národní
úložiště
šedé
literatury

Static load balancing of parallel mining of frequent itemsets using reservoir sampling

Kessl, Robert
2010

Dostupný z http://www.nusl.cz/ntk/nusl-42270

Dílo je chráněno podle autorského zákona č. 121/2000 Sb.

Tento dokument byl stažen z Národního úložiště šedé literatury (NUŠL).

Datum stažení: 09.05.2024

Další dokumenty můžete najít prostřednictvím vyhledávacího rozhraní nusl.cz .

Institute of Computer Science
Academy of Sciences of the Czech Republic

Static load balancing of parallel mining of frequent itemsets using reservoir sampling

Robert Kessl

Technical report No. 1100

December 2010

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

Institute of Computer Science
Academy of Sciences of the Czech Republic

Static load balancing of parallel mining of frequent itemsets using reservoir sampling

Robert Kessl

Technical report No. 1100

December 2010

Abstract:

In this paper, we present a novel method for parallelization of an arbitrary depth-first search (DFS in short) algorithm for mining of all FIs. The method is based on the so called reservoir sampling algorithm. The reservoir sampling algorithm in combination with an arbitrary DFS mining algorithm executed on a database sample takes an uniformly but not independently distributed sample of all FIs using the reservoir sampling. The sample is then used for static load- balancing of the computational load of a DFS algorithm for mining of all FIs.

Keywords:
Parallel, algorithms, frequent itemsets, reservoir sampling, associatio

Static load balancing of parallel mining of frequent itemsets using reservoir sampling

Robert Kessl*

Abstract

In this paper, we present a novel method for parallelization of an arbitrary depth-first search (DFS in short) algorithm for mining of all FIs. The method is based on the so called reservoir sampling algorithm. The reservoir sampling algorithm in combination with an arbitrary DFS mining algorithm executed on a database sample takes an uniformly but not independently distributed sample of all FIs using the reservoir sampling. The sample is then used for static load-balancing of the computational load of a DFS algorithm for mining of all FIs.

1 Introduction

The automated data collection causes that companies own huge databases. The companies are interested in analysing their databases, so they can use them for making better decisions. The process of analysing the data is called *data mining*. Unfortunately, extreme growth of the database sizes make using ordinary data mining techniques unfeasible. Processing of databases of such sizes is almost impossible with a single processor. Therefore, new parallel algorithms that are able to process such amount of data are needed. Today, large shared-memory machines parallel are still quite expensive. Distributed-memory multiprocessors can be easily built from cheap computers connected with a special network. Therefore, we consider designing algorithms for distributed-memory parallel machines.

One of the important data mining tasks is the search for co-occurences among the data, so called frequent itemset mining [1]. The frequent itemset mining was introduced in the context of analysing the market basket of a consumer in a retail store (hence the term market basket analysis). In the retail store, we track the contents of baskets of customers. The content of the baskets is stored in the database as transactions. In this database, we search for sets of items (itemsets in short) that occurs in at least *min_support* transactions. These itemsets are so called *frequent itemsets* (or FIs in short). From FIs, we create rules of type $X \Rightarrow Y$, where X, Y are two FIs. For example {butter, bread} $\Rightarrow$ {milk}. The search of these co-occurences is divided into two parts: 1) find all *frequent itemsets*; 2) create *association rules* from the FIs.

This task is computationally and memory demanding. It seems that the finding of all frequent itemsets is the most time-consuming part of the whole process. With the growth of retail-store databases it is important to design parallel algorithms for mining of FIs.

The first parallel algorithm for this task was proposed in [2]. In [3] and [4] we have proposed new parallel methods for mining of FIs. First, we denote the set of all FIs by $\mathcal{F}$. The basic idea of the algorithms proposed in [3, 4] is to create a sample of FIs $\widetilde{\mathcal{F}}_s$ that is used to create disjoint partitions $F_i, F_j \subseteq \mathcal{F}$ such that $|F_i|/|\mathcal{F}| \approx 1/P$, where P is the number of processors. The relative size $|F_i|/|\mathcal{F}|$ is estimated using the sample $\widetilde{\mathcal{F}}_s$. The partitions F_i are then independently processed by each processor. In [3, 4], we have proposed a method of creation of $\widetilde{\mathcal{F}}_s$, based on a *modified* coverage algorithm. The problem with these two methods is that the sample $\widetilde{\mathcal{F}}_s$ is *non-uniform* and additionally the two methods needs an algorithm for mining of maximal frequent itemsets (MFIs in short). Since the number of MFIs can be quite large the methods are in some cases very memory consuming.

In this paper, we show how to create a *uniform* sample $\widetilde{\mathcal{F}}_s$ using a different, faster and less memory consuming, method. The new method does not need to compute the MFIs (and store them in main memory) and therefore is less memory consuming. Additionally, our new method needs only the algorithm for mining of FIs and its modifications.

2 Notation

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 *itemset*. Further, we need to view the baseset $\mathcal{B}$ as an ordered set. The items are 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 $U = (b_{u_1}, b_{u_2}, \ldots, b_{u_{|U|}})$.

*This paper was supported from the Czech Science Foundation, grant number GA ČR P202/10/1333.

Let $U \subseteq \mathcal{B}$ be an itemset and `id` a unique identifier. We call the pair (id, U) a *transaction*. The `id` is called the *transaction id*. A database $\mathcal{D}$ is a set of transactions. In our algorithms, we need to sample the database $\mathcal{D}$. A *database sample* is denoted by $\widetilde{\mathcal{D}}$. We define the support as the number of transactions containing U, but in some literature, the relative support is defined by $Supp^*(U) = Supp(U)/|\mathcal{D}|$. We call U *frequent* in database $\mathcal{D}$ if $Supp(U, \mathcal{D}) \geq min\_support$. We can also define the frequent itemset using the relative support, denoted by $min\_support^*$, $0 \leq min\_support^* \leq 1$, i.e., an itemset is frequent iff $Supp^*(U, \mathcal{D}) \geq min\_support^*$.

We denote the set of all frequent itemsets computed from $\mathcal{D}$ by $\mathcal{F}$. The set of all frequent itemsets computed from $\widetilde{\mathcal{D}}$ is denoted by $\widetilde{\mathcal{F}}$. A sample of frequent itemsets is denoted by $\widetilde{\mathcal{F}}_s$. In our case, the set $\widetilde{\mathcal{F}}_s$ is a sample of $\widetilde{\mathcal{F}}$, i.e., $\widetilde{\mathcal{F}}_s \subseteq \widetilde{\mathcal{F}}$.

The basic property of frequent itemsets is the so called *monotonicity of support*. It is an important property of all FIs mining algorithms and is defined as follows:

THEOREM 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 for the supports of U and V we have $Supp(U, \mathcal{D}) \geq Supp(V, \mathcal{D})$.*

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 from balls and X_i balls, such that $n = \sum_{i=1}^{C} X_i$, are colored by the ith color. Then the probability mass function is:

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

We denote the number of processors by P and processor i by p_i.

