Distributed Algorithm for Computing Formal Concepts Using Map-Reduce Framework*

Petr Krajca[1,2] and Vilem Vychodil[1,2]

[1] T. J. Watson School, State University of New York at Binghamton
[2] Dept. Computer Science, Palacky University, Olomouc
{petr.krajca, vychodil}@binghamton.edu

Abstract. Searching for interesting patterns in binary matrices plays an important role in data mining and, in particular, in formal concept analysis and related disciplines. Several algorithms for computing particular patterns represented by maximal rectangles in binary matrices were proposed but their major drawback is their computational complexity limiting their application on relatively small datasets. In this paper we introduce a scalable distributed algorithm for computing maximal rectangles that uses the map-reduce approach to data processing.

1 Introduction

We introduce a novel distributed algorithm for extracting rectangular patterns in binary object-attribute relational data. Our approach is unique among other approaches in that we employ the map-reduce framework which is traditionally used for searching and querying in large data collections. This paper contains a preliminary study and a proof of concept of how the map-reduce framework can be used for particular data-mining tasks.

In this paper, we focus on extracting rectangular patterns, so called formal concepts, in binary object-attribute relational data. The input data, we are interested in, takes form of a two-dimensional data table with rows corresponding to objects, columns corresponding to attributes (features), and table entries being crosses (or 1's) and blanks (or 0's) indicating presence/absence of attributes (features): a table has $\times$ on the intersection of row corresponding to object x and column corresponding to attribute y iff "object x has attributes y" ("feature y is present in object x"). Given a data table, we wish to find all maximal submatrices full of $\times$'s that are present in the table. These submatrices have a direct interpretation: they represent natural concepts hidden in the data which are the subjects of study of formal concept analysis [3,7] invented by Rudolf Wille [19]. Recently, it has been shown by Belohlavek and Vychodil [1] that maximal rectangles can be used to find optimal factorization of Boolean matrices. In fact, maximal rectangles correspond with optimal solutions to the discrete basis problem discussed by Miettinen et al. [11]. Finding maximal rectangles in data tables is therefore an important task.

* Supported by institutional support, research plan MSM 6198959214.

The algorithm we propose in this paper may help overcome problems with generating all formal concepts from large data sets. In general, the problem of listing all formal concepts is #P-complete [14]. Fortunately, if the input data is sparse, one can get sets of all formal concepts in a reasonable time. Still, listing all formal concepts can be time and space demanding and, therefore, there is a need to provide scalable distributed algorithms that may help distribute the burden over a large amount of low-cost computation nodes. In this paper we offer a possible solution using the map-reduce framework.

In the sequel we present a survey of notions from the formal concept analysis and principles of the map-reduce framework. Section 2 describes the algorithm. Furthermore, in Section 3, we describe the implementation and provide experimental evaluation of scalability of the proposed algorithm. The paper is concluded by related works and future research directions.

1.1 Formal Concept Analysis

In this section we recall basic notions of the formal concept analysis (FCA). More details can be found in monographs [7] and [3].

Formal concept analysis deals with binary data tables describing relationship between objects and attributes, respectively. The input for FCA is a data table with rows corresponding to objects, columns corresponding to attributes (or features), and table entries being ×'s and blanks, indicating whether an object given by row has or does not have an attribute given by column. An example of such a data table is depicted in Fig. 1 (left). A data table like that in Fig. 1 can be seen as a binary relation $I \subseteq X \times Y$ such that $\langle x, y \rangle \in I$ iff object x has attribute y. In FCA, I is usually called a *formal context* [7]. In this paper, we are going to use a set $X = \{0, 1, \ldots, m\}$ of objects and a set $Y = \{0, 1, \ldots, n\}$ of attributes, respectively. There is no danger of confusing objects with attributes because we do not mix elements from the sets X and Y in any way.

	0	1	2	3	4	5	6
0		×	×		×	×	×
1	×		×	×			×
2	×	×					
3	×		×	×		×	
4	×		×		×		×

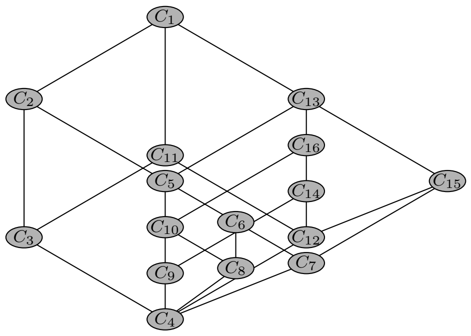

Fig. 1. Formal context (left) and the corresponding concept lattice (right)

