The Journal of Supercomputing (2019) 75:7909–7920
https://doi.org/10.1007/s11227-018-2473-8

A formally based parallelization of data mining algorithms for multi-core systems

Ivan Kholod[1] · Andrey Shorov[1] · Evgenii Titkov[1] · Sergei Gorlatch[2]

Published online: 7 July 2018
© Springer Science+Business Media, LLC, part of Springer Nature 2018

Abstract
We describe a novel, systematic approach to efficiently parallelizing data mining algorithms: starting with the representation of an algorithm as a sequential composition of functions, we formally transform it into a parallel form using higher-order functions for specifying parallelism. We implement the approach as an extension of the industrial-strength Java-based library Xelopes, and we illustrate its use by developing a multi-threaded Java program for the popular naive Bayes classification algorithm. In comparison with the popular MapReduce programming model, our resulting programs enable not only data-parallel, but also task-parallel implementation and a combination of both. Our experiments demonstrate an efficient parallelization and good scalability on multi-core processors.

Keywords Parallel algorithms · Data mining · Parallel data mining · Program transformation · Functional programming · Parallel programming

1 Introduction

Data mining algorithms have recently become increasingly popular, especially for analyzing Big data. Since data mining is often very time intensive, its parallelization for modern multi-core processors is in increasing demand.

✉ Ivan Kholod
iiholod@mail.ru

Andrey Shorov
ashxz@mail.ru

Evgenii Titkov
titkov.evgen@gmail.com

Sergei Gorlatch
gorlatch@uni-muenster.de

[1] Saint Petersburg Electrotechnical University “LETI”, Saint Petersburg, Russia

[2] University of Muenster, Münster, Germany

Several parallel algorithms for solving different data mining tasks [1] have been developed recently, e.g., frequent item set mining [2], clustering [3], and building decision trees [4]. Our detailed survey of related work is in [5]. This research has demonstrated that creating a parallel algorithm for a specific task is complicated and requires a lot of effort for developing and debugging. A more generic alternative is the popular MapReduce programming model [6]. It uses the abstraction inspired by the *map* and *reduce* primitives present in many functional programming languages. If an application can be expressed using suitable functions *map* and *reduce*, then the MapReduce approach allows the developer to abstract from the problems of parallelization, data management, errors handling, etc. The MapReduce programming model has been widely used for different tasks of data mining. Paper [7] shows how some data mining algorithms can be decomposed into *map* and *reduce* functions by transforming an algorithm into the summation form: the *map* function computes the statistic sufficiency for each portion of data, and the *reduce* function performs the aggregation of results.

The MapReduce programming model has good scalability on cluster or cloud systems, because it is originally oriented toward system with distributed memory: *map* functions are executed in a distributed manner, and their results are combined by the *reduce* functions. However, this restriction does not allow MapReduce to use advantages of shared memory in a multi-core system, in particular to combine partial results more efficiently.

We propose a novel approach to parallelizing algorithms of data mining. Our approach facilitates a more flexible parallelization than MapReduce, including task-parallel execution, and it allows to compute common result without an additional combining phase. At the same time, we use the formal principles of functional programming for ensuring the correctness of parallelization. We implement our approach as an extension of the Xelopes commercial library [8]. The extended library supports the developer in systematically transforming a sequential data mining algorithm into several efficient parallel versions.

2 Formally based parallelization

In our formalism, we use capital letters for types and lowercase letters for variables of these types and functions on them.

