $n = \sum_{1 \leq j \leq P} s_j$. However, using the modified coverage algorithm makes from the sample just a heuristic. Therefore, we do not have any guarantees on the error and p_i takes $|\widetilde{\mathcal{F}}_s| \cdot \frac{s_i}{\sum_{1 \leq j \leq P} s_j}$ number of samples.

7.1.2 The sampling based on the reservoir algorithm

In the previous section, we have shown a variant of Phase 1, based on the *modified coverage algorithm*, that samples $\mathcal{F}$ non-uniformly. In this section, we **propose** a new variant of the Phase 1: a sampling process based on the *reservoir sampling* [32] that samples $\widetilde{\mathcal{F}}$ uniformly, i.e., it creates an identically distributed sample of $\widetilde{\mathcal{F}}$. This work is also described in [19].

To speedup the sampling phase of our parallel method, we execute the reservoir sampling in parallel. The database sample $\widetilde{\mathcal{D}}$ is distributed among the processors – each processor having a copy of the database sample $\widetilde{\mathcal{D}}$. The baseset $\mathcal{B}$ is partitioned into P parts $B_i \subseteq \mathcal{B}$ of size $|B_i| \approx |\mathcal{B}|/P$ such that $B_i \cap B_j = \emptyset, i \neq j$. Processor p_i then takes part B_i and executes an arbitrary sequential DFS algorithm for mining of FIs, enumerating $[(b_j)] \cap \widetilde{\mathcal{F}}, b_j \in B_i$. The output, the itemsets $[(b_j)] \cap \widetilde{\mathcal{F}}$, of the sequential DFS algorithm are read by the reservoir sampling algorithm. If a processor finished its part B_i, it asks other processors for work. For terminating the parallel execution, we use the Dijkstra's token termination algorithm.

The task of the process is to take $|\widetilde{\mathcal{F}}_s| = -\frac{\log(\delta_{\widetilde{\mathcal{F}}_s}/2)}{D(\rho+\epsilon_{\widetilde{\mathcal{F}}_s}||\rho)}$ samples, see Theorem 5.3. Because the reservoir algorithm and the sequential algorithm is executed in parallel, it is not known how many FIs is computed by each processor. Denote the unknown number of FIs computed on p_i by f_i, the total number of FIs is denoted by $f = \sum_{1 \leq i \leq P} f_i$. Because, we do not know f_i in advance, each processor samples $|\widetilde{\mathcal{F}}_s|$ frequent itemsets using the reservoir sampling algorithm, producing $\widetilde{\mathcal{F}}_s$, and counts the number of FIs computed by the sequential algorithm. When the reservoir sampling finishes, each processor p_i sends f_i to p_1. p_1 picks P random variables $X_i, 1 \leq i \leq P$ from multivariate hypergeometrical distribution with parameters $M_i = f_i$. The value of X_i is send to p_i. p_i then choose X_i itemsets $U \in \widetilde{\mathcal{F}}_s$ at random out of the $|\widetilde{\mathcal{F}}_s|$ sampled frequent itemsets computed by p_i. The samples are then send to processor p_1. p_1 stores the received samples in $\widetilde{\mathcal{F}}_s$. This procedure guarantees us the bounds made in the Theorem 5.3.

7.2 Detailed description of Phase 2

In Phase 2 the method partitions $\mathcal{F}$ sequentially on processor p_1. As an input of the partitioning, we use the samples $\widetilde{\mathcal{F}}_s$, the database $\widetilde{\mathcal{D}}$ (computed in Phase 1), and a real number $\alpha, 0 < \alpha \leq 1$.

The process of creation of prefixes follows directly from Proposition 2.4 and Corollary 2.5. The size of each created PBEC is estimated using $\widetilde{\mathcal{F}}_s$. We create PBECs with relative size $\leq \alpha \cdot \frac{1}{P}$. The reason is the granularity of the relative sizes of PBECs.

In Chapter 2, we defined without loss of generality a single order of $b_i \in \mathcal{B}$: $b_1 < b_2 < \ldots < b_{|\mathcal{B}|}$. But: a sequential DFS algorithms (like Eclat and FPGrowth) expands every prefix W_k using the extensions Σ_k sorted by the support in ascending order by the support of $b, b' \in \Sigma_k$ and $b < b'$ if and only if $Supp(W \cup \{b\}, \mathcal{D}) < Supp(W \cup \{b'\}, \mathcal{D})$, i.e., each prefix W_k can have different order of the extensions Σ_k. The dynamic re-ordering of items can significantly reduce the execution time of the sequential algorithm executed in Phase 4. To make the parallel algorithm fast, we have to use the same order as the sequential algorithm for mining of FIs. To make the order the same as the sequential algorithm, we esti-

mate the order of extensions Σ_k for prefix W_k using the supports from $\widetilde{\mathcal{D}}$, i.e., $Supp(W \cup \{b\}, \widetilde{\mathcal{D}}), b \in \Sigma_k$. The different order of items for different prefix does not influence the output of a sequential algorithm for mining of FIs.

The only problem that remains to show is how to schedule the created PBECs. That is: we need to create index sets L_i, such that $F_i = \bigcup_{k \in L_i}([U_k] \cap \mathcal{F})$ and $\max_i |F_i|/|\mathcal{F}|$ is minimized, i.e., we want to schedule $\sum_i |L_i|$ tasks on P equivalent processors. The scheduling task is known NP-complete problem with known approximation algorithms. We use the LPT-Schedule algorithm (LPT stands for least processing time). The LPT-Schedule algorithm, see [13], is a best-fit algorithm: it schedules the largest PBECs on the least loaded processor.

Lemma 7.1. *[13]* LPT-Schedule *is* 4/3-*approximation algorithm.*

Let OPT be the time of the optimum schedule. The lemma says that the LPT-Schedule algorithm finds a schedule with the time at most $4/3 \cdot \text{OPT}$.

The index sets L_i together with U_k and Σ_k are then broadcast to all processors.

7.3 Detailed description of Phase 3

The input of Phase 3 for a processor p_i is the set of indexes of the assigned PBECs L_i together with the prefixes U_k and its extensions Σ_k, created in Phase 2. The processor p_i needs for the computation of $F_i = \bigcup_{k \in L_i}([U_k] \cap \mathcal{F})$ a database partition D'_i that contain all the information needed for computation of F_i. For the description of the algorithm of Phase 3, we expect that we have a distributed memory machine whose nodes are interconnected using a network such as Myrinet[24] or Infiniband[16], i.e., a network that is not congested while an arbitrary permutation of two nodes communicates with each other. The problem is the congestion of the network in Phase 3.

To construct D'_i on processor p_i, every processor $p_j, i \neq j$, has to send a part of its database partition D_j needed by the other processors to all other processors (an all-to-all scatter takes place[3]). That is: processor p_i send to processor p_j the set of transactions $\{t | t \in D_i, k \in L_j, \text{ and } U_k \dot{\subseteq} t\}$, i.e., all transactions that contain at least one $U_k, k \in L_j$ as a subset. Each processor p_i then has the database part $D'_i = \bigcup_\ell \{t | t \in D_\ell, k \in L_i, \text{ and } U_k \dot{\subseteq} t\} = \{t | t \in \mathcal{D}, \text{exists } k \in L_i, U_k \dot{\subseteq} t\}$. Each round of the all-to-all scatter is done in $\lfloor \frac{P}{2} \rfloor$ parallel communication steps. We can consider the scatter as a round-robin tournament of P players [28]: each processor is a player and has to play(exchange data) with all other players.

7.4 Detailed description of Phase 4

The input to this phase, for processor $p_q, 1 \leq q \leq P$, is the database partition D_q (the database partition that is the input of the whole method, the database partition), the set $Q = \{(U_k, \Sigma_k) | U_k \subseteq \mathcal{B}, \Sigma_k \subseteq \mathcal{B}, U_k \cap \Sigma_k = \emptyset\}$ of prefixes U_k and the extensions Σ_k (forming disjoint PBECs $[U_k|\Sigma_k]$), and the sets of indexes L_q of prefixes U_k and extensions Σ_k assigned to processor p_q, and $D'_q = \bigcup_{1 \leq i \leq P} \{t | t \in \mathcal{D}, \text{such that for each } k \in L_q \text{ holds } U_k \dot{\subseteq} t\}$ (the database received in Phase 3 from other processors).

