Distributed Formal Concept Analysis Algorithms Based on an Iterative MapReduce Framework

Biao Xu, Ruairí de Fréin, Eric Robson, and Mícheál Ó Foghlú

Telecommunications Software & Systems Group,
Waterford Institute of Technology, Ireland
{bxu,rdefrein,erobson,mofoghlu}@tssg.org

Abstract. While many existing formal concept analysis algorithms are efficient, they are typically unsuitable for distributed implementation. Taking the MapReduce (MR) framework as our inspiration we introduce a distributed approach for performing formal concept mining. Our method has its novelty in that we use a light-weight MapReduce runtime called Twister which is better suited to iterative algorithms than recent distributed approaches. First, we describe the theoretical foundations underpinning our distributed formal concept analysis approach. Second, we provide a representative exemplar of how a classic centralized algorithm can be implemented in a distributed fashion using our methodology: we modify Ganter's classic algorithm by introducing a family of MR* algorithms, namely MRGanter and MRGanter+ where the prefix denotes the algorithm's lineage. To evaluate the factors that impact distributed algorithm performance, we compare our MR* algorithms with the state-of-the-art. Experiments conducted on real datasets demonstrate that MRGanter+ is efficient, scalable and an appealing algorithm for distributed problems.

Keywords: Formal Concept Analysis, Distributed Mining, MapReduce.

1 Introduction

Formal Concept Analysis (FCA), pioneered in the 80's by Wille [1], is a method for extracting formal concepts –natural clusters of objects and attributes– from binary object-attribute relational data. FCA has great appeal in the context of knowledge discovery [2], information retrieval [3] and social networking analysis applications [4] because arranging data as a concept lattice yields a powerful and intuitive representation of the dataset [1,5].

The main short-coming of FCA –which has curtailed a more widespread uptake of the approach– is that FCA becomes prohibitively time consuming as the dataset size increases. However, association rules mining tends to deal with large datasets. FCA relies on the fact that the set of concept intents is closed under intersection [6], namely, a closure system. Appealingly, using this property, new formal concepts may be extracted iteratively by extending an existing intent,

F. Domenach, D.I. Ignatov, and J. Poelmans (Eds.): ICFCA 2012, LNAI 7278, pp. 292–308, 2012.

in practice, by intersecting it with a new attribute and shrinking the extent in an iteration. While existing FCA algorithms perform this iterative procedure efficiently for small centralized datasets, the recent explosion in dataset sizes, privacy protection concerns, and the distributed nature of the systems that collect this data, suggests that efficient *distributed* FCA algorithms are required. In this paper we introduce a distributed FCA approach based on a light-weight MapReduce runtime called Twister [7], which is suited to iterative algorithms, scales well and reduces communication overhead.

1.1 Related Work

Some well-known algorithms for performing FCA include Ganter's algorithm [8], Lindig's algorithm [9] and CloseByOne [10,11] and their variants [12,13]. Ganter introduces *lectic* ordering so that all possible attribute subsets of the data do not have to be scanned when performing FCA. Ganter's algorithm computes concepts iteratively based on the previous concept without incurring exponential memory requirements. In contrast, CloseByOne produces many concepts in each iteration. Bordat's algorithm [14] runs in almost the same amount of time as Ganter's algorithm, however, it takes a local concept generation approach. Bordat's algorithm introduces a data structure to store previously found concepts, which results in considerable time savings. Berry proposes an efficient algorithm based on Bordat's approach which requires a data structure of exponential size in [15]. A comparison of theoretical and empirical complexity of many well-known FCA algorithms is given in [16]. In addition, some useful principles for evaluating algorithm performance for sparse and dense data are suggested by Kuznetsov and Obiedkov; We consider data density when evaluating our approach.

The main disadvantage of the batch algorithms discussed above is that they require that the entire lattice is reconstructed from scratch if the database changes. Incremental algorithms address this problem by updating the lattice structure when a new object is added to database. Incremental approaches have been made popular by Norris [17], Dowling [18], Godin et al. [19], Capineto and Romano [20], Valtchev et al. [21] and Yu et al. [22]. In recent years, to reduce concept enumeration time, some parallel and distributed algorithms have been proposed. Krajca et al., proposed a parallel version based on CloseByOne [13]. The first distributed algorithm [23] was developed by Krajca and Vychodil in 2009 using the MapReduce framework [24]. In order to encourage more widespread usage of FCA, beyond the traditional FCA audience, we propose the development and implementation of efficient, distributed FCA algorithms. Distributed FCA is appealing as distributed approaches that can take advantage of cloud infrastructures to reduce enumeration time, are attractive for practitioners.

1.2 Contributions

We utilize the MapReduce framework in this paper to execute distributed algorithms on different nodes. Several implementations of MapReduce have been developed by a number of companies and organizations, such as Hadoop MapReduce by Apache[1], and Twister Iterative MapReduce[2], since its introduction by Google in 2004. A crucial distinction between the present paper and the work of Krajca and Vychodil [23] is that we use a Twister implementation of MapReduce. Twister supports iterative algorithms [7]: we leverage this property to reduce the computation time of our distributed FCA algorithms. In contrast, Hadoop architecture is designed for performing single step MapReduce. We implement new distributed versions (MRGanter and MRGanter+) of Ganter's algorithm and empirically evaluate their performance. In order to provide an established and credible benchmark under equivalent experimental conditions, MRCbo, the distributed version of CloseByOne is implemented as well using Twister.

This paper is organized as follows. Section 2 reviews Formal Concept Analysis and Ganter's algorithm. The theoretical underpinnings for implementing FCA using distributed databases are described in Section 3 to support our approach. Our main contribution is a set of Twister-based distributed versions of Ganter's algorithm. Section 4 presents an implementation overview and comparison of MapReduce, Hadoop and Twister. Empirical evaluation of the algorithms proposed in this paper is performed using datasets from the UCI KDD machine learning repository; experimental results are discussed in Section 5. In summary, MRGanter+ performs favourably in comparison to centralized versions.