3 The lattice of all itemsets

It is well known that the powerset $\mathcal{P}(\mathcal{B})$ of a set $\mathcal{B}$ is a complete lattice. The *join* operation is the *set union operation* and *meet* the *set intersection operation*. To decompose the $\mathcal{P}(\mathcal{B})$ into the prefix-based equivalence classes, 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*:

DEFINITION 3.1. (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 *(PBEC in short) of U, denoted by $[U]_\ell$, is a set of all itemsets that have the same prefix of length ℓ, i.e., $[U]_\ell = \{W = (w_1, w_2, \ldots, w_m) | u_i = w_i, i \leq \ell, m = |W| \geq |U|, \ell \leq |U|, U, W \subseteq \mathcal{B}\}$*

To simplify the notation, we use $[W]$ for the PBEC $[W]_\ell$ iff $\ell = |W|$. Each $[W], W \subseteq \mathcal{B}$ is a sublattice of $(\mathcal{P}(\mathcal{B}), \subseteq)$. Additionally, we use the term prefix for both: (a) *ordered set*; (b) *unordered set*; if clear from context, e.g., let $\mathcal{B} = \{1, 2, 3, 4, 5\}$ with the order $1 < 2 < 3 < 4 < 5$ and $U = \{3, 1, 2\}$ then the term prefix means $U = (1, 2, 3)$.

DEFINITION 3.2. (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 PBEC $[U]$ is an ordered set $\Sigma \subseteq \mathcal{B}$ such that $U \cap \Sigma = \emptyset$ and for each $W \in [U], W \setminus U \subseteq \Sigma$. We denote the PBEC together with the extensions Σ by $[U|\Sigma]$.*

We omit the extensions from the notation if clear from context.

Let $\mathcal{B} = \{b_1, \ldots, b_n\}, b_1 < \ldots < b_n$. Let $U_i = \{b_i\}$ and $\Sigma_i = \{b_j | b_i < b_j\}$ and $Q = \{(U_i, \Sigma_i)\}$ be a set of pairs such that each pair forms a PBEC $[U_i|\Sigma_i]$. The $[U_i|\Sigma_i]$ forms *disjoint* PBECs. Each PBEC $[U_i|\Sigma_i]$ can be recursively divided into disjoint PBECs in the following way: let $q = (U, \Sigma_U) \in Q$ and $W_k = U \cup \{b_k\}, b_k \in \Sigma_U$ and $\Sigma'_k = \{b | b_k < b; b, b_k \in \Sigma_U\}$ then $[W_k|\Sigma'_k]$ forms disjoint PBECs. The PBEC $[U|\Sigma_U]$ is a union of the new PBECs: $[U|\Sigma_U] = (\bigcup_k [W_k|\Sigma'_k]) \cup \{U\}$. We can replace $q = (U, \Sigma_U)$ by pairs (W_k, Σ'_k), i.e., $Q' \leftarrow (Q \setminus q) \cup (\bigcup_k \{(W_k, \Sigma_k)\})$. Since the pairs in Q forms *disjoint* PBECs and the new PBECs are also disjoint and $[U|\Sigma_U] = (\bigcup_k [W_k|\Sigma'_k]) \cup \{U\}$, the set Q' also forms disjoint PBECs and contains almost the same FIs. This process can be recursively repeated, making the PBECs smaller. During the partitioning, we can freely change the order of items in the extensions Σ_U: the resulting PBECs remain disjoint.

Further, we need to partition $\mathcal{F}$ into P disjoint sets, F_i, such that $\bigcup_i F_i = \mathcal{F}$ and $|F_i|/|\mathcal{F}| \approx 1/P$. This partitioning can be done using the PBECs. The PBECs can be collated to a single partition: let have m *disjoint* PBECs $[U_l]$, such that $\bigcup_l [U_l] = \mathcal{F}$ and sets of indexes of the PBECs $L_i \subseteq \{l | 1 \leq l \leq m\}, 1 \leq i \leq P$ such that $L_i \cap L_j = \emptyset$, $\sum_i |L_i| = m$ and $F_i = \bigcup_{l \in L_i} ([U_l] \cap \mathcal{F})$. The sets L_i can be chosen in such a way that F_i make the partitions of relative size $|F_i|/|\mathcal{F}| \approx 1/P$

4 Existing parallel algorithms

There is a number of parallel algorithms for parallel mining of FIs on distributed memory machines. A lot of these algorithms are based on the Apriori algorithm [1]. Because the Apriori algorithm is a breadth-first search algorithm, it is very slow and needs huge amount of memory. A first parallelization of the Apriori algorithm is the count distribution algorithm [2]. A prunning techniques for the parallel Apriori algorithm were proposed by Cheung [5, 6].

Parallelizations of the ECLAT algorithm were proposed by Zaki[7]. The parallelization of the FP-GROWTH algorithm was proposed by [8]. The idea behind these two algorithms is to estimate the amount of FIs in a PBEC and schedule the PBECs based on the estimation. In both cases, the estimation of the size of the PBECs is a heuristic. Unfortunately, the heuristic cannot be viewed as a static load-balancing method, because it does not capture the real amount of FIs in a PBEC.

Another parallelization of the algorithm for mining of all FIs was proposed by Veloso[9]. The method in fact uses only a parallel algorithm for mining of MFIs. The method first distributes the database among processors, processor p_i having a database partition D_i such that $D_i \cap D_j = \emptyset, |D_i| \approx |\mathcal{D}|/P$. The method then computes the MFIs from the whole database in parallel and then each p_i enumerates all FIs using the computed MFIs, computing support for each FI in D_i. The supports are then added together in parallel. First, we have to argue that in the case of having large amount of FIs, the parallel computation of supports can be very expensive in the terms of communication. Second, it is not clear from the paper which algorithm is used for computation of the speedup, i.e., to which sequential algorithm is compared the execution time of the parallel algorithm.

In [4], we have proposed the PARALLEL-FIMI-PAR method for parallel mining of all FIs. Our method also computes the MFIs. However, we compute the MFIs from a database sample and we use the MFIs for different purpose. The MFIs are used for making a sample of FIs. We discuss the PARALLEL-FIMI-PAR method in a more detail in Section 7.

5 Sampling methods

The basic idea of our method is to create a database sample $\widetilde{\mathcal{D}}$ and from $\widetilde{\mathcal{D}}$ we create $\widetilde{\mathcal{F}}_s$ that allow us to estimate the relative size of an arbitary PBEC. The reason of making $\widetilde{\mathcal{F}}_s$ using $\widetilde{\mathcal{D}}$ is the speed of the "double sampling process".

5.1 Database sample The time complexity of the decision 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 decide 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 $Supp^*(U, \mathcal{D})$ – this significantly reduces the time complexity. The approach of estimating the relative support of U was described by Toivonen [10].

Toivonen uses a *database sample* $\widetilde{\mathcal{D}}$ for the *sequential* mining of frequent itemsets and for the efficient estimation of theirs supports. Toivonen's algorithm works as follows: 1) create a database sample $\widetilde{\mathcal{D}}$ of $\mathcal{D}$; 2) compute all frequent itemsets in $\widetilde{\mathcal{D}}$; 3) check that all these FIs computed using $\widetilde{\mathcal{D}}$ are also FIs in $\mathcal{D}$ and correct the output. If an itemset is frequent in $\mathcal{D}$ and not in $\widetilde{\mathcal{D}}$, correct the output using $\mathcal{D}$. Toivonen's algorithm is based on an efficient probabilistic estimate of the support of an itemset U. We reuse this idea of estimating the support of U in our method for parallel mining of FIs, i.e., we use only the first two steps.

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 database sample $\widetilde{\mathcal{D}}$ is sampled with replacement. 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. *[10] Given an itemset $U \subseteq \mathcal{B}$ 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 $\delta_{\widetilde{\mathcal{D}}}$.

