Parallel Recursive Algorithm for FCA*

Petr Krajca, Jan Outrata and Vilem Vychodil

Data Analysis and Modelling Laboratory, SUNY Binghamton
Vestal Parkway E, Binghamton, NY 13902–6000, USA
petr.krajca@binghamton.edu, vychodil@binghamton.edu

Department of Computer Science, Palacky University, Olomouc
Tomkova 40, CZ-779 00 Olomouc, Czech Republic
jan.outrata@upol.cz

Abstract. This paper presents a parallel algorithm for computing formal concepts. Presented is a sequential version upon which we build the parallel one. We describe the algorithm, its implementation, scalability, and provide an initial experimental evaluation of its efficiency. The algorithm is fast, memory efficient, and can be optimized so that all critical operations are reduced to low-level bit-array operations. One of the key features of the algorithm is that it avoids synchronization which has positive impacts on its speed and implementation.

1 Introduction

In this paper, we focus on extracting formal concepts, i.e. particular rectangular patterns, 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 1's and 0's indicating presence/absence of attributes. Tables like these represent a fundamental form of incidence data. Given a data table, we wish to find all formal concepts [9, 18] present in the table.

There are several algorithms for computing formal concepts, see [13] for an overview and comparison. Among the best known algorithms are Ganter's algorithm [8] and Lindig's algorithm [14] and their variants. Almost all algorithms proposed to date are sequential ones. Since parallel computing is recently gaining interests as hardware manufactures are shifting their focus from improving computing power by increasing clock frequencies to developing processors with multiple cores, there is a need to have scalable parallel algorithms for formal concept analysis (FCA) which can fully utilize the power of such milticore systems and deliver results faster than sequential algorithms. In this paper, we propose a parallel version of an algorithm presented in [16, 17] which is closely related to algorithm Close-by-One [12]. Our algorithm is light weight, fast, memory efficient, and can be implemented so that it uses just static linear data structures utilizing only low-level operations present in arithmetic logic units of contemporary

* Supported by grant No. 1ET101370417 of GA AV ČR and by institutional support, research plan MSM 6198959214.

microchips which significantly improves the performance of its implementations. We describe the algorithm and compare its performance with the other algorithms. We also focus on scalability, i.e. the growth of algorithm's performance with respect to the growing number of processors.

Let us note that computing all formal concepts is interesting not only for FCA itself but has a wide range of applications. For instance, it has been shown in [3] that formal concepts can be used to find optimal factorization of Boolean matrices. In fact, formal concepts correspond with optimal solutions to the discrete basis problem discussed by Miettinen et al. [15]. Finding formal concepts in data tables is therefore an important task.

2 Preliminaries from FCA

In this section we recall basic notions of the formal concept analysis. More details can be found in monographs [9] and [5].

Let $X = \{0, 1, \ldots, m\}$ and $Y = \{0, 1, \ldots, n\}$ be our 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, $\langle x, y\rangle \in I$ meaning that object x has attribute y. As usual, we consider a couple of concept-forming operators [9] ${}^{\uparrow}\colon 2^X \to 2^Y$ and ${}^{\downarrow}\colon 2^Y \to 2^X$ defined, for each $A \subseteq X$ and $B \subseteq Y$, by

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

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

By definition (1), $A^{\uparrow}$ is the set of all attributes shared by all objects from A and, by (2), $B^{\downarrow}$ is the set of all objects sharing all attributes from B. Operators ${}^{\uparrow}\colon 2^X \to 2^Y$ and ${}^{\downarrow}\colon 2^Y \to 2^X$ defined by (1) and (2) form the so-called Galois connection [9]. A formal concept (in $\langle X, Y, I\rangle$) is any couple $\langle A, B\rangle \in 2^X \times 2^Y$ such that $A^{\uparrow} = B$ and $B^{\downarrow} = A$. If $\langle A, B\rangle$ is a formal concept then A and B will be called the extent and intent of that concept, respectively. The subconcept-superconcept hierarchy $\leq$ is defined as $\langle A_1, B_1\rangle \leq \langle A_2, B_2\rangle$ iff $A_1 \subseteq A_2$ (or, iff $B_2 \subseteq B_1$, both the ways are equivalent), see [5, 9] for details.