2 Formal Concept Analysis

We continue by introducing the notational conventions used in the sequel. Let O and P denote a finite set of objects and attributes respectively. The data ensemble, S, may be arranged in Boolean matrix form as follows: the objects and attributes are listed along the rows and columns of the matrix respectively; The symbol × is entered in a row-column position to denote an object has that attribute; An empty entry denotes that the object does not have that attribute. Formally, this matrix describes the binary relation between the sets O and P. The object X has attribute Y if $(X, Y) \in I$, $X \subseteq O$ and $Y \subseteq P$. The triple (O, P, I) is called a formal context. For example, in Table 1, $O = \{1, 2, 3, 4, 5, 6\}$ and $P = \{a, b, c, d, e, f, g\}$, thus object $\{2\}$ has attributes $\{a, c, e, g\}$. We define a derivation operator on X and Y where $X \subseteq O$ and $Y \subseteq P$ as:

$$X' = \{p \in P \mid \forall t \in O : (t, p) \in I\} \tag{1}$$

$$Y' = \{t \in O \mid \forall p \in P : (t, p) \in I\}. \tag{2}$$

The operation X' generates the set of attributes which are common to objects in X. Similarly, Y' generates the set of objects which are common to attributes in Y. A pair $\langle X, Y\rangle$ is called a formal concept of (O, P, I) if and only if $X \subseteq O$, $Y \subseteq P$, $X' = Y$, and $Y' = X$. Given a formal concept $\langle X, Y\rangle$, X and Y are its *extent* and *intent*. The crucial property here is that the mappings $X \mapsto X''$ and

[1] http://hadoop.apache.org/mapreduce/

[2] http://www.iterativemapreduce.org/

Table 1. The symbol × indicates that an object has the corresponding attribute

	a	b	c	d	e	f	g
1	×	×		×		×	
2	×		×		×		×
3		×	×	×		×	×
4		×		×	×		
5	×			×	×	×	
6		×	×			×	×

$Y \mapsto Y''$, commonly known as *closure operators*, hold. The closure operator can be used to calculate the extent and intent that form a formal concept.

Establishing some notion of concept ordering, that is engendering a sub/super-concept hierarchy, is crucial in what follows. Given X_1, $X_2 \subseteq O$ and Y_1, $Y_2 \subseteq P$ the concepts of a context are ordered as follows: $\langle X_1, Y_1\rangle \leqslant \langle X_2, Y_2\rangle$: $\Longleftrightarrow X_1 \subseteq X_2 \Longleftrightarrow Y_2 \subseteq Y_1$, an ordering which is interesting because it facilitates the iterative formation of a complete lattice which is called the concept lattice of the context [6]. In the following sections we describe algorithms for concept lattice formation, namely Ganter's algorithm (also known as NextClosure) and CloseByOne. We then introduce our distributed extensions of these approaches.

2.1 Ganter: Iterative Closure Mining Algorithm

The NextClosure algorithm describes a method for generating new closures which guarantees every closure is enumerated once. Closures are generated iteratively using a pre-defined order, namely lectic ordering. The set of all formal concepts is denoted by $\mathcal{F}$. Let us arrange the elements of $P = \{p_1, \cdots, p_i, \cdots, p_m\}$ in an arbitrary linear order $p_1 < p_2 < \cdots < p_i < \ldots < p_m$, where m is the cardinality of the attribute set, P. The decision to use lectic ordering dictates that any arbitrarily chosen subset of P is also ordered according to the *lectic* ordering which was defined *ab initio*. Given two subsets Y_1, $Y_2 \subseteq P$, Y_1 is lectically smaller than Y_2 if the smallest element in which Y_1 and Y_2 differ belongs to Y_2.

$$Y_1 \leq Y_2 :\Longleftrightarrow \exists_{p_i}(p_i \in Y_2, p_i \notin Y_1, \forall_{p_j < p_i}(p_j \in Y_1 \Longleftrightarrow p_j \in Y_2)). \qquad (3)$$

NextClosure uses (Eqn. 3) as a feasibility condition for accepting new candidate formal concepts. Typically this difference in set membership is made more explicit by denoting the smallest element, p_i, in which the set Y_1 and Y_2 differ.

$$Y_1 \leq_{p_i} Y_2 :\Longleftrightarrow \exists_{p_i}(p_i \in Y_2, p_i \notin Y_1, \forall_{p_j < p_i}(p_j \in Y_1 \Longleftrightarrow p_j \in Y_2)). \qquad (4)$$

To fix ideas, if the order of $P = \{a, b, c, d, e, f, g\}$ is defined as $a < b < c < d < e < f < g$, and two subsets of P, or *itemsets*, $Y_1 = \{a, c, e, g\}$ and $Y_2 = \{a, b, e, g\}$ are examined then $Y_1 \leq Y_2$ because the smallest element in which the two sets differ is b and this element belongs to Y_2.

Each subset $Y \subseteq P$ may yield a closure, $Y'' \subseteq P$; The NextClosure algorithm attempts to find all closures systematically by exploiting lectic ordering.

Table 2. Formal concepts mined from Table 1, including empty concepts

F_1: $\langle\{1,2,3,4,5,6\},\{\}\rangle$
F_2: $\langle\{1,3,5,6\},\{f\}\rangle$
F_3: $\langle\{2,4,5\},\{e\}\rangle$
F_4: $\langle\{1,3,4,5\},\{d\}\rangle$
F_5: $\langle\{1,3,5\},\{d,f\}\rangle$
F_6: $\langle\{4,5\},\{d,e\}\rangle$
F_7: $\langle\{2,3,6\},\{c,g\}\rangle$
F_8: $\langle\{1,3,4,6\},\{b\}\rangle$
F_9: $\langle\{1,3,6\},\{b,f\}\rangle$
F_{10}: $\langle\{1,3,4\},\{b,d\}\rangle$
F_{11}: $\langle\{1,3\},\{b,d,f\}\rangle$
F_{12}: $\langle\{4\},\{b,d,e\}\rangle$
F_{13}: $\langle\{3\ 6\},\{b,c,f,g\}\rangle$
F_{14}: $\langle\{3\},\{b,c,d,f,g\}\rangle$
F_{15}: $\langle\{1,2,5\},\{a\}\rangle$
F_{16}: $\langle\{2,5\},\{a,e\}\rangle$
F_{17}: $\langle\{1,5\},\{a,d,f\}\rangle$
F_{18}: $\langle\{5\},\{a,d,e,f\}\rangle$
F_{19}: $\langle\{2\},\{a,c,e,g\}\rangle$
F_{20}: $\langle\{1\},\{a,b,d,f\}\rangle$
F_{21}: $\langle\{\},\{a,b,c,d,e,f,g\}\rangle$

Let the ordering of P be $p_1 < p_2 < \ldots < p_i < \ldots < p_m$, and consider the subset $Y \subseteq P$. The generative operation is the $\oplus$-operation: a new intent is generated by applying $\oplus$ on an existing intent and an attribute, and is defined as

$$Y \oplus p_i := ((Y \cap \{p_1, \ldots, p_{i-1}\}) \cup \{p_i\})'', \quad \text{where } Y \subseteq P \text{ and } p_i \in P. \tag{5}$$

NextClosure then compares the new candidate formal concept with the previous concept. If the condition in (Eqn. 4) is satisfied the candidate concept produced by (Eqn. 5) is kept and added to the lattice.

