Ann Math Artif Intell (2010) 59:257–272
DOI 10.1007/s10472-010-9199-5

Parallel algorithm for computing fixpoints of Galois connections

Petr Krajca · Jan Outrata · Vilem Vychodil

Published online: 10 July 2010

Abstract This paper presents a parallel algorithm for computing fixpoints of Galois connections induced by object-attribute relational data. The algorithm results as a parallelization of CbO (Kuznetsov 1999) in which we process disjoint sets of fixpoints simultaneously. One of the distinctive features of the algorithm compared to other parallel algorithms is that it avoids synchronization which has positive impacts on its speed and implementation. We describe the parallel algorithm, prove its correctness, and analyze its asymptotic complexity. Furthermore, we focus on implementation issues, scalability of the algorithm, and provide an evaluation of its efficiency on various data sets.

Keywords Galois connection · Fixpoint · Formal concept · Parallel algorithm

Mathematics Subject Classifications (2010) 03G10 · 62H30 · 11Y16

Supported by research plan MSM 6198959214. Partly supported by grant P103/10/1056 of the Czech Science Foundation.

P. Krajca · J. Outrata · V. Vychodil (✉)
Department of Computer Science, Palacky University, Olomouc, Czech Republic
e-mail: vychodil@binghamton.edu

P. Krajca
e-mail: petr.krajca@binghamton.edu

J. Outrata
e-mail: jan.outrata@upol.cz

1 Introduction

We propose a parallel algorithm for computing all fixpoints of Galois connections induced by object-attribute incidence data. The fixpoints, called *formal concepts* [8, 19], represent fundamental rectangular patterns that can be found in the data. Besides their geometrical meaning, the fixpoints can be interpreted as formalizations of natural concepts found in the input incidence data: each formal concept is given by its *extent*, i.e. a set of all objects that fall under the concept, and *intent*, i.e. a set of all attributes (features) that are covered by the concept. The set of all formal concepts equipped with a subconcept–superconcept ordering forms a complete lattice which is commonly called a *concept lattice.* Concept lattices and related incidence structures are thoroughly studied by formal concept analysis—a discipline founded by Rudolf Wille in the early 1980s. Since then, many theoretical results and applications of formal concept analysis (FCA) appeared, see monograph [8] and a recent book [5] for an overview.

The basic task which appears in virtually any application of FCA is to take the input incidence data and compute the set of all formal concepts. The incidence data is represented by a binary relation $I \subseteq X \times Y$ between a set X of objects and a set Y of attributes (features). The data can be depicted by a two-dimensional table with rows corresponding to objects, columns corresponding to attributes, and table entries being ones and zeros indicating presence/absence of attributes. The limiting factor of listing all formal concepts is that the problem is apparently hard as the associated counting problem is $\#P$-complete [13]. Fortunately, if $|I|$ is considerably small, one can get sets of all formal concepts in reasonable time even if X and Y are large. The latter observation resulted in efforts of developing algorithms for FCA specialized on sparse incidence data.

This paper contributes to the family of algorithms for FCA by showing a clear and efficient way to parallelize the computation of concepts by splitting the set of all formal concepts into disjoint subsets which can be computed simultaneously with a minimal overhead. Our motivation for focusing on a parallel algorithm is twofold. First, one of the main problems of FCA is how to deal with large-scale data. The problem has become important recently as FCA is increasingly popular in the data-mining community as a preprocessing technique. Efficient parallelization and distribution over network may help overcome problems with delivering results in a reasonable time (for input data of reasonable size). Second, parallel computing is recently gaining interest as hardware manufactures are shifting their focus from improving computing power by increasing clock frequencies to developing processors with multiple cores. As the multiprocessor systems are becoming more affordable, there will be an increasing pressure to deliver parallel algorithms to better utilize the hardware. From these two points of view, research on parallel algorithms for FCA seems to be promising.

There are several algorithms for computing formal concepts which are closely related to our algorithm. Our algorithm can be seen as a parallelization of a simplified version of CbO [14, 15] and the algorithm proposed by Norris [18]. Our algorithm uses the same canonicity test for avoiding to process the same concept multiple times. This idea also appears in Ganter's algorithm [7] but our algorithm produces

formal concepts in a different order. A detailed comparison will be presented in Section 3.

The paper is organized as follows. In Section 2 we recall notions from formal concept analysis. Section 3 describes the algorithm, shows its correctness, and presents comments on the relationship to other algorithms. Furthermore, in Section 4 we discuss complexity and efficiency issues of the algorithm both theoretically and experimentally. We focus on the scalability of the algorithm, i.e. the growth of its performance with respect to the growing number of processors.

2 Preliminaries and notation

In this section we recall basic notions of the formal concept analysis. More details can be found in monographs [8, 9] and [5]. Let $X = \{0, 1, \ldots, m\}$ and $Y = \{0, 1, \ldots, n\}$ denote finite nonempty sets of objects and attributes, respectively. A formal context is a triplet $\langle X, Y, I\rangle$ where $I \subseteq X \times Y$, i.e. I is a binary relation between X and Y. As usual, given $\langle X, Y, I\rangle$, we consider a pair of concept-forming operators [8] ${}^{\uparrow_I}\colon 2^X \to 2^Y$ and ${}^{\downarrow_I}\colon 2^Y \to 2^X$ defined, for each $A \subseteq X$ and $B \subseteq Y$, by $A^{\uparrow_I} = \{y \in Y \mid \text{for each } x \in A\colon \langle x, y\rangle \in I\}$ and $B^{\downarrow_I} = \{x \in X \mid \text{for each } y \in B\colon \langle x, y\rangle \in I\}$, respectively. If there is no danger of confusion, we omit I and write just ${}^{\uparrow}$ and ${}^{\downarrow}$ instead of ${}^{\uparrow_I}$ and ${}^{\downarrow_I}$, respectively. By a formal concept (in $\langle X, Y, I\rangle$) with extent A and intent B we mean any pair $\langle A, B\rangle \in 2^X \times 2^Y$ such that $A^{\uparrow_I} = B$ and $B^{\downarrow_I} = A$. Thus, formal concepts are fixpoints of the concept-forming operators. The set of all fixpoints of $\langle {}^{\uparrow_I}, {}^{\downarrow_I}\rangle$ will be denoted by $\mathcal{B}(X, Y, I)$. The set $\mathcal{B}(X, Y, I)$ of all formal concepts in $\langle X, Y, I\rangle$ can be equipped with a partial order $\leq$ modeling the subconcept–superconcept hierarchy:

$$\langle A_1, B_1\rangle \leq \langle A_2, B_2\rangle \text{ iff } A_1 \subseteq A_2 \text{ (or, equivalently, iff } B_2 \subseteq B_1). \tag{1}$$