Each formal context $I \subseteq X \times Y$ induces a couple of operators $^\uparrow$ and $^\downarrow$ defined, for each $A \subseteq X$ and $B \subseteq Y$, as follows:

$$A^{\uparrow} = \{y \in Y \mid \text{for each } x \in A\colon \langle x, y\rangle \in I\}, \tag{1}$$
$$B^{\downarrow} = \{x \in X \mid \text{for each } y \in B\colon \langle x, y\rangle \in I\}. \tag{2}$$

Operators $^{\uparrow}: 2^X \to 2^Y$ and $^{\downarrow}: 2^Y \to 2^X$ defined by (1) and (2) form so-called Galois connection [7]. By definition (1), $A^{\uparrow}$ is a set of all attributes shared by all objects from A and, by (2), $B^{\downarrow}$ is a set of all objects sharing all attributes from B.

A pair $\langle A, B\rangle$ where $A \subseteq X$, $B \subseteq Y$, $A^{\uparrow} = B$, and $B^{\downarrow} = A$, is called a *formal concept* (*in* $I \subseteq X \times Y$). Formal concepts can be seen as particular clusters hidden in the data. Namely, if $\langle A, B\rangle$ is a formal concept, A (called an *extent of* $\langle A, B\rangle$) is the set all objects sharing all attributes from B and, conversely, B (called an *intent of* $\langle A, B\rangle$) is the set of all attributes shared by all objects from A. Note that this approach to "concepts" as entities given by their extent and intent goes back to classical Port Royal logic. From the technical point of view, formal concepts are fixed points of the Galois connection $\langle^{\uparrow}, ^{\downarrow}\rangle$ induced by the formal context. Formal concepts in $I \subseteq X \times Y$ correspond to so-called maximal rectangles in I. In a more detail, any $\langle A, B\rangle \in 2^X \times 2^Y$ such that $A \times B \subseteq I$ shall be called a rectangle in I. Rectangle $\langle A, B\rangle$ in I is a maximal one if, for each rectangle $\langle A', B'\rangle$ in I such that $A \times B \subseteq A' \times B'$, we have $A = A'$ and $B = B'$. We have that $\langle A, B\rangle \in 2^X \times 2^Y$ is a maximal rectangle in I iff $A^{\uparrow} = B$ and $B^{\downarrow} = A$, i.e. maximal rectangles = formal concepts. Hence, maximal rectangles give us an alternative interpretation of formal concepts.

Let $\mathcal{B}(X, Y, I)$ denote the set of all formal concepts in $I \subseteq X \times Y$. The set $\mathcal{B}(X, Y, I)$ can be equipped with a partial order $\leq$ modeling the subconcept-superconcept hierarchy:

$$\langle A_1, B_1\rangle \leq \langle A_2, B_2\rangle \text{ iff } A_1 \subseteq A_2 \text{ (or, equivalently, iff } B_2 \subseteq B_1). \tag{3}$$

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 $\leq$ form a complete lattice whose structure is described by the Main Theorem of Formal Concept Analysis [7]. The above-described notions are illustrated in the following example.

Example 1. Consider a formal context $I \subseteq X \times Y$ corresponding to the incidence data table from Fig. 1 (left). The concept-forming operators induced by this context have exactly 15 fixpoints $C_1, \ldots, C_{16}$:

$$\begin{aligned}
C_1 &= \langle\{0,1,2,3,4\}, \{\}\rangle, & C_9 &= \langle\{4\}, \{0,2,4,6\}\rangle,\\
C_2 &= \langle\{1,2,3,4\}, \{0\}\rangle, & C_{10} &= \langle\{1,4\}, \{0,2,6\}\rangle,\\
C_3 &= \langle\{2\}, \{0,1\}\rangle, & C_{11} &= \langle\{0,2\}, \{1\}\rangle,\\
C_4 &= \langle\{\}, \{0,1,2,3,4,5,6\}\rangle, & C_{12} &= \langle\{0\}, \{1,2,4,5,6\}\rangle,\\
C_5 &= \langle\{1,3,4\}, \{0,2\}\rangle, & C_{13} &= \langle\{0,1,3,4\}, \{2\}\rangle,\\
C_6 &= \langle\{1,3\}, \{0,2,3\}\rangle, & C_{14} &= \langle\{0,4\}, \{2,4,6\}\rangle,\\
C_7 &= \langle\{3\}, \{0,2,3,5\}\rangle, & C_{15} &= \langle\{0,3\}, \{2,5\}\rangle,\\
C_8 &= \langle\{1\}, \{0,2,3,6\}\rangle, & C_{16} &= \langle\{0,1,4\}, \{2,6\}\rangle.
\end{aligned}$$

Hence, $\mathcal{B}(X, Y, I) = \{C_1, \ldots, C_{16}\}$. If we equip $\mathcal{B}(X, Y, I)$ with the partial order (3), we obtain a concept lattice shown in Fig. 1 (right).