The $\oplus$-operator in (Eqn. 5) consists of intersection, union and closure operations; Lectic ordering and the associated complexity of these operations explains why NextClosure's ordered approach incurs high computational expense. Consequently the largest dataset-size NextClosure can practically process is small.

Example 1. *Consider the formal context in Table 1. Assume we have a concept $\langle\{1,5\},\{a,d,f\}\rangle$. We take the attribute set, $Y = \{a,d,f\}$, and calculate, $Y \oplus e$. First, we compute, $\{a,d,f\} \cap \{a,b,c,d\} = \{a,d\}$, then we append e and generate $\{a,d\} \cup \{e\} = \{a,d,e\}$. Performing $\{a,d,f\} \oplus e = \{a,d,e\}''$ yields the set, $\{a,d,e,f\}$. To demonstrate the role of lectic ordering, we compute $Y \oplus c = \{a,c,e\}$. According to the feasibility condition in (Eqn. 4), $\{a,d,e,f\} \leq_c \{a,c,e\}$. Thus, the set, $\{a,c,e\}$, is added to the concept lattice, $\mathcal{F}$. By repeating this process, NextClosure determines that there are 21 formal concepts in the concept lattice representation of the formal context in Table 1. The set of concepts, $\mathcal{F}$, is listed in Table 2.*

Pseudo code for NextClosure is described in the Algorithm 1 and 2 as background to our distributed approach. Algorithm 1 applies the closure operator on the null attribute set and generates the first intent, Y, which is the base for all subsequent formal concepts. New concepts are generated in turn by calling Algorithm 2 and concatenating the resultant concepts to the set of formal concepts, $\mathcal{F}$. As each candidate intent is extended with new attributes, the intent for the last iteration of this loop consists of the complete set of attributes. This feature is used to terminate the loop (in Line 2 of the Algorithm 1). Algorithm 2 accepts the formal context triple, (O, P, I) and current intent, Y, as inputs. By convention, the attribute set P is sorted in descending order. The $\oplus$-operator described in (Eqn. 5) is applied to produce candidate formal concepts. The concept feasibility condition (Eqn. 4) is used to verify whether a

Algorithm 1. AllClosure

```
Input: $\emptyset$: null attribute set.
Output: $\mathcal{F}$: Formal concepts set.
Y ← $\emptyset''$;
while $Y$ is not the last closure do
    $Y$ ← NextClosure();
    $\mathcal{F} \leftarrow \mathcal{F} \cup Y$;
end while
return $\mathcal{F}$
```

Algorithm 2. NextClosure

```
Input: $O, P, I, Y$: formal context & current intent.
Output: $Y$.
for $p_i$ from $p_m$ down to $p_1$ do
    if $p_i \notin Y$ then
        candidate ← $Y \oplus p_i$;
        if candidate $\leq_{p_i} Y$ then
            $Y$ ← candidate;
            break;
        end if
    end if
end for
return $Y$
```

new candidate should be added to the set of formal concepts, $\mathcal{F}$. The approach taken in the CloseByOne algorithm is similar in spirit to the approach taken by the NextClosure algorithm: CloseByOne generates new formal concepts based on concept(s) generated in the previous iteration and tests their feasibility using the operator, $\leq_{p_i}$. The crucial difference is that the CloseByOne algorithm generates many concepts in each iteration. CloseByOne terminates when there are no more concepts that satisfy (Eqn. 4). In short, NextClosure only finds the first feasible formal concept in each iteration whereas CloseByOne potentially generates many. As a consequence, CloseByOne requires far fewer iterations.

The appeal of NextClosure, and explanation for our desire to make it more efficient lies in its thoroughness; the guarantee of a complete lattice structure which is a consequence of the main theorem of Formal Concept Analysis [6]. This thoroughness is due to lectic ordering and the iterative approach deployed by NextClosure; however, thoroughness comes at the cost of high complexity. The advent of efficient mechanisms for dealing with iterative algorithms using MapReduce captured by Twister allow us to couple NextClosure's thoroughness with a practical distributed implementation in this paper.

3 Distributed Algorithms for Formal Concept Mining

We continue by describing two methods for performing distributed NextClosure, namely, MRGanter and MRGanter+. An introduction to Twister is deferred to Section 4. We start by describing the properties of a partitioned dataset compared to its unpartitioned form. In many cases these properties are simply restatements of the properties of the derivations operators.

Given a dataset S, we partition its objects into n subsets and distribute the subsets over n different nodes. Without loss of generality, it is convenient to limit $n = 2$ here. We denote the partitions by S_1 and S_2. Alternatively we can think in terms of formal contexts and write the formal context, (O, P, I), in terms of the partitioned formal contexts (O_{S_1}, P, I_{S_1}) and (O_{S_2}, P, I_{S_2}). To fix ideas, we use the dataset in Table 1 as an exemplar and generate the partitions in Table 3.

Table 3. Partitioned datasets derived from Table 1, S_1 and S_2

S_1 or (O_{S_1}, P, I_{S_1})								S_2 or (O_{S_2}, P, I_{S_2})							
	a	b	c	d	e	f	g		a	b	c	d	e	f	g
1	×	×		×		×		4		×		×	×		
2	×		×		×		×	5	×			×	×	×	
3		×	×	×		×	×	6		×	×			×	×

The partitions are non-overlapping: the intersection of the partitions is the null set, $S_1 \cap S_2 = \emptyset$ and their union gives the full dataset $S = S_1 \cup S_2$. It follows that the partitions, S_1, S_2, have the same attributes sets, P, as the entire dataset S, however, the set of objects is different in each partition, e.g. O_{S_1} and O_{S_2}. Let Y_S, Y_{S_1} and Y_{S_2} denote an arbitrary attribute set Y with respect to the entire dataset S, and each of its partitions S_1 and S_2 respectively. By construction they are equivalent: $Y_S \equiv Y_{S_1} \equiv Y_{S_2}$. Similarly, Y'_S, Y'_{S_1} and Y'_{S_2} are the sets of objects derived by the derivation operation in each of the partitions S_1, S_2 and the entire dataset S respectively.