If $\langle A_1, B_1\rangle \leq \langle A_2, B_2\rangle$ then $\langle A_1, B_1\rangle$ is called a subconcept of $\langle A_2, B_2\rangle$. The set $\mathcal{B}(X, Y, I)$ together with $\leq$ defined by (1) form a complete lattice whose structure is described by the Basic Theorem of FCA [8]. For the purpose of illustration, we are going to use the following

Example 1 Consider a formal context $\langle X, Y, I\rangle$ corresponding to the incidence data table from Fig. 1 (left). The concept-forming operators induced by this context have exactly 15 fixpoints (formal concepts) $C_1, \ldots, C_{15}$:

$C_1 = \langle X, \emptyset\rangle$, $C_6 = \langle\{4\}, \{0, 1, 4, 5, 6, 7\}\rangle$, $C_{11} = \langle\{0, 2\}, \{1, 2, 5\}\rangle$,
$C_2 = \langle\{1, 2, 4\}, \{0, 6\}\rangle$, $C_7 = \langle\{1, 2\}, \{0, 3, 6\}\rangle$, $C_{12} = \langle\{0\}, \{1, 2, 4, 5, 7\}\rangle$,
$C_3 = \langle\{2, 4\}, \{0, 1, 5, 6\}\rangle$, $C_8 = \langle\{1\}, \{0, 3, 6, 7\}\rangle$, $C_{13} = \langle\{0, 3, 4\}, \{1, 4, 5\}\rangle$,
$C_4 = \langle\{2\}, \{0, 1, 2, 3, 5, 6\}\rangle$, $C_9 = \langle\{1, 4\}, \{0, 6, 7\}\rangle$, $C_{14} = \langle\{0, 4\}, \{1, 4, 5, 7\}\rangle$,
$C_5 = \langle\emptyset, Y\rangle$, $C_{10} = \langle\{0, 2, 3, 4\}, \{1, 5\}\rangle$, $C_{15} = \langle\{0, 1, 4\}, \{7\}\rangle$.

I	0	1	2	3	4	5	6	7
0	0	1	1	0	1	1	0	1
1	1	0	0	1	0	0	1	1
2	1	1	1	1	0	1	1	0
3	0	1	0	0	1	1	0	0
4	1	1	0	0	1	1	1	1

I	0	1	2	3	4	5	6	7
0	0	1	1	0	1	1	0	1
1	1	0	0	1	0	0	1	1
2	1	1	1	1	0	1	1	0
3	0	1	0	0	1	1	0	0
4	1	1	0	0	1	1	1	1

Fig. 1 Formal context (*left*) and maximal rectangles (*right*) corresponding to C_9 and C_{13}

Hence, $\mathcal{B}(X, Y, I) = \{C_1, \ldots, C_{15}\}$. If we equip $\mathcal{B}(X, Y, I)$ with the partial order (1), the resulting structure is the concept lattice of $\langle X, Y, I \rangle$.

Note that formal concepts in $\langle X, Y, I \rangle$ correspond to so-called maximal rectangles [8] in $\langle X, Y, I \rangle$, cf. Fig. 1 (right).

3 Algorithm for computing all fixpoints

In this section we describe the algorithm for computing all fixpoints of a Galois connection. We start by describing a subroutine which can be seen as a serial version of the algorithm. The main idea behind the serial subroutine of our algorithm is the same as in case of the algorithm Close-by-One (CbO) proposed by Kuznetsov in [15]. The parallel algorithm can be seen as several instances of the serial version working simultaneously on disjoint subsets of concepts. Since Galois connections induced by formal contexts are in fact the most general ones, we focus on fixpoints of $\langle {}^{\uparrow_I}, {}^{\downarrow_I} \rangle$ for a given formal context $\langle X, Y, I \rangle$ such that $X = \{0, 1, \ldots, m\}$ and $Y = \{0, 1, \ldots, n\}$.

Algorithm 1 Procedure GenerateFrom $(\langle A, B \rangle, y)$

```
   Input: formal concept ⟨A, B⟩ and a number y ∈ Y ∪ {n + 1} such that y ∉ B
 1 process ⟨A, B⟩ (e.g., print ⟨A, B⟩ on the screen);
 2 if B = Y or y > n then
 3 |  return
 4 end
 5 for j from y upto n do
 6 |  if j ∉ B then
 7 |  |  set C to A ∩ {j}↓;
 8 |  |  set D to C↑;
 9 |  |  if B ∩ Y_j = D ∩ Y_j then
10 |  |  |  GenerateFrom(⟨C, D⟩, j + 1);
11 |  |  end
12 |  end
13 end
14 return
```

The core of the serial algorithm is a recursive procedure GenerateFrom, see Algorithm 1, which lists all formal concepts using a depth-first search through the space of all formal concepts. The procedure accepts a formal concept $\langle A, B \rangle$ (an

initial formal concept) and an attribute $y \in Y$ (first attribute to be processed) as its arguments. The procedure recursively descends through the space of formal concepts, beginning with the formal concept $\langle A, B\rangle$.

When invoked with $\langle A, B\rangle$ and $y \in Y$, GenerateFrom first processes $\langle A, B\rangle$ (e.g., prints it on the screen or stores it in a data structure, see line 1 of Algorithm 1) and then it checks its halting condition, see lines 2–4. According to the halting condition, the computation stops either when $\langle A, B\rangle$ equals $\langle Y^{\downarrow}, Y\rangle$ (the least formal concept has been reached) or $y > n$ (there are no more remaining attributes to be processed). Otherwise, the procedure goes through all attributes $j \in Y$ such that $j \geq y$ which are not contained in the intent B (see lines 5 and 6). For each $j \in Y$ having these properties, a new pair $\langle C, D\rangle \in 2^X \times 2^Y$ such that

$$\langle C, D\rangle = \langle A \cap \{j\}^{\downarrow}, (A \cap \{j\}^{\downarrow})^{\uparrow}\rangle \tag{2}$$

is computed (lines 7 and 8). One can show that $\langle C, D\rangle$ is always a formal concept such that $B \subset D$ (see Remark 1 below). After obtaining $\langle C, D\rangle$, the algorithm checks whether it should continue with $\langle C, D\rangle$ by recursively calling GenerateFrom or whether $\langle C, D\rangle$ should be "skipped". The test is based on comparing $B \cap Y_j = D \cap Y_j$ where $Y_j \subseteq Y$ is defined as follows:

$$Y_j = \{y \in Y \mid y < j\}. \tag{3}$$

The role of the test (see lines 9–11) is to prevent processing the same formal concept multiple times. In the sequel we prove that GenerateFrom computes formal concepts in a unique order which ensures that each formal concept is processed exactly once.