See [10] for proof. Using a database sample $\widetilde{\mathcal{D}}$ with size given by the previous theorem, we can estimate $Supp^*(U, \mathcal{D})$ with error $\epsilon_{\widetilde{\mathcal{D}}}$ that occurs with probability at most $\delta_{\widetilde{\mathcal{D}}}$: It follows from Lemma 5.1 that if we compute the approximation $\widetilde{\mathcal{F}}$ of $\mathcal{F}$ from the database sample $\widetilde{\mathcal{D}}$ of size $|\widetilde{\mathcal{D}}| \geq \frac{1}{2\epsilon^2_{\widetilde{\mathcal{D}}}} \ln \frac{2}{\delta_{\widetilde{\mathcal{D}}}}$, we could get a close approximation $\widetilde{\mathcal{F}}$ of $\mathcal{F}$.

5.2 The reservoir sampling algorithm In this section, we show the *reservoir sampling algorithm* [11] that creates an uniformly but not independently distributed sample $\widetilde{\mathcal{F}}_s \subseteq \widetilde{\mathcal{F}}$.

We can formulate the problem of *reservoir sampling* in the terms of $\widetilde{\mathcal{F}}$ and $\widetilde{\mathcal{F}}_s$: let have a stream of frequent

itemsets $U \in \widetilde{\mathcal{F}}$ mined from a database $\widetilde{\mathcal{D}}$. We do not know the number of FIs $|\widetilde{\mathcal{F}}|$ in advance and the task is to take $|\widetilde{\mathcal{F}}_s|$ samples of $\widetilde{\mathcal{F}}$. The solution of this task solves our problem of making a uniform sample $\widetilde{\mathcal{F}}_s \subseteq \widetilde{\mathcal{F}}$. The sampling is done using an array of FIs (a buffer, or in the terminology of [11] a reservoir) that holds $\widetilde{\mathcal{F}}_s$ during the processing of the itemsets from $\widetilde{\mathcal{F}}$.

The reservoir sampling uses the following two procedures: 1) ReadNextFI(L): reads next FI from an output of an arbitrary sequential algorithm for mining of FIs and stores the itemset at the location L in memory; 2) SkipFIs(k): skips k FIs from the output of an arbitrary algorithm for mining of FIs. And the following function: Random() which returns an uniformly distributed real number from the interval $[0, 1]$

The simplest reservoir sampling algorithm is summarized in Algorithm 1. It takes as an input an array R (reservoir/buffer) of size $n = |\widetilde{\mathcal{F}}_s|$, the function ReadNextFI(L) that reads an FI from the output of an FI mining algorithm and stores it in memory at location L, and finally the function SkipFIs(k) that skips k FIs. The algorithm samples n FIs and stores them in main memory into the buffer R.

Algorithm 1 The Reservoir-Sampling algorithm

Reservoir-Sampling(**In/Out:** Array R of size n,
In: Integer n,
In: Function ReadNextFI,
In: Function SkipFIs)

1: **for** $j = 0$ to $n - 1$ **do**
2: ReadNextFI($R[j]$)
3: **end for**
4: $t = n$
5: **while** not eof **do**
6: $t = t + 1$
7: $m = \lfloor t \times \text{Random}() \rfloor$ {pick uniformly a number from the set $\{1, \ldots, t-1\}$}
8: **if** $m < n$ **then**
9: ReadNextFI($R[m]$)
10: **else**
11: SkipFIs(1)
12: **end if**
13: **end while**

The Reservoir-Sampling is quite slow, it is linear in the number of input records read by ReadNextFI(R), i.e., it is linear in $|\widetilde{\mathcal{F}}|$. Vitter [11] created a *faster* algorithm with the average running time $\mathcal{O}(|\widetilde{\mathcal{F}}_s|(1 + \log \frac{|\widetilde{\mathcal{F}}|}{|\widetilde{\mathcal{F}}_s|}))$, where $|\widetilde{\mathcal{F}}_s|$ is the size of the array R used by Reservoir-Sampling. The algorithm has the same parameters as the Reservoir-Sampling and we denote the Vitter's variant of the reservoir sampling algorithm by Vitter-Reservoir-Sampling.

Now, we analyse the relative size of a PBEC using the samples taken by the reservoir sampling algorithm. The reservoir sampling samples the set $\widetilde{\mathcal{F}}$ **without replacement**, resulting in $\widetilde{\mathcal{F}}_s$.

Using the bounds from [12, 13] we can analyse the size of the sample $\widetilde{\mathcal{F}}_s$ in order to estimate the relative size of a PBEC using an uniformly but *not independently* distributed sample. From these bounds follows the following theorem:

Theorem 5.2. *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 $\epsilon_{\widetilde{\mathcal{F}}_s}$ with probability $\delta_{\widetilde{\mathcal{F}}_s}$ 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. In [12, 13] is shown that $P[\mathrm{E}[i] - \epsilon|\widetilde{\mathcal{F}}_s| \leq i \leq \mathrm{E}[i] + \epsilon|\widetilde{\mathcal{F}}_s|] \leq 1 - (e^{-|\widetilde{\mathcal{F}}_s|D(\rho - \epsilon_{\widetilde{\mathcal{F}}_s}||\rho)} + e^{-|\widetilde{\mathcal{F}}_s|D(\rho + \epsilon_{\widetilde{\mathcal{F}}_s}||\rho)})$. From this and the fact that $D(\rho + \epsilon_{\widetilde{\mathcal{F}}_s}||\rho) > D(\rho - \epsilon_{\widetilde{\mathcal{F}}_s}||\rho)$ we have:

$$1 - (e^{-|\widetilde{\mathcal{F}}_s| \cdot D(p - \epsilon||p)} + e^{-|\widetilde{\mathcal{F}}_s| \cdot D(p + \epsilon||p)}) \leq 1 - \delta_{\widetilde{\mathcal{F}}_s}$$

$$1 - 2e^{-|\widetilde{\mathcal{F}}_s| \cdot D(p + \epsilon||p)} \leq 1 - \delta_{\widetilde{\mathcal{F}}_s}$$

■

6 Error of the estimation of the size of a union of PBECs using $\widetilde{\mathcal{D}}$

In our method, we create a sample $\widetilde{\mathcal{F}}_s \subseteq \widetilde{\mathcal{F}}$ using the reservoir sampling algorithm. Let have $n \geq 1$ prefixes U_i, we need to have $|\bigcup_i [U_i] \cap \widetilde{\mathcal{F}}|/|\widetilde{\mathcal{F}}|$ very close to $|\bigcup_i [U_i] \cap \mathcal{F}|/|\mathcal{F}|$. The following theorem gives bounds the size of $|\bigcup_i [U_i] \cap \mathcal{F}|/|\mathcal{F}|$ in terms of $|\bigcup_i [U_i] \cap \widetilde{\mathcal{F}}|/|\widetilde{\mathcal{F}}|$:

Theorem 6.1. (bounds on the size of union of FIs) *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. *$A = \{U | Supp^*(U, \mathcal{D}) < min\_support^*$ 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. $B = \{U | Supp^*(U, \mathcal{D}) \geq min\_support^*$ *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:

$$|\widetilde{C}| \leq |C| + a \cdot |\mathcal{F}| \tag{6.1}$$

$$\frac{|\widetilde{C}|}{|\mathcal{F}|} \leq \frac{|C|}{|\mathcal{F}|} + a \tag{6.2}$$