In Phase 4, we execute an arbitrary algorithm for mining of FIs. The sequential algorithm is run on processor p_q for every prefix and extensions $(U_k, \Sigma_k) \in Q, k \in L_q$ assigned to the processor, i.e., p_q enumerates all itemsets $W \in [U_k|\Sigma_k], k \in L_q$. Therefore, the datastructures used by a sequential algorithm, must be prepared in order to execute the sequential algorithm for mining of FIs with particular prefix and extensions. To make the parallel execution of a DFS algorithm fast, we prepare the datastructures by simulation of the execution of the sequential DFS algorithm, e.g., to enumerate all FIs in a PBEC $[U_k|\Sigma_k]$ Phase 4 simulates the sequential branch of a DFS algorithm for mining of FIs up to the point the sequential algorithm can compute the FIs in $[U_k|\Sigma_k]$.

[3] all-to-all scatter is a well known communication operation: each processor p_i sends a message m_{ij} to processor p_j such that $m_{ij} \neq m_{ik}, i \neq k$

7.5 The summary of the new parallel FIMI methods

From the previous discussion it follows that we can create three parallel FIMI methods. Two of the methods are based on the *modified coverage algorithm*. We call these two methods Parallel-FIMI-Seq($\widetilde{\mathcal{M}}$ computed sequentially) and Parallel-FIMI-Par (set M, such that $\widetilde{\mathcal{M}} \subseteq M$, computed in parallel). The third method that leverages the *reservoir sampling*, called Parallel-FIMI-Reservoir. The Parallel-FIMI-Reservoir computes the sample $\widetilde{\mathcal{F}}_s$ in parallel. The methods can be parametrized using an arbitrary algorithm for mining of MFIs and/or an arbitrary algorithm for mining of FIs. The pseudocode showing the overall process is shown in Method 1. The Parallel-FIMI method becomes one of the three methods by using appropriate sampling method at line 5 in Method 1.

Method 1 The Parallel-FIMI method scheme

```
Parallel-FIMI(In: Integers N_D̃, N_F̃s, In: Double α,
              Out: Sets F_i)
 1 for all p_i do-in-parallel
 2   // Phase 1: sampling.
 3   Read D_i.
 4   create a database sample of size N_D̃
 5   create a sample of FIs F̃_s of size N_F̃s (in parallel on all
     processors or on p_1)
 6   // Phase 2: partitioning.
 7   p_1 creates prefixes U_k and its extensions Σ_{U_k} of PBECs
     such that |[U_k|Σ_{U_k}] ∩ F̃_s|/|F̃_s| < α · 1/P and stores them
     in X = {(U_k, Σ_{U_k})}.
 8   p_1 partitions F on disjoint sets F_ℓ, 1 ≤ ℓ ≤ P by
     scheduling the PBECs [U, Σ], (U, Σ) ∈ X, using the
     LPT-Makespan algorithm.
 9   p_1 stores to L_ℓ the indexes of the prefixes and its exten-
     sions from X assigned to p_ℓ
10   // Phase 3: data distribution.
11   send to p_ℓ, ℓ ≠ i, database part D'_ℓ = {t|t ∈ D_i, k ∈
     L_ℓ, and U_k ⊆̇ t}.
12   // Phase 4: execution of arbitrary sequential algorithm
     for computation of FIs.
13   execute an arbitrary algorithm for mining of FIs in the
     assigned PBECs F_i
14 end for
```

8 Experimental evaluation

We have proposed a method for dynamic load-balancing of an arbitrary DFS or BFS sequential algorithm for mining of FIs. In this section, we evaluate the speedup of our method on databases generated by the IBM generator.

8.1 Implementation and experimental setup

We have implemented our methods using the `C++` language and the `g++` compiler version 4.4.3 with the `-O4` option (highest optimalizations on speed of the resulting code). As the sequential algorithm, we have used the Eclat algorithm [34, 11]. As the algorithm for mining of MFIs, we have chosen the *fpmax** [14] algorithm. As the algorithm for mining of FIs, we have chosen the Eclat algorithm [34, 11].

The *fpmax** algorithm was downloaded from [12] and the Eclat algorithm was downloaded from [11]. Both algorithms were modified, so we can run them in parallel using dynamic load-balancing (balancing on prefixes of size 1). We have preformed all the experiments with our methods on a cluster of workstations interconnected with the Infiniband network. Every node in the cluster has two dual-core 2.6GHz AMD Opteron processors and 8GB of main memory.

8.2 Datasets

The experiments were performed on datasets generated using the IBM database generator – a standart way of assessing parallel algorithms. We have used datasets with 500k transactions and supports for each dataset such that the sequential run of the Eclat algorithm is between 100 and 12000 seconds ($\approx$ 3.3 hours) and two cases with running time 33764 seconds (9.37 hours) and 132186 seconds (36.71 hours). The IBM generator is parametrized by the average transaction length `TL` (in thousands), the number of items `I` (in thousands), by the number of patterns `P` used for creation of the parameters, and by the average length of the patterns `PL`. To clearly differentiate the parameters of a database we are using the string `T[number in thousands]I[items count in`

Dataset	Supports
TXI0.1P100PL20TL50	$0.11, 0.12, 0.13, 0.14, 0.15, 0.16, 0.17, 0.18$
TXI0.1P250PL10TL40	$0.05, 0.07, 0.09, 0.1$
TXI0.1P50PL10TL40	$0.09, 0.1, 0.13, 0.15, 0.18$
TXI0.1P50PL20TL40	$0.05, 0.07, 0.09, 0.1$
TXI0.4P250PL10TL120	$0.2, 0.25, 0.26, 0.27, 0.3$
TXI0.4P250PL20TL80	$0.02, 0.03, 0.05, 0.07, 0.09$
TXI0.4P50PL10TL40	$0.02, 0.05, 0.07, 0.09$
TXI1P100PL20TL50	$0.02, 0.03, 0.05, 0.07, 0.09$

Table 2: Databases used for measuring of the speedup and used supports values for each dataset, $X \in \{500, 1000, 2000, 3000\}$.

`1000]P[number]PL[number]TL[number]`, e.g. the string `T500I0.4P150PL40TL80` labels a database with 500K transactions 400 items, 150 patterns of average length 40 and with average transaction length 80. All speedup experiments were performed with various values of the support parameter on 2, 4, 6, 10, 16, and 20 processors. The databases and supports used for evaluation of our methods is summarized in the Table 2.

We have chosen the parameters of the IBM generator so that the distribution of lengths of FI, the lengths of intersections of MFI, and lengths of MFI for particular support of min_support are similar to the same characteristics of some real databases. For details, see Appendix A.* There is a possibility to take a real database $\mathcal{D} = \{t_i\}$ and create a bigger database $\mathcal{D}' = \{t'_\ell = (\mathrm{id} \times \ell, U)|$ for each $t = (\mathrm{id}, U) \in \mathcal{D}, 1 \leq \ell \leq N\}$ by N time replication of the database $\mathcal{D}$. The reason is that some algorithms could behave in the same way when run on $\mathcal{D}$ or $\mathcal{D}'$, e.g., the FPGrowth algorithm and the *fmpax** algorithm that is based on the FPGrowth algorithm.

8.3 Evaluation of the estimate of the size of PBECs

In the previous sections, we have shown that the parallel mining of FIs is a two stage sampling process. Theorem 5.4 and Corollary 5.5 suggests that the results of the double sampling process can be very bad. In this section, we show that the results are not that pesimistic.

We use the notation from our previous sections: by U_j, we denote the prefixes of PBECs, by L_i we denote a set of indexes of prefixes assigned to processor p_i. The indexsets L_i are chosen as described in Section 7.2, i.e., $\frac{|\bigcup_{j \in L_i} [U_j] \cap \widetilde{\mathcal{F}}_s|}{|\widetilde{\mathcal{F}}_s|} \approx 1/P$.