Property 1. *Given the formal context, (O, P, I), the two partitions (O_{S_1}, P, I_{S_1}) and (O_{S_2}, P, I_{S_2}) and an arbitrary itemset, $Y \subseteq P$, the property $Y'_S = Y'_{S_1} \cup Y'_{S_2}$ holds: the union of the sets of objects generated by the derivation of the attribute set Y in each of the partitions is equivalent to the set of objects generated by the derivation of the attribute set over the entire dataset, S.*

Appealing to the definition of the derivation operator proposed by Wille in [1], the set, Y'_S, is a subset of O, $Y'_S \subseteq O$. Moreover, $Y'_{S_1} \subseteq O_{S_1}$ and $Y'_{S_2} \subseteq O_{S_2}$. Given $S_1 \cup S_2 = S$ and $S_1 \cap S_2 = \emptyset$, it follows that, $O_{S_1} \cup O_{S_2} = O$ and $O_{S_1} \cap O_{S_2} = \emptyset$; Therefore, $Y'_{S_1} \subseteq Y'_S$ and $Y'_{S_2} \subseteq Y'_S$. Finally, $Y'_{S_1} \cup Y'_{S_2} \equiv Y'_S$. As a counterexample, an object t that exists in Y'_S, but not in Y'_{S_1} or Y'_{S_2}, cannot exist because $O_{S_1} \cup O_{S_2} = O$ and $O_{S_1} \cap O_{S_2} = \emptyset$ and $Y_S = Y_{S_1} = Y_{S_2}$. If t is in Y'_S it must appear in Y'_{S_1} or Y'_{S_2}. In short, Property 1 allows us to process all objects independently: the objects can be distributed and processed in an arbitrary order and this will not affect the result of Y'. Property 1 is trivially extended to the case of n partitions. Now we describe how formal concepts can be combined from different partitions.

Property 2. *Given the formal context, (O, P, I), the two partitions (O_{S_1}, P, I_{S_1}) and (O_{S_2}, P, I_{S_2}) and an arbitrary itemset, $Y \subseteq P$, the property $Y''_S = Y''_{S_1} \cap Y''_{S_2}$ holds: The intersection of the closures of the attribute set, Y, with respect to each of the partitions S_1 and S_2 is equivalent to the closure of the attribute set, Y, with respect to the entire dataset S.*

By the definition of the partition construction method above, $S_1 \cup S_2 = S$, which implies that, $S_1 \subset S$ and $S_2 \subset S$. Recall that, $Y'_{S_1} \subset Y'_S$ and $Y'_{S_2} \subset Y'_S$, and from Property 1 we have that $Y'_S = Y'_{S_1} \cup Y'_{S_2}$. Appealing to the properties of the derivation operators, in [1], we have, $Y''_{S_1} \supseteq Y''_S$ and $Y''_{S_2} \supseteq Y''_S$. It is clear that Y''_{S_1} and Y''_{S_2} need not equal Y''_S, but by the definition of a closure

$(Y'_{S_1} \cup Y'_{S_2})' = (Y'_S)' = Y_S$, thus, $(Y'_{S_1} \cup Y'_{S_2})' = Y''_{S_1} \cap Y''_{S_2}$ follows trivially from the definition of the derivations operators.

Example 2. *Consider the following example. Taking itemset $Y = \{b, d\}$. We derive $Y''_{S_1} = \{b, d, f\}$ from the first partition S_1, and $Y''_{S_2} = \{b, d, e\}$ from S_2. We derive $Y''_S = \{b, d\}$ for the entire dataset S. Therefore $Y''_S = Y''_{S_1} \cap Y''_{S_2}$.*

Theorem 1. *Given a set of attributes Y, $Y \subset P$. Let $\mathcal{F}^Y_{S_1}$ and $\mathcal{F}^Y_{S_2}$ be the sets of closures based on Y in relation to S_1 and S_2 respectively. Then the closure set of Y in relation to S can be calculated from: $\mathcal{F}^Y_S = \mathcal{F}^Y_{S_1} \cap \mathcal{F}^Y_{S_2}$*

This is simply a consequence of Property 2 as, $\mathcal{F}^Y_S = Y''_S = Y''_{S_1} \cap Y''_{S_2} = \mathcal{F}^Y_{S_1} \cap \mathcal{F}^Y_{S_2}$ and $Y_S \equiv Y_{S_1} \equiv Y_{S_2}$ by definition of the partition.

Example 3. *Consider again Example 2. Appealing to Theorem 1, the formal concept with respect to the entire data set is the intersection of the formal concepts from each partition $F^Y_S = F^Y_{S_1} \cap F^Y_{S_2} = \{b,d,f\} \cap \{b,d,e\} = \{b,d\}$.*

We denote the k-th partition as S_k and then propose:

Theorem 2. *Given the closures $\mathcal{F}^Y_{S_1}, \ldots, \mathcal{F}^Y_{S_n}$ from n disjoint partitions, $\mathcal{F}^Y_S = \mathcal{F}^Y_{S_1} \cap \ldots \cap \mathcal{F}^Y_{S_n}$.*

A trivial inductive argument establishes that Theorem 2 holds. Theorem 1 proves the $n = 2$ case. Theorem 2 follows by recognizing that the dataset S at the $(k-1)$-th step of the proof can be thought as of consisting of two partitions only, the partition $S_1 \cup \cdots \cup S_{k-1}$ and a second partition S_k.

Calling on nothing more complex than: 1) the properties of the derivation operators, and 2) construction of non-overlapping partitions, we leverage Theorem 2 in order to apply the MapReduce, specifically the Twister variant, to calculate closures from arbitrary number of distributed nodes sure in the knowledge that the thoroughness of NextClosure is preserved.