Remark 1 If $\langle A, B\rangle$ is a formal concept then $\langle C, D\rangle$ computed in lines 7 and 8 of Algorithm 1 is also a formal concept such that $B \subset D$ and $C \subset A$ provided that $j \notin B$. Indeed, $D = C^{\uparrow}$ by definition. Moreover, $C = A \cap \{j\}^{\downarrow} = B^{\downarrow} \cap \{j\}^{\downarrow} = (B \cup \{j\})^{\downarrow}$. Since $^{\downarrow\uparrow\downarrow}$ equals $^{\downarrow}$, we get $D^{\downarrow} = C^{\uparrow\downarrow} = (B \cup \{j\})^{\downarrow\uparrow\downarrow} = (B \cup \{j\})^{\downarrow} = C$, i.e. $\langle C, D\rangle$ is a formal concept. The facts $B \subset D$ and $C \subset A$ follow from properties of the concept-forming operators $^{\downarrow}$ and $^{\uparrow}$ using $j \notin B$.

In order to prove the correctness of Algorithm 1, we introduce so-called derivations which will correspond to recursive invocations of the procedure GenerateFrom. Later, the derivations will be used to describe the parallel algorithm.

Definition 1 (Derivations of Formal Concepts) Let $\langle X, Y, I\rangle$ be a formal context with $Y = \{0, \ldots, n\}$. For formal concepts $\langle A_1, B_1\rangle, \langle A_2, B_2\rangle \in \mathcal{B}(X, Y, I)$ and integers $y_1, y_2 \in Y \cup \{n+1\}$ let $\langle\langle A_1, B_1\rangle, y_1\rangle \vdash \langle\langle A_2, B_2\rangle, y_2\rangle$ denote that for $m = y_2 - 1$ the following conditions

(i) $m \notin B_1$,
(ii) $y_1 < y_2$,

(iii) $B_2 = (B_1 \cup \{m\})^{\downarrow\uparrow}$, and
(iv) $B_1 \cap Y_m = B_2 \cap Y_m$, where Y_m is defined by (3)

are all satisfied. A *derivation of* $\langle A, B\rangle \in \mathcal{B}(X, Y, I)$ *of length* $k+1$ is any sequence

$$\langle\langle \emptyset^{\downarrow}, \emptyset^{\downarrow\uparrow}\rangle, 0\rangle = \langle\langle A_0, B_0\rangle, y_0\rangle, \langle\langle A_1, B_1\rangle, y_1\rangle, \ldots, \langle\langle A_k, B_k\rangle, y_k\rangle = \langle\langle A, B\rangle, y_k\rangle \quad (4)$$

such that $\langle\langle A_i, B_i\rangle, y_i\rangle \vdash \langle\langle A_{i+1}, B_{i+1}\rangle, y_{i+1}\rangle$ for each $i = 0, \ldots, k-1$. If $\langle A, B\rangle$ has a derivation of length k we say that $\langle A, B\rangle$ is *derivable in k steps.*

It is easily seen that $\langle\langle A, B\rangle, y\rangle \vdash \langle\langle C, D\rangle, k\rangle$ iff the invocation of GenerateFrom$(\langle A, B\rangle, y)$ causes GenerateFrom$(\langle C, D\rangle, k)$ to be called in line 10. Indeed (i) ensures that the condition in line 6 of Algorithm 1 is satisfied, (ii) corresponds to the fact that the loop between lines 5–13 goes from y upwards, (iii) is the intent computed in line 8, and (iv) is true iff the condition in line 9 is true. Algorithm 1 and derivations are further demonstrated by the following example.

Example 2 Consider the formal context $\langle X, Y, I\rangle$ from Fig. 1 (left). According to Example 1, denote $\langle \emptyset^{\downarrow}, \emptyset^{\downarrow\uparrow}\rangle = \langle X, \emptyset\rangle$ by C_1. If GenerateFrom$(C_1, 0)$ is called, j goes over all attributes from Y, starting with $y = 0$, see line 5. For $j = 0$, new formal concept $\langle C, D\rangle$ with $C = \{1, 2, 4\}$ and $D = \{0, 6\}$ is computed (lines 7 and 8). Denote the concept by C_2. Since $D \cap Y_0 = \emptyset = B \cap Y_0$, i.e. the test in line 9 is successful, GenerateFrom$(C_2, 1)$ is invoked. In terms of derivations, we have $\langle C_1, 0\rangle \vdash \langle C_2, 1\rangle$. During the invocation of GenerateFrom$(C_2, 1)$, j goes over all attributes starting with 1. For $j = 1$, we get $C = \{2, 4\}$, $D = \{0, 1, 5, 6\}$. Since $\{0, 6\} \cap \{0\} = \{0, 1, 5, 6\} \cap \{0\}$, the test is successful and GenerateFrom$(C_3, 2)$ is invoked where C_3 denotes $\langle\{2, 4\}, \{0, 1, 5, 6\}\rangle$. Thus, $\langle C_2, 1\rangle \vdash \langle C_3, 2\rangle$. In a similar way we get $\langle C_3, 2\rangle \vdash \langle C_4, 3\rangle$ and $\langle C_4, 3\rangle \vdash \langle C_5, 5\rangle$. When GenerateFrom$(C_5, 5)$ is invoked, all attributes are already present in the intent, i.e., the invocation of GenerateFrom$(C_5, 5)$ is terminated and the computation goes back to GenerateFrom$(C_4, 3)$ with $j \geq 5$. Since the intent of C_4 contains both 5 and 6, we continue with $j = 7$, for which we obtain a formal concept $\langle C, D\rangle = \langle \emptyset, Y\rangle = C_5$ which has already been found. In this case, the test in line 9 fails because $B \cap Y_7 = \{0, 1, 2, 3, 5, 6\} \neq \{0, 1, 2, 3, 4, 5, 6\} = D \cap Y_7$. Therefore, the invocation of GenerateFrom$(C_4, 3)$ is terminated because $j = n = 7$ is the last attribute and the computation proceeds with GenerateFrom$(C_3, 2)$ with $j \geq 3$. For $j = 3$, we obtain a concept $\langle C, D\rangle = C_4$ which has also been found and the test fails because $B \cap Y_3 = \{0, 1\} \neq \{0, 1, 2\} = D \cap Y_3$. For $j = 4$, we obtain a new concept $\langle C, D\rangle = \langle\{4\}, \{0, 1, 4, 5, 6, 7\}\rangle = C_6$ which has not been considered so far. The test succeeds, GenerateFrom$(C_6, 5)$ is invoked, meaning $\langle C_3, 2\rangle \vdash \langle C_6, 5\rangle$, and the computation continues in a similar way as before.