For each dataset we have measured the error of the estimate of the amount of work per processor: for a set of prefixes $\{U_k\}$ and for P processors such that $\frac{|\bigcup_{j \in L_i} [U_j] \cap \widetilde{\mathcal{F}}_s|}{|\widetilde{\mathcal{F}}_s|} \approx 1/P$, we show a graph of probability of the error $\left| \frac{1}{P} - \frac{|\bigcup_{j \in L_i} [U_j] \cap \mathcal{F}|}{|\mathcal{F}|} \right|$. This graph is very important for our work.

The Figures 2–4 show the size of the union of the PBECs created in Phase 2. There are four lines per graph: combination of dashed and solid line with red and blue color. The red color indicates measurement with $|\widetilde{\mathcal{D}}| = 42856$ and the blue color indicates measurements with $|\widetilde{\mathcal{D}}| = 14450$. The solid line shows the probability of the error with $|\widetilde{\mathcal{F}}_s| = 1001268$ and the dashed line shows the probability of the error with $|\widetilde{\mathcal{F}}_s| = 26492$. The left-hand graph show the measurements for $P = 5$ and the right hand graph show the measurements for $P = 10$. The graphs shows a summarization of the errors for different values of supports. It can be seen from the graphs that the larger database sample the smaller the probability of the error. The probability of the error is similar for different size of $|\widetilde{\mathcal{F}}_s|$.

In addition to the measurements, we have computed for each dataset the number of PBECs that make 96% of the total number of FIs. We denote the set of the prefixes of the 96% of PBECs by $S = \{U\}$, i.e., $\sum_{U \in S} |[U] \cap \mathcal{F}| \geq 0.96 \cdot |\widetilde{\mathcal{F}}_s|$. We have discovered that 96% of all FIs are contained in $\approx$ 100–200 PBECs (the number of all PBECs varies from $\approx$ 300–3000). Let $V_{min} = \arg\min_{W \in S} |[W] \cap \mathcal{F}|$ be the prefix of the smallest PBEC, we have measured the relative size of the smallest PBEC $|[V_{min}] \cap \mathcal{F}| \approx 0.0007$-$0.003$. Therefore, the value of ρ can be chosen between 0.0007-0.003, depending on the dataset.

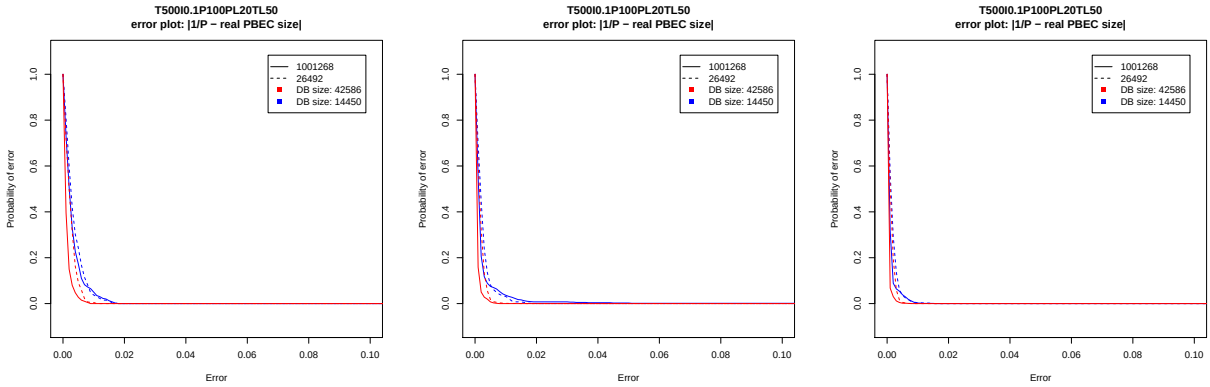


Figure 2: Probability of error of the estimation of the union of PBECs using a database sample created in Phase 1 and 2. Experiments made using 5 processors (left) and 10 processors (right). The `T500I0.1P100PL20TL50` dataset.

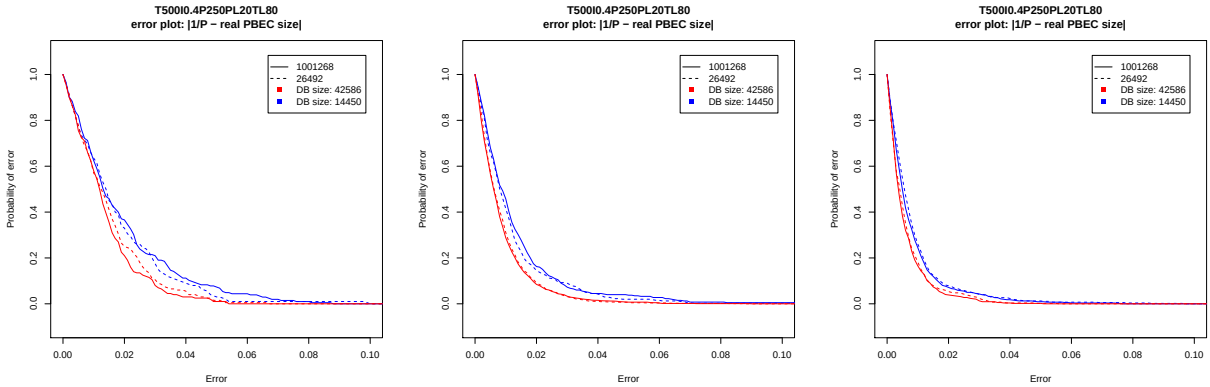


Figure 3: Probability of error of the estimation of the union of PBECs using a database sample created in Phase 1 and 2. Experiments made using 5 processors (left) and 10 processors (right). The `T500I0.4P250PL20TL80` dataset.

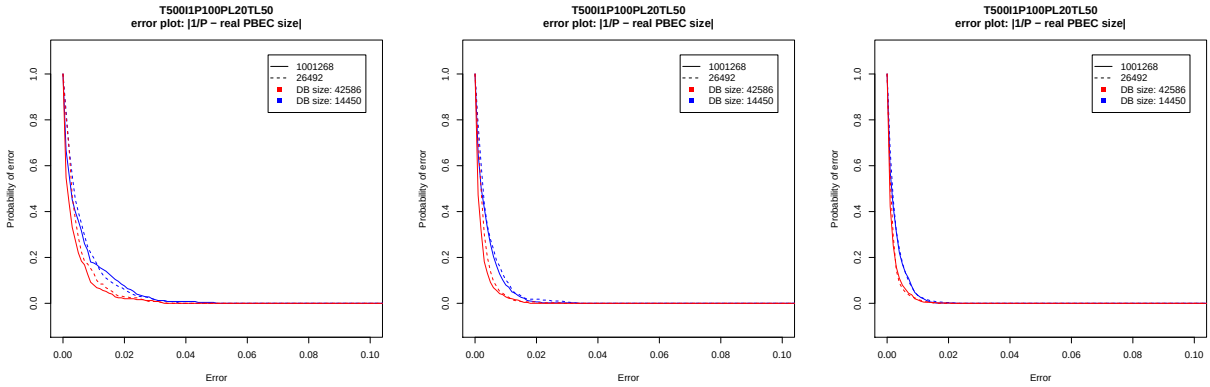


Figure 4: Probability of error of the estimation of the union of PBECs using a database sample created in Phase 1 and 2. Experiments made using 5 processors (left) and 10 processors (right). The `T500I1P100PL20TL50` dataset.

Variant of our method	$\lvert\widetilde{\mathcal{D}}\rvert$	$\lvert\widetilde{\mathcal{F}}_s\rvert$
Parallel-FIMI-Reservoir	10000	19869
Parallel-FIMI-Par	10000	33115
Parallel-FIMI-Seq	10000	19869

Table 4: Best combintations of $\lvert\widetilde{\mathcal{D}}\rvert$ and $\lvert\widetilde{\mathcal{F}}_s\rvert$ for $P = 20$

8.4 Evaluation of the speedup

Two of the proposed parallel methods, namely the Parallel-FIMI-Seq method and the Parallel-FIMI-Par method, need to compute the MFIs $\widetilde{\mathcal{M}}$ from a database sample $\widetilde{\mathcal{D}}$. In the expriments, in Phase 1, we use the *fpmax** [14] algorithm that computes the MFIs. In the case of the Parallel-FIMI-Seq the *fpmax** algorithm is executed sequentially on processor p_1. In the case of the Parallel-FIMI-Par, we execute the *fpmax** algorithm in parallel.