(6.3) follows from (6.2) using the fact that $|\mathcal{F}| = \frac{|\widetilde{\mathcal{F}}|}{(1+a-b)}$.

$$\frac{|\widetilde{C}|}{|\widetilde{\mathcal{F}}|}(1 + a - b) \leq \frac{|\widetilde{C}|}{|\mathcal{F}|} \leq \frac{|C|}{|\mathcal{F}|} + a \tag{6.3}$$

$$\frac{|\widetilde{C}|}{|\widetilde{\mathcal{F}}|}(1 + a - b) - a \leq \frac{|C|}{|\mathcal{F}|} \tag{6.4}$$

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:

$$|C| - b \cdot |\mathcal{F}| \leq |\widetilde{C}| \tag{6.5}$$

$$\frac{|C|}{|\mathcal{F}|} - b \leq \frac{|\widetilde{C}|}{|\mathcal{F}|} \tag{6.6}$$

$$\frac{|C|}{|\mathcal{F}|} \leq \frac{|\widetilde{C}|}{|\widetilde{\mathcal{F}}|} \cdot (1 + a - b) + b \tag{6.7}$$

COROLLARY 6.1. *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:*

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

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 $\widetilde{\mathcal{F}}_s$ (that was computed using $\widetilde{\mathcal{D}}$) 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.

7 Summary of the previous two methods

In [3] we have proposed the PARALLEL-FIMI-SEQ method and in [4] we have proposed the PARALLEL-FIMI-PAR method. The idea of the two methods 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 whole method consists of four phases.

The input and the parameters of the whole method are the following: 1) Minimal support $min\_support^*$; 2) The sampling parameters: real numbers $0 \leq \epsilon_{\widetilde{\mathcal{D}}}, \delta_{\widetilde{\mathcal{D}}}, \epsilon_{\widetilde{\mathcal{F}}_s}, \delta_{\widetilde{\mathcal{F}}_s} \leq 1$, see Sections 5.1 and 5.2; 3) The relative size of a PBEC: the parameter $\rho, 0 \leq \rho \leq 1$, see Sections 5.2; 4) Partition parameter: real number $\alpha, 0 \leq \alpha \leq 1$; 5) Database parts $D_i, 1 \leq i \leq P$. The database partitions are loaded by each processor at the beginning of the methods.

The four phases are designed in such a way that they statically load-balance the computation of all FIs. In Phases 1–2, we prepare the static load-balancing for Phase 4. In Phase 3, we redistribute the database partitions so each processor can proceed independently with some PBECs. In the Phase 4, we execute an arbitrary algorithm for mining of FIs. To speedup Phases 1–2, we can execute each of Phase 1–2 in parallel.

We assume that at the beginning of the computation, processor p_i loads its database partition D_i to a local memory. The database partitions D_i has the following properties: $D_i \cap D_j = \emptyset, i \neq j$, and $|D_i| \approx \frac{|\mathcal{D}|}{P}$. Additionally, without loss of generality, we expect that each $b_i \in \mathcal{B}$ is frequent. If some of the items $b_i \in \mathcal{B}$ are not frequent, each processor p_i computes local support of all items $b_j \in \mathcal{B}$ in its database part D_i. The support is then broadcast and each p_i removes all b_j that are not globally frequent. The four phases are summarized below:

Phase 1 (sampling of FIs): the input of Phase 1 is the minimal support $min\_support^*$, a partitioning of the

database $\mathcal{D}$ into P disjoint partitions D_i, and the real numbers $0 \leq \epsilon_{\widetilde{\mathcal{D}}}, \delta_{\widetilde{\mathcal{D}}}, \epsilon_{\widetilde{\mathcal{F}}_s}, \delta_{\widetilde{\mathcal{F}}_s} \leq 1$. Output of Phase 1 is a sample of frequent itemsets $\widetilde{\mathcal{F}}_s \subsetneq \widetilde{\mathcal{F}}$. Generally, the only purpose of the first phase is to compute a sample of FIs $\widetilde{\mathcal{F}}_s$ and a database sample $\widetilde{\mathcal{D}}$. First, each processor samples D_i (in parallel) and creates part $\mathcal{D}'_i$ and broadcasts them to other processors (all-to-all scatter[1]). Each processor p_i then creates $\widetilde{\mathcal{D}} = \bigcup_i \mathcal{D}'_i$. The processors compute the set of all MFIs from $\widetilde{\mathcal{D}}$, denoted by $\widetilde{\mathcal{M}}$. The set $\widetilde{\mathcal{M}}$ is the upper boundary in the sense of set inclusion of the set $\widetilde{\mathcal{F}}$, i.e., due to the monotonicity of support for each $U \in \widetilde{\mathcal{F}}$ exists $m \in \widetilde{\mathcal{M}}$ such that $U \subseteq m$. We need to take a sample $\widetilde{\mathcal{F}}_s \subsetneq \widetilde{\mathcal{F}}$. This can be done using a *modified coverage algorithm*. The modified coverage algorithm randomly chooses $m \in \widetilde{\mathcal{M}}$ with probability $|\mathcal{P}(m)| / \sum_{m' \in \widetilde{\mathcal{M}}} |\mathcal{P}(m')|$. Then it picks uniformly an itemset $U \in \mathcal{P}(m)$ – the itemset U is frequent due to the monotonicity principle. Because the FIs $U \in \widetilde{\mathcal{F}}$ have different probability of being chosen in the sample $\widetilde{\mathcal{F}}_s$ the approach does not generates an uniformly distributed sample $\widetilde{\mathcal{F}}_s$ and neither does not guarantee the probability of the error of the relative size of a PBEC. Therefore, we do not have any guarantees on the error of the estimate of the relative size. But the estimates using the sample $\widetilde{\mathcal{F}}_s$ gives reasonably good results and the process is very quick. For the coverage algorithm that creates independently and identically (uniformly) distributed sample, see [14]. The difference between Parallel-FIMI-Seq and Parallel-FIMI-Par methods is that the second computes a set M in parallel, such that $\widetilde{\mathcal{M}} \subseteq M \subseteq \widetilde{\mathcal{F}}$, i.e., it computes a superset of $\widetilde{\mathcal{M}}$. Computation of M in parallel makes the Parallel-FIMI-Par faster then Parallel-FIMI-Seq but it also needs more memory.

Phase 2 (lattice partitioning): the input of this phase is the sample $\widetilde{\mathcal{F}}_s$, the database sample $\widetilde{\mathcal{D}}$ (both computed in Phase 1) and the parameter α. In Phase 2, processor p_1 creates prefixes $U_i \subseteq \mathcal{B}$ and the extensions Σ_i of each PBEC $[U_i|\Sigma_i]$, and estimates the size of $[U_i|\Sigma] \cap \mathcal{F}$ using $\widetilde{\mathcal{F}}_s$: $|[U_i|\Sigma] \cap \mathcal{F}|/|\mathcal{F}| \approx |[U_i|\Sigma] \cap \widetilde{\mathcal{F}}_s|/|\widetilde{\mathcal{F}}_s|$. p_1 then creates indexsets $L_i, 1 \leq i \leq P$ that makes the partition $F_i = \mathcal{F} \cap \bigcup_{j \in L} [U_j|\Sigma_j]$ such that $F_i \cap F_j = \emptyset, i \neq j$ and $|F_i|/|\mathcal{F}| \approx 1/P$.