Remark 1. There is a useful view of formal concepts which is often neglected in literature. Namely, formal concepts in $\langle X, Y, I\rangle$ correspond to maximal rectangles in $\langle X, Y, I\rangle$. 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 $B = B'$. Now, it is easily seen 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.

3 Computing Closures

Here we describe a procedure common to both the sequential and parallel versions of our algorithm. It generates a new concept from an existing one by enlarging its intent and shrinking its extent (at the same time).

Procedure ComputeClosure($\langle A, B\rangle, y$)

```
for $i$ from 0 upto $m$ do
    set $C[i]$ to 0;
end
for $j$ from 0 upto $n$ do
    set $D[j]$ to 1;
end
foreach $i$ in $A \cap \text{rows}[y]$ do
    set $C[i]$ to 1;
    for $j$ from 0 upto $n$ do
        if $\mathit{table}[i, j] = 0$ then
            set $D[j]$ to 0;
        end
    end
end
return $\langle C, D\rangle$
```

Representation of the Input Data For the sake of efficiency, we represent each $\langle X, Y, I\rangle$ two ways. First, by a two-dimensional array, denoted *table*, which corresponds with I in the usual sense. That is, the array *table* is filled with 1s and 0s so that $\mathit{table}[i, j] = 1$ iff $\langle i, j\rangle \in I$ and $\mathit{table}[i, j] = 0$ iff $\langle i, j\rangle \notin I$.

The second representation of the data is an array of ordered lists of objects. For each attribute $y \in Y$, we let $\mathit{rows}[y]$ be a list of all objects having the attribute y. Thus, $\mathit{rows}[y]$ contains $x \in X$ iff $\langle x, y\rangle \in I$. In addition to that, the numbers of rows contained in $\mathit{rows}[y]$ will be ordered in the ascending order (this is for the sake of efficiency). For instance, $\mathit{rows}[y] = (2, 4, 7)$ means that the only objects from X having y in I are the objects 2, 4, and 7. The two-dimensional array *table* and the array of lists *rows* will be used by the subsequent algorithms.

All the algorithms we are going to describe will use sets of objects and attributes represented by their characteristic arrays. That is, in case of attributes, a subset $B \subseteq Y = \{0, 1, \ldots, n\}$ will be represented by an $(n + 1)$-element linear array b of 1s and 0s such $b[k] = 1$ iff $k \in B$ (and $b[k] = 0$ iff $k \notin B$). By a slight abuse of notation, we will identify B with b and write $B[k] = 1$ to denote $k \in B$.

Description of the Algorithm If $\langle A, B\rangle$ is a formal concept then due to the monotony of ${}^{\downarrow\uparrow}$, all the formal concepts whose intents are strictly greater than B can be written as $\big\langle (B \cup C)^{\downarrow}, (B \cup C)^{\downarrow\uparrow}\big\rangle$, where $C \subseteq Y$ is a set of attributes such that there is at least one attribute $y \in Y$ such that $y \in C$ and $y \notin B$. In particular, if we consider $C = \{y\} \subseteq Y$ such that $y \notin B$, then

$$\big\langle (B \cup \{y\})^{\downarrow}, (B \cup \{y\})^{\downarrow\uparrow}\big\rangle \tag{3}$$

is a formal concept such that $(B \cup \{y\})^{\downarrow} \subset A$ and $B \subset (B \cup \{y\})^{\downarrow\uparrow}$. This is important from the computational point of view because if we want to compute