We have evaluated our methods on various combinations of $\lvert\widetilde{\mathcal{D}}\rvert$ and $\lvert\widetilde{\mathcal{F}}_s\rvert$. The combinations of $\lvert\widetilde{\mathcal{F}}_s\rvert$ and $\lvert\widetilde{\mathcal{D}}\rvert$ are shown in Table 3. The best combination of parameters for $P = 20$ are summarized in Table 4. Additionally to the datasets with size $500'000$ transactions, we have choosed the best combination of sampling parameters and made experiments with the Parallel-FIMI-Reservoir on datasets of size $1'000'000$, $2'000'000$, and $3'000'000$.

We show an example of speedup graphs for $\lvert\widetilde{\mathcal{D}}\rvert = 10000$ and $\widetilde{\mathcal{F}}_s = 19869$ in Figures 5–10 shows the Figures 5–10 clearly demonstrate that for the smallest databases with reasonable structure, the speedup is up to ≈ 12 on 20 processors. The Parallel-FIMI-Seq achieves speedup up to ≈ 8 on 20 processor, the Parallel-FIMI-Par method achieves maximal speedup up to ≈ 11 on 20 processors and the Parallel-FIMI-Reservoir method achieves speedup up to ≈ 13 on 20 processors. The speedup for datasets of size $> 500'000$, using the Parallel-FIMI-Reservoir, have bigger values of speedups up to ≈ 17 on 20 processors. The speedups are usually bigger with lower values of support. In the case of `T3000I0.4P250PL10TL120` the communication network get congested on 20 processors, see Figure 8.

The speedups 0, in the graphs, indicates that the program run out of memory. The reason of the memory exhaustion is the large amount of upper bounds used in the coverage algorithm, see Theorem 6.4. That is: if the program implementing the Parallel-FIMI-Seq method runs out of memory then the program implementing the Parallel-FIMI-Par method also runs out of main memory. The program implementing the Parallel-FIMI-Reservoir method never runs out of memory be-

$\lvert\widetilde{\mathcal{D}}\rvert$	10000	10000	10000	14450	14450	14450	14450	14450	20000	20000	20000
$\lvert\widetilde{\mathcal{F}}_s\rvert$	19869	26492	33115	13246	19869	26492	33115	39738	19869	26492	33115

Table 3: Sizes of $|\widetilde{\mathcal{D}}|$ and $|\widetilde{\mathcal{F}}_s|$ used in our experiments.

cause it need not to store the MFIs in main memory. The evaluation of the sampling process in Section 8.3 shows that the estimates are quite good. The question is, why the speedup is not almost linear with the value of the speedup being close to the number of processors? The answer to this question is obvious: making the sample takes some time. Additionally, we can observe that lower values of $min\_support^*$ makes better speedup with the two cases for the `T500I0.4P250PL20TL80` dataset having a very good speedup of ≈ 13 on $P = 20$ processors. The reason is obvious: the sampling process taking the same number of sample on the database of the same size makes better speedup, i.e., if it takes more time to compute sequentiall the FIs for given support in the given dataset then the speedup is usually better.

From the graphs on Figures 5–10 and tables on the Tables 5–7 follows that the Parallel-FIMI-Par is usually faster then the Parallel-FIMI-Seq. The average speedup of Parallel-FIMI-Par is better, compared to the Parallel-FIMI-Seq, for $P < 20$. The Parallel-FIMI-Reservoir performs usually better compared to Parallel-FIMI-Par. Still, there is a possibility to improve the speedup of the Parallel-FIMI-Reservoir method. However, the average speedup is always better, compared to the Parallel-FIMI-Seq and Parallel-FIMI-Par methods, see Tables 5–7. The bold values in the tables for the Parallel-FIMI-Par and Parallel-FIMI-Seq represents the better speedup of the two methods. The value in the Table for the speedup if Parallel-FIMI-Reservoir is bold if it is the best average speedup from the three methods. Additionally, there is an advantage of the Parallel-FIMI-Reservoir over the two other methods: the need of computation of MFIs. The number of MFIs can be very large and the program implementing the Parallel-FIMI-Seq method or the Parallel-FIMI-Par can run out of main memory. This happens for some supports of the following datasets: `T500I0.4P250PL20TL80`, `T500I0.4P50PL10TL40`, and `T500I1P100PL20TL50`.

Examples of the results of the experiments with datasets of size $> 500'000$ are on Figures 7–8. The average speedups are shown in Tables 8–10.

Other complicated cases are the datasets with 1000 items in the Figure 10. The reason for such a bad speedup lies in Phase 1 and 2. There is always a processor that has much bigger running time in Phase 4. For example, for $min\_support^* = 0.02$ and for $P = 10$ the execution time of Phase 4 is (in seconds): 194, 1199, 319, 245, 536, 357, 477, 212, 332, 212. A sum of these times is 4087 seconds, the sequential algorithm runs ≈ 3800 seconds. The probability of error of the estimates made in Phase 2 of `T500I1P100PL20TL50` are competitive to other datasets, see Figure 4. The best speedup that achieved by Parallel-FIMI-Reservoir is ≈ 8 on 20 processors for $min\_support^* = 0.02$. In other cases the speedup is not so good. The reason of such behaviour is unknown.

Another unexpected behavior is the slighlty super-linear speedup on 2 processors for the datasets `T1000I0.4P250PL10TL120`, `T2000I0.4P250PL10TL120`, and `T3000I0.4P250PL10TL120`. We are sure that we do not have an error on our implementation due to several testing of our implementation: 1) in some cases comparison of the output with the sequential algorithm; 2) comparison of the number of all FIs of the parallel algorithm against the number of FIs of the sequential algorithm. The hypothesis for such behaviour is the following: due to the usage of diffsets and the fact that the ordering of items for the PBECs for the Phase 4 is taken from the database sample, we can have slightly different ordering of items. This ordering of items is more efficient in this case. Additionally, these datasets show the bottleneck of the algorithm hidden in Phase 3.

Another consequence of the different ordering of

items can be also a slow-down of the execution of the sequential algorithm in the Phase 4, resulting in lower speedups. Generally speaking: in some cases it may not be easy to use the sample for estimation of the amount of work in a single PBEC resulting in need to add weight for each sample. The weighting then may be specific for certain domain, or type of data.

8.5 The evaluation of the database replication experiments

Let D_i be the database part obtained by processor p_i in Phase 3, i.e., the database partition loaded by p_i in Phase 1 and the transactions received by p_i in Phase 3. We define the database replication factor by: $\frac{\sum_{i=1}^{P}|D_i|}{|\mathcal{D}|}$.

The LPT-SCHEDULE algorithm assigns the prefix-based equivalence classes to the processors based solely on their sizes. If we want to minimize the database replication, we have to consider the mutual sharing of the database portions among the prefix-based classes. In this sub-section, we will show how the problem of scheduling of the PBECs with respect to the mutual share of transactions is related to the Quadratic Knapsack Problem (QKP in short) and the values of the database replication factor. For a good source of information on knapsack problems, see [18].