3.1 MRGanter

To address the dataset size limitations imposed on NextClosure –owing to the complexity of the $\oplus$-operation– we deploy FCA across multiple nodes to reduce the execution time. We demonstrate how decompose NextClosure so that each sub-task is executed in parallel. In Algorithm 2, there were two stages involved in computing NextClosure: 1) computing a new candidate closure, and 2) making a judgment on whether to add it to the evaluated formal concepts. In MapReduce parlance, computing a new candidate closure corresponds to the map stage, and validating its feasibility corresponds to the reduce phase. In this paper, we only calculate the intent of a formal concept. The variables and constants used by distributed algorithms are summarized in Table 4. The main operation in the merging function is the intersection operator, which is applied on the set of local closures L_k generated at each node. Algorithm 3 gives the pseudo code for the merging function based on Theorem 2. To describe the merging operation, we

Table 4. Variables and constants used in distributed FCA

Variables/Constants	Description
p_i	an attribute in P, where $i = 1, \cdots, m$
L_k	complete set of local closures in data partition k, $k = 1, \cdots, n$.
l_i	an intent in L_k which is derived from p_i
d	the intent produced in the previous iteration
f	the newly generated intent
G	a container for storing newly generated intents

Algorithm 3. Merging function

```
Input: p_i, L_k, f.
Output: $f$.
 1: l_i ← the local closure in L_k in terms of $p\_i$;
 2: f ← $\Psi$(l_i, f);
 3: return f
```

Algorithm 4. Map: MRGanter

```
Input: d.
Output: (d, L_k).
 1: for p_i from p_m down to p_1 do
 2:     if p_i is not in d then
 3:         l_i ← d $\oplus$ p_i;
 4:         associate l_i with p_i;
 5:         L_k ← L_k $\cup$ l_i;
 6:     end if
 7:     return (d, L_k);
 8: end for
```

Algorithm 5. Reduce: MRGanter

```
Input: (d,L_k).
Output: f.
 1: for p_i in P do
 2:     f ← initialize new intent;
 3:     for $i$ from 1 up to $m$ do
 4:         f ← merging(p_i, L_k, f);
 5:     end for
 6:     if f $\leq_{p\_i}$ d then
 7:         break;
 8:     else
 9:         continue;
10:     end if
11: end for
12: return f
```

Algorithm 6. Reduce: MRGanter+

```
Input: (d, L_k).
Output: G.
 1: H ← initialize a two-level hash table;
 2: for $p_i$ in $P$ do
 3:     f ← initialize new intent;
 4:     for $i$ from 1 up to $m$ do
 5:         f ← merging(p_i, L_k, f);
 6:     end for
 7:     if f is not in H then
 8:         add f into H;
 9:         add f into G;
10:     end if
11: end for
12: return G
```

introduce the notation, Ψ(l_i, f) =l_i $\cap$ f, which acts on two intents. The merging function is deployed at the reduce phase and only processes local closures derived from the same attribute (Line 1).

The Map phase described in the Algorithm 4 produces all local closures. The output consists of the previous intent d and a set of local intents L_k. In order to be used in the merging function the attribute which was used to form local closures should be recorded and passed (Line 4). All pairs with the same key, d, are sent to the same reducer. All local intents are used to form global intents and then filtered by the closure validation condition (Line 6 in Algorithm 5). Algorithm 5 accepts (d,L_k) from the k-th *mapper*s (see Section 4), where $k = 1, \cdots, n$. Only pairs with the same key, d, are accepted by a Reducer. Line 4

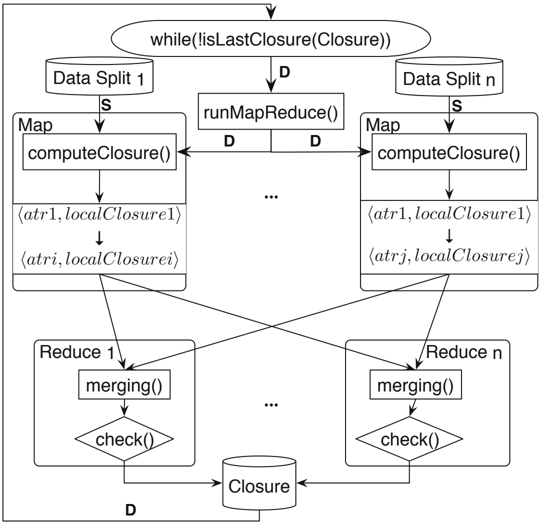

Fig. 1. MRGanter work flow: static data is loaded at the start of the procedure (labeled by S) and the dynamic data (Closures produced during each iteration) is passed and used in the next iteration (labeled by D)

generates an candidate closure f. This candidate is then validated. Successful candidates are outputted as a global closure f.

Fig. 1 depicits the iterative flow of control of MRGanter; the lines marked with "S" import static data from each partition, while the lines marked with "D" configure each map with the previous closure. Each new closure is tested to see if it is the last, e.g. it contains all attributes, P. If this condition is not met MRGanter continues. We present a worked example using the dataset in Table 3. Table 5, does not illustrate all results due to space limitations. MRGanter performs 20 iterations to determine all concepts.

3.2 MRGanter+

NextClosure calculates closures in lectic ordering to ensure every concept appears only once. This approach allows a single concept to be tested with the closure validation condition during each iteration. This is efficient when the algorithm runs on a single machine. For multi-machine computation, the extra computation and redundancy resulting from keeping only one concept after each iteration across many machines is costly. We modify NextClosure to reduce the number of iterations and name the corresponding distributed algorithm MRGanter+.

Rather than using redundancy checking, we keep as many closures as possible in each iteration; All closures are maintained and used to generate the next batch of closures. To this end, we modify Algorithm 5: the Map algorithm remains the same as in Algorithm 4. Algorithm 6 describes the ReduceTask for

Table 5. MRGanter: Only single a intent (bold) produced per iteration.

d	p_i	l_i from S_1	l_i from S_2	f
∅	g	{c,g}	{b,c,f,g}	{c,g}
	f	{b,d,f}	{f}	**{f}**
	e	{a,c,e,g}	{d,e}	{e}
	d	{b,d,f}	{d,e}	{d}
	c	{c,g}	{b,c,f,g}	{c,g}
	b	{b,d,f}	{b}	{b}
	a	{a}	{a,d,e,f}	{a}
{f}	g	{b,c,d,f,g}	{b,c,f,g}	{b,c,f,g}
	e	{a,c,e,g}	{d,e}	**{e}**
	d	{b,d,f}	{d,e}	{d}
	c	{c,g}	{b,c,f,g}	{c,g}
	b	{b,d,f}	{b}	{b}
	a	{a}	{a,d,e,f}	{a}
{e}	g	{a,c,e,g}	{a,. . . ,g}	{a,c,e,g}
	f	{a,. . . ,g}	{a,d,e,f}	{a,d,e,f}
	d	{b,d,f}	{d,e}	**{d}**
	c	{c,g}	{b,c,f,g}	{c,g}
	b	{b,d,f}	{b}	{b}
	a	{a}	{a,d,e,f}	{a}
{d}	g	{b,c,d,f,g}	{a,. . . ,g}	{b,c,d,f,g}
	f	{b,d,f}	{a,d,e,f}	**{d,f}**
	e	{a,. . . ,g}	{d,e}	{d,e}
	c	{c,g}	{b,c,f,g}	{c,g}
	b	{b,d,f}	{b}	{b}
	a	{a}	{a,d,e,f}	{a}