$(B \cup \{y\})^{\downarrow}$, it suffices to go exactly through all objects in A having attribute y:

$$(B \cup \{y\})^{\downarrow} = \{x \in A \,|\, \langle x, y \rangle \in I\} = A \cap \{y\}^{\downarrow}. \tag{4}$$

The common attributes of objects from (4) form the intent of (3). We have just outlined the idea behind our algorithm which generates formal concept (3) given formal concept $\langle A, B \rangle$ and attribute $y \in Y$ which does not belong to B. The corresponding procedure will be called ComputeClosure. It accepts a formal concept $\langle A, B \rangle$ and an attribute $y \notin B$ and produces a new formal concept $\langle C, D \rangle$ which equals to (3). We can show that the algorithm is sound, see [16].

Remark 2. We have used two representations of the input data to establish desired efficiency of computing new formal concepts, i.e. the redundancy in representation is a trade-off for efficiency. The two-dimensional array representation is used to determine which attributes are not present in the intent of the newly computed formal concept (see lines 7–14 of ComputeClosure). The second representation is used to skip rows in which y does not appear. Such rows do not contribute to the closure $(B \cup \{y\})^{\downarrow\uparrow}$, i.e. they can be disregarded. Our representation is most efficient for mid-size data sets (hundreds of attributes + thousands of objects) stored in RAM.

4 Sequential Algorithm

The previous section described how we can efficiently compute a new formal concept (3) given an initial formal concept $\langle A, B \rangle$. In this section we present a simplified version of our sequential algorithm for computing formal concepts [16, 17] which is suitable for parallelization. The main idea behind this algorithm is the same as in case of the algorithm Close-by-One proposed by Kuznetsov in [12].

Listing Formal Concepts in a Unique Order The core of our algorithm is a recursive procedure GenerateFrom which lists all formal concepts using a depth-first search through the space of all formal concepts. The procedure starts with an initial formal concept $\langle \emptyset^{\downarrow}, \emptyset^{\downarrow\uparrow} \rangle$. During the search, the procedure first generates a new formal concept R by adding attributes to the intent of the current formal concept, i.e. it applies the procedure described in ComputeClosure. Then, it is checked whether R has already been found. If not, it processes R (e.g., prints it on the screen), and proceeds with generating further formal concepts resulting from R by adding attributes to its intent, i.e. here GenerateFrom recursively calls itself with R being the current formal concept.

The key issue here is to have a quick procedure testing whether a newly generated formal concept has been generated before. We generate the formal concepts in a unique order which ensures that each formal concept is processed exactly once. The principle is the following. Let $\langle A, B \rangle$ be a formal concept, $y \in Y$ such that $y \notin B$. Put $D = (B \cup \{y\})^{\downarrow\uparrow}$, i.e. the new formal concept is $\langle (B \cup \{y\})^{\downarrow}, D \rangle$, see (3). Once D is computed using ComputeClosure, we check whether

$$D \cap \{0, 1, \ldots, y-1\} = B \cap \{0, 1, \ldots, y-1\} \tag{5}$$

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

```
1  process $B$ (e.g., print $B$ on screen);
2  if $B = Y$ or $y > n$ then
3      return
4  end
5  for $j$ from $y$ upto $n$ do
6      if $B[j] = 0$ then
7          set $\langle C, D\rangle$ to ComputeClosure($\langle A, B\rangle, j$);
8          set $skip$ to false;
9          for $k$ from $0$ upto $j - 1$ do
10             if $D[k] \neq B[k]$ then
11                 set $skip$ to true;
12                 break for loop;
13             end
14         end
15         if $skip$ = false then
16             GenerateFrom($\langle C, D\rangle, j + 1$);
17         end
18     end
19 end
20 return
```

is true. Note that the "$\supseteq$"-part of (5) is trivial. Moreover, (5) is true iff D agrees with B on the attributes $0, 1, \ldots, y - 1$. In other words, (5) is true iff, for each $i \in \{0, 1, \ldots, y - 1\}$: $i \in D$ iff $i \in B$. Thus, condition (5) expresses the fact that the closure D of $B \cup \{y\}$ does not contain any new attributes which are "before y". Condition (5) will be used to check whether we should process D. If (5) will be false, we will not process D because due to the depth-first search method, D has already been processed.