Remark 2 The computation of Algorithm 1 and the corresponding derivations can be depicted by a tree as in Fig. 2. The tree contains two types of nodes. Nodes represented by pairs $\langle C_i, y_i\rangle$ represent arguments of GenerateFrom, i.e. each node of this type represents an invocation of GenerateFrom. Leaf nodes denoted by black squares represent computed concepts for which the test in line 9 fails. Each edge in the tree is labeled by the current value of j which is used to compute a (new) formal concept, see lines 7 and 8. We call such a tree a call tree of GenerateFrom for given

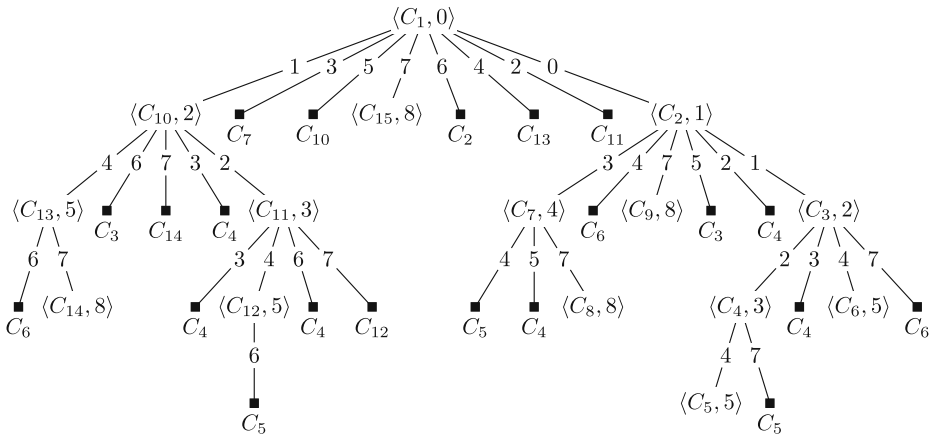

Fig. 2 Example of a call tree for GenerateFrom($\langle \emptyset^{\downarrow}, \emptyset^{\downarrow\uparrow} \rangle, 0$) with input data from Fig. 1

$\langle X, Y, I \rangle$. A path from the root of the tree to any node labeled by $\langle C_i, y_i \rangle$ corresponds to a derivation of $\langle C_i, y_i \rangle$. Later, we prove that the nodes labeled by $\langle C_i, y_i \rangle$ are always in a one-to-one correspondence with formal concepts in $\mathcal{B}(X, Y, I)$, showing that the algorithm is correct.

The following assertions show the existence and uniqueness of derivations.

Lemma 1 (Existence of Derivations) *For each formal concept* $\langle A, B \rangle \in \mathcal{B}(X, Y, I)$ *there is a derivation* (4) *such that* $y_i = m_i + 1$ *where*

$$m_i = \min\{y \in B \mid y \notin B_{i-1}\} \tag{5}$$

for each $0 < i \leq k$.

Proof We prove by induction over i that $\langle \langle A_0, B_0 \rangle, y_0 \rangle, \ldots, \langle \langle A_i, B_i \rangle, y_i \rangle$ is a derivation. Assume that the claim holds for $0, \ldots, i-1 < k$. We prove that it holds for i. Since $i - 1 < k$, $B \setminus B_{i-1} \neq \emptyset$. Therefore, m_i given by (5) and consequently $y_i = m_i + 1$ are well defined. Put $B_i = (B_{i-1} \cup \{m_i\})^{\downarrow\uparrow}$ and $A_i = A_{i-1} \cap \{m_i\}^{\downarrow}$. We now prove that $\langle \langle A_{i-1}, B_{i-1} \rangle, y_{i-1} \rangle \vdash \langle \langle A_i, B_i \rangle, y_i \rangle$ by checking Definition 1 (i)–(iv). Using (5), $y_i - 1 = m_i \notin B_{i-1}$, i.e. (i) is true. In order to prove (ii), we check that $m_{i-1} < m_i$. By contradiction, assume that $m_i \leq m_{i-1}$. Obviously, $m_{i-1} \neq m_i$ because $m_i \notin B_{i-1}$ and $m_{i-1} \in B_{i-1}$. Thus, assume $m_i < m_{i-1} \neq 0$. Since $m_i \notin B_{i-1}$ and $B_{i-2} \subset B_{i-1}$, we get $m_i \notin B_{i-2}$. Using the induction hypothesis, $m_{i-1} = \min\{y \in B \mid y \notin B_{i-2}\}$ which contradicts the facts that $m_i < m_{i-1}$ and $m_i \notin B_{i-2}$, proving (ii). Condition (iii) agrees with the definition of B_i. It remains to check that $B_{i-1} \cap Y_{m_i} = B_i \cap Y_{m_i}$. Since $B_{i-1} \subset B_i = (B_{i-1} \cup \{m_i\})^{\downarrow\uparrow} \subseteq B$, m_i is a minimum attribute such that $m_i \in B_i$ and $m_i \notin B_{i-1}$. That is, for each $y < m_i$, $y \in B_{i-1}$ iff $y \in B_i$. The latter is equivalent to $B_{i-1} \cap Y_{m_i} = B_i \cap Y_{m_i}$, showing (iv). □

Lemma 2 (Uniqueness of Derivations) *Each formal concept* $\langle A, B\rangle \in \mathcal{B}(X, Y, I)$ *has at most one derivation.*

Proof According to Lemma 1, we prove that each derivation of $\langle A, B\rangle$ equals to (4) where $y_i = m_i + 1$ and m_i are given by (5). By contradiction, let

$$\langle\langle A'_0, B'_0\rangle, y'_0\rangle, \langle\langle A'_1, B'_1\rangle, y'_1\rangle, \ldots, \langle\langle A'_l, B'_l\rangle, y'_l\rangle$$

be another derivation of $\langle A, B\rangle$. Let i by the index such that $y_j = y'_j$ for all $j < i$ and $y_i \neq y'_i$. It is easily seen that $A_j = A'_j$ and $B_j = B'_j$ for all $j < i$. Furthermore, for m_i given by (5), we get $m_i \in B_i$. The observations that $m_i \notin B_j = B'_j$ for all $j < i$ and that m_i is the minimum attribute in $B \setminus B_{i-1} = B \setminus B'_{i-1}$ yield $m_i \notin B'_i$ because otherwise $B'_{i-1} \cap Y_{y'_i - 1} = B'_i \cap Y_{y'_i - 1}$ would be violated. On the other hand, $m_i \in B = B'_l$, i.e. there must be an index $j > i$ such that $m_i \in B'_j$ and $m_i \notin B'_h$ for all $h < j$. In addition to that, we have $m_i < y'_i - 1 < y'_j - 1$. Therefore, $m_i \in B'_j \cap Y_{y'_j - 1}$ and $m_i \notin B'_{j-1} \cap Y_{y'_j - 1}$, contradicting the fact that $B'_{j-1} \cap Y_{y'_j - 1} = B'_j \cap Y_{y'_j - 1}$. □

We now get the following consequence of Lemmas 1 and 2:

Theorem 1 (Correctness of Algorithm 1) *When invoked with* $\langle \emptyset^{\downarrow}, \emptyset^{\downarrow\uparrow}\rangle$ *and* $y = 0$*, Algorithm 1 derives all formal concepts in* $\langle X, Y, I\rangle$*, each of them exactly once.* □