The QKP can be defined as follows: let have n items and the j-th item having a positive integer weight w_j, and a limit on the total weight of the chosen items is given by a positive integer knapsack capacity c. In addition, we have a $n \times n$ *profit matrix* $S = (S_{ij})$, where S_{ij} is the profit of having item i together with item j in the knapsack. Additionally, we have indicator variables $x_i \in \{0, 1\}$ where $x_i = 1$ if the item i was selected to the knapsack and 0 otherwise. The QKP selects subset of items that fit in the knapsack and have maximal profit. The problem can be stated in the following way:

$$\begin{aligned} \text{maximize} \quad & \sum_i \sum_j S_{ij} x_i x_j \\ \text{subject to} \quad & \textstyle\sum_j w_j x_j \leq c \end{aligned}$$

We can reformulate the QKP in the terms of our problem: let have a list of prefixes $P = \{U_i | U_i \subseteq \mathcal{B}\}$. The *profit matrix* S, contains the number of shared transactions for every two PBECs, i.e., $S_{ij} = Supp(U_i \cup U_j), i \neq j$ and $S_{ii} = 0$. The weight w_i is defined as the size of the prefix-based class $[U_i] \cap \mathcal{F}$. The size $|[U_i] \cap \mathcal{F}|$ is determined by the relative number of samples $\widetilde{\mathcal{F}}_s$ belonging to $[U_i]$, i.e., $|[U_i] \cap \widetilde{\mathcal{F}}_s|/|\widetilde{\mathcal{F}}_s|$. The task is to put prefix-based equivalence classes into the knapsack, such that the size of the knapsack $c = \sum_i s_i/P$ while maximizing the share of transactions. This task is the same as solving the QKP. When we have a set of prefixes, we assign them to a processor, remove them from the set Q, update the matrix and the weight vector, and repeat the process until we assign all the prefix-based classes. We can assign some PBECs to a processor p_i and repeat the process of assigning the PBECs not yet assigned to other processor then p_i.

Due to the randomnes of the data generated by the IBM generator the replication factor is almost P on P processors. Therefore, we have used real datasets downloaded from the internet [12]: `kosarak`, `accidents`, `chess`, `connect`, `mushroom`, `pumsb_star`, and `pumsb` [12]. As the implementation of the QKP algorithm, we have downloaded the source code from [27], which is the implementation of the algorithm from [5].

The results of the experiments are summarized in tables. For each dataset there are three tables: improvement of the QKP scheduling against the LPT-MAKESPAN algorithm, the database replication using the greedy schedule, and the database replication using the QKP schedule. We have chosen the number of processors: 4, 6, 10, and 14.

The biggest improvement of the database replication (28%) is on the `mushroom` dataset. It can be seen that the biggest improvement is at the relative support level 0.001. The improvements are much smaller, when the relative support is > 0.01. The `mushroom` dataset is also one of the two datasets where we have achieved a replication factor after minimalization $\ll P - 1$ (for $P = 14$ processors). The lowest replication factor 2.7 on 14 processors was measured on the `mushroom` dataset. In most cases the replication factor is between $P - 1$ and P for the

datafile/Parallel-FIMI-Seq	2	4	6	10	16	20
T500I0.4P50PL10TL40	1.349	1.918	2.423	**3.037**	**3.532**	**3.578**
T500I0.1P50PL20TL40	1.417	2.399	3.040	4.280	6.113	6.761
T500I0.4P250PL20TL80	0.771	1.471	1.906	2.928	4.108	4.703
T500I1P100PL20TL50	1.020	**1.520**	1.824	**1.939**	2.281	**2.273**
T500I0.1P50PL10TL40	1.345	**2.264**	**3.103**	4.538	**6.386**	7.385
T500I0.4P250PL10TL120	0.759	1.471	2.101	3.044	4.159	4.985
T500I0.1P100PL20TL50	1.062	1.832	**2.413**	3.660	5.349	**6.110**
T500I0.4P150PL40TL80	0.965	1.635	2.282	3.163	4.121	**4.658**
T500I0.1P250PL10TL40	0.985	1.724	2.285	3.513	4.830	5.778

Table 5: Average speedups of Parallel-FIMI-Seq for $|\widetilde{\mathcal{D}}| = 10000$, $|\widetilde{\mathcal{F}}_s| = 19869$, and for database with $500'000$ transactions.

datafile/Parallel-FIMI-Par	2	4	6	10	16	20
T500I0.4P50PL10TL40	**1.524**	**2.209**	**2.889**	2.649	2.972	3.255
T500I0.1P50PL20TL40	**1.502**	**2.553**	**3.452**	**4.829**	**6.372**	**7.973**
T500I0.4P250PL20TL80	**0.945**	**1.838**	**2.882**	**3.679**	**4.813**	**5.374**
T500I1P100PL20TL50	**1.116**	1.498	**1.980**	1.697	**2.763**	2.051
T500I0.1P50PL10TL40	**1.371**	2.244	2.633	**4.862**	6.194	**7.461**
T500I0.4P250PL10TL120	**0.879**	**2.030**	**2.549**	**4.011**	**5.553**	**6.273**
T500I0.1P100PL20TL50	**1.122**	**1.932**	2.237	**3.869**	**5.583**	5.885
T500I0.4P150PL40TL80	**1.076**	**1.955**	**2.591**	**3.630**	**4.776**	3.156
T500I0.1P250PL10TL40	**1.100**	**1.898**	**2.791**	**3.954**	**5.525**	**6.239**

Table 6: Average speedups of Parallel-FIMI-Par for $|\widetilde{\mathcal{D}}| = 10000$, $|\widetilde{\mathcal{F}}_s| = 19869$ and for database with $500'000$ transactions.

datafile/Parallel-FIMI-Reservoir	2	4	6	10	16	20
T500I0.4P50PL10TL40	**1.628**	**2.347**	**3.372**	**4.776**	**6.202**	**6.893**
T500I0.1P50PL20TL40	1.455	2.550	3.093	4.076	5.670	6.342
T500I0.4P250PL20TL80	**1.453**	**2.757**	**3.965**	**5.761**	**6.208**	**8.430**
T500I1P100PL20TL50	**1.121**	**1.923**	**2.374**	**2.841**	**4.026**	**5.753**
T500I0.1P50PL10TL40	**1.380**	**2.521**	**3.281**	4.464	**6.772**	**8.860**
T500I0.4P250PL10TL120	**1.216**	**2.249**	**3.117**	**4.584**	**6.179**	**7.067**
T500I0.1P100PL20TL50	**1.172**	**2.071**	**2.767**	3.802	**6.840**	**8.967**
T500I0.4P150PL40TL80	**1.211**	**2.155**	**2.858**	**3.789**	**4.976**	**5.423**
T500I0.1P250PL10TL40	**1.243**	**2.161**	**2.930**	**4.606**	**6.863**	**7.793**

Table 7: Average speedups of Parallel-FIMI-Reservoir for $|\widetilde{\mathcal{D}}| = 10000$ and $|\widetilde{\mathcal{F}}_s| = 19869$

datafile/Parallel-FIMI-Reservoir	2	4	6	10	16	20
T3000I0.4P50PL10TL40	1.869	2.757	3.324	4.324	5.477	5.802
T3000I0.1P50PL20TL40	1.958	3.120	4.531	5.659	7.858	8.345
T3000I0.4P250PL20TL80	1.910	3.557	4.944	7.324	10.205	11.060
T3000I1P100PL20TL50	1.731	2.838	3.891	5.412	8.158	9.696
T3000I0.1P50PL10TL40	1.969	3.441	4.771	6.569	10.604	13.357
T3000I0.4P250PL10TL120	2.205	3.927	5.229	8.250	10.292	7.192
T3000I0.1P100PL20TL50	1.920	3.278	4.377	6.046	11.572	14.358
T3000I0.4P150PL40TL80	1.773	3.185	4.427	6.332	8.061	8.475
T3000I0.1P250PL10TL40	1.914	3.568	4.879	7.180	11.626	14.160

Table 8: Average speedups of Parallel-FIMI-Reservoir on database of size $3'000'000$ for $|\widetilde{\mathcal{D}}| = 10000$ and $|\widetilde{\mathcal{F}}_s| = 19869$

datafile/Parallel-FIMI-Reservoir	2	4	6	10	16	20
T2000I0.4P50PL10TL40	1.792	2.829	3.342	4.655	4.759	5.465
T2000I0.1P50PL20TL40	1.865	3.218	3.920	5.919	8.675	8.650
T2000I0.4P250PL20TL80	1.935	3.166	4.767	6.775	8.884	9.315
T2000I1P100PL20TL50	1.861	2.890	4.078	4.799	8.146	10.081
T2000I0.1P50PL10TL40	1.951	3.426	4.462	6.604	8.563	12.164
T2000I0.4P250PL10TL120	2.134	3.773	5.543	8.284	10.448	12.141
T2000I0.1P100PL20TL50	1.905	3.613	4.332	6.327	12.093	14.223
T2000I0.4P150PL40TL80	1.938	3.362	4.655	6.531	8.304	9.299
T2000I0.1P250PL10TL40	1.897	3.582	4.369	7.137	11.703	13.817