Description of the Algorithm The algorithm is represented by a procedure GenerateFrom that accepts two arguments. First, a formal concept $\langle A, B\rangle$ represented by characteristic vectors of objects A and attributes B covered by the concept. Second, an attribute y which is the first attribute to be added to B. $\langle A, B\rangle$ serves as an initial concept from which we start generating other formal concepts. After its invocation, GenerateFrom proceeds as follows:

– It processes the formal concept $\langle A, B\rangle$ (e.g., it prints A and B on screen).
– Then, the procedure checks whether B contains all the attributes from Y, i.e. whether B represents the greatest intent, in which case we exit current branch of recursion (lines 2–4).
– The main loop (lines 5–20) iterates over all remaining attributes, starting with the attribute y. In the body of the main loop (lines 6–18), j denotes the current attribute which we are about to add to B. The if-condition at line 6 checks whether j is already present in B. If so, we proceed with another attribute. If j is not present in B, we try to generate new intent from $B \cup \{j\}$ (lines 7–17).

– At line 7, we compute a new formal concept denoted $\langle C, D \rangle$. The loop between lines 9–14 checks whether B and D satisfy condition (5) for y being j. A flag *skip* is initially set to false (line 8). The flag is reset to true iff there is $k < j$ such that B and D disagree on k.
– If *skip* is false, i.e. if D and B agree on all attributes up to $j - 1$, we make a recursive call of the procedure GenerateFrom to compute descendant intents of D, starting with the next attribute $j + 1$ (line 16).

In order to compute all the formal concepts, we invoke GenerateFrom with $\langle \emptyset^{\downarrow}, \emptyset^{\downarrow\uparrow} \rangle$ and $y = 0$ as its arguments. Then, after finitely many steps, the algorithm produces all formal concepts, each of them exactly once. The soundness of the algorithm is proved in [16], cf. also [12].

Relationship to Other Sequential Algorithms Conceptually, GenerateFrom is the same algorithm as Close-by-One proposed by Kuznetsov [12] although there are some technical differences. GenerateFrom can be seen as simpler version of Close-by-One since we are not interested in the order of generated concepts. On the other hand, we utilize ComputeClosure which results to a much better performance. The algorithm is similar to Lindig's algorithm [13, 14] in that it performs a depth-first search through the search space of all formal concepts. The key difference between our algorithm and that proposed by Lindig [14] and its variants is the way how we test that new formal concept has already been found. Lindig's algorithm and its variants use additional data structures to store intents of found formal concepts. Thus, after a new formal concept is computed, Lindig's algorithm looks up for the concept in a data structure, typically a search tree or a hashing table. Our algorithm uses similar idea as Ganter's algorithm [8] to ensure that no concept is generated multiple times, see (5). Compared to Ganter's algorithm, the number of concepts which are computed multiple times and "dropped" is much lower, see [16].

5 Parallel Algorithm

The sequential version of our algorithm, described in previous section, lists all formal concepts using a depth-first search through the space of all formal concepts. Consider a calling tree of the recursive procedure GenerateFrom. The parallel version consists in modification of GenerateFrom so that subtrees of the calling tree are executed simultaneously by independent processes. The problem to solve is, given a process, which subtree(s) will be executed in the process, or, put in other words, how to distribute computed formal concepts among the processes.

Computing Formal Concepts in More Processes In the following we describe our approach for computing formal concepts in a given fixed number P of separate processes running in parallel. In the approach, processes are executing subtrees (of the calling tree of GenerateFrom) containing, in the root node, a call of GenerateFrom for a formal concept generated by a predefined number of attributes. The number of attributes, denoted by L, is a second parameter of the parallel algorithm. The parameter has an impact on the distribution of computed formal concepts among the processes, see Remark 3 on page 9.