Remark 3 Algorithm 1 can be seen as a simplified version of CbO [14, 15]. We formulate the algorithm by a recursive procedure GenerateFrom rather than by backtracking as it is used in [15]. This has several benefits. First, GenerateFrom is much closer to the actual implementation than the abstract description from [15]. Second, there is no need for explicit labeling of attributes which have been processed, see [15], because each invocation of GenerateFrom has all the necessary information in a local variable *j*. When computing new closures, we improve the efficiency of the algorithm by going through only a subset of all attributes from Y, see line 5 of Algorithm 1. Finally, there is no need to build the CbO-tree [15] as a data structure. The CbO-tree corresponds to the recursive invocations of GenerateFrom: derivations from Definition 1 correspond to canonical paths in the CbO-tree, see [15]. Paths which are not canonical according to [15] can be seen as paths from the root node of the call tree of GenerateFrom to nodes labeled by black squares, see Fig. 2.

Ganter's algorithm [7] is also closely related to our algorithm but it lists formal concepts in a different order. On the other hand, our algorithm can be easily modified to produce formal concepts in the same order with a slight loss of the performance. Indeed, during each invocation of GenerateFrom$(\langle A, B\rangle, y)$ it suffices to (i) build a list $\mathcal{L}$ of all concepts $\langle A_i, B_i\rangle$ such that $\langle\langle A, B\rangle, y\rangle \vdash \langle\langle A_i, B_i\rangle, j_i\rangle$ $(j_i > y)$ without invoking GenerateFrom$(\langle A_i, B_i\rangle, j_i)$, then (ii) sort the list $\mathcal{L}$ according to the lexicographic order [7] on the intents B_i, and (iii) recursively invoke GenerateFrom$(\langle A_i, B_i\rangle, j_i)$ for all $\langle A_i, B_i\rangle$ in the sorted list $\mathcal{L}$ according to the lexicographic order.

We now turn our attention to the parallel algorithm. Assume that we have P *independent processors* which can execute instructions simultaneously. These may represent separate computers in a network or multiple processors in a system with

shared memory. We assume that each processor has access to the context $\langle X, Y, I\rangle$. Since $\langle X, Y, I\rangle$ is not altered during the computation, each processor can have its own copy of $\langle X, Y, I\rangle$ or share one copy among multiple processors (in systems with shared memory).

Algorithm 2 Procedur ParallelGenerateFrom ($\langle A, B\rangle, y, l$)

```
   Input: formal concept ⟨A, B⟩,
          number y ∈ Y ∪ {n + 1} such that y ∉ B,
          level of recursion L ≥ 2,
          number of processors P ≥ 1, and
          counter l such that 1 ≤ l ≤ L
 1 if l = L then
 2 |  select r ∈ {1, ..., P};
 3 |  store ⟨⟨A, B⟩, y⟩ to queue_r;
 4 |  return
 5 end
 6 process ⟨A, B⟩ (e.g., print ⟨A, B⟩ on screen);
 7 if not (B = Y or y > n) then
 8 |  for j from y upto n do
 9 |  |  if j ∉ B then
10 |  |  |  set C to A ∩ {j}↓;
11 |  |  |  set D to C↑;
12 |  |  |  if B ∩ Y_j = D ∩ Y_j then
13 |  |  |  |  ParallelGenerateFrom(⟨C, D⟩, j + 1, l + 1)
14 |  |  |  end
15 |  |  end
16 |  end
17 end
18 if l = 1 then
19 |  for r from 1 upto P do
20 |  |  with processor r
21 |  |  |  foreach ⟨⟨C, D⟩, j⟩ ∈ queue_r do
22 |  |  |  |  GenerateFrom(⟨C, D⟩, j);
23 |  |  |  end
24 |  |  end
25 |  end
26 |  wait for all processors
27 end
28 return
```

The parallelization we propose consists in modification of GenerateFrom so that particular subtrees of the call tree are computed simultaneously by P processors. The idea is best explained when we consider a call tree like the one in Fig. 2. Recall that GenerateFrom is a recursive procedure and its invocations during the computation agree with the nodes labeled $\langle C_i, y_i\rangle$ in the tree. Moreover, the order in which the concepts are processed can be read directly from the call tree. It suffices to go through the $\langle C_i, y_i\rangle$ nodes in the depth-first order following the labels of edges from smallest to biggest numbers. At any level of the call tree, we obtain a set of nodes which are root nodes of disjoint subtrees. For instance, in Fig. 2, the second level of the call tree contains nodes $\langle C_{10}, 2\rangle$, $\langle C_{15}, 8\rangle$, and $\langle C_2, 1\rangle$. Two of the nodes are root nodes of nontrivial subtrees which may be processed independently by two processors. This suggests to modify GenerateFrom so that is goes through the call tree only up to a certain predefined level L and then it lets P independent processors compute the remaining concepts descendant to those on the Lth level. In terms of derivations, see Definition 1, the algorithm first processes all concepts which are derivable in less

than L steps. The remaining concepts are computed in parallel. Therefore, a parallel procedure for computing concepts can be summarized by three consecutive stages:

Stage 1: Compute and process all concepts that are derivable in less than L steps.

Stage 2: Store all concepts derivable in L steps in P independent queues.

Stage 3: Initiate P processors and run the parallel computation: (i) let each of the processors take exactly one of the queues; (ii) let each processor compute all concepts (using Algorithm 1) beginning with those in its queue.

A parallel algorithm following this idea is represented by procedure ParallelGenerateFrom, see Algorithm 2. It is important to note that Algorithm 2 has two parameters which are constant during the computation: $P \geq 1$ (*number of processors*) and $L \geq 2$ (*level of recursion*, i.e. the maximum length of derivations which are computed sequentially in Stage 1). The choice of values of P and L has an influence of the practical performance of the algorithm. This issue will be addressed later. Procedure ParallelGenerateFrom is a modification of GenerateFrom and accepts one additional argument: a *counter* l which goes from 1 up to L and is used to indicate lengths of derivations that are processed in Stage 1.

After its invocation, ParallelGenerateFrom proceeds as follows: The procedure simulates the original GenerateFrom until is reaches the recursion level L, see the code between lines 1–17. This agrees with Stage 1 as outlined above. There are two technical differences between GenerateFrom and ParallelGenerateFrom:

- ParallelGenerateFrom increases the counter l upon each invocation, see line 13. Obviously, if the procedure is initially called with $l = 1$ then during the computation l is always equal to the current recursion level (call tree level). In addition to that, formal concepts that are processed in line 6 are exactly the concepts derivable in less than L steps.
- Instead of returning from the recursion, see the condition in line 7, the procedure continues to the point where the original GenerateFrom ends. This step is taken because ParallelGenerateFrom has to initiate the parallel computation after the first two stages are finished, see lines 18–27.