The most common algorithms for computing formal concepts include Ganter's algorithm [6], Lindig's algorithm [18], and Berry's [2] algorithm. The algorithm we are going to introduce in Section 2 can be seen as a distributed version of the algorithm proposed in [12,13]. A survey and comparison of algorithms for FCA can be found in [17].

1.2 Processing Data Using Map-Reduce Approach

Distributed computing represents a common approach to processing large data but the complexity of distributed computing usually limits its application only on problems unsolvable in other ways. Common distributed algorithm implementations usually consist of modifications of more or less known algorithms that distribute particular parts of the data to be processed on other computers. This approach is quite comprehensible for programmers but brings several issues. Especially, issues connected with granularity of the task, reliability of the used hardware platform, etc.

A general framework for processing large data in distributed networks consisting of commodity hardware is proposed in [4]. In essence, this framework is based on two basic operations *map* and *reduce* that are applied on the data. These two operations have given the name of the framework—a *map-reduce* framework or (in short, an M/R framework). This approach to data processing has originally been developed by Google for their data centers but has shown to be very practical and later has been adopted by other software companies interested in storing and querying large amounts of data. In the rest of this section we provide a brief overview of the map-reduce framework.

Data in the M/R framework are generally represented in the form of key-value pairs $\langle key, value \rangle$. In the first step of the computation, the framework reads input data and optionally converts it into the desired key-value pairs.

In the second step—the *map phase*, a function f is applied on each pair $\langle k, v \rangle$ and returns a multiset of new key-value pairs, i.e.,

$$f(\langle k, v \rangle) = \{\langle k_1, v_1 \rangle, \ldots, \langle k_n, v_n \rangle\}$$

Notice that there is certain similarity with function `map` present in many programming languages (e.g., LISP, Python, and Scheme). In contrast with the usual `map`, function f may return arbitrary number of results and they are all collected during the *map phase*.

Subsequently, in the *reduce phase*, all pairs generated in the previous step are grouped by their keys and their values are aggregated (reduced) by a function g:

$$g(\{\langle k, v_1 \rangle, \langle k, v_2 \rangle, \ldots, \langle k, v_n \rangle\}) = \langle k, v \rangle.$$

The following example illustrate how the M/R framework can be used to perform a computation which might appear in information retrieval.

Example 2. Let us consider that we want to compute frequencies of letters in a text consisting of three words—*alice, barbara*, and *carol*. Now, consider function

f accepting a word and returning a multiset where each letter in the word is represented as a pair $\langle \mathit{letter}, 1\rangle$. The map phase will produce the following results:

$$\begin{aligned}
&f(\mathit{alice}) = \{\langle a,1\rangle, \langle l,1\rangle, \langle i,1\rangle, \langle c,1\rangle, \langle e,1\rangle\},\\
&f(\mathit{barbara}) = \{\langle b,1\rangle, \langle a,1\rangle, \langle r,1\rangle, \langle b,1\rangle, \langle a,1\rangle, \langle r,1\rangle, \langle a,1\rangle\},\\
&f(\mathit{carol}) = \{\langle c,1\rangle, \langle a,1\rangle, \langle r,1\rangle, \langle o,1\rangle, \langle l,1\rangle\}.
\end{aligned}$$

In the reduce phase, we group all pairs by the key and function g sums all 1's in the key-value pairs as follows:

$$\begin{aligned}
&g(\{\langle a,1\rangle, \langle a,1\rangle, \langle a,1\rangle, \langle a,1\rangle, \langle a,1\rangle\}) = \langle a,5\rangle,\\
&g(\{\langle b,1\rangle, \langle b,1\rangle\}) = \langle b,2\rangle,\\
&g(\{\langle c,1\rangle, \langle c,1\rangle\}) = \langle c,2\rangle,\\
&g(\{\langle e,1\rangle\}) = \langle e,1\rangle,\\
&g(\{\langle l,1\rangle, \langle l,1\rangle\}) = \langle l,2\rangle,\\
&g(\{\langle i,1\rangle\}) = \langle i,1\rangle,\\
&g(\{\langle o,1\rangle\}) = \langle o,1\rangle,\\
&g(\{\langle r,1\rangle, \langle r,1\rangle, \langle r,1\rangle\}) = \langle r,3\rangle.
\end{aligned}$$

One can see from the previous illustrative example and the sketch of the algorithm that since the functions f and g are applied only on particular pairs, it is possible to easily distribute the computation through several computers. For the sake of completeness, we should stress that functions f and g used to map and aggregate values must always be implemented by procedures (i.e., computer functions) that do not have any side effects. That means, the results of f and g depend only on given arguments, i.e., the procedures computing results of f and g behave as maps.