The algorithm, consisting in modification of GenerateFrom, first simulates original sequential GenerateFrom until it reaches the recursion level at which formal concepts generated by $0 < L \leq n$ attributes are to be processed. The initial recursion halts at level which equals L, counting recursion levels from 0 upwards. The formal concepts generated by L attributes, i.e. formal concepts $\langle C, D\rangle = \langle\{y_0, \ldots, y_{L-1}\}^{\downarrow}, \{y_0, \ldots, y_{L-1}\}^{\downarrow\uparrow}\rangle$ such that $y_i \in Y$, are stored in a queue instead of being processed. For each of the P processes there is exactly one queue and the selection of the queue to which we store $\langle C, D\rangle$ is the key point of the algorithm. In fact, by selecting a queue we select a process which will list all formal concepts descendant to $\langle C, D\rangle$. The optimal selection method should distribute all formal concepts to processes equally. 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. In the present version of the algorithm we select process r, where r is the total number of stored formal concepts so far modulo the number P of processes.

After filling up the queues, the modified procedure then forks itself into P processes (or, alternatively, runs the following in $P - 1$ new processes too), and in each process the original sequential GenerateFrom is called for each formal concept in the queue of the respective process. This will list all the remaining descendant formal concepts, in parallel.

Description of the Algorithm The algorithm is represented by a procedure ParallelGenerateFrom, the modification of GenerateFrom which accepts one additional argument: the recursion level counter l, which is used to recognize the recursion level L at which formal concepts generated by L attributes are to be stored in a queue rather than processed. After its invocation, ParallelGenerateFrom proceeds as follows:

- Until it reaches the recursion level $L > 0$, the procedure simulates original GenerateFrom (lines 6–24). The code is identical, with two exceptions: first, instead of exiting at line 8 it skips to the point where original GenerateFrom ends and, second, upon each recursive call of itself it increases the recursion level counter l (line 21). In this step it (sequentially) processes all formal concepts generated by up to $L - 1$ attributes.
- When recursion level counter l is equal to L, i.e. the procedure is about to process formal concept $\langle A, B\rangle$ generated by L attributes, it (instead of processing $\langle A, B\rangle$) stores $\langle A, B\rangle$ and y (the attribute to be added to B) to queue *queue*$[r]$ of selected process r and exits current branch of recursion (lines 2–4). In this step, all formal concepts generated by L attributes are stored in the queues.
- Notice that when ParallelGenerateFrom exits a branch of recursion at line 4, the execution continues at line 22 because line 21 is the only place where ParallelGenerateFrom is recursively called. Therefore, it continues at line

Procedure ParallelGenerateFrom($\langle A, B\rangle, y, l$)

```
 1 if $l = L$ then
 2     select $r$ from 0 to $P - 1$ (e.g. $r = (\sum_{s=0}^{P-1} queue[s]) \mod P$);
 3     store $(\langle A, B\rangle, y)$ to $queue[r]$;
 4     return
 5 end
 6 process $B$ (e.g., print $B$ on screen);
 7 if $B = Y$ or $y > n$ then
 8     goto line 25;
 9 end
10 for $j$ from $y$ upto $n$ do
11     if $B[j] = 0$ then
12         set $\langle C, D\rangle$ to ComputeClosure($\langle A, B\rangle, j$);
13         set $skip$ to false;
14         for $k$ from 0 upto $j - 1$ do
15             if $D[k] \neq B[k]$ then
16                 set $skip$ to true;
17                 break for loop;
18             end
19         end
20         if $skip$ = false then
21             ParallelGenerateFrom($\langle C, D\rangle, j + 1, l + 1$);
22         end
23     end
24 end
25 if $l = 0$ then
26     for $r$ from 1 upto $P - 1$ do
27         new process
28             while set $(\langle C, D\rangle, j)$ to load from $queue[r]$ do
29                 GenerateFrom($\langle C, D\rangle, j$);
30             end
31         end
32     end
33     while set $(\langle C, D\rangle, j)$ to load from $queue[0]$ do
34         GenerateFrom($\langle C, D\rangle, j$);
35     end
36 end
37 return
```

25 after exiting the loop between line 10–24. Here, it either exits the current branch of recursion (if $l \neq 0$) or continues if the top recursion level ($l = 0$) has been reached (i.e., no more branches of recursion are on the call stack).