2.1 Representing algorithms as functions composition

In our approach, we represent a data mining algorithm as a composition of functions, e.g.,

$$dma = f_n \circ f_{n-1} \circ \cdots \circ f_1 \circ f_0 \tag{1}$$

Here, function $f_0 \colon D \to M$ takes a data set $d \in D$ as an argument and returns a mining model $m_0 \in M$. Functions $f_t \colon M \to M, t = 1 \dots n$ take the mining model $m_{t-1} \in M$ created by the previous function f_{t-1} and return the changed mining model

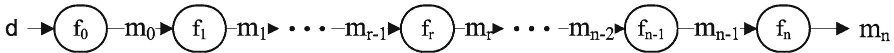

Fig. 1 Algorithm as a composition of functions

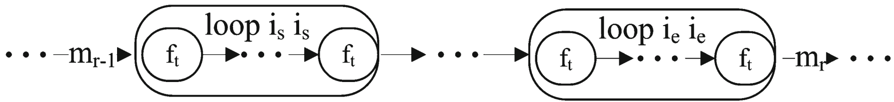

Fig. 2 The loop function as a composition of FMBs

```
1. for j = 1...z // loop for each vector
2. //increment count of vectors for value x_{j.t} of the dependent t^{th} attribute of vector x_j;
       m[o1 + x_{j.t}] + +;
3.     for k = 1...p // loop for each attribute
4.         // increment count of vectors with value x_{j.k} of the independent k^{th} attribute
           // and value x_{j.t} of the dependent t^{th} attribute of vector x_j
           m[i1 + k× | Def_k | +x_{j.k} + x_{j.t}] + +;
       end for;
   end for;
```

Fig. 3 The naive Bayes algorithm: pseudocode

$m_t \in M$. Note that the functions in (1) are applied from right to left. We call functions in (1) functional mining blocks (FMB).

An algorithm expressed as a composition of functions (1) is intuitively represented in Fig. 1.

Loops in data mining algorithms are expressed using higher-order function *loop* that applies an FMB f_t to the mining model's elements from index i_s till index i_e (see Fig. 2):

$$loop\colon I \to I \to (M \to M) \to M \to M \tag{2}$$
$$loop\ i_s\ i_e\ f_t\ m = ((loop\ i_e\ i_e\ f_t) \circ \cdots \circ (loop\ i_s\ i_s\ f_t))\ m$$

Here, function *loop* executes the FMB f_t for the mining model's elements with indices from i_s till i_e. I is a set of mining model's arrays indices.

2.2 The naive Bayes algorithm: illustration

To illustrate our approach, we use a classification algorithm—the naive Bayes [10]—as simple example data mining application that belongs to the top 10 data mining algorithms [9] and allows us to illustrate the features of our approach.

In Fig. 3, we show the pseudocode of the naive Bayes algorithm. Here, data set d is a set of objects, each of which belongs to a known class and has a known vector of attributes. The attribute which defines the class of object is called dependent attribute

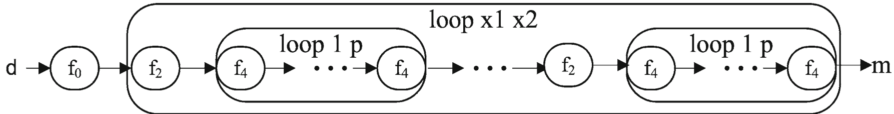

Fig. 4 Naive Bayes algorithm as a function composition

a_t (for example, risk of loan default). The attributes with other characteristics of the object are independent attributes $a_k : k = 1 \ldots m$ (for example, for a borrower, they can be: age, income, etc).

The naive Bayes algorithm applies Bayes' theorem with the naive assumption of independence between every pair of attributes. It calculates:

- the number of vectors with the value $x_{j.t} = v_{t.p}$, for each value $v_{t.p}$ ($v_{t.p} \in Def_t$) of a dependent attribute a_t (line 2 in Fig. 3);
- the number of vectors with value $x_{j.k} = v_{k.q}$ of the independent attribute a_k and with value $x_{j.t} = v_{t.p}$ of the dependent attribute a_t, for each value $v_{k.q}$ of each independent attribute a_k (line 4 in Fig. 3).