Table 9: Average speedups of Parallel-FIMI-Reservoir on database of size $2'000'000$ for $|\widetilde{\mathcal{D}}| = 10000$ and $|\widetilde{\mathcal{F}}_s| = 19869$

datafile/Parallel-FIMI-Reservoir	2	4	6	10	16	20
T1000I0.1P50PL10TL40	1.993	3.666	4.403	6.191	10.884	11.991
T1000I0.1P50PL20TL40	1.940	3.472	4.515	5.482	8.544	8.936
T1000I0.4P250PL20TL80	1.935	3.560	4.995	7.497	9.537	10.043
T1000I1P100PL20TL50	1.816	3.438	3.483	4.914	8.180	11.410
T1000I0.1P50PL10TL40	1.993	3.666	4.403	6.191	10.884	11.991
T1000I0.4P250PL10TL120	2.083	3.837	5.289	7.778	10.741	12.229
T1000I0.1P100PL20TL50	1.877	3.530	4.620	6.098	11.651	14.002
T1000I0.4P150PL40TL80	1.841	3.256	4.487	6.548	8.300	8.731
T1000I0.1P250PL10TL40	1.901	3.643	4.576	7.130	11.171	13.736

Table 10: Average speedups of Parallel-FIMI-Reservoir on database of size $1'000'000$ for $|\widetilde{\mathcal{D}}| = 10000$ and $|\widetilde{\mathcal{F}}_s| = 19869$

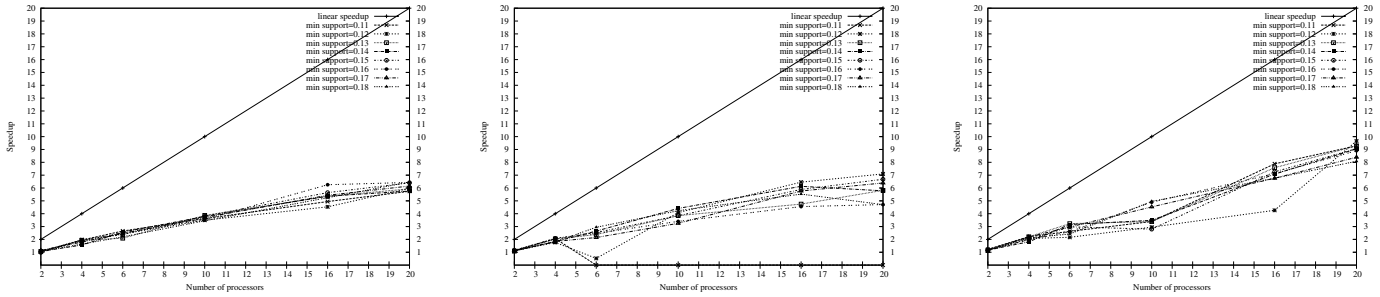


Figure 5: Speedups of the Parallel-FIMI-Seq, Parallel-FIMI-Par, and Parallel-FIMI-Reservoir methods (from left to right) on the `T500I0.1P100PL20TL50` database.

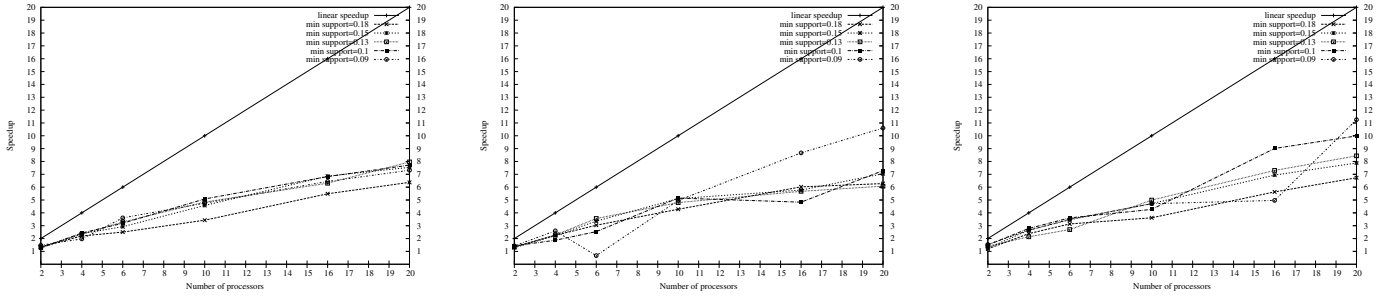


Figure 6: Speedups of the Parallel-FIMI-Seq, Parallel-FIMI-Par, and Parallel-FIMI-Reservoir methods (from left to right) on the `T500I0.1P50PL10TL40` database.

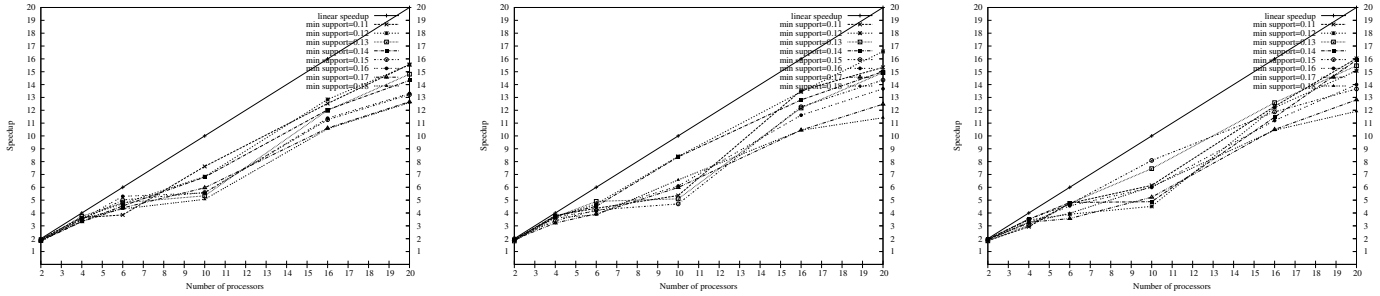


Figure 7: Speedups of the Parallel-FIMI-Reservoir method on `T1000I0.1P100PL20TL50`, `T2000I0.1P100PL20TL50`, and `T3000I0.1P100PL20TL50` datasets (from left to right).

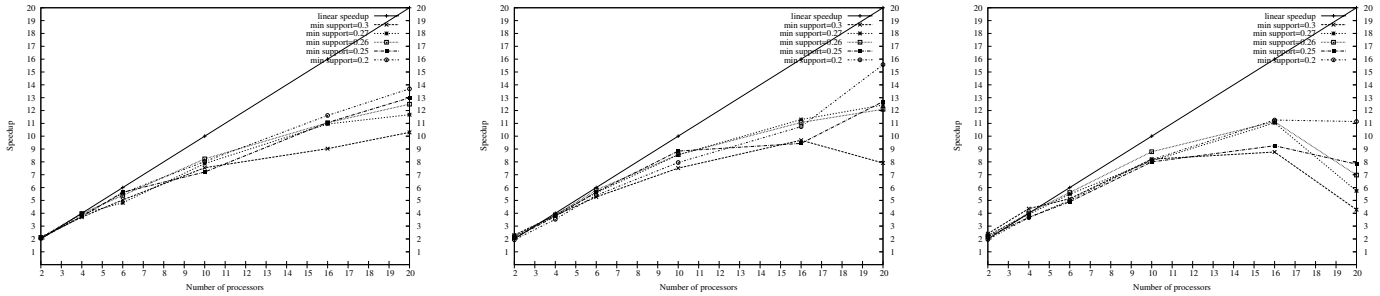


Figure 8: Speedups of the Parallel-FIMI-Reservoir method on `T1000I0.4P250PL10TL120`, `T2000I0.4P250PL10TL120`, and `T3000I0.4P250PL10TL120` datasets (from left to right).