- On the top recursion level ($l = 0$), it runs new $P - 1$ processes running in parallel (lines 26, 27) and the last step is performed by the new processes too.
- Finally, still on the top recursion level only, in each process, it calls original GenerateFrom for each formal concept $\langle C, D\rangle$ and attribute j in the queue

of the respective process (lines 28–30 and 33–35). That means, all formal concepts generated by L or more attributes are processed in separate processes running in parallel.

In order to compute all the formal concepts, we invoke ParallelGenerateFrom with $\langle \emptyset^{\downarrow}, \emptyset^{\downarrow\uparrow} \rangle$, $y = 0$ and $l = 0$ as its arguments. Then, after finitely many steps, the algorithm produces all formal concepts, each of them exactly once. The soundness of the algorithm follows directly from the soundness of the sequential version [12, 16] and the fact that processes compute predefined disjoint sub-collections of all formal concepts. This also means that the processes do not interfere with each other and hence the algorithm needs no synchronization. We postpone the proof to the full version of the paper. The parallelization also does not increase the overall theoretical complexity of the algorithm which is the same as for the sequential version.

Remark 3. Note that the parameter L, in addition to the process selection method, also determines the number of formal concepts computed by each process. If $L = 1$, most of the formal concepts (formal concepts descendant to a formal concept generated by a single attribute) are computed by one or two processes. With increasing L, formal concepts are distributed to processes more equally. On the other hand, however, with increasing L more formal concepts are computed sequentially and less in parallel. From our experimentation it seems a good trade-off value is already $L = 2$, where almost all formal concepts (for $n \gg L$) are computed in parallel and are distributed to processes nearly optimally. This will be further discussed in Section 6.

Remark 4. There have been several approaches to parallel algorithms in FCA. For instance, [7] 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 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 [11] where the algorithm is based on dividing the input data into disjoin 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.

6 Experimental Evaluation

We have run several experiments to compare the algorithm with other algorithms for computing formal concepts. In the experiments, we have used Ganter's [8], Lindig's [14] and Berry's [4] algorithms and were interested in the performance of the algorithms measured by the running time. Furthermore, we have run several experiments to compare algorithm performances in dependence on number of

dataset	mushroom	tic-tac-toe	Debian tags	anonymous web
size	8124×119	958×29	14315×475	32710×295
density	19 %	34 %	< 1 %	1 %
our (1 CPU)	6.543	0.092	12.746	65.221
our (2 CPUs)	3.541	0.047	7.710	33.364
our (4 CPUs)	2.343	0.035	4.545	18.520
our (8 CPUs)	1.393	0.029	3.043	11.466
Ganter's	834.409	2.158	1720.827	10039.733
Lindig's	5271.988	14.530	2639.670	13422.643
Berry's	934.507	5.783	1531.944	3615.078

Fig. 1. Performance for selected datasets (seconds)

used CPUs. For the sake of comparison, we have implemented all the algorithms in ANSI C. The experiments were done on otherwise idle 64-bit x86_64 hardware with 8 independent processors (dual processor workstation with Quad-core Intel Xeon Processor E5345, 2.33 GHz, 12 GB RAM).

Note that even the serial version of our algorithm significantly outperforms the most commonly used algorithms for FCA. A detailed comparison can be found in [16]. In this section, we focus primarily on the scalability of our algorithm, i.e., we focus on the speed improvement with growing number of hardware processors.