Phase 3 (data distribution): the input of this phase is the assignment of the prefixes U_j and the extensions Σ_j to the processors p_i and the database partitioning $D_i, i = 1, \ldots, P$. Now, the processors exchange database partitions: processor p_i sends $S_{ik} \subseteq D_i$ to processor p_k such that S_{ik} contains transactions needed by p_k for computing support of the itemsets of its assigned PBECs.

Phase 4 (computation of FIs): as the input to each processor are the prefixes U_k, its extensions Σ_k, and the database parts needed for computation of supports of itemsets $V \in [U_k] \cap \mathcal{F}$ and the original D_i. Each processor computes the FIs in $[U_k] \cap \mathcal{F}$ by executing an arbitrary sequential algorithm for mining of FIs. Additionally, each processor computes support of $W \subseteq U_k$ in D_i, i.e., $Supp(W, D_i)$. The supports are then send to p_1 and p_1 computes $Supp(W, \mathcal{D}) = \sum_{1 \leq i \leq P} Supp(W, D_i)$

The difference between our new method and the previous two methods is only in Phase 1: we use the reservoir sampling instead of the modified coverage algorithm. However, the use of the reservoir sampling is quite important, because the number of MFIs can be quite large (and therefore sometims do not fit into main memory) and we have a better values of speedup.

8 Proposal of a new DM parallel method

Our new method is called *Parallel Frequent Itemset MIning – Reservoir* (Parallel-FIMI-Reservoir in short). This method works for any number of processors $P \ll |\mathcal{B}|$. The basic idea is the same as in the Parallel-FIMI-Par method. The main difference is the usage of the so called reservoir sampling algorithm instead of the modified coverage algorithm. This allow us to take an uniformly but *not independently* distributed sample $\widetilde{\mathcal{F}}_s$. We make the sample $\widetilde{\mathcal{F}}_s$ in parallel: in Phase 1, we execute an arbitrary algorithm for mining of FIs in parallel and the output of the FI mining algorithm is sampled using the reservoir sampling (in parallel). The input parameters are the same as in the Parallel-FIMI-Par method. In the rest of this section, we give detailed description of the four phases.

8.1 Detailed description of Phase 1

In this Section, we give a detailed description of the sampling process based on the *reservoir sampling* [11] that samples $\widetilde{\mathcal{F}}$ uniformly, i.e., it creates an identically distributed sample $\widetilde{\mathcal{F}}_s$ of $\widetilde{\mathcal{F}}$. At the beggining the processors create the database sample $\widetilde{\mathcal{D}}$ by sampling its database parts D_i.

In our parallel method, we are using the Vitter-Reservoir-Sampling Algorithm, the faster reservoir sampling algorithm. To speedup the sampling phase of our parallel method, we execute the sequential algorithm for mining of FIs on the database $\widetilde{\mathcal{D}}$ 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

[1] 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 $m_{ij} \neq m_{ik}, i \neq k$

sequential depth-first search (DFS in short) algorithm for mining of FIs, enumerating $[(b_j)] \cap \widetilde{\mathcal{F}}, b_j \in B_i$. Because the sequential algorithm is run in parallel with dynamic load balancing, processor p_i ask other processors for work when it finishes the items in B_i, i.e., processor p_i must not process only the items in B_i. For terminating the parallel execution, we use the Dijkstra's token termination algorithm. The output of the sequential DFS algorithm is read by the reservoir sampling algorithm that creates the sample $\widetilde{\mathcal{F}}_s$.

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 5.2. Because, the sequential algorithm for mining of MFIs with the reservoir sampling is executed in parallel with dynamic load-balancing, it is not known how many FIs are 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, processor p_i sends f_i to processor p_1. p_1 picks P random variables $X_i, 1 \leq i \leq P$ from multivariate hypergeometrical distribution with parameters: number of colors $C = P$, $M_i = f_i$. Each processor p_i choose X_i itemsets $U \in \widetilde{\mathcal{F}}_s$ at random out of the $N = |\widetilde{\mathcal{F}}_s|$ sampled frequent itemsets computed by p_i. The samples are then send to processor p_1. p_1 stores the received samples in $\widetilde{\mathcal{F}}_s$.

8.2 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), the set $\mathcal{B}$, and a real number $\alpha, 0 < \alpha \leq 1$. For the purpose of this section, we denote the prefixes by U_k, the extensions of U_k by Σ_k, i.e., U_k and Σ_k forms a PBEC $[U_k|\Sigma_k]$. The output of this phase are the indexsets L_i. The indexsets make the disjoint sets F_i such that $|F_i|/|\mathcal{F}| \approx 1/P$, see Section 3. Each processor p_i then in Phase 4 processes the FIs contained in F_i. *The output of Phase 2* are the index sets L_i of PBECs, computed on p_1, and the PBECs $[U_k|\Sigma_k]$.

The DFS sequential FI mining algorithm usually dynamically changes the order of items in Σ_k for each PBEC $[U_k|\Sigma_k]$, i.e., the algorithm uses different order of items in the extensions. The PBECs are still disjoint and additionally the sequential algorithm is faster. Therefore, we need to prepare the PBECs in the same way as the sequential algorithm does. Let U be a prefix and $\Sigma = \{b_1, \ldots, b_n\} \subseteq \mathcal{B}$ the extensions. The sequential algorithm orders the items b_i: $b_1 < \ldots < b_n$ such that $Supp(U \cup \{b_1\}) < \ldots < Supp(U \cup \{b_n\})$. We use the supports estimated using $\widetilde{\mathcal{D}}$ for estimating the order of the extensions.

The partitioning of $\mathcal{F}$ is a two step process:

(1) p_1 creates a list of prefixes U_k such that the estimated relative size of the PBEC $[U_k] \cap \mathcal{F}$ satisfies $\frac{|[U_k] \cap \widetilde{\mathcal{F}}_s|}{|\widetilde{\mathcal{F}}_s|} \leq \alpha \cdot \frac{1}{P}$, where $0 < \alpha < 1$ is a parameter of the computation set by the user. The PBECs are created recursively, see Section 3. The reason for making the PBECs of relative size $\leq \alpha \cdot \frac{1}{P}$ is to make the PBECs small enough so that they can be scheduled and the schedule is balanced, i.e., each processor having a fraction $\approx 1/P$ of FIs. Smaller number of large PBECs could make the scheduling unbalanced.

(2) p_1 creates set of indexes L_i such that $|F_i|/|\mathcal{F}| \approx 1/P$. Making the sets L_i is a well known NP-complete problem of scheduling tasks on P equivalent machines. We use the well-known LPT-Schedule algorithm. The LPT-Schedule algorithm is a best-fit algorithm, see Algorithm 2 and [15].

Algorithm 2 The LPT-Schedule algorithm

LPT-Schedule(**In:** Set $S = \{(U_i, \Sigma_i, s_i)\}$, **Out:** Sets L_i)

1: Sort the set S such that $s_i < s_j, i \neq j$.
2: Assign each (U_i, Σ_i, s_i) (in decreasing order by s_i) to the least loaded processor p_i. The indexes assigned to p_i, are stored in L_i.