The actual implementation details of M/R processing are far more complex but their detailed description is out of the scope of this paper. The basic outline we have provided should be sufficient for understanding of our algorithm for computing formal concepts.

2 The Algorithm

In this section we present an overview of a parallel algorithm PCbO we have proposed in [12]. Then, we introduce a distributed variant of PCbO based on the M/R framework.

2.1 Overview

The distributed algorithm is an adaptation of Kuznetsov's Close-by-One (CbO, see [15,16]) and its parallel variant PCbO [12]. The CbO can be formalized by a recursive procedure GenerateFrom($\langle A, B\rangle, y$), 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) and then it checks its halting condition. 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 present in the intent B. For each $j \in Y$ having these properties, a new couple $\langle C, D\rangle \in 2^X \times 2^Y$ such that

$$\langle C, D\rangle = \langle A \cap \{j\}^{\downarrow}, (A \cap \{j\}^{\downarrow})^{\uparrow}\rangle \tag{4}$$

is computed. The pair $\langle C, D\rangle$ is always a formal concept [12] such that $B \subset D$. 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 (so-called canonicity test) is based on comparing $B \cap Y_j = D \cap Y_j$ where $Y_j \subseteq Y$ is defined as follows:

$$Y_j = \{y \in Y \mid y < j\}. \tag{5}$$

The role of the canonicity test is to prevent computing the same formal concept multiple times. GenerateFrom computes formal concepts in a unique order which ensures that each formal concept is processed exactly once. The proof is elaborated in [13].

Remark 1. Recursive invocations of GenerateFrom form a tree. The tree corresponding to data from Fig. 1 is depicted in Fig. 2. The root corresponds to the first invocation GenerateFrom($\langle\emptyset^{\downarrow}, \emptyset^{\downarrow\uparrow}\rangle, 0$). Each node labeled $\langle C_i, k\rangle$ corresponds to an invocation of GenerateFrom, where C_i is a formal concept, see Example 1. The nodes denoted by black squares represent concepts which are

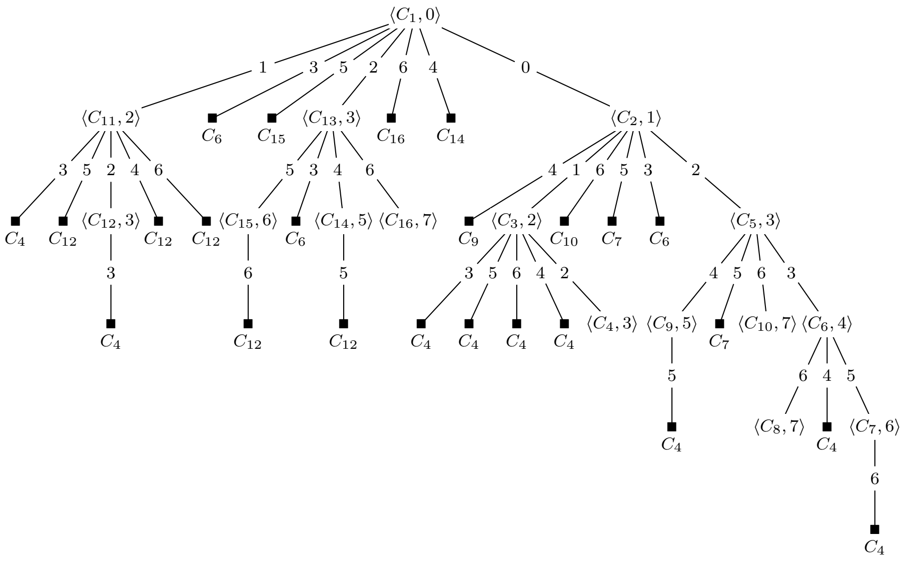

Fig. 2. Example of a call tree for GenerateFrom($\langle\emptyset^{\downarrow}, \emptyset^{\downarrow\uparrow}\rangle, 0$) with data from Fig. 1

computed but not processed because the canonicity test fails. The edges in the tree are labeled by the number of attributes that are used to compute new concepts, cf. (4). More details can be found in [13].

2.2 Adaptation for M/R Framework

The procedure GenerateFrom outlined in Section 2.1 uses a depth-first search strategy to generate new formal concepts. Since GenerateFrom depends only on its arguments, the strategy in which formal concepts are generated does not play any significant role and can be replaced by another strategy. In our distributed algorithm, we are going to use the breadth-first search strategy. Conversion of the algorithm from the depth-first to the breadth-first search is necessary for the adaptation of CbO algorithm to the M/R framework. Moreover, the original GenerateFrom will be split into two functions: a map function called MapConcepts and a reduction function ReduceConcepts. The map function will take care of generating new formal concepts and the reduce function will take care of performing the canonicity tests. The breadth-first search strategy is beneficial since it allows us to compute formal concepts by *levels*, where each level is computed from the previous one by consecutive applications of MapConcepts and ReduceConcepts on formal concepts that were computed in the previous level.