If we represent naive Bayes in the format (1), then function f_0 initializes the array of the mining model's elements, adding the following elements for the naive Bayes algorithm:

- elements $e_1,\ldots, e_p$ with information about the attributes of a data set d (where p is the number of attributes);
- elements $e_{v1},\ldots, e_{v2}$ with information about the independent attributes values (where $v1 = p + 1$, $v2 = v1 + p\times \mid Def_k \mid$);
- elements $e_{t1}, \ldots, e_{t2}$ with information about the dependent attributes values (where $t1 = v1$, $t2 = t1+ \mid Def_t \mid$);
- elements $e_{x1},\ldots, e_{x2}$ with information about the vectors of a data set d (where $x1 = t2 + 1$, $x2 = x1 + z + 1$ and z is the number of vectors);
- elements $e_{o1},\ldots, e_{o2}$, for saving information about the number of vectors with value $x_{j.t} = v_{t.p}$ of the dependent attribute a_t (where $o1 = x2 + 1$, $o2 = o1+ \mid Def_t \mid$);
- elements $e_{i1}, \ldots, e_{i2}$, for saving information about the number of vectors with value $x_{j.k} = v_{k.q}$ of independent attribute a_k and with value $x_{j.t} = v_{t.p}$ of the dependent attribute a_t (where i1=o2+1, $i2 = i1 + p \times (\mid Def_k \mid + \mid Def_t \mid)$).

Figure 4 represents the naive Bayes algorithm described by pseudocode in Fig. 3 according to our approach as a composition of the FMBs as follows:

$$NB = f_1 \circ f_0 = (loop\ x1\ x2\ (f_3 \circ f_2)) \circ f_0 = (loop\ x1\ x2\ ((loop\ 1\ p\ f_4) \circ f_2)) \circ f_0 \tag{3}$$

- f_1 is the loop for mining model's elements $e_{x1},\ldots, e_{x2}$ (line 1 in Fig. 3):

$$f_1 = loop\ x1\ x2\ (f_3 \circ f_2);$$

- f_2 increments the number n of the vectors with the value $v_{t.p}$ of the tth dependent attribute (line 2 in Fig. 3);

– f_3 is the loop for the mining model's elements $e_1, \ldots, e_p$ (line 3 in Fig. 3):

$$f_3 = loop\ 1\ p\ f_4;$$

– f_4 increments the number n of the vectors with the value $v_{k.q}$ of the kth independent attribute and value $v_{t.p}$ of the tth dependent attribute (line 4 in Fig. 3).

2.3 Formally based parallelization

Note that in (1), f_t and f_{t+1} can be executed in parallel iff there are no data dependencies between them. We use the following notation:

– $Out(f_t)$ is the subset of mining model's elements modified by FMB f_t;
– $In(f_t)$ is the subset of mining model's elements used by FMB f_t.

We use the classical Bernstein's conditions [11]: two FMBs f_t and f_{t+1} can be executed in parallel in a system with shared memory if:

– there is no data flow dependency: $Out(f_t) \cap In(f_{t+1}) = \oslash$;
– there is no output dependency: $Out(f_t) \cap Out(f_{t+1}) = \oslash$;
– there is no data anti-dependency: $In(f_t) \cap Out(f_{t+1}) = \oslash$.

For expressing the parallel execution of FMBs in systems with shared memory, we introduce the higher-order function *parallel*:

$$\begin{aligned} ¶llel\colon [(M \to M)] \to M \to M \\ ¶llel\ [f_1, \ldots, f_r]\ m = head\ \ fork\ [f_1, \ldots, f_r]\ m, \end{aligned} \tag{4}$$

where function *fork* invokes FMBs in parallel:

$$\begin{aligned} &fork\colon [M \to M] \to M \to [M] \\ &fork\ [f_1, \ldots, f_r]\ m = [f_1\ m, \ldots, f_r\ m]. \end{aligned} \tag{5}$$

Parallel FMBs compute the common mining model in shared memory. Therefore, function *fork* calls FMBs on the same mining model m and returns a list of references on the common mining models. Thus, function *parallel* can return any (for example, first) mining model from the list as result. To return the first element of a list, we use the *head* function.