Our first experiment compares our algorithm with various FCA algorithms using several data tables from the UCI Machine Learning Repository [1], UCI Knowledge Discovery in Databases Archive [10], and our dataset describing packages in the Debian GNU/Linux [6]. The results, along with the information on size and density (percentage of 1s) of used data sets, are depicted in Figure 1. First four rows contain computation times measured in seconds in case of our algorithm which has been run on 1 (sequential version), 2, 4, and 8 hardware processors. From all the graphs and tables we can see that our algorithm (significantly) outperforms all the other algorithms.

We now focus on the scalability of the algorithm, i.e., ability to decrease running time using multiple CPUs (or more precisely CPU cores). We have used selected data sets and various randomly generated data tables. Fig. 2 (left) contains results for selected datasets while Fig. 2 (right) contains results for randomly generated tables with 10000 objects and 5 % density of 1's. By a *relative speedup* which is shown on y-axes in the graphs, we mean the theoretical speedup given by 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 CPU (the sequential version of the algorithm) and running time using multiple CPU cores. Note that the theoretical maximum of speedup is equal to the number of used CPUs but real speedup is always smaller due to certain overhead caused by managing of multiple threads of computation. Nevertheless, from the point of view of the speedup, we can see from the experiments

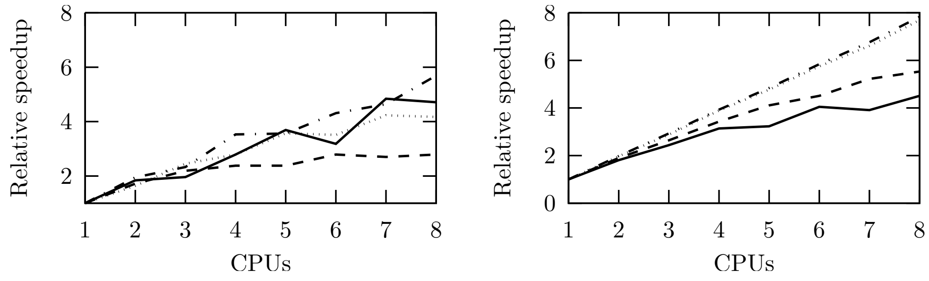

Fig. 2. Relative speedup dependent on various data tables (solid line—mushrooms, dashed line—tic-tac-toe, dotted line—Debian tags, dot-and-dashed line—annonymous web) and used CPU cores (on the left); relative speedup dependent on number of attributes (solid line—50 attributes, dashed line—100 attributes, dotted line—150 attributes, dot-and-dashed line—200 attributes) and used CPU cores measured using randomly generated contexts with 10000 objects and 5 % density (on the right).

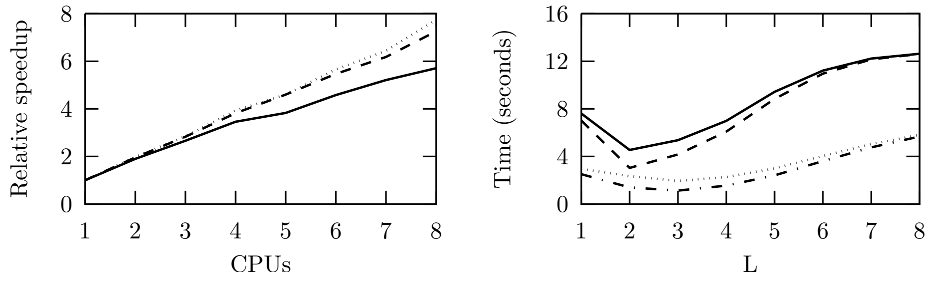

Fig. 3. Relative speedup dependent on density of 1's (solid line—5 %, dashed line—10 %, dotted line—20 %) and used CPU cores (on the left); running time dependent on the argument L (the solid line is for the Debian tags data table and 4 CPUs used, the dashed line is for the Debian tags data table and 8 CPUs used, the dotted lines is for the mushrooms data table and 4 CPUs used and dot-and-dashed lines is for the mushrooms data table and 8 CPUs used) (on the right).

that with growing number of attributes, the real speedup of the algorithm is near its theoretical limits.