Table 6. MRGanter+: Many intents (bold) produced per iteration

d	p_i	l_i from S_1	l_i from S_2	f
∅	g	{c,g}	{b,c,f,g}	**{c,g}**
	f	{b,d,f}	{f}	**{f}**
	e	{a,c,e,g}	{d,e}	**{e}**
	d	{b,d,f}	{d,e	**{d}**
	c	{c,g}	{b,c,f,g}	{c,g}
	b	{b,d,f}	{b}	**{b}**
	a	{a}	{a,d,e,f}	**{a}**
{cg}	f	{b,c,d,f,g}	{b,c,f,g}	**{b,c,f,g}**
	e	{a,c,e,g}	{a,. . . ,g}	**{a,c,e,g}**
	d	{b,c,d,f,g}	{a,. . . ,g}	**{b,c,d,f,g}**
	b	{b,d,f}	{b}	{b}
	a	{a}	{a,d,e,f}	{a}
{f}	g	{b,c,d,f,g}	{b,c,f,g}	{b,c,f,g}
	e	{a,c,e,g}	{d,e}	{e}
	d	{b,d,f}	{d,e}	{d}
	c	{c,g}	{b,c,f,g}	{c,g}
	b	{b,d,f}	{b}	{b}
	a	{a}	{a,d,e,f}	{a}
{e}	g	{a,c,e,g}	{a,. . . ,g}	{a,c,e,g}
	f	{a,. . . ,g}	{a,d,e,f}	**{a,d,e,f}**
	d	{b,d,f}	{d,e}	{d}
	c	{c,g}	{b,c,f,g}	{c,g}
	b	{b,d,f}	{b}	{b}
	a	{a}	{a,d,e,f}	{a}

MRGanter+. The Reduce in MRGanter+ merges local closures first in Line 5, and then recursively examines if they already exist in the set of global formal concepts H (Line 7). The set H is used to fast index and search a specified closure; it is designed as a two-level hash table to reduce its costs. The first level is indexed by the head attribute of the closure, while the second level is indexed by the length of the closure. New closures are stored in G. We present a running example based on the dataset in Table 3 for comparison. MRGanter+ produces many intents in each iteration. New intents are kept if they are not already in H. Notably, MRGanter+ requires 3 iterations to mine all concepts. Moreover, we implement CloseByOne proposed by Krajca and Vychodil in [23] based on the MapReduce framework and call it, MRCbo. Comparing MRGanter+ with MRCbo, we demonstrate that MRGanter+ typically generates more concepts in each iteration and uses fewer iterations. Detailed analysis is given in Section 5.2.

4 Twister MapReduce

The MapReduce framework adopts a divide-conquer strategy to deal with huge datasets and is applicable to many classes of problems [25]. A large number of computers, collectively referred to as a cluster, are used to run the algorithm.

MapReduce was inspired by the map and reduce functions commonly used in functional programming, for example Lisp. It was introduced by Google [24] and then implemented by many companies (Google, Yahoo!) and organizations (Apache). These implementations provide automatic parallelization and distribution, fault-tolerance, I/O scheduling, status and monitoring. The only demand made of the user is the formulation of the problem in terms of map and reduce functions. We use the terminology *mapper* and *reducer* when we refer to the map and reduce function respectively. The map function takes an input pair and produces a set of intermediate key/value pairs. The MapReduce library provides the ability to acquire input pairs from files or databases which are stored in distributed way. Additionally, it can group all intermediate values associated with the same intermediate key I and pass them to the same reducer. The reduce function accepts an intermediate key I and a set of values associated with I. It merges these values to form a possibly smaller set of values.

Twister [7] was designed to enhance MapReduce's functionality by efficiently supporting iterative algorithms. Twister uses a public/subscribe messaging infrastructure for communication and data transfer, and introduces long running map/reduce tasks which can be re-used in different iterations. These long running tasks, which last for the duration of the entire computation, ensures that Twister avoids reading static data in each execution of MapReduce; a considerable saving. For iterative algorithms, Twister categorizes data as being either static or dynamic. Static data is the distributed data in local machines. Dynamic data is typically the data produced by the previous iteration. Twister's *configure* phase allows the specification of where the mapper reads the static data. Calculation is performed cyclically based upon the dynamic and static data. All communication between the mappers and the reducers is handled by a broker network[3].

Unlike Twister, Hadoop focuses on single step MapReduce and lacks built-in support for iterative programs. For iterative algorithms, Hadoop MapReduce chains multiple jobs together. The output of a previous MapReduce task is used as the input for the next MapReduce task[4]. This approach is suboptimal; it incurs the additional cost of repetitively applying MapReduce –the disadvantage is that new map/reduce tasks are created repetitively for different iterations. This incurs considerable performance overhead costs.

5 Evaluation

We provide evidence of the effectiveness and scalability of our algorithm in this section. First we describe the experimental environment and the dataset characteristics for the datasets used. Then, we describe our experimental results.

5.1 Test Environment and Datasets

MRGanter and MRGanter+ are implemented in Java using the Twister runtime as the distributed environment. In addition, MRCbo, a distributed version of

[3] NaradaBrokering is used in this paper http://www.naradabrokering.org/

[4] http://hadooptutorial.wikispaces.com/Iterative+MapReduce+and+Counters

Table 7. UCI dataset characteristics: numbers of objects, attributes, and density

Dataset	mushroom	anon-web	census-income
objects	8124	32711	103950
attributes	125	294	133
density	17.36%	1.03%	6.7%

Table 8. Execution time: Distributed algorithms are the fastest (in seconds)

Dataset	mushroom	anon-web	census-income
concepts	219010	129009	96531
NextClosure	618	14671	18230
CloseByOne	2543	656	7465
MRGanter	20269(5 nodes)	20110 (3 nodes)	9654 (11 nodes)
MRCbo	241 (11 nodes)	693 (11 nodes)	803 (11 nodes)
MRGanter+	198 (9 nodes)	496 (9 nodes)	358 (11 nodes)

CloseByOne proposed by Krajca and Vychodil [23] is implemented using the Twister model in order to provide a fair comparison with the algorithms proposed in the present paper. To illustrate the performance improvement of our distributed approach, we also evaluate NextClosure and CloseByOne.