Lemma 8.1. *[15]* LPT-Schedule *is* 4/3-*approximation algorithm.*

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

The only problem with the scheduling proposed in this Section is the following: the Theorem 5.2 allows us to estimate the size of a single PBEC. Unfortunatelly, we need to estimate the size of the union of PBECs that are dependent on the sample $\widetilde{\mathcal{F}}_s$. 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 the PBECs and estimating its size using the sample, i.e., $|[U_i] \cap \widetilde{\mathcal{F}}|/|\widetilde{\mathcal{F}}| \approx |[U_i] \cap \widetilde{\mathcal{F}}_s|/|\widetilde{\mathcal{F}}_s|$. Processor p_i is assigned with prefixes from the index set L_i, i.e., it is assigned with FIs $F_i = \bigcup_{j \in L_i} [U_j] \cap \widetilde{\mathcal{F}}$ such that $|F_i|/|\widetilde{\mathcal{F}}| \approx 1/P$. The problem is that we make the set F_i using the

sample $\widetilde{\mathcal{F}}_s$. Therefore, we are not able to use the estimate of the relative size of a single PBEC with error $\epsilon_{\widetilde{\mathcal{F}}_s}$ with probability $\delta_{\widetilde{\mathcal{F}}_s}$ for estimation of the relative size of $|F_i|/|\widetilde{\mathcal{F}}|$ with error $\epsilon_{\widetilde{\mathcal{F}}_s}$ with probability $\delta_{\widetilde{\mathcal{F}}_s}$. The error of $|F_i|/|\widetilde{\mathcal{F}}|$ is $\epsilon = |L_i| \cdot \epsilon_{\widetilde{\mathcal{F}}_s}$ with probability $\delta = |L_i| \cdot \delta_{\widetilde{\mathcal{F}}_s}$. We will discuss the error of the estimate later, in Section 10.1.

8.3 Detailed description of Phase 3 The input of Phase 3 for 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. Processor p_i needs for the computation of $F_i = \bigcup_{k \in L_i}([U_k] \cap \mathcal{F})$ a database partition $D'_i = \bigcup_j \{t | t = (\mathrm{id}, W) \in D_j, k \in L_i, \text{ and } U_k \subseteq W\}$. The database partition D'_i should contain all the information needed for computation of F_i. At the begginning of this phase, the processors has disjoint database partitions D_i such that $|D_i| \approx \frac{|\mathcal{D}|}{P}$. We expect that we have a distributed memory machine whose nodes are interconnected using a network such as Myrinet or Infiniband, 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[2]). That is: processor p_i send to processor p_j the set of transactions $S_{ij} = \{t | t = (\mathrm{id}, W) \in D_i, k \in L_j, \text{ and } U_k \subseteq W\}$, i.e., all transactions that contain at least one $U_k, k \in L_j$, as a subset: $D'_j = \bigcup_i \{t | t = (\mathrm{id}, W) \in D_i, k \in L_j, \text{ and } U_k \subseteq W\} = \{t | t = (\mathrm{id}, W) \in \mathcal{D}, \text{exists } k \in L_j, U_k \subseteq W\}$. The all-to-all scatter is done in $\lfloor \frac{P}{2} \rfloor$ communication rounds.

We can consider the scatter as a round-robin tournament of P players [16]. Making the schedule of the database exchange (tournament) is the following procedure: if P is odd, a dummy processor(player) can be added, whose scheduled opponent waits for the next round and the processors(player) performs P communication rounds(plays). For example let have 14 processors, in the first round the following processors exchange their database portions:

1	2	3	4	5	6	7
14	13	12	11	10	9	8

The processors are paired by the numbers in the columns. That is, database parts are exchanged between processors p_1 and p_{14}, p_2 and p_{13}, etc. In the second round one processor is fixed (number one in this case) and the other are rotated clockwise:

1	14	2	3	4	5	6
13	12	11	10	9	8	7

This process is iterated until the processors are almost in the initial position:

1	3	4	5	6	7	8
2	14	13	12	11	10	9

8.4 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), D'_q (computed in Phase 3), 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, and the sets of indexes L_q of prefixes U_k and extensions Σ_k assigned to processor p_q.

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 $W \in [U_k | \Sigma_k], (U_k, \Sigma_k) \in Q$. 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 $[U_k | \Sigma_k]$.

8.5 The Parallel-FIMI-Reservoir method This method samples $\widetilde{\mathcal{F}}$ using the *reservoir sampling*. The reservoir sampling samples $\widetilde{\mathcal{F}}$ uniformly. To make the method faster, the reservoir sampling is executed in parallel. The method is summarized in the Parallel-FIMI-Reservoir method, see Method 1. The advantage of the Parallel-FIMI-Reservoir method over the two previous methods are the following: 1) the Parallel-FIMI-Reservoir method does not need to compute the MFIs and therefore does not need an additional very quick algorithm for mining MFIs; 2) the number of MFIs can be very large, since we do not need to computed the MFIs, we do not need to store the MFIs in main memory and therefore it has lower memory consumption; 3) it creates uniform sample, so the error of estimate of relative size of a PBEC is guaranteed by Theorem 5.2.

Even that we have bounds on the estimate of the relative size of each PBEC, it is hard to make

[2] 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 $m_{ij} \neq m_{ik}, i \neq k$

tight bounds on the error of the union of the PBECs. Therefore, we show the speedups of our new method and leave the theoretical analyse of the estimate of the relative size of a union of PBECs as an opened problem, showing only experimental results of the quality of the estimates.

Method 1 The PARALLEL-FIMI-RESERVOIR method

PARALLEL-FIMI-RESERVOIR **In:** Double $min\_support^*$,
In: Doubles $\epsilon_{\widetilde{\mathcal{D}}}, \delta_{\widetilde{\mathcal{D}}}, \epsilon_{\widetilde{\mathcal{F}}_s}, \delta_{\widetilde{\mathcal{F}}_s}$,
In: Doubles ρ, α,
Out: Set $\mathcal{F}$)

1: **for** all p_i **do-in-parallel**
   // Phase 1: sampling.
2: Read D_i and set $N_{\widetilde{\mathcal{D}}} \leftarrow \frac{1}{2\epsilon^2_{\widetilde{\mathcal{D}}}} \ln \frac{2}{\delta_{\widetilde{\mathcal{D}}}}$.
3: Creates a sample $D'_i \subseteq D_i$ and broadcast it to each other processor.
4: $\widetilde{\mathcal{D}} \leftarrow \bigcup_i D'_i$.
5: Execute in parallel an arbitrary algorithm for mining of FIs on database $\widetilde{\mathcal{D}}$ in parallel and create the sample $\widetilde{\mathcal{F}}_s$ using the VITTER-RESERVOIR-SAMPLING.
   // Phase 2: partitioning.
6: p_1 creates PBECs $[U_k]$ such that $\frac{|[U_k] \cap \widetilde{\mathcal{F}}_s|}{|\widetilde{\mathcal{F}}_s|} < \alpha \cdot \frac{1}{P}$.
7: p_1 creates $L_j, 1 \leq j \leq P$ using the LPT-MAKESPAN algorithm.
8: *// Phase 3: data re-distribution.*
9: Redistribute the database partition D_i
10: *// Phase 4: parallel computation of FIs.*
11: compute support of $W \subseteq U_k$ in D_i and send the supports to p_1
12: p_1 outputs W
13: all p_i executes an arbitrary algorithm for mining of FIs in parallel that computes supports of $Supp(W, D'_q), W \in \bigcup_{k \in L_q} [U_k|\Sigma_k], (U_k, \Sigma_k) \in Q$.
14: **end for**

9 Database replication

At the begginning of the computation, each processor p_i has a database partition D_i such that $D_i \cap D_j = \emptyset$ and $|D_i| \approx |\mathcal{D}|/P$. That is: the database is almost perfectly distributed among the processors. In Phase 3, we have to redistribute the database $\mathcal{D}$ so each processor can compute its FIs.