Remark 2. The adaptation for the map-reduce framework has several aspects. First, instead of using a single recursive procedure GenerateFrom, we employ two functions which are not recursive (i.e., they do not invoke themselves) but serve as a mapping and a reduction functions used by the framework. Second, the arguments to MapConcepts and ReduceConcepts, which somehow encode the arguments of the original GenerateFrom, must be presented as key/value pairs to ensure compatibility with the map-reduce framework. These issues will be addressed in the sequel.

Recall that the M/R framework assumes that all values processed by MapConcepts and ReduceConcepts are in the form of $\langle key, value \rangle$. In order to ensure the input/output compatibility with the M/R framework, we are going to encode the arguments for MapConcepts and ReduceConcepts as follows: We consider pairs $\langle key, value \rangle$ such that

– *key* is a tuple $\langle B, y \rangle$ where B is an intent of a concept $\langle A, B \rangle \in \mathcal{B}(X, Y, I)$ and $y \in Y$ is an attribute;
– *value* is a new concept $\langle C, D \rangle \in \mathcal{B}(X, Y, I)$.

The exact meaning of the key/value pair during the computation will become apparent later. The way in which MapConcepts and ReduceConcepts compute all formal concepts can be summarized by the following steps:

(1) Initially, the first formal concept $\langle \emptyset^{\downarrow}, \emptyset^{\downarrow\uparrow} \rangle$ is computed and MapConcepts is called with the initial key/value pair $\langle \langle \emptyset^{\downarrow\uparrow}, 0 \rangle, \langle \emptyset^{\downarrow}, \emptyset^{\downarrow\uparrow} \rangle \rangle$ which produces

a multiset of new key/value pairs representing new concepts. The multiset is further reduced by ReduceConcepts by removing key/value pairs with concepts that do not pass the canonicity test. These two steps represents the first iteration.

(2) The MapConcepts function is applied on each key/value pair from the previous nth iteration and the result is reduced by ReduceConcepts; the returned key/value pairs containing formal concepts are stored as result of the $(n+1)$th iteration.

(3) If the $(n+1)$th iteration produces any new concepts, the computation continues with the step (2) for the next iteration; otherwise the computation is stopped.

In the sequel, we provide a detailed description of MapConcepts and ReduceConcepts.

2.3 Details on MapConcepts and ReduceConcepts

The mapping function is described in Algorithm 1. It accepts an encoded formal concept $\langle A, B\rangle$ and iterates over all attributes that are equal or greater than y (lines 2–7). If the attribute is not present in the intent B (line 3), it computes a new formal concept $\langle C, D\rangle$ by extending intent B with an attribute j (lines 4 and 5). This corresponds to getting a formal concept of the form (4).

Algorithm 1: MapConcepts

```
Input: Pair $\langle key, value\rangle$ where $key$ is $\langle B_0, y\rangle$ and $value$ is $\langle A, B\rangle$.
1 set $result$ to $\emptyset$
2 for $j = y$ to $|Y|$ do
3     if $j \in B$ then continue;
4     set $C$ to $A \cap \{j\}^{\downarrow}$
5     set $D$ to $C^{\uparrow}$
6     set $result$ to $result \cup \{\langle\langle B, j\rangle, \langle C, D\rangle\rangle\}$
7 end
8 return $result$
```

Algorithm 1 computes new formal concepts that are derived from $\langle A, B\rangle$. Notice that some of the new concepts obtained this way may be the same. In general, a single concept may result by computing (4) multiple times during the entire computation. To identify redundantly computed formal concepts and to remove them, we use the same canonicity test as the ordinary CbO and its parallel variant PCbO. In our case, the canonicity test will appear in the reduction function. Also note that the value of B_0 is not used by MapConcepts.

The ReduceConcepts function accepts an encoded tuple $\langle\langle B, j\rangle, \langle C, D\rangle\rangle$ and returns a value as follows:

$$\textsc{ReduceConcepts}(\langle\langle B, j\rangle, \langle C, D\rangle\rangle) = \begin{cases} \langle\langle B, j+1\rangle, \langle C, D\rangle\rangle, & \text{if } B \cap Y_j = D \cap Y_j, \\ \textit{void-value}, & \text{otherwise.} \end{cases}$$

Therefore, if the canonicity test is satisfied then the input pair $\langle\langle B, j\rangle, \langle C, D\rangle\rangle$ is reduced to $\langle\langle B, j+1\rangle, \langle C, D\rangle\rangle$ which will be used in the next iteration.