Note that the general form of function *parallel* (4) can parallelize different FMBs $f_s, \ldots, f_r$ in a task-parallel manner. On the other hand, data parallelism can also be implemented by applying *parallel* to function *loop* (2). If an algorithm is represented by (1), then FMB f_r is a loop ($f_r \equiv loop\ i_s\ i_e\ f_t$), such that applying FMB f_t to a pair of mining model's elements, ($loop\ i\ i\ f_t$) and ($loop\ i+1\ i+1\ f_t$), where $i_s < i < i_e$, satisfies the Bernstein's conditions, then FMB f_r can be executed in parallel:

$$parallel\ [loop\ i_s\ i_e\ f_t] = parallel\ [(loop\ i_s\ i_s\ f_t), \ldots, (loop\ i_e\ i_e\ f_t)] \tag{6}$$

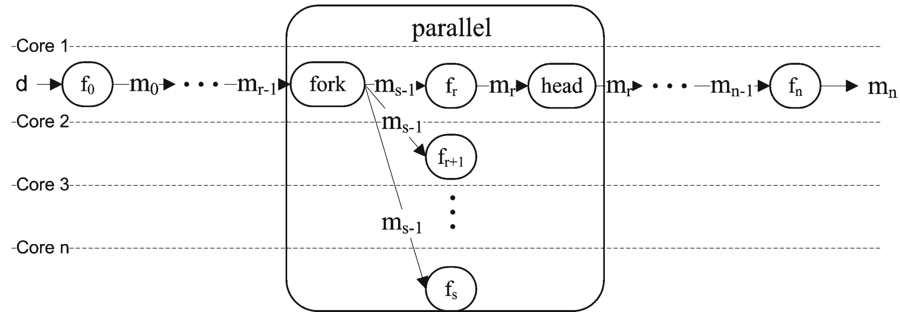

Fig. 5 Parallel execution of a data mining algorithm on multiple cores

This parallelization of a loop over vectors is a generalization of the MapReduce approach, where FMB f_t is an analog of the *map* function. Note that, unlike MapReduce, function *parallel* can be used several times for parallelizing different parts of the algorithm.

Summarizing, we parallelize a data mining algorithm in our approach in the following three steps:

1. the algorithm is represented as a composition (1) of functions $f_t, t = 0 \dots n$;
2. the sets of used and modified mining model's elements are calculated for each FMB, and Bernstein's conditions for each pair of consecutive FMBs are verified;
3. if Bernstein's conditions are satisfied, the sequential execution of $f_s, \dots, f_r$ is transformed into the parallel execution as follows:

$$f_n \circ \cdots \circ f_r \circ \cdots \circ f_s \circ \cdots \circ f_0 = f_n \circ \cdots \circ (parallel\ [f_s, \dots, f_r]) \circ \cdots \circ f_0$$

The parallel execution according to this approach is shown in Fig. 5.

For the example of naive Bayes algorithm, we show how to calculate the sets in Step 2) manually; see below. For the general case, in our future work we plan to adapt the well-developed methods of dependence analysis in compilers (like in [12,13]) to perform Step 2) automatically.

2.4 The naive Bayes algorithm: illustration

Let consider parallelizing the naive Bayes algorithm as an example. The first step was described in Sect. 2.2; its result is the algorithm representation as a composition expressed by (3). In the second step, we determine the sets of used and modified mining model's elements for the FMBs of the algorithm: f_1, f_2, f_3, f_4. We do it based on the pseudocode in Fig. 3. For example, the sets of used and modified elements for f_2 and f_3 are determined as follows based on the lines 2, 3 of the pseudocode in Fig. 3:

$$In(f_2) = e_{o1},\ \ Out(f_2) = e_{o1}, In(f_3) = e_{i1}, \dots, e_{i2}, Out(f_3) = e_{i1}, \dots, e_{i2}.$$

Table 1 Sets In and Out for the FMBs of the naive Bayes algorithm

FMB	In	Out
$f_1 = loop\ x1\ x2\ f_3 \circ f_2$	$e_{o1}, \ldots, e_{o2}, e_{i1}, \ldots, e_{i2}$	$e_{o1}, \ldots, e_{o2}, e_{i1}, \ldots, e_{i2}$
f_2	e_{o1}	e_{o1}
$f_3 = loop\ 1\ p\ f_4$	$e_{i1}, \ldots, e_{i2}$	$e_{i1}, \ldots, e_{i2}$
f_4	e_{i1}	e_{i1}