When l equals L, ParallelGenerateFrom has reached the level of recursion at which the serial algorithm stops, entering the Stage 2. In other words, $l = L$ means that the current formal concept $\langle A, B\rangle$ is derivable in L steps. Instead of processing $\langle A, B\rangle$ in line 6, the procedure performs the code between lines 2–4, i.e., it selects one of the queues numbered $1, \ldots, P$, stores $\langle\langle A, B\rangle, y\rangle$ in the queue, and exits this branch of recursion. During this stage of computation, all formal concepts derivable in L steps are stored in the queues.

Notice that the limit condition in line 1 also ensures that there are only finitely many recursive invocations of ParallelGenerateFrom. Since $L \geq 2$ and the initial value of the counter l equals 1, the initial invocation of ParallelGenerateFrom is never terminated in line 4. As a consequence, after finitely many steps, the initial invocation of ParallelGenerateFrom gets to the line 18. Here, the condition succeeds because $l = 1$. Thus, the initial invocation proceeds with lines 19–26 which take care of initiating the parallel computation: each processor goes over all $\langle\langle A, B\rangle, y\rangle$ in its queue and invokes the serial procedure GenerateFrom with $\langle A, B\rangle$ and y as its arguments. The only synchronization that is used in the algorithm is that the initial invocation waits until all processors finish the computation, see line 26. Also note that

the condition in line 1 ensures that the parallel computation will be initiated exactly once because there is only one invocation of ParallelGenerateFrom with $l = 1$.

Remark 4 The key issue with Algorithm 2 is how to distribute formal concepts derivable in L steps into P queues. In fact, by selecting a queue in which we put $\langle\langle C, D\rangle, y\rangle$ we select a processor which will list all formal concepts descendant to $\langle C, D\rangle$. The optimal selection method should distribute all formal concepts to processors uniformly. This is, however, very hard to achieve since we do not know the distribution of formal concepts in the search space of all formal concepts until we actually compute them all and reveal the structure of the call tree. In the present version of the algorithm we select $queue_r$ based on a simple round-robin principle: the index r is computed as $r = (N \bmod P) + 1$ where N denotes the number of formal concepts stored so far. This principle, albeit simple, turned out to be efficient for both the real-world datasets and randomly generated data, see Section 4.

Our algorithm can be seen as having two parts: first, a part which distributes concepts into queues and, second, a part which runs several instances of the ordinary Close-by-One in parallel. Because of this reliance on CbO, we call our algorithm *Parallel Close-by-One* (*PCbO*). The following assertion shows correctness of PCbO:

Theorem 2 (Correctness of PCbO) *When invoked with* $\langle \emptyset^{\downarrow}, \emptyset^{\downarrow\uparrow}\rangle$, $y = 0$, *and* $l = 1$, *Algorithm* 2 *derives all formal concepts in* $\langle X, Y, I\rangle$, *each of them exactly once.*

Proof The correctness is a consequence of properties of derivations, see Lemmas 1 and 2. First, it is easy to observe that Algorithm 2 finishes after finitely many steps. Moreover, each concept that is derivable in less than L steps is processed in the first stage, each of them is processed exactly once. This follows from the fact that ParallelGenerateFrom simulates GenerateFrom. If a concept is derivable in $> L$ steps, it will be computed by one of the independent processors. Indeed, let (4) be the derivation of $\langle A, B\rangle$ where $k + 1 > L$. Then the $(L-1)$th element $\langle\langle A_{L-1}, B_{L-1}\rangle, y_{L-1}\rangle$ of the derivation (4) will be put in one of the queues, say $queue_r$, in the second stage of the algorithm because $\langle A_{L-1}, B_{L-1}\rangle$ is derivable in L steps. Therefore, $\langle A, B\rangle$ will be computed by the processor r. In addition to that, $\langle A, B\rangle$ will be computed exactly once on the account of Lemma 2. □

Remark 5 Let us comment on the role of P and L which influence Algorithm 2. Both the parameters have an impact on the distribution of computed formal concepts among the processors. Note that the practical range of the parameter P is somewhat limited by the hardware on which we run the algorithm (e.g., we are limited by hardware processors or network nodes). On the other hand, L can be set to any value ≥ 2. The performance of the algorithm in dependence of the value of L is experimentally evaluated in Section 4. According to our observations, if $L = 2$, most of the formal concepts are computed by one or two processors. With increasing L, formal concepts are distributed to processors more equally. On the other hand, large values of L tend to degenerate the parallel computation. For instance, if $L \geq |Y| + 1$ then all concepts will be computed in the first (sequential) stage because the depth of the call tree is at most $|Y| + 1$. From our experiments it seems that on average, a good trade-off value is already $L = 3$ provided that $|Y|$ is large. In such a case, almost all

formal concepts are computed in parallel and are distributed among the processors nearly optimally.

Example 3 We illustrate the influence of P and L on how Algorithm 2 computes the concepts. Consider a formal context $\langle X, X, \neq \rangle$ where $|X| = 5$. The corresponding $\mathcal{B}(X, X, \neq)$ is isomorphic to the Boolean algebra 2^5. Figure 3 contains results for four combinations of values of P and L. Each of the diagrams in Fig. 3 depicts the Hasse diagram of the concept lattice where nodes denoted by black circles correspond to concepts processed during the initial sequential stage. Nodes denoted by numbers are processed by independent processors of the corresponding numbers. In case of $P = 2$ and $L = 2$, only the topmost concept is processed in the first stage. During the second stage, three concepts are put in the queue of the first processor, the remaining two concepts are put in the queue of the second processor. The total number of concepts that are processed by the two processors are 21 and 10, respectively. If $P = 2$ and $L = 3$ (second diagram), the concepts are distributed among the processors more equally: 16 and 10. A similar situation applies for $P = 3$ where we have 18, 9, and 4 concepts processed by three processors in case of $L = 2$ and 11, 10, and 5 in case of $L = 3$, see last two diagrams.

Remark 6 The parallel computation of Algorithm 2 can be degenerate, meaning that in certain situations, only one of the P processors is computing all the remaining concepts while other processors are idle. Such a situation occurs iff the Lth level of the call tree contains at most one node $\langle C_i, y_i \rangle$. In particular, the situation occurs when $\mathcal{B}(X, Y, I)$ is isomorphic to an ordinal sum $\mathbf{L}_1 \oplus \mathbf{L}_2$ of a lattice $\mathbf{L}_1$ and an n-element chain $\mathbf{L}_2$ where n equals L (the recursion level), see Fig. 4. Such pathologic situations can be (partially) avoided by modifying the condition in line 1 of Algorithm 2 so that is checks whether at least a given number of queues are nonempty. More details on the utilization of processors can be found in Section 4.