In next experiment, that is depicted in Fig. 3 (left), we were focusing on the impact of density of 1's. That is, we have generated data tables with various densities and observed the impact on the scalability. We have used data tables of size 100×10000. Finally, Fig. 3 (right) illustrates the influence of parameter L on various data tables and amounts of CPU cores. The experiments indicate that good choice is $L \in \{2, 3\}$, see Remark 3.

7 Conclusions

We have introduced a parallel algorithm for computing formal concepts in object-attribute data tables. The parallel algorithm is an extension of the serial algorithm we have proposed in [16]. The algorithm consists of a procedure for computing closures and a recursive procedure for computing formal concepts. The main feature of the recursive procedure is that it simulates the sequential one up to a point where the procedure forks into multiple processes and each process computes a disjoint set of formal concepts. Due to our design of the algorithm, there is no need for synchronization which 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 future research will focus on further refinements of the algorithm and comparison with other approaches.

References

1. Asuncion A., Newman D. UCI Machine Learning Repository. University of California, Irvine, School of Information and Computer Sciences, 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 2005*, pp. 207–221.
3. Belohlavek R., Vychodil V. On boolean factor analysis with formal concept as factors. *Proceedings of SCIS & ISIS 2006,* pp. 1054–1059, 2006. Tokyo, Japan: Tokyo Institute of Technology.
4. Berry A., Bordat J.-P., Sigayret A. A local approach to concept generation. *Annals of Mathematics and Artificial Intelligence*, **49**(2007), 117–136.
5. Carpineto C., Romano G. *Concept data analysis. Theory and applications.* J. Wiley, 2004.
6. *DAMOL Dataset Repository* (in preparation).
7. Fu H., Mephu Nguifo E.: A Parallel Algorithm to Generate Formal Concepts for Large Data. *ICFCA 2004, LNCS* **2961**, pp. 394–401.
8. Ganter B. *Two basic algorithms in concept analysis.* (Technical Report FB4-Preprint No. 831). TH Darmstadt, 1984.
9. Ganter B., Wille R. *Formal concept analysis. Mathematical foundations.* Berlin: Springer, 1999.
10. Hettich S., Bay S. D.: The UCI KDD Archive University of California, Irvine, School of Information and Computer Sciences, 1999.
11. Kengue J. F. D., Valtchev P., Djamégni C. T.: A Parallel Algorithm for Lattice Construction. *ICFCA 2005, LNCS* **3403**, pp. 249–264.
12. Kuznetsov S.: Learning of Simple Conceptual Graphs from Positive and Negative Examples. *PKDD 1999*, pp. 384–391.
13. Kuznetsov S., Obiedkov S. Comparing performance of algorithms for generating concept lattices. *J. Exp. Theor. Artif. Int.*, **14**(2002), 189–216.
14. Lindig C. Fast concept analysis. *Working with Conceptual Structures– –Contributions to ICCS 2000,* pp. 152–161, 2000. Aachen: Shaker Verlag.
15. Miettinen P., Mielikäinen T., Gionis A., Das G., Mannila H. The discrete basis problem. *PKDD*, pp. 335–346, 2006. Springer.
16. Outrata J., Vychodil V. Fast algorithm for computing maximal rectangles from object-attribute relational data (submitted).
17. Vychodil V.: A new algorithm for computing formal concepts. In: Trappl R. (Ed.): *Cybernetics and Systems 2008: Proc. 19th EMCSR*, 2008, pp. 15–21.
18. Wille R. Restructuring lattice theory: an approach based on hierarchies of concepts. *Ordered Sets*, pp. 445–470, 1982. Dordrecht-Boston.