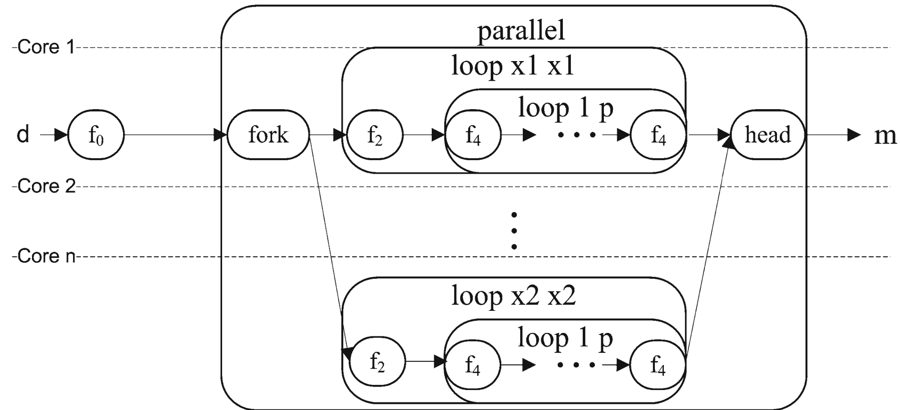

Fig. 6 Parallel execution of naive Bayes (variant NBParVec) on multiple cores

The full sets In and Out for all FMBs of the naive Bayes algorithm are presented in Table 1.

Verifying the Bernstein's conditions for all FMBs allows us to obtain the following variants of the parallelized naive Bayes algorithm:

– with parallel execution of the loop for vectors f_1 (variant NBParVec);
– with a parallel execution of the loop for attributes f_3 (variant NBParAttr);
– with parallel execution of FMBs f_2 and f_3 (variant NBParFMB);

as explained in the following.

The variant with parallel execution of the loop for vectors applies the *parallel* function to the FMB f_1 (Fig. 6):

$$NBParVec = (parallel\ [loop\ x1\ x2\ ((loop\ 1\ p\ f_4) \circ f_2)]) \circ f_0. \tag{7}$$

This variant is the traditional data-parallel way of parallelization using the MapReduce.

The variant with a parallel execution of the loop for attributes f_3 (variant NBParAttr) applies the parallel function to the loop f_3 (Fig. 7):

$$NBParAttr = (loop\ x1\ x2\ (parallel\ [loop\ 1\ p\ f_4]) \circ f_2) \circ f_0. \tag{8}$$