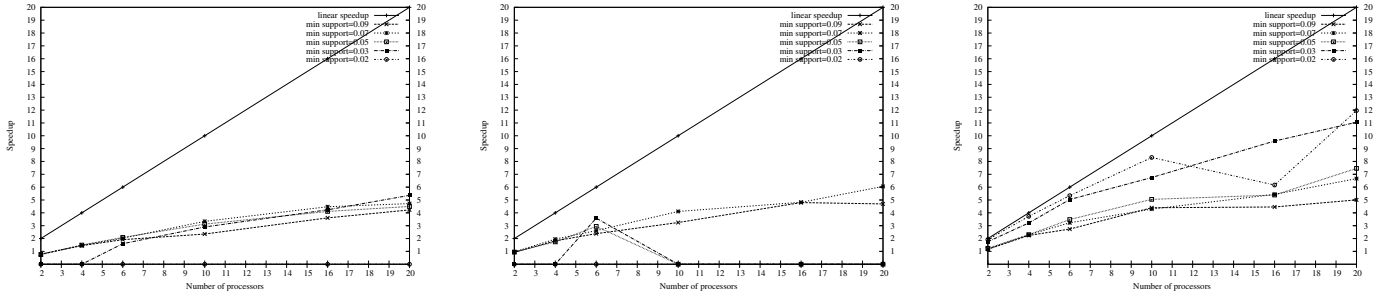


Figure 9: Speedups of the Parallel-FIMI-Seq, Parallel-FIMI-Par, and Parallel-FIMI-Reservoir methods (from top to bottom) on the `T500I0.4P250PL20TL80` database.

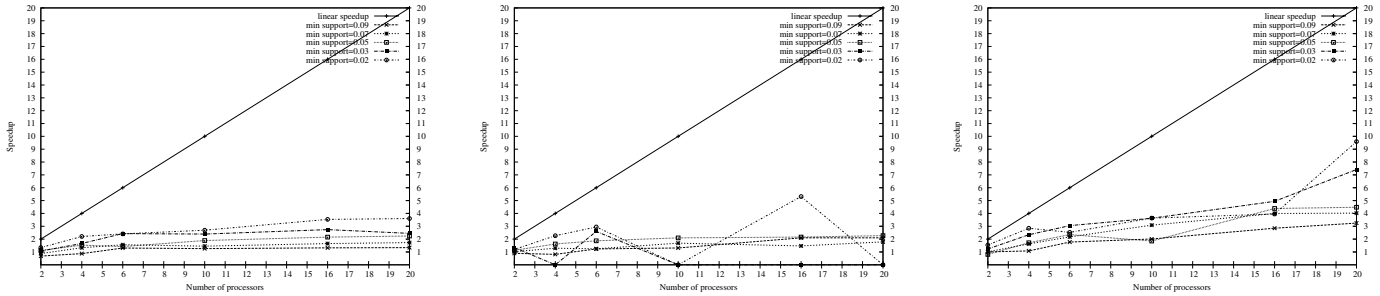


Figure 10: Speedups of the Parallel-FIMI-Seq, Parallel-FIMI-Par, and Parallel-FIMI-Reservoir methods (from top to bottom) on the `T500I1P100PL20TL50` database.

datasets generated by the IBM generator and some of the real datasets.

Overall, the improvement of the replication factor mostly ranges between $\approx 1\%$ and $\approx 13\%$. It sometimes happens that the replication factor is worse after minimalization. The worsening is for the `pumsb` dataset -0.0464%, `pumsb_star` -2.2881% and -0.2538%. We consider these values as outliers.

Generally it holds that for two processors the database replication is very high, but mostly does not reach P for P processors. However, in most cases the replication factor is between $P-1$ and P. The most interesting case is the `mushroom` dataset. From the experiments it can be seen that the lower the support the better results. The best database replication factor is ≈ 10 on 14 processors for the `mushroom` dataset.

P/*min_support**	0.0050	0.0040	0.0030
4	12.6096	11.6921	15.9684
6	13.6673	19.4744	20.7549
10	18.0931	18.0157	18.7086
14	17.6054	20.2953	21.7529

Table 11: Database replication improvement (in %)

P/*min_support**	0.0050	0.0040	0.0030
4	1.76357	1.9325	1.86456
6	2.08358	2.14564	2.18368
10	2.36798	2.4311	2.49938
14	2.55512	2.55404	2.74345

Table 12: Database replication *without minimalization.*

P/*min_support**	0.0050	0.0040	0.0030
4	1.54119	1.70655	1.56682
6	1.79881	1.72779	1.73046
10	1.93954	1.99312	2.03178
14	2.10528	2.03569	2.14667

Table 13: Database replication *after minimalization*

9 Conlusions and future work

We have two already published methods for parallel mining of frequent itemsets, namely: the Parallel-FIMI-Seq method [20], the Parallel-FIMI-Par method [21], and the Parallel-FIMI-Reservoir method [19], that is a qualitative major improvement of the two previous methods. The results are important from the following reasons:

1. they are very general, they can be used for parallelization of an arbitrary DFS algorithm for mining of FIs;
2. the proposed static load balancing captures the amount of work for each processor: none of previously proposed methods does this;
3. we are measuring the speedup of our methods against a very fast sequential algorithm, i.e., it is hard to achieve good speedup;

Even that the speedup of the Parallel-FIMI-Reservoir method is only slightly better then the speedup of the previous methods, the new method is a major improvement for the following reasons:

1. the previous two methods suffer from very high memory consumption due to the fact that they have to store the MFIs or a super set of MFIs in main memory;
2. the Parallel-FIMI-Reservoir method needs only to store the sample, which could consume much smaller amount of memory then the MFIs $\widetilde{\mathcal{M}}$ or the superset of all MFIs M;
3. the new method allow us to have bounds on the estimation of the load balancing, see Theorem 5.4 and Theorem 5.3.

Our method can fail in the following case: if the size of the intersections of tidlists(or diffsets) differ substantialy in each PBEC then the sample does not represent the amount of work in each PBEC.

There is still room for improvement: instead of using the Eclat algorithm together with the reservoir sampling algorithm, we can use a modified *fpmax** algorithm or any other algorithm for mining of MFIs together with the reservoir sampling algorithm. That is: we can use similar techniques of optimalizations used in the MFI mining algorithm. The reason is, that we do not need to compute the support of the FIs, we need only to enumerate the FIs. This approach can be used only in the case when the relative

$P/min\_support^*$	0.1	0.08	0.06	0.04	0.02	0.001
4	0.6155	0.8617	0.5663	1.0093	4.7883	11.0753
6	2.6015	3.9635	1.4903	1.8668	2.1500	14.6110
10	3.8776	3.2556	3.5445	9.6659	8.0022	22.7943
14	5.8516	5.9913	7.9623	7.7287	10.0239	28.9319

$P/min\_support^*$	0.1	0.08	0.06	0.04	0.02	0.001
4	4	4	4	4	4	4
6	5.99951	6	5.99606	6	6	6
10	9.93599	9.98929	9.98769	9.99926	9.99852	9.96972
14	13.9791	13.8902	13.9357	13.979	13.9547	13.8765

$P/min\_support^*$	0.1	0.08	0.06	0.04	0.02	0.001
4	3.97538	3.96553	3.97735	3.95963	3.80847	3.55699
6	5.84343	5.76219	5.9067	5.88799	5.871	5.12334
10	9.55071	9.66408	9.63368	9.03274	9.19842	7.69719
14	13.1611	13.058	12.8261	12.8986	12.5559	9.86177

Table 14: Improvement of the database replication of the `mushroom` dataset.

number of samples in a PBEC represents the amount of work for enumeration of that PBEC.

Another possibility to improve the performance is the weighting of each set $U \in \widetilde{\mathcal{F}}_s$: instead of estimating the amount of work in a PBEC $[W]$ by $[W] \cap \widetilde{\mathcal{F}}_s$, we can weight each $U \in \widetilde{\mathcal{F}}_s$ by the amount of work needed to compute U.

It seems that there is a possibility to use a recently proposed method for approximating the number of frequent sets [3]. In the beginning of our work, we have been considering a similar technique. However, we have dismissed the method due to the fact that in some cases the methods could make a big error. For the same reasons the Parallel-FIMI-Par and Parallel-FIMI-Seq.