We define the database replication factor as a real number that determines the number of copies of a database that is spread among the processors. Let D'_i is the database partition received by p_i in Phase 3. The database replication factor is defined as:

$$\frac{\sum_{i=1}^{P} |D'_i|}{|\mathcal{D}|}$$

At the begginning of the computation the replication factor is almost 1 (an ideal case). The question is, how the replication factor changes after the database redistribution in Phase 3. Unfortunatelly, the database replication factor for *artificial* datasets is almost $\approx P$, i.e., each processor having almost the whole database $\mathcal{D}$. Minimalization of the database replication factor is a hard optimalization task.

10 Experimental evaluation

We have measured the speedup of our new method, the PARALLEL-FIMI-RESERVOIR method, on a cluster of workstations using three datasets.

The cluster of workstations was interconnected with the Infiniband network. Every node out of 8 nodes in the cluster has two dual-core 2.6GHz AMD Opteron processors and 8GB of main memory.

We have implemented the PARALLEL-FIMI-PAR method and the PARALLEL-FIMI-RESERVOIR method in `C++` using the `g++` compiler version 4.4.3 and `STLPort` library (an implementation of Standart Template Libarary). The sequential algorithm used in the PARALLEL-FIMI-RESERVOIR method is the ECLAT algorithm. We have also implemented a modified ECLAT algorithm, used in Phase 1 for making the sample using the VITTER-RESERVOIR-SAMPLING algorithm. As the algorithm for mining of MFIs in Phase 1 of the PARALLEL-FIMI-PAR method was used the *fpmax** [17]. We have also implemented a modified *fpmax** algorithm that executes in parallel.

The datasets were generated using the IBM database generator. 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 (9.37 hours) and 132186 (36.71 hours) seconds. 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 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 experiments were performed with various values of the support parameter on 2, 4, 6, and 10 processors. We have used the follow-

ing datasets: `T500I0.1P100PL20TL50` with minimal supports $0.11, 0.12, 0.13, 0.14, 0.15, 0.16, 0.17, 0.18$; `T500I0.4P250PL10TL120` with minimal supports $0.2, 0.25, 0.26, 0.27, 0.3$; and `T500I1P100PL20TL50` with minimal supports $0.02, 0.03, 0.05, 0.07, 0.09$.

10.1 The error of the estimate of the relative size of a union of PBECs In Section 5.2, we have discussed that we have to bound the error of the estimate the union of PBECs. We need to have the size $\frac{|\bigcup_{j\in L_i}[U_j]\cap\mathcal{F}|}{|\mathcal{F}|} \approx 1/P$. Therefore, we show a graph of the errors $\left|\frac{1}{P} - \frac{|\bigcup_{j\in L_i}[U_j]\cap\mathcal{F}|}{|\mathcal{F}|}\right|$. This graph is the very important for our work because it shows the probability of the error of the whole process, i.e., the error of the estimation of the union of PBECs using $\widetilde{\mathcal{D}}$ and $\widetilde{\mathcal{F}}_s$.

In Figures 3 and 4 is shown the probability of the errors for our three datasets. On the x-axis is the error and on the y-axis is the probability of the error. We have made the experiments with database sample sizes $|\widetilde{\mathcal{D}}| = 42586$, shown as red curves, and $|\widetilde{\mathcal{D}}| = 14450$, shown as blue curves. We have chosen the size of the sample $|\widetilde{\mathcal{F}}_s| = 1001268$ (solid line) and $|\widetilde{\mathcal{F}}_s| = 26492$ (dashed line). We have chosen $P = 5$ and $P = 10$. Each line is a mixture of measurements for different values of minimal support, i.e., we have measured the size of a union of PBECs for different values of minimal support for each dataset and made the graphs from all these measurements. From the graphs can be seen that the most important factor of the precision of the estimate is the size of the database sample. Having a bigger database sample implies smaller probabbility of the error. From the graphs can be also seen in most cases that having a very large sample $\widetilde{\mathcal{F}}_s$ gives only slightly smaller probability of the error or no improvement at all.

10.2 Evaluation of the speedup Figure 1 clearly demonstrate that for reasonably large and reasonably structured datasets, the speedup is linear with average speedup ≈ 6 on 10 processors with maximum value of speedup ≈ 8.6 on 10 processors. The numeric values of the speedup are located in Table 1.

In [4], we have evaluated the Parallel-FIMI-Par as faster then the Parallel-FIMI-Seq method. We can compare the speedup of the Parallel-FIMI-Reservoir method with the Parallel-FIMI-Par method. In the Parallel-FIMI-Par method we have used the Eclat algorithm in Phase 4 and the *fpmax** [17] algorithm in Phase 1 as the algorithm for mining of MFIs. The speedup of Parallel-FIMI-Par method is shown in Figure 1 the numerical average speedup values are located in Table 1. We can see that the speedup of the Parallel-FIMI-Par is a bit smaller then the speedup of Parallel-FIMI-Reservoir. Additionally, in some cases, we were not able to finish the execution of the Parallel-FIMI-Par due to large amount of memory used by the MFIs. In such cases the speedup is shown to be 0.

In the two cases with the computational time > 9 hours the algorithm exhibits very good performance with speedup 8 for $P = 10$.

11 Conclusion and future work

In this paper, we have proposed a static load-balancing method for parallel mining of all FIs. Our method needs a sequential algorithm for mining of all FIs and a reservoir sampling algorithm. In our previous papers, we have proposed another two methods that are based on the modified coverage algorithm. We have demostrated that our new method has a clear advantage of being less memory consuming and having a better speedup than our two previous methods.

There is still a room for improvement. Instead of using a modified sequential algorithm for mining of all FIs in Phase 1, we can use a faster algorithm that does not need to count the support of *all* FIs. It is possible that we can use a modified algorithm for mining of MFIs that enumerates *all* FIs, but counts support only for handsome of FIs in order to take the sample of all FIs.

12 Acknowledgment

This paper was supported from the Czech Science Foundation, grant number GA ČR P202/10/1333.

References

[1] 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.

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

[3] Robert Kessl and Pavel Tvrdík. Probabilistic load balancing method for parallel mining of all frequent itemsets. In *PDCS '06: Proceedings of the 18th IASTED International Conference on Parallel and Distributed Computing and Systems*, pages 578–586, Anaheim, CA, USA, 2006. ACTA Press.

[4] Robert Kessl and Pavel Tvrdík. Toward more parallel frequent itemset mining algorithms. In *PDCS '07: Proceedings of the 19th IASTED International Conference on Parallel and Distributed Computing and Systems*, pages 97–103, Anaheim, CA, USA, 2007. ACTA Press.

[5] 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.

[6] 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.

[7] 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.

[8] A. Javed and A. Khokhar. Frequent Pattern Mining on Message Passing Multiprocessor Systems. *Distributed and Parallel Databases*, 16(3):321–334, November 2004.

[9] 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.

