Parallelization of the PC Algorithm

Anders L. Madsen[1 2], Frank Jensen[1], Antonio Salmerón[3], Helge Langseth[4], and Thomas D. Nielsen[2]

[1] HUGIN EXPERT A/S, Aalborg, Denmark
[2] Department of Computer Science, Aalborg University, Denmark
[3] Department of Mathematics, University of Almería, Spain
[4] Department of Computer and Information Science, Norwegian University of Science and Technology, Norway

Abstract. This paper describes a parallel version of the PC algorithm for learning the structure of a Bayesian network from data. The PC algorithm is a constraint-based algorithm consisting of five steps where the first step is to perform a set of (conditional) independence tests while the remaining four steps relate to identifying the structure of the Bayesian network using the results of the (conditional) independence tests. In this paper, we describe a new approach to parallelization of the (conditional) independence testing as experiments illustrate that this is by far the most time consuming step. The proposed parallel PC algorithm is evaluated on data sets generated at random from five different real-world Bayesian networks. The results demonstrate that significant time performance improvements are possible using the proposed algorithm.

Keywords: Bayesian network, PC algorithm, parallelization

1 Introduction

A *Bayesian network* (BN) [9, 11] is a powerful model for probabilistic inference. It consists of two main parts: a graphical structure specifying a set of dependence and independence relations between its variables and a set of conditional probability distributions quantifying the strengths of the dependence relations. The graphical nature of a Bayesian network makes it well-suited for representing complex problems, where the interactions between entities, represented as variables, are described using *conditional probability distributions* (CPDs). Both parts can be elicited from experts or learnt from data, or a combination. Here we focus on learning the graphical structure from data using the PC algorithm [17] exploiting parallel computations.

Large data sets both in terms of variables and cases may challenge the efficiency of pure sequential algorithms for learning the structure of a Bayesian network from data. Since the computational power of computers is ever increasing and access to computers supporting parallel processing is improving, it is natural to consider exploiting parallel computations to improve performance of learning algorithms. In [7] the authors describe a MapReduce-based method for

learning Bayesian networks from massive data using a search & score algorithm while [5] describes a MapReduce-based method for machine learning on multicore computers. Also, [16] presents the R package **bnlearn** which provides implementations of some structure learning algorithms including support for parallel computing. [3] introduces a method for accelerating Bayesian network parameter learning using Hadoop and MapReduce. Other relevant work on parallelization of learning Bayesian network from data include [10], [6], [14], [4] and [7]. In this paper, we describe a parallel version of the PC algorithm for learning the structure of a Bayesian network from large data sets on a shared memory computer using threads. The proposed parallel PC algorithm is inspired by the work of [13] on vertical parallization of TAN learning using Balanced Incomplete Block (BIB) designs [18]. The results of an empirical evaluation shows a significant improvement in time performance over a purely sequential implementation.

2 Preliminaries and Notation

2.1 Bayesian Networks

A BN $\mathcal{N} = (\mathcal{X}, G, \mathcal{P})$ over the set of random variables $\mathcal{X} = \{X_1, \ldots, X_n\}$ consists of an acyclic directed graph (DAG) $G = (V, E)$ with vertices V and edges E and a set of CPDs $\mathcal{P} = \{P(X \,|\, \mathrm{pa}(X)) : X \in \mathcal{X}\}$, where $\mathrm{pa}(X)$ denotes the parents of X in G. The BN $\mathcal{N}$ specifies a joint probability distribution over $\mathcal{X}$

$$P(\mathcal{X}) = \prod_{i=1}^{n} P(X_i \,|\, \mathrm{pa}(X_i)).$$

We use upper case letters, e.g., X_i and Y, to denote variables while sets of variables are denoted using calligraphy letters, e.g., $\mathcal{X}$ and $\mathcal{S}$. If the Bayesian network contains continuous variables that are not discretized, then we assume these to have a Conditional Linear Gaussian distribution.

We let $\mathcal{D} = (c_1, \ldots, c_N)$ denote a data set of N complete cases over variables $\mathcal{X} = \{X_1, \ldots, X_n\}$ and we let $I(X, Y; \mathcal{S})$ denote conditional independence between X and Y given $\mathcal{S}$. When learning the structure of a DAG G from data, we use a test statistic to test the hypothesis $I(X, Y; \mathcal{S})$ using $\mathcal{D}$.

2.2 PC Algorithm

The task of learning the structure of a Bayesian network from $\mathcal{D}$ amounts to determining the structure G. The PC algorithm of [17] is basically:

1. Determine pairwise (conditional) independence $I(X, Y; \mathcal{S})$.
2. Identify skeleton of G.
3. Identify v-structures in G.
4. Identify derived directions in G.
5. Complete orientation of G making it a DAG.

Step 1 is performed such that tests for marginal independence (i.e., $\mathcal{S} = \emptyset$) are performed first followed by conditional independence tests where the size of $\mathcal{S}$ iterates over $1, 2, 3, \ldots$ taking the adjacency of vertices into consideration. That is, in the process of determining the set of conditional independence statements $I(X, Y; \mathcal{S})$, the results produced earlier are exploited to reduce the number of tests. This means, that we stop testing conditional independence of X and Y once a subset $\mathcal{S}$ has been identified such that the independence hypothesis is not rejected. When testing the conditional independence hypothesis $I(X, Y; \mathcal{S})$, the conditioning set $\mathcal{S}$ is restricted, e.g., to contain only potential neighbours of either X or Y, i.e., a variable Z is excluded from $\mathcal{S}$, if the independence test between X (or Y) and Z was previously not rejected. This is referred to as the PC* algorithm by [17], but we will refer to it as the PC algorithm.