Remark 3. (a) During the computation, each value $\langle B, j\rangle$ of key appears at most once. Thus, we can assume that ReduceConcepts accepts only one argument instead of a set of arguments with the same key.

(b) Practical implementations of M/R frameworks have support for dealing with *void-values.* In practice, the *void-values* are not included into the results.

(c) The process of computing formal concepts may be seen as building a tree (e.g., as depicted in Fig. 2) by its levels as it is shown in Fig. 3.

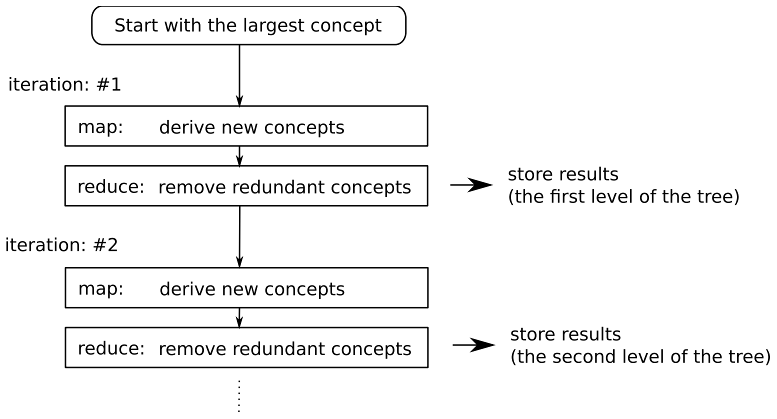
Start with the largest concept
iteration: #1
map: derive new concepts
reduce: remove redundant concepts
store results
(the first level of the tree)
iteration: #2
map: derive new concepts
reduce: remove redundant concepts
store results
(the second level of the tree)

Fig. 3. Particular iterations of the computation

Example 3. This example shows how the algorithm processes input data. As the input dataset we use formal context described in the Fig. 1. We start with the first iteration and create the initial value $\langle\langle\emptyset, 0\rangle, C_1\rangle$ and continue with the first iteration. In the first iteration, map function is applied only on the initial value and generates tuples: $\langle\langle\emptyset, 0\rangle, C_2\rangle$, $\langle\langle\emptyset, 1\rangle, C_{11}\rangle$, $\langle\langle\emptyset, 2\rangle, C_{13}\rangle$, $\langle\langle\emptyset, 3\rangle, C_6\rangle$, $\langle\langle\emptyset, 4\rangle, C_{14}\rangle$, $\langle\langle\emptyset, 5\rangle, C_{15}\rangle$, $\langle\langle\emptyset, 6\rangle, C_{16}\rangle$. In the sequel, i.e., the reduction phase, only tuples passing the canonicity test are retained. That is, tuples $\langle\langle\emptyset, 1\rangle, C_2\rangle$, $\langle\langle\emptyset, 2\rangle, C_{11}\rangle$, $\langle\langle\emptyset, 3\rangle, C_{13}\rangle$ are stored as results of the first iteration and used as input values for the second iteration. The second iteration takes values from the first one and applies map function on them. The map function applied on $\langle\langle\emptyset, 1\rangle, C_2\rangle$ generates tuples: $\langle\langle\{0\}, 1\rangle, C_3\rangle$, $\langle\langle\{0\}, 2\rangle, C_5\rangle$, $\langle\langle\{0\}, 3\rangle, C_6\rangle$, $\langle\langle\{0\}, 4\rangle, C_9\rangle$, $\langle\langle\{0\}, 5\rangle, C_7\rangle$, $\langle\langle\{0\}, 6\rangle, C_{10}\rangle$. Similarly, the map function applied on $\langle\langle\emptyset, 2\rangle, C_{11}\rangle$ returns: $\langle\langle\{1\}, 2\rangle, C_{12}\rangle$, $\langle\langle\{1\}, 3\rangle, C_4\rangle$, $\langle\langle\{1\}, 4\rangle, C_{12}\rangle$, $\langle\langle\{1\}, 5\rangle, C_{12}\rangle$, $\langle\langle\{1\}, 6\rangle, C_{12}\rangle$, and application of map function on $\langle\langle\emptyset, 3\rangle, C_{13}\rangle$ produces: $\langle\langle\{2\}, 3\rangle, C_6\rangle$, $\langle\langle\{2\}, 4\rangle, C_{14}\rangle$, $\langle\langle\{2\}, 5\rangle, C_{15}\rangle$, $\langle\langle\{2\}, 6\rangle, C_{16}\rangle$. From these values only $\langle\langle\{0\}, 2\rangle, C_3\rangle$, $\langle\langle\{0\}, 3\rangle, C_5\rangle$, $\langle\langle\{1\}, 3\rangle, C_{12}\rangle$, $\langle\langle\{2\}, 5\rangle, C_{14}\rangle$, $\langle\langle\{2\}, 6\rangle, C_{15}\rangle$, $\langle\langle\{2\}, 7\rangle, C_{16}\rangle$ pass the canonicity test and thus are stored as results of the second iteration and used in the following iteration. The computation continues in much the same way with the next iteration and stops if the reduction phase does not return any value. One can see that the computation directly corresponds to the tree depicted in Fig. 2. Each iteration represents one level of the tree whereas map function computes descendant nodes and the reduce function determines which nodes should be used in further computation.