The experiments were run on the Amazon EC2 cloud computing platform. We used High-CPU Medium Instances which had 1.7 GB of memory, 5 EC2 Compute Units (2 virtual cores with 2.5 EC2 Compute Units each), 350 GB of local instance storage, and a 32-bit platform. We selected 3 datasets from UCI KDD machine learning repository, mushroom, anon-web, and census-income for this evaluation[5]. These datasets have 8124, 32711, 103950 records and 125, 294, 133 attributes respectively. We used the percentage of 1s to measure the dataset density (see row 4 in Table 7). CPU time was used as the metric for comparing the performance of each of the algorithm. The number of iterations used by each algorithms was also recorded in Table 9.

5.2 Results and Analysis

In Table 8, we present the best test results for the centralized algorithms, NextClosure and CloseByOne, and the distributed algorithms, MRGanter, MRCbo and MRGanter+. In short, it is clear that MRGanter+ has the best overall performance for the mushroom, anon-web and census datasets when 9 nodes and 11 nodes are used respectively. In comparison with NextClosure, MRGanter+ demonstrates a 97.6% time saving improvement. MRGanter+ runs 102 times faster than MRGanter and 1.4 times faster than MRCbo. MRCbo runs much faster than CloseByOne when 11 nodes are used. It presents a 90.5% saving in time when dealing with the mushroom dataset compared to CloseByOne, but

[5] http://archive.ics.uci.edu/ml/index.html

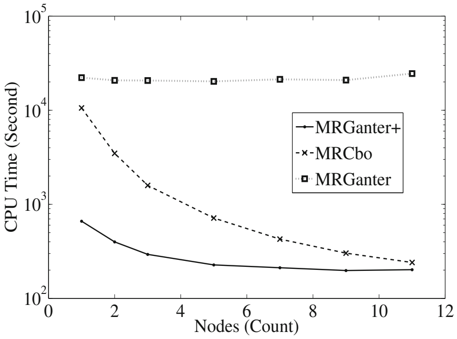

Fig. 2. Mushroom dataset: comparison of MRGanter+, MRCbo and MRGanter. MRGanter+ outperforms MRCbo and MRGanter when dense data is processed.

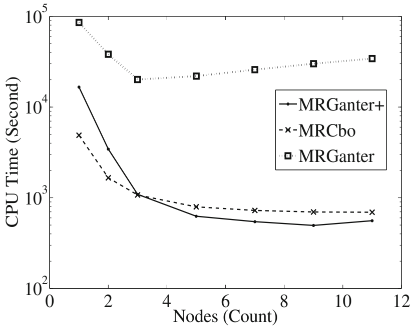

Fig. 3. Anon-web dataset: comparison of MRGanter+, MRCbo and MRGanter. MRGanter+ is faster when more than 3 nodes are used.

there is not much of difference when the anon-web dataset is processed. MRGanter takes the longest time to calculate the formal concepts for both the mushroom and anon-web datasets. It is much slower than even the centralized version, NextClosure. The census-income dataset is an exception because MRGanter saves up to half the time with 11 nodes. Among the MR* algorithms and centralized algorithms, MRGanter+ achieves the best performance.

Taking scalability into account, we tested MR* algorithms on a range of nodes to demonstrate the ability of the algorithms to decrease computation time by utilizing more computers. These results are presented in Fig. 2, 3 and 4 for each dataset.

In Fig. 2, MRCbo is slower than MRGanter+ although this curve decreases faster than MRGanter+ when we increase the number of nodes. The execution time of MRGanter+ is fast even on a single node and the execution time keeps decreasing up to the maximum number of nodes, 11. The performance of MRGanter is not beneficially affected by increasing the number of nodes. One explanation for this is the overhead incurred by distributing the computation, for example network communication overhead. This is markedly different from MRGanter+, because MRGanter+ produces substantially more intermediate data than MRGanter and MRCbo. Moreover, there is additional computation involved in the distributed algorithms in comparison with the centralized versions of these algorithms. Consider, for instance, the extra operation needed by the merging operation. The best number of nodes, in terms of performance speed, depends on the density characteristics of the dataset.

Fig. 3 demonstrates that MRGanter+ outperforms MRGanter for the anon-web dataset. One reason for this performance improvement is that both algorithms produce different numbers of concepts during each iteration. Table 9 indicates that MRGanter+ requires 12, 11 and 9 iterations for each of the datasets, whereas MRGanter requires 219010, 129009 and 96531 iterations to obtain all

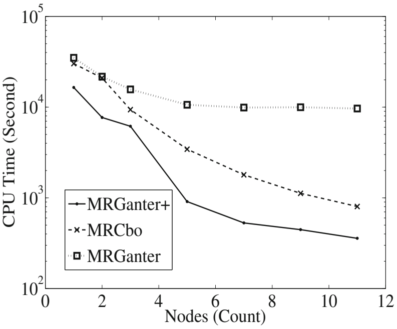

Fig. 4. Census dataset: comparison of MRGanter+, MRCbo and MRGanter. MRGanter+ is fastest when a large dataset is processed.

Table 9. Number of iterations required for each of the three datasets

Dataset	mushroom	anon-web	census-income
concepts	219010	129009	96531
NextClosure	219010	129009	96531
CloseByOne	14	11	11
MRGanter	219010	129009	96531
MRCbo	14	11	11
MRGanter+	12	11	9

concepts. These additional iterations incur higher network communication costs. Fig. 4 demonstrates that this is also the case for the census dataset. In addition, the curves in Fig. 4 are steeper than the curves in Fig. 2 and 3. These figures give evidence that the performance of the MR* algorithms is related to size and density of the data. Based on these results we posit that MR* algorithms scale well for large and sparse datasets. This evidence suggests that MR* algorithms may be a viable candidate tool for handling large datasets, particularly when it is impractical to use a traditional centralized technique.