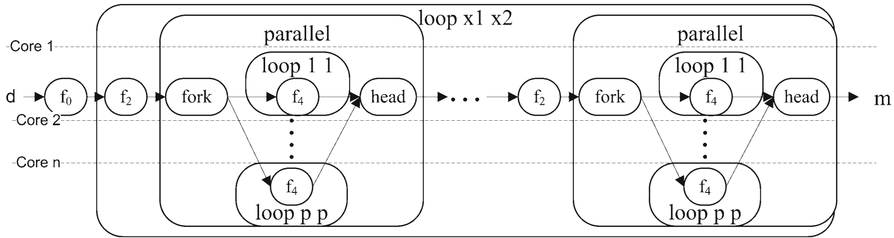

Fig. 7 Parallel execution of naive Bayes (variant NBParAttr) on multiple cores

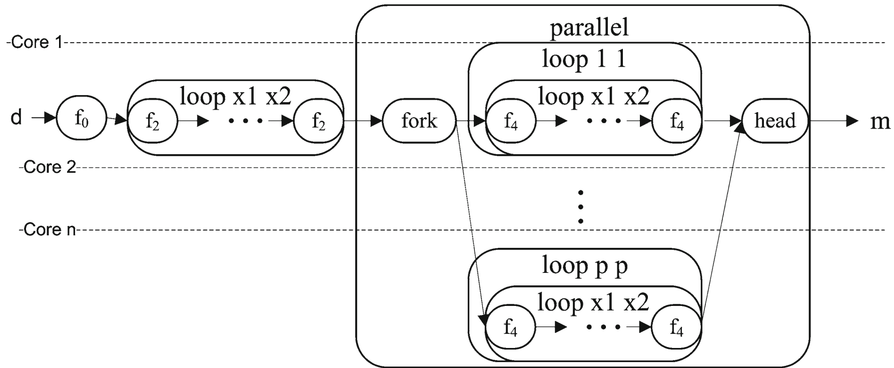

Fig. 8 Parallel execution of naive Bayes (variant NBParAttr') on multiple cores

This variant can be implemented by applying MapReduce to the attributes of the input data set. In the variant NBParAttr, the *fork* and *head* functions are invoked (from function *parallel* (4)) in the loop for each vector. The runtime of them may be long for a large number of vectors.

To avoid this, we reconstruct the Naive Base algorithm (3) by two loop transformations [13]: loop fission (for loop x1 x2) and loop reversal (for loop x1 x2 and loop 1 p):

$$NB' = (loop\ 1\ p\ (loop\ x1\ x2\ f_4)) \circ (loop\ x1\ x2\ f_2) \circ f_0. \tag{9}$$

This variant can be parallelized as follows (Fig. 8):

$$NBParAttr' = (parallel\ [loop\ 1\ p\ (loop\ x1\ x2\ f_4)]) \circ (loop\ x1\ x2\ f_2) \circ f_0. \tag{10}$$

In this variant, the *fork* and *head* functions will be invoked (from function *parallel* (4)) in the loop for each attribute.

In the variant NBParFMB, the parallel function is applied to both loops (Fig. 9):

$$NBParFMB = (parallel\ [loop\ 1\ p\ (loop\ x1\ x2\ f_4), (loop\ x1\ x2\ f_2)]) \circ f_0. \tag{11}$$

This variant realizes task parallelism that cannot be implemented by MapReduce.

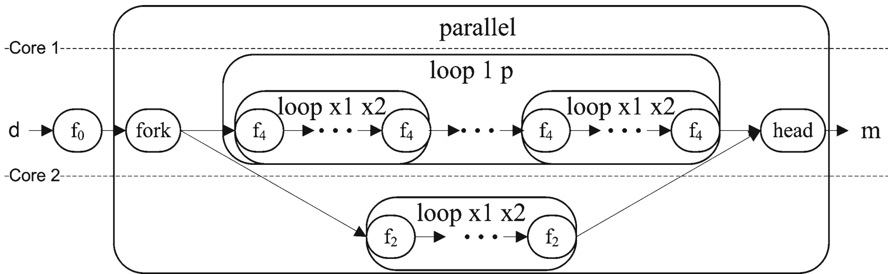

Fig. 9 Parallel execution of naive Bayes (variant NBParFMB) on multiple cores

Table 2 Parameters of experimental data sets

Input data set	ds1	ds2	ds3	ds4	ds5	ds6	ds7
Number of vectors (thousand)	851	3403	6805	11,059	14,461	18,715	28,993
Data volume (Gb)	0.1	0.5	1	1.5	2	2.5	3

Additionally, we can combine all three variants (NBParFMB, NBParAttr and NBParVec) as follows, which is also not possible in MapReduce:

$$NBParAll = (parallel\ [(parallel\ [loop\ 1\ p\ (parallel\ loop\ x1\ x2\ f_4)]), (parallel\ [loop\ x1\ x2\ f_2])]) \circ f_0. \quad (12)$$

3 Results of experiments

Our approach is implemented as an extension of the commercial Java-based library Xelopes [8] containing a variety of algorithms for data mining . Using it, we perform several experiments for the implemented parallel versions of the naive Bayes algorithm.