3 Implementation and Experiments

We have implemented our algorithm as a Java application using Hadoop Core framework [8] providing infrastructure for map-reduce computations along with distributed file system for storing input data and results. For our experiments, we have used cluster consisting of 15 idle desktop computers equipped with Intel Core 2 Duo (3.0 GHz) processors, 2 GB RAM, 36 GB of disk space, and GNU/Linux.

We present here two sets of preliminary experiments. The first one focuses on the total time needed to compute all formal concepts present in real-world datasets. We selected three datasets from [9] and our repository with various properties and measured time it took to compute all formal using cluster consisting of one, five, and ten nodes. The results are depicted in the Fig. 4. For the comparison, we have also included the running time it takes to compute all formal concepts by the usual Ganter's NextClosure algorithm [7].

dataset	mushroom	debian tags	anon. web
size	8124×119	14315×475	32710×295
density	19 %	< 1 %	1 %
our (1 node)	1259	1379	2935
our (5 nodes)	436	426	830
our (10 nodes)	397	366	577
NextClosure	743	1793	10115

Fig. 4. Time needed to compute all formal concepts for various datasets (in seconds)

While evaluating the distributed algorithm, besides the overall performance, we have to take into account an important feature called scalability. In other words, the ability to decrease the time of the computation by utilizing more computers. In the second set of experiments we focus on this feature. To represent scalability we are using *relative speedup*, a ratio $S = \frac{T_1}{T_n}$, where T_1 is the time of the computation running only one computer and T_n is the time of the computation running on n computers. The theoretical maximal speedup is equal to the number of computers. The real speedup is always smaller than the theoretical one due to many factors including especially communication overhead between particular nodes, network throughput, etc.

Fig. 5 (left) shows scalability of our algorithm for selected datasets. One can see that for small numbers of nodes the speedup is almost linear but with the increasing number the speedup is not so significant. In case of the *mushrooms* dataset, there is even a decline of the speedup, i.e., with increasing number

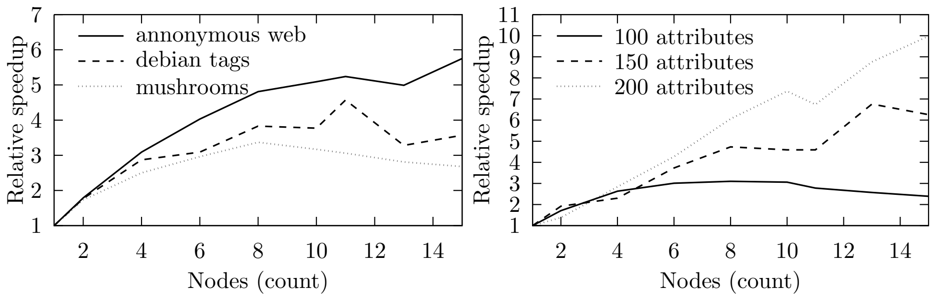

Fig. 5. Relative speedup for real world datasets (left) and for contexts with various counts of attributes (right)

of nodes, the speedup is no longer increasing. This is caused by the overhead of distributed computation that cannot be counterweighted by utilizing more computers. This means, the size of the cluster has to be adequate to the size of the input data. This fact also supports Fig. 5 (right), depicting scalability of the algorithm on randomly generated contexts consisting of 10000 objects, 100, 150, and 200 attributes, where density of 1's is 10%. From the figure Fig. 5 (right) follows that with the increasing size of data the scalability grows and for the data table of size 10000×200 the computation on cluster consisting of 15 may be done even $10\times$ faster than on a single computer.