Another possibility to obtain the sample $\widetilde{\mathcal{F}}_s$ is to use directly the transactions $t \in \mathcal{D}$ as the input to the modified coverage algorithm or to the unmodified coverage algorithm. However, we have been considering this approach and it fails due to the following reasons: the size of the set $\bigcup_{t \in \mathcal{D}} \mathcal{P}(t)$ is much bigger then the size of $\bigcup_{m \in \mathcal{M}} \mathcal{P}(m)$. Let have two finite sets A, B such that $A \subset B$. Let $\rho = |A|/|B|$. The problem is that the running time is proportional to $1/\rho$.

We believe that our method can be applied also on other frequent pattern mining algorithms such as: mining of frequent trees, mining of frequent sequences, mining of frequent graphs.

A Database characteristics

In this appendix, we explain the database characteristics used for database selection in a more detail.

Mimicking the real dataset using the IBM generator is a hard task. In order to choose databases that are similar to the real databases, we have created the following database characteristics:

1. The distribution of intersections of MFIs: let have a set of MFIs $\mathcal{M}$. We have measured $|m_i \cap m_j|, m_i, m_j \in \mathcal{M}$ for particular choice of $min\_support^*$ and compared the histograms of real databases and databases generated by the IBM generator.

2. The distribution of FIs of certain length: let have a set of FIs $\mathcal{F}$. We have measured $|U|, U \in \mathcal{F}$. We have measured the lengths for various values of $min\_support^*$: we have split the interval $[0, 1]$ on $n = 1000$ values $i \cdot \frac{1}{n}$ for $i = 0, \ldots, n-1$ and compared histograms of real datbases and databases generated by the IBM generator for each value of $i \cdot \frac{1}{n}$.

3. The distribution of lengths of MFI: let have a set of MFIs $\mathcal{M}$. We have drawn the histograms of $|m|, m \in \mathcal{M}$ for various values of $min\_support^*$ and compared the histograms of the databases generated by the IBM generator to the histograms of real databases.

We have chosen the datasets so these characteristics are close to the characteristics of real datasets, e.g., `connect`, `pumsb`, see [12]. The only exception to this choice is the `T500I1P100PL20TL50` dataset. We omit details of the measurements because they are out of the scope of this paper.

References

[1] R. Agrawal and J. C. Shafer. Parallel mining of association rules. *IEEE Transactions On Knowledge And Data Engineering*, 8(6):962–969, 1996.

[2] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Proceedings of 20th International Conference on Very Large Data Bases*, pages 487–499. Morgan Kaufmann, 1994.

[3] M. Boley and H. Grosskreutz. Approximating the number of frequent sets in dense data. *Knowledge and Information Systems*, 21(1):65–89, Oct 2009.

[4] G. Buehrer, S. Parthasarathy, S. Tatikonda, T. Kurc, and J. Saltz. Toward terabyte pattern mining: an architecture-conscious solution. In *Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '07, pages 2–12, New York, NY, USA, 2007. ACM Press.

[5] A. Caprara, D. Pisinger, and P. Toth. Exact solution of the quadratic knapsack problem. *INFORMS Journal on Computing*, 11(6):125–137, 1999.

[6] D. W.-L. Cheung, S. D. Lee, and Y. Xiao. Effect of data skewness and workload balance in parallel data mining. *Knowledge and Data Engineering*, 14(3):498–514, 2002.

[7] D. W.-L. Cheung and Y. Xiao. Effect of data distribution in parallel mining of associations. *Data Mining and Knowledge Discovery*, 3(3):291–314, 1999.

[8] V. Chvátal. The tail of the hypergeometric distribution. *Discrete Mathematics*, 25(3):285 – 287, 1979.

[9] S. Cong, J. Han, J. Hoeflinger, and D. Padua. A sampling-based framework for parallel data mining. In *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 255–265, New York, NY, USA, 2005. ACM Press.

[10] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51:107–113, January 2008.

[11] B. Goethals.
`http://adrem.ua.ac.be/~goethals/software/`, 2010.

[12] B. Goethals and M. J. Zaki. Frequent itemset mining repository.
`http://fimi.ua.ac.be`, 2011.

[13] R. L. Graham. Bounds on multiprocessing timing anomalies. *SIAM Journal of Applied Mathematics*, 17(2):416–429, 1969.

[14] G. Grahne and J. Zhu. Efficiently using prefix-trees in mining frequent itemsets. In *FIMI '03, Frequent Itemset Mining Implementations, Proceedings of the ICDM 2003 Workshop on Frequent Itemset Mining Implementations*, volume 90 of *CEUR Workshop Proceedings*, 2003.

[15] D. Gunopulos, R. Khardon, and R. S. Sharma. Discovering all most specific sentences. *ACM Transactions on Database Systems*, 28:140–174, 2003.

[16] Description of the Infiniband network on Wikipedia.
`http://en.wikipedia.org/wiki/InfiniBand`.

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

[18] H. Kellerer, U. Pferschy, and D. Pisinger. *Knapsack problems*. Springer-Verlag, 2004.

[19] R. Kessl. Static load balancing of parallel mining of frequent itemsets using reservoir sampling. In *Machine Learning and Data Mining in Pattern Recognition - 7th International Conference, MLDM 2011, New York, NY, USA, August 30 - September 3, 2011.*, volume 6871, pages 553–567, 2011.

[20] R. Kessl and P. Tvrdík. Probabilistic load balancing method for parallel mining of all frequent itemsets. In *Proceedings of the 18th IASTED International Conference on Parallel and distributed computing systems 2006*, pages 578–586. ACTA Press, 2006.

[21] R. Kessl and P. Tvrdík. Toward more parallel frequent itemset mining algorithms. In *Proceedings of the 19th IASTED International Conference on Parallel and Distributed Computing and Systems*, pages 97–103. ACTA Press, 2007.

[22] H. Li, Y. Wang, D. Zhang, M. Zhang, and E. Y. Chang. Pfp: parallel fp-growth for query recommendation. In *Proceedings of the 2008 ACM conference on Recommender systems*, RecSys '08, pages 107–114, New York, NY, USA, 2008. ACM Press.

[23] R. Motwani and P. Raghavan. *Randomized algorithms*. Cambridge university press, 1995.

[24] Description of the Myrinet network on Wikipedia.

`http://en.wikipedia.org/wiki/Myrinet`, 2010.

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

[26] I. Pramudiono and M. Kitsuregawa. Parallel FP-Growth on PC Cluster. In *Proceedings of the 7th Pacific-Asia Conference of Knowledge Discovery and Data Mining*, pages 467–473. Springer, 2003.

[27] The source code of the solution of the quadratic knapsack problem.

`http://www.diku.dk/hjemmesider/ansatte/pisinger/codes.html`, 2010.

[28] Description of the round robin tournament on Wikipedia.

`http://en.wikipedia.org/wiki/Round_robin_tournament`, 2010.

[29] M. Skala. Hypergeometric tail inequalities: ending the insanity . `http://ansuz.sooke.bc.ca/professional/hypergeometric.pdf`, 2009.

[30] H. Toivonen. Sampling large databases for association rules. In *International Conference on Very Large Data Bases*, pages 134–145. Morgan Kaufman, 1996.

[31] A. Veloso. New parallel algorithms for frequent itemset mining in large databases. In *Proceedings of the Symposium on Computer Architectures and High Performance Computing*, pages 158–166, 2003.

[32] J. S. Vitter. Random sampling with a reservoir. *ACM Transactions on Mathematical Software*, 11(1):37–57, 1985.

[33] O. R. Zaiane, M. El-Hajj, and P. Lu. Fast parallel association rule mining without candidacy generation. *First IEEE International Conference on Data Mining*, pages 665–668, 2001.

[34] M. J. Zaki. Scalable algorithms for association mining. *IEEE Transactions on Knowledge and Data Engineering*, 12:372–390, 2000.

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

[36] M. J. Zaki, S. Parthasarathy, M. Ogihara, and W. Li. New algorithms for fast discovery of association rules. In *3rd International Conference on Knowledge Discovery and Data Mining*, pages 283–286. AAAI Press, 1997.