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

[11] J. S. Vitter. Random sampling with a reservoir. *ACM Trans. Math. Softw.*, 11(1):37–57.

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

[13] M. Skala. Hypergeometric tail inequalities: ending the insanity. Published on-line. `http://ansuz.sooke.bc.ca/professional/hypergeometric.pdf`.

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

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

[16] `http://en.wikipedia.org/wiki/Round_robin_tournament`.

[17] 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.

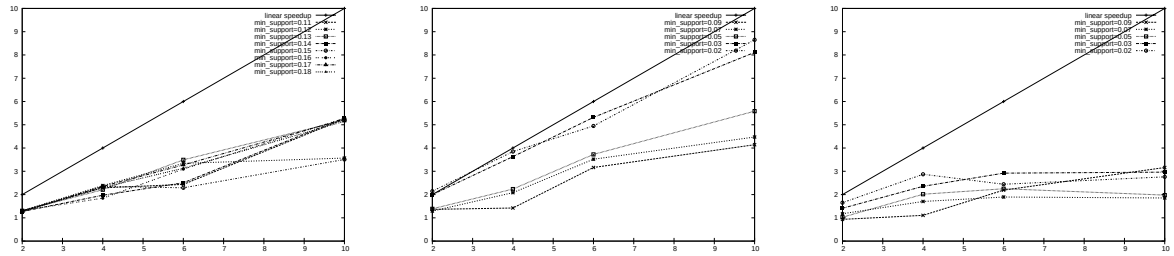

Figure 1: Speedup of the Parallel-FIMI-Reservoir method parametrized with the Eclat algorithm, measured on the `T500I0.1P100PL20TL50`, `T500I0.4P250PL20TL80`, `T500I1P100PL20TL50` (from left to right)

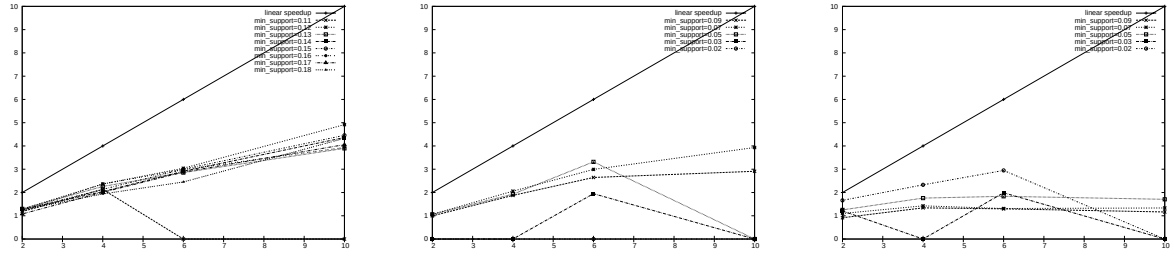

Figure 2: Speedup of the Parallel-FIMI-Par method parametrized with the Eclat algorithm, measured on the `T500I0.1P100PL20TL50`, `T500I0.4P250PL20TL80`, `T500I1P100PL20TL50` (from left to right)

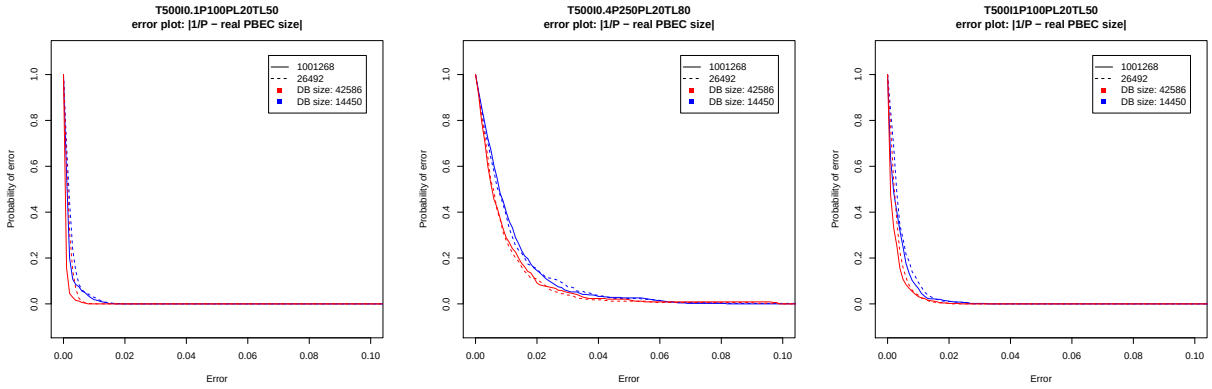

Figure 3: Error of the estimation of the relative size of a union of a PBECs, for $P = 10$. The datasets `T500I0.1P100PL20TL50`, `T500I0.4P250PL20TL80`, `T500I1P100PL20TL50` (from left to right)

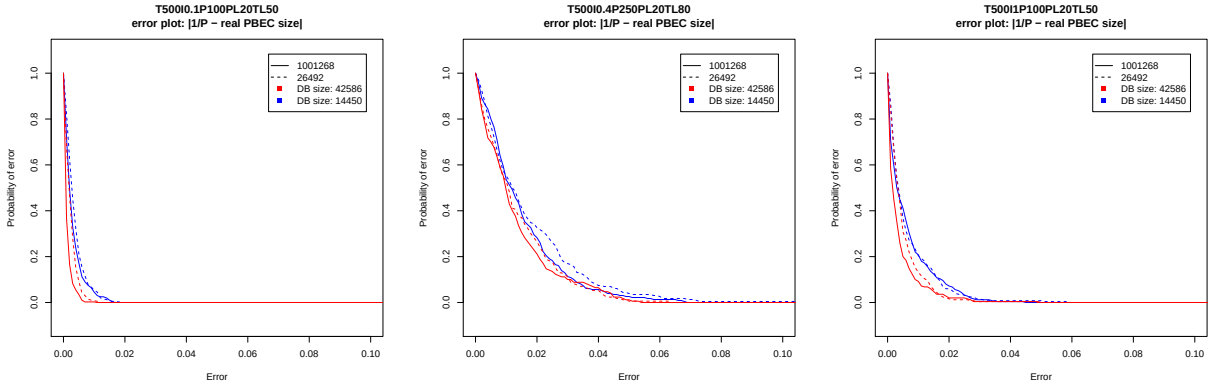

Figure 4: Error of the estimation of the relative size of a union of a PBECs, for $P = 5$. The datasets `T500I0.1P100PL20TL50`, `T500I0.4P250PL20TL80`, `T500I1P100PL20TL50` (from left to right)

datafile/Parallel-FIMI-Reservoir	2	4	6	10
T500I0.1P50PL10TL40	1.523	2.633	3.380	5.342
T500I0.4P250PL20TL80	1.637	2.644	4.135	6.191
T500I1P100PL20TL50	1.240	2.010	2.340	2.544
Total average	1.466	2.428	3.285	4.692

datafile/Parallel-FIMI-Par	2	4	6	10
T500I0.1P50PL10TL40	1.596	2.668	3.438	5.135
T500I0.4P250PL20TL80	1.039	1.954	2.726	3.422
T500I1P100PL20TL50	1.227	1.714	1.876	1.401
Total average	1.304	2.112	2.679	3.193

Table 1: Numerical values of average speedup of the Parallel-FIMI-Reservoir and Parallel-FIMI-Par methods for number of processors $P = 2, 4, 6, 10$