For experiments, we use the data set predict outcome of pregnancy from the Kaggle data sets [14]—a collection of data sets that are used by the machine learning community for the empirical analysis of machine learning algorithms. This data set contains data on Annual Health Survey: Woman Schedule. We use data with 68 independent attributes about birth history: type of medical attention at delivery; details of maternal health care; and other. The data set has one dependent attribute related to the outcome of pregnancy(s) (live birth/still birth/abortion).

The data set is represented as a cvs file with volume 3 Gb. We experiment with 7 files of different volumes (Table 2).

The experiments run on the following multi-core computer: CPU Intel Xenon (12 physical cores), 2.90 GHz, 4 Gb of memory.

In Figs. 10 and 11, we show the results of our experiments on the data sets from Table 2 with the following parallel variants of the naive Bayes algorithm: NBParVec 7); NBParAttr (10); and NBParAll (12).

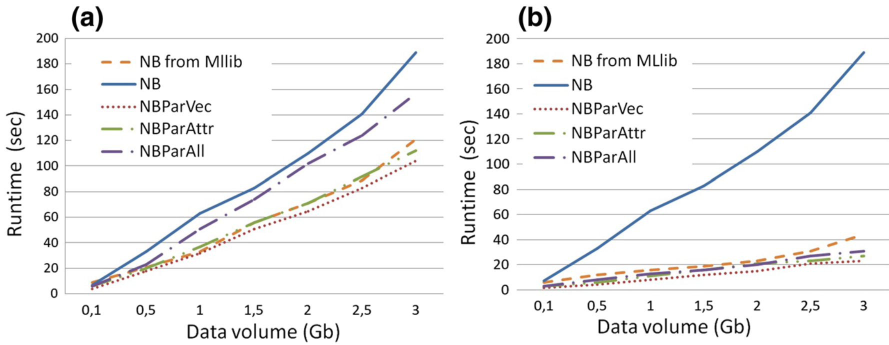

Fig. 10 Execution time of the naive Bayes algorithm: **a** for 2 cores; **b** for 12 cores

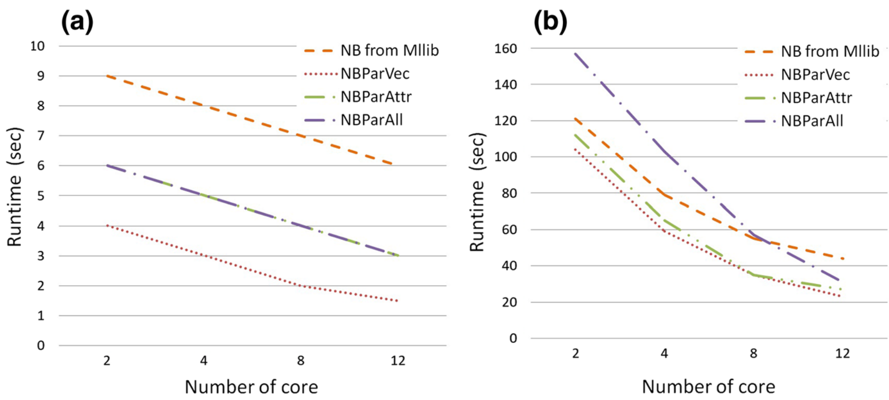

Fig. 11 Execution time of the naive Bayes algorithm: **a** for 1 Gb; **b** for 3 Gb

We compare our results with Apache Spark MLib (MLlib) [15]—a popular data mining library for the Apache Spark platform. We compare our results to the sequential implementation of naive Bayes (NB) according to (3) and its parallel variants for 2 cores (Fig. 10a) and 12 cores (Fig. 10b). Figure 11 shows a comparison of parallel variants of naive Bayes for different numbers of cores for data sets of 1Gb (Fig. 11a) and 3Gb (Fig. 11b).

All parallel variants of the naive Bayes algorithm have a lower runtime than the sequential variant (Fig. 10). Their efficiency is increasing with the increasing volume of analyzed data sets, because parallel handling of a large volume of data set compensates the overhead of parallel execution (creation and running of threads, execution of *fork* and *head* functions, etc.).