Let us conclude this section with bibliographical remarks on existing approaches to parallel algorithms in FCA. For instance, [6] proposes a parallelization of Ganter's algorithm by decomposing the set of all concepts into non-overlapping subsets which are computed simultaneously. Another parallelization of Ganter's algorithm is presented in [2]. The basic idea in [2] is that the lexicographically ordered power

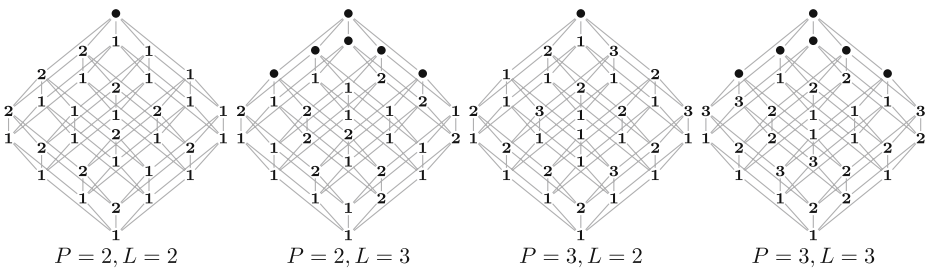

Fig. 3 Examples of parallelization for various values of P and L

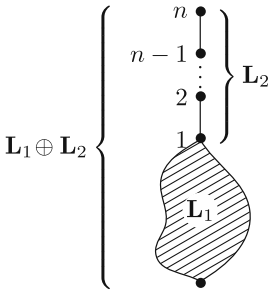

Fig. 4 Ordinal sum $\mathbf{L}_1 \oplus \mathbf{L}_2$ of a lattice $\mathbf{L}_1$ and an n-element chain $\mathbf{L}_2$

set 2^Y is split into p intervals of the same length (p indicates a number of processes). Then, each of the p intervals is executed by an independent process using a serial version of Ganter's algorithm. A different approach is shown, e.g., in [12] where the algorithm is based on dividing the input data into disjoint fragments which are then computed by independent processes. A detailed comparison of the algorithms in terms of their efficiency and scalability is beyond the scope of this paper and will be a subject of future investigation.

4 Efficiency and implementation issues

From the point of view of the worst-case complexity, PCbO is a polynomial time delay [11] algorithm with asymptotic complexity $O(|\mathcal{B}|\cdot|Y|^2\cdot|X|)$ because in the worst case, PCbO can degenerate into the sequential CbO [14, 15]. The actual performance compared to CbO is influenced by the number of processors P and their utilization. In case of optimal utilization of processors, PCbO can run P times faster than CbO, i.e. the reciprocal P^{-1} can be seen as a multiplicative constant of the running time of CbO. In practice, the multiplicative constant is greater than P^{-1} because (i) concepts are not distributed over the processors uniformly and (ii) the parallelization has certain overhead. In order to show how PCbO behaves on average data, we should provide theoretical and experimental average-case complexity analysis. The theoretical analysis seems to be an interesting and challenging problem which is

Table 1 Performance for selected datasets (real time, in seconds; time in parentheses represents total processor time used by all the processors together)

Dataset	Mushroom	Tic-tac-toe	Debian tags	Anon. web
Size	8,124 × 119	958 × 29	14,315 × 475	32,710 × 295
Density	19 %	34 %	< 1 %	1 %
PCbO ($P = 1$)	4.89	0.06	7.79	40.32
PCbO ($P = 2$)	2.78 (5.16)	0.04 (0.07)	5.52 (9.34)	22.16 (43.33)
PCbO ($P = 4$)	1.90 (5.39)	0.03 (0.07)	3.65 (10.88)	13.38 (47.81)
PCbO ($P = 8$)	1.18 (5.58)	0.02 (0.07)	2.51 (11.08)	8.09 (46.68)
Ganter's	834.40	2.15	1,720.82	10,039.73
Lindig's	5,271.98	14.53	2,639.67	13,422.64
Berry's	934.50	5.78	1,531.94	3,615.07

Table 2 Utilization of processors (number of concepts processed by particular processors)

CPU	#0	#1	#2	#3	#4	#5	#6
Mushroom ($P = 2$)	440	103,005	135,265				
Mushroom ($P = 4$)	440	78,825	89,174	24,180	46,091		
Mushroom ($P = 6$)	440	35,486	78,348	23,040	33,398	44,479	23,519
Tic-tac-toe ($P = 2$)	409	31,986	27,110				
Tic-tac-toe ($P = 4$)	409	16,518	13,832	15,468	13,278		
Tic-tac-toe ($P = 6$)	409	11,407	9,962	10,635	7,759	9,944	9,389

yet to be explored. In the sequel we present results of experiments with randomly generated and real data sets which may give hint how PCbO behaves for different values of P and L.

We first compare PCbO with other algorithms [16] for computing formal concepts. Namely, we compare it with Ganter's [7], Lindig's [17] and Berry's [4] algorithms (all implemented in ANSI C). The comparison is made using datasets from [1, 10] and a dataset generated from package descriptions in Debian GNU/Linux. The results, along with the information on sizes and densities (percentage of 1s) of used data sets, are depicted in Table 1. The first four rows contain running times of PCbO that has been run on 1 (sequential version), 2, 4, and 8 hardware processors. The measurements have been done on an otherwise idle 64-bit x86_64 hardware with 8 independent processors (2× Quad-Core Intel Xeon E5345, 2.33 GHz, 12 GB RAM). For $P > 1$, the table in Table 1 contains total processor time used to compute all formal concepts (the time written in parentheses). This time allows us to make a rough estimate of the overhead that is needed to manage multiple threads of computation: the overhead can be computed as the real processor time minus the total processor time divided by P. As it is expected, larger values of P lead to a larger overhead. The utilization of processors can be observed from the number of concepts that are processed by each processor. For instance, Table 2 shows the distribution of computed concepts among particular processors. The processor marked #0, is the initial sequential stage of the algorithm. It should be mentioned that the number of computed concepts by each processor is entirely given by parameters P, L, and by the context. This means, if one processor completes its computation, it cannot "help" other processors to process their load.