3.1 Related Works, Conclusions, and Future Research

Several parallel algorithms for computing formal concepts have been proposed. For instance, [5], [10], or [12]. In general, parallel algorithms have a disadvantage of requiring a hardware equipped with several processors or processor cores. Despite the shift in hardware development toward to multicore microprocessors, hardware configuration with large amounts (more than ten) of processor cores are still relatively expensive and rare. Contrary to that, distributed algorithms may run on a coupled commodity hardware. Typically, parallel programs have a smaller overhead of the computation management than the distributed ones but the distributed algorithms are more cost-effective (they can be run on ordinary personal computers connected by a network). Although all mentioned algorithms may be modified to their *ad hoc* distributed versions, as far as we know, there are no distributed implementations of these algorithms. The approach introduced in this paper should be seen as a proof of concept of computing formal concepts by isolated nodes. We have shown that the algorithm is scalable. Therefore, there is a potential to apply techniques of formal concept analysis for much larger data sets than previously indicated. Our future research will focus on improving the algorithm by employing more efficient canonicity tests and providing a more efficient implementation. Furthermore, we intend to test the behavior of the algorithm on larger data sets and with larger amount of nodes.

References

1. Belohlavek, R., Vychodil, V.: Discovery of optimal factors in binary data via a novel method of matrix decomposition. Journal of Computer and System Sciences (to appear)
2. Berry, A., Bordat, J.-P., Sigayret, A.: A local approach to concept generation. Annals of Mathematics and Artificial Intelligence 49, 117–136 (2007)
3. Carpineto, C., Romano, G.: Concept data analysis. Theory and applications. J. Wiley, Chichester (2004)
4. Dean, J., Ghemawat, S.: MapReduce: simplified data processing on large clusters. Commun. ACM 51(1), 107–113 (2008)
5. Fu, H., Nguifo, E.M.: A parallel algorithm to generate formal concepts for large data. In: Eklund, P. (ed.) ICFCA 2004. LNCS (LNAI), vol. 2961, pp. 394–401. Springer, Heidelberg (2004)
6. Ganter, B.: Two basic algorithms in concept analysis (Technical Report FB4-Preprint No. 831). TH Darmstadt (1984)
7. Ganter, B., Wille, R.: Formal concept analysis. Mathematical foundations. Springer, Berlin (1999)
8. Hadoop Core Framework, `http://hadoop.apache.org/`
9. Hettich, S., Bay, S.D.: The UCI KDD Archive University of California, Irvine, School of Information and Computer Sciences (1999)
10. Kengue, J.F.D., Valtchev, P., Djamegni, C.T.: A parallel algorithm for lattice construction. In: Ganter, B., Godin, R. (eds.) ICFCA 2005. LNCS (LNAI), vol. 3403, pp. 249–264. Springer, Heidelberg (2005)
11. Miettinen, P., Mielikäinen, T., Gionis, A., Das, G., Mannila, H.: The discrete basis problem. In: Fürnkranz, J., Scheffer, T., Spiliopoulou, M. (eds.) PKDD 2006. LNCS (LNAI), vol. 4213, pp. 335–346. Springer, Heidelberg (2006)
12. Krajca, P., Outrata, J., Vychodil, V.: Parallel Recursive Algorithm for FCA. In: Belohlavek, R., Kuznetsov, S.O. (eds.) Proc. CLA 2008, vol. 433, pp. 71–82. CEUR WS (2008) ISBN 978–80–244–2111–7
13. Krajca, P., Outrata, J., Vychodil, V.: Parallel Algorithm for Computing Fixpoints of Galois Connections. Annals of Mathematics and Artificial Intelligence (submitted)
14. Kuznetsov, S.: Interpretation on graphs and complexity characteristics of a search for specific patterns. Automatic Documentation and Mathematical Linguistics 24(1), 37–45 (1989)
15. Kuznetsov, S.: A fast algorithm for computing all intersections of objects in a finite semi-lattice (Быстрый алгоритм построения всех пересечений объектов из конечной полурешетки, in Russian). Automatic Documentation and Mathematical Linguistics 27(5), 11–21 (1993)
16. Kuznetsov, S.O.: Learning of simple conceptual graphs from positive and negative examples. In: Żytkow, J.M., Rauch, J. (eds.) PKDD 1999. LNCS (LNAI), vol. 1704, pp. 384–391. Springer, Heidelberg (1999)
17. Kuznetsov, S., Obiedkov, S.: Comparing performance of algorithms for generating concept lattices. J. Exp. Theor. Artif. Int. 14, 189–216 (2002)
18. Lindig, C.: Fast concept analysis. In: Working with Conceptual Structures —-Contributions to ICCS 2000, pp. 152–161. Shaker Verlag, Aachen (2000)
19. Wille, R.: Restructuring lattice theory: an approach based on hierarchies of concepts. In: Ordered Sets, Dordrecht, Boston, pp. 445–470 (1982)