The implementation of parallel variants in our framework shows a better runtime than in MLib of Apache Spark, because Apache Spark implements the MapReduce model and uses the *reduce* function to combine partial results in a resulting mining model, which requires additional time.

The parallel variant NBParVec is the fastest, because it performs a one-time reading of the cvs file with the data set and parallelizes the loop for vectors that has the maximum number of iterations. The variant NBParAttr is slower, because it opens the

cvs file in each iteration of the loop for attributes (68 times). The variant NBParAll is the slowest for 2 cores because it parallelizes only two FMBs (the variant NBParAll for 2 cores is equivalent to the variant NBParFMB (11)) where the first FMB is the longest loop for attributes and vectors. Therefore, this variant is unbalanced. For 12 cores, this variant is close to NBParAttr, because the first FMB is additionally parallelized.

4 Conclusion

We describe a new, formally based approach to the parallelization of data mining algorithms using their functional representation.

We demonstrate that our approach is more general than the MapReduce programming model. Our approach has the following advantages:

1. we cover both data and task parallelism and a combination of both;
2. we can obtain several parallel variants of a data mining algorithm;
3. we can use shared memory and decrease the overhead of parallelism;
4. we implement the approach as an extension of the commercial library Xelopes.

We plan to extend, our approach to a broader variety of environments, including heterogeneous and distributed systems.

Acknowledgements This work was supported by the Ministry of Education and Science of the Russian Federation in the framework of the state order "Organization of Scientific Research," task 2.6113.2017/6.7, and by the German Ministry of Education and Research (BMBF) in the framework of the HPC2SE project at the University of Muenster.

References

1. Wu X, Zhu X, Wu GQ, Ding W (2014) Data mining with big data. IEEE Trans Knowl Data Eng 26(1):97–107
2. Zaki M (1999) Parallel and distributed association mining: a survey. IEEE Concurr 7(4):14–25
3. Kadam P, Jadhav S, Kulkarni A, Kulkarni S (2017) Survey of parallel implementations of clustering algorithms. Int J Adv Res Comput Commun Eng 6(10):46–52
4. Zaki MJ, Ho C-T, Agrawal R (1999) Parallel classification for data mining on shared-memory multiprocessors. In: ICDE: IEEE International Conference on Data Engineering, pp 198–205
5. Kholod I, Shorov A, Gorlatch S (2017) Creation of data mining algorithms as functional expression for parallel and distributed execution. In: Malyshkin V (ed) PaCT 2017, LNCS, vol 10421. Springer, Basel, pp 459–472
6. Dean J, Ghemawat S (2008) MapReduce: simplified data processing on large clusters. Commun ACM 51:107–113
7. Chu C-T et al (2006) Map-reduce for machine learning on multicore. In: Proceedings of the Twentieth Annual Conference on Neural Information Processing Systems, Vancouver, Canada, pp 281–288
8. Prudsys Xelopes. https://prudsys.de/en/knowledge/technology/prudsys-xelopes/
9. Wu X et al (2007) Top 10 algorithms in data mining. Knowl Inf Syst 14(1):1–37
10. John GH, Langley P (1995) Estimating continuous distributions in Bayesian classifiers. In: Eleventh Conference on Uncertainty in Artificial Intelligence, pp 338–345
11. Bernstein J (1966) Program analysis for parallel processing. IEEE Trans Electron Comput EC–15:757–762
12. Li Z, Yew P-C, Zhu C-Q (1990) An efficient data dependence analysis for parallelizing compilers. IEEE Trans Parallel Distrib Syst 1:26–34

13. Allen R, Kennedy K (2002) Optimizing compilers for modern architectures. Morgan Kaufmann, San Francisco
14. Kaggle Dataset. https://www.kaggle.com/rajanand/ahs-woman-1
15. Machine Learning Library (MLlib) Guide. http://spark.apache.org/docs/latest/mllib-guide.html