Classical formal concept computing methods usually act on, and have local access to the entire database. Network communication is the primary concern when developing distributed FCA approaches: Frequent requests to remote databases incur significant time and resource costs. Performance improvements of the algorithms proposed in this paper may potentially arise from preprocessing the dataset so that the dataset is partitioned in a more efficient manner. One direction for improving these algorithms lies in making the partitions more even, in terms of density, so that the complexity is distributed more equably. In future work we we intend to explore the effect of data distribution between cluster nodes in more detail. We propose to extend this empirical study in a companion paper which examines algorithm performance on larger dataset sizes. We will also study the affects the data distribution has on the optimal number of nodes. In addition, we intend to extend these methods so that they reduce the size of intermediate data produced in each iteration. We posit that further improvement of the methods proposed here could motivate a more widespread adoption of FCA using the Map-Reduce framework.

6 Conclusion

In this paper we considered methods for extending the NextClosure FCA algorithm. A formal description of dealing with distributed datasets for the

NextClosure FCA was discussed. Two new distributed FCA algorithms, MRGanter and MRGanter+, were proposed based on this discussion. Various implementation aspects of these approaches were discussed based on empirical evaluation of the algorithms. These experiments demonstrated the advantages of our approach and the scalability in particular of MRGanter+. By comparing MRGanter+ with MRCbo and MRGanter, we found that the number of iterations significantly impacted the performance of distributed FCA, a promising result. In future work we hope to capitalize on this by improving the MR* methodology by reducing the number of iterations of these approaches and to further reduce computation time.

References

1. Wille, R.: Restructuring Lattice Theory: an Approach Based on Hierarchies of Concepts. In: Rival, I. (ed.) Ordered Sets, pp. 445–470. Reidel (1982)
2. Lakhal, L., Stumme, G.: Efficient Mining of Association Rules Based on Formal Concept Analysis. In: Ganter, B., Stumme, G., Wille, R. (eds.) Formal Concept Analysis. LNCS (LNAI), vol. 3626, pp. 180–195. Springer, Heidelberg (2005)
3. Polaillon, G., Aufaure, M.-A., Le Grand, B., Soto, M.: FCA for Contextual Semantic Navigation and Information Retrieval in Heterogeneous Information Systems. In: DEXA Workshops 2007, pp. 534–539 (2007)
4. Snásel, V., Horak, Z., Kocibova, J., Abraham, A.: Analyzing Social Networks Using FCA: Complexity Aspects. In: Web Intelligence/IAT Workshops 2009, pp. 38–41 (2009)
5. Caspard, N., Monjardet, B.: The Lattices of Closure Systems, Closure Operators, and Implicational Systems on a Finite Set: A Survey. Discrete Applied Mathematics, 241–269 (2003)
6. Ganter, B., Wille, R.: Formal Concept Analysis: Mathematical Foundations. Springer, Heidelberg (1999)
7. Ekanayake, J., Li, H., Zhang, B., Gunarathne, T., Bae, S.-H., Qiu, J., Fox, G.: Twister: a Runtime for Iterative MapReduce. In: Hariri, S., Keahey, K. (eds.) HPDC, pp. 810–818. ACM (2010)
8. Ganter, B.: Two Basic Algorithms in Concept Analysis. In: Kwuida, L., Sertkaya, B. (eds.) ICFCA 2010. LNCS, vol. 5986, pp. 312–340. Springer, Heidelberg (2010)
9. Lindig, C.: Fast Concept Analysis. In: Working with Conceptual Structures-Contributions to ICCS, pp. 235–248 (2000)
10. Kuznetsov, S.O.: A Fast Algorithm for Computing All Intersections of Objects in a Finite Semi-Lattice. Automatic Documentation and Mathematical Linguistics 27(5), 11–21 (1993)
11. Andrews, S.: In-Close, a Fast Algorithm for Computing Formal Concepts. In: The Seventeenth International Conference on Conceptual Structures (2009)
12. Vychodil, V.: A New Algorithm for Computing Formal Concepts. Cybernetics and Systems, 15–21 (2008)
13. Krajca, P., Outrata, J., Vychodil, V.: Parallel Recursive Algorithm for FCA. In: CLA 2008, vol. 433, pp. 71–82. CLA (2008)
14. Bordat, J.-P.: Calcul pratique du treillis de Galois d'une correspondance. Mathématiques et Sciences Humaines 96, 31–47 (1986)
15. Berry, A., Bordat, J.-P., Sigayret, A.: A Local Approach to Concept Generation. Ann. Math. Artif. Intell. 49(1), 117–136 (2006)

16. Kuznetsov, S.O., Obiedkov, S.A.: Comparing Performance of Algorithms for Generating Concept Lattices. J. Exp. Theor. Artif. Intell. 14, 189–216 (2002)
17. Norris, E.M.: An Algorithm for Computing the Maximal Rectangles in a Binary Relation. Rev. Roum. Math. Pures et Appl. 23(2), 243–250 (1978)
18. Dowling, C.E.: On the Irredundant Generation of Knowledge Spaces. J. Math. Psychol. 37, 49–62 (1993)
19. Godin, R., Missaoui, R., Alaoui, H.: Incremental Concept Formation Algorithms Based on Galois (Concept) Lattices. Computational Intelligence 11, 246–267 (1995)
20. Carpineto, C., Romano, G.: A Lattice Conceptual Clustering System and Its Application to Browsing Retrieval. Machine Learning, 95–122 (1996)
21. Valtchev, P., Missaoui, R., Lebrun, P.: A Partition-based Approach Towards Constructing Galois (concept) Lattices. Discrete Mathematics, 801–829 (2002)
22. Yu, Y., Qian, X., Zhong, F., Li, X.-R.: An Improved Incremental Algorithm for Constructing Concept Lattices. In: Proceedings of the 2009 WRI World Congress on Software Engineering, WCSE 2009, vol. 04, pp. 401–405. IEEE Computer Society, Washington, DC (2009)
23. Krajca, P., Vychodil, V.: Distributed Algorithm for Computing Formal Concepts Using Map-Reduce Framework. In: Adams, N.M., Robardet, C., Siebes, A., Boulicaut, J.-F. (eds.) IDA 2009. LNCS, vol. 5772, pp. 333–344. Springer, Heidelberg (2009)
24. Dean, J., Ghemawat, S.: MapReduce: Simplified Data Processing on Large Clusters. In: OSDI, p. 13 (2004)
25. Chu, C.T., Kim, S.K., Lin, Y.A., Yu, Y., Bradski, G.R., Ng, A.Y., Olukotun, K.: Map-Reduce for Machine Learning on Multicore. In: Schölkopf, B., Platt, J.C., Hoffman, T. (eds.) NIPS, pp. 281–288. MIT Press (2006)