The next experiment focuses on the scalability of PCbO, i.e., the ability to decrease the running time using multiple processors. For this set of experiments we have used

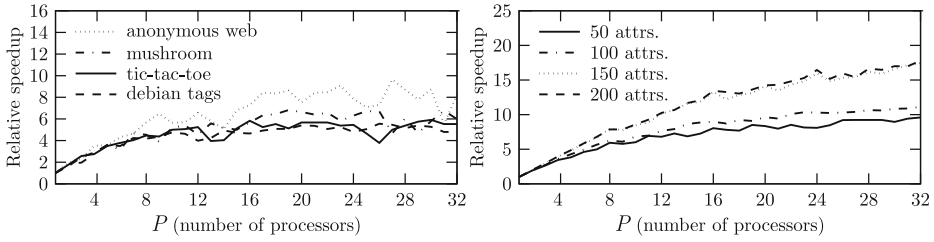

Fig. 5 Relative speedup in various data tables (on the *left*); relative speedup in contexts with various counts of attributes (on the *right*)

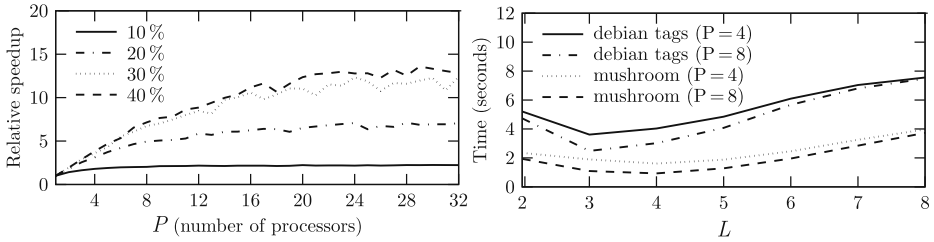

Fig. 6 Relative speedup dependent on density of 1's (on the *left*); running time dependent on the argument L (on the *right*)

computer equipped with eight core UltraSPARC T1 processor that is able to process up to 32 simultaneously running threads. Fig. 5 (left) contains results for selected datasets while Fig. 5 (right) contains results for randomly generated tables with 10,000 objects and 5 % density [16] of 1's. By a *relative speedup* which is shown on the y-axes in the graphs, we mean the theoretical speedup given by the number of hardware processors (e.g., if we have 4 processors, the execution can be 4 times faster). Therefore, the relative speedup is a ratio of running time using a single processor (the sequential algorithm) and running time using multiple processors. Note that the theoretical maximum of the speedup is equal to P but the real speedup is always smaller due to the overhead caused by managing of multiple threads (cf. also Table 1). The experiment in Fig. 6 (left) shows results of the impact of the data density. That is, we have generated data tables with various densities of 1's and observed the impact on the scalability. We have used data tables of size $5{,}000 \times 100$. Finally, Fig. 6 (right) illustrates the influence of parameter L on various data tables and amounts of processors. The experiments indicate that good choice is $L \in \{3, 4\}$, see Remark 5.

Let us note that the actual performance of an implementation of the algorithm depends on used data structures. We have used boolean vectors as basic data structures which turned out to be very efficient. The data structures and optimized algorithms for computing closures are further discussed in Outrata and Vychodil (submitted for publication).

5 Conclusions

We have introduced a parallel algorithm called PCbO for computing formal concepts in object-attribute data tables. The parallel algorithm results as a parallelization of CbO [14, 15] and is formalized by a recursive procedure which simulates the ordinary CbO up to a point where it forks into multiple processes and each process computes a disjoint set of formal concepts. The algorithm has minimal overhead because the concurrent processes computing disjoint sets of concepts are fully independent. This significantly improves efficiency of the algorithm. We have shown that the algorithm is scalable. With growing numbers of CPUs, the speedup of the computation given by increasing number of CPUs is near its theoretical limit. The implementation of the algorithm can be downloaded from

http://fcalgs.sourceforge.net/pcbo-amai.html.

The future research will focus on

- refinements of the algorithm including new approaches to reducing the number of concepts which are computed multiple times, some advances towards this direction can be found in Outrata and Vychodil (submitted for publication);
- comparison of various strategies for selecting queues and advanced conditions preventing degenerate computation, see Remark 6;
- performance comparison with other parallel algorithms, performance and scalability tests of various data structures for representing contexts, extents, and intents;
- specialized variants of the algorithm focused to solve particular problems related to FCA, e.g., factorization of binary matrices [3].

References

1. Asuncion, A., Newman, D: UCI Machine learning repository. School of Information and Computer Sciences, University of California, Irvine (2007)
2. Baklouti, F., Levy G.: A distributed version of the Ganter algorithm for general Galois Lattices. In: Belohlavek, R., Snasel, V. (eds.) Proc. CLA, pp. 207–221 (2005)
3. Belohlavek, R., Vychodil V.: Discovery of optimal factors in binary data via a novel method of matrix decomposition. J. Comput. Syst. Sci. **76**, 3–20 (2010)
4. Berry, A., Bordat, J.-P., Sigayret, A.: A local approach to concept generation. Ann. Math. Artif. Intell. **49**, 117–136 (2007)
5. Carpineto, C., Romano, G.: Concept Data Analysis. Theory and Applications. Wiley, New York (2004)
6. Fu, H., Mephu Nguifo, E.: A Parallel Algorithm to Generate Formal Concepts for Large Data. ICFCA, LNCS **2961**, 394–401 (2004)
7. Ganter, B.: Two basic algorithms in concept analysis. (Technical Report FB4-Preprint No. 831). TH Darmstadt (1984)
8. Ganter, B., Wille, R.: Formal Concept Analysis. Mathematical Foundations. Springer, Berlin (1999)
9. Grätzer G. et al.: General Lattice Theory, 2nd edn. Birkhäuser, Basel (2003)
10. Hettich, S., Bay, S.D.: The UCI KDD Archive University of California, Irvine, School of Information and Computer Sciences (1999)
11. Johnson, D.S., Yannakakis, M., Papadimitriou, C.H.: On generating all maximal independent sets. Inf. Process. Lett. **27**(3), 119–123 (1988)
12. Kengue J. F. D., Valtchev P., Djamégni C. T.: A parallel algorithm for lattice construction. ICFCA, LNCS **3403**, 249–264 (2005)
13. Kuznetsov, S.: Interpretation on graphs and complexity characteristics of a search for specific patterns. Autom. Doc. Math. Linguist. **24**(1), 37–45 (1989)
14. Kuznetsov, S.: A fast algorithm for computing all intersections of objects in a finite semi-lattice (Быстрый алгоритм построения всех пересечений обIектов из конечной полурешетки, in Russian). Automatic Documentation and Mathematical Linguistics, **27**(5), 11–21 (1993)
15. Kuznetsov, S.: Learning of simple conceptual graphs from positive and negative examples. PKDD, pp. 384–391 (1999)
16. Kuznetsov, S., Obiedkov, S.: Comparing performance of algorithms for generating concept lattices. J. Exp. Theor. Artif. Int. **14**, 189–216 (2002)
17. Lindig, C.: Fast concept analysis. Working with Conceptual Structures—Contributions to ICCS 2000, pp. 152–161. Shaker, Aachen (2000)
18. Norris, E.M.: An Algorithm for computing the maximal rectangles in a binary relation. Rev. Roum. Math. Pures. Appl. **23**(2), 243–250 (1978)
19. Wille, R.: Restructuring Lattice Theory: An Approach Based on Hierarchies of Concepts. Ordered Sets, pp. 445–470, Reidel, Dordrecht (1982)