Step 2 to Step 5 use the results of Step 1 to determine the DAG G. We will not consider Step 2 to Step 5 further in this paper as experiments demonstrate that the combined time cost of these steps is negligible compared to the time cost of Step 1. The reader is referred to, e.g., [17] for more details.

2.3 Balanced Incomplete Block Designs

[13] describes how BIB designs can be applied to learn the structure of a TAN model from data by parallelization using processes on a distributed memory system. Here, we will use BIB designs to control the process of testing for marginal independence on a shared memory computer using threads.

This section provides the necessary background information on BIB designs to follow the presentation of the method proposed. A design is defined as:

Definition 1 (Design [18]). *A design is a pair $(X, \mathcal{A})$ s. t. the following properties are satisfied:*

1. *X is a set of elements called points, and*
2. *$\mathcal{A}$ is a collection of nonempty subsets of X called blocks.*

In this paper, we only consider cases where each block is a set (and not a multiset). A BIB design is defined as:

Definition 2 (BIB design [18]). *Let v, k and λ be positive integers s. t. $v > k \geq 2$. A (v, k, λ)-BIB design is a design $(X, \mathcal{A})$ s. t. the following properties are satisfied:*

1. *$|X| = v$,*
2. *each block contains exactly k points, and*
3. *every pair of distinct points is contained in exactly λ blocks.*

The number of blocks in a design is denoted by b and r denotes the *replication number*, i.e., how often each point appears in a block. Property 3 in the definition is the *balance* property that we need. We want to test each pair exactly once and therefore require $\lambda = 1$. A BIB design is *symmetric* when the number of blocks equals the number of points. This will not be the case in general.

Example 1. Consider the $(7, 3, 1)$-BIB design. The blocks are (one out of a number of possibilities):

$$\{123\}, \{145\}, \{167\}, \{246\}, \{257\}, \{347\}, \{356\}, \tag{1}$$

where $\{abc\}$ is shorthand notation for $\{a, b, c\}$. This BIB design is symmetric

There is no single method to construct all BIB designs. However, a difference set can be used to generate some symmetric BIB designs.

Definition 3 (Difference Set[18]). *Assume $(G, +)$ is a finite group of order v in which the identity element is 0. Let k and λ be positive integers such that $2 \leq k < v$. A (v, k, λ)-difference set in $(G, +)$ is a subset $D \subseteq G$ that satisfies the following properties:*

1. $|D| = k$,
2. *the multiset $[x+y : x, y \in D, x \neq y]$ contains every element in $G \setminus \{0\}$ exactly λ times.*

In our case, we are restricted to using $(\mathbb{Z}_v, +)$, the integers modulo v. If $D \subseteq \mathbb{Z}_v$ is a difference set in group $(G, +)$, then $D + g = \{x + g | x \in D\}$ is a translate of D for any $g \in G$. The multiset of all v translates of D is denoted $Dev(D)$ and called the development of D [18], page 42. It is important to know that BIB designs do not exist for all possible combinations of v, k, and λ.

The concept of a difference set can be generalized to a *difference family*. A difference family is a set of base blocks. A difference family can be used to generate a BIB design similarly to how difference sets are used to generate BIB designs. Table 1 shows a set of difference families for BIB designs on the form $(q, 6, 1)$, which we will use later.

Table 1. Examples of difference families for a set of $(q, 6, 1)$ BIB designs.

BIB design	Difference family	#(base blocks)	b = $v \cdot$ #(base blocks)
(31,6,1)	$\{(1, 2, 7, 19, 23, 30)\}$	1	31
(91,6,1)	$\{(0, 1, 3, 7, 25, 38),$ $(0, 5, 20, 32, 46, 75),$ $(0, 8, 17, 47, 57, 80)\}$	3	273
(151,6,1)	$\{(1, 32, 118, 7, 73, 71), \ldots\}$	5	755
(211,6,1)	$\{(0, 1, 107, 55, 188, 71), \ldots\}$	7	1477
(271,6,1)	$\{(1, 242, 28, 9, 10, 232), \ldots\}$	9	2439

The base blocks in Table 1 have been generated using SageMath[5]. The value $q = 6$ is chosen for practical reasons as difference families for generating the blocks need to be known to exist and we need to be able to store the corresponding count tables in memory.

[5] http://www.sagemath.org

3 Parallelisation of PC Structure Learning

Improving the performance of the PC algorithm on large data sets can be achieved in a number of ways, see, for instance, [16, 10, 14]. Here we consider an approach where the tests for (conditional) independence are performed in parallel. We use two different approaches based on threads. When testing for marginal independence the set of tests to be performed are known in advance and we use BIB designs to obtain parallization by threads. For the higher order tests we create an edge index array, which the threads iterate over to select the next edge to evaluate for each iteration. The edge index array contains all edges that has not been removed at an earlier step and it is sorted in decreasing order of the test score. Step 1 of the PC algorithm is implemented as three steps:

1. Test all pairs X and Y for marginal independence.
2. Perform the most promising higher-order conditional independence tests.
3. Test of conditional independence $(X, Y; \mathcal{S})$ where $|\mathcal{S}| = 1, 2, 3$.

In [17] bounding the order of the conditional independence relations is suggested as a natural heuristic to reduce the number of tests. Experiments show that by far the most edges are removed for low order tests and statistical tests become increasingly unreliable as the size of the conditioning set increases. For these reasons, the size of the conditioning set is limited to three in the implementation. In Step 3 of the process of testing for conditional independence between X and Y given $\mathcal{S}$, we select $\mathcal{S}$ as a subset of the potential neighbours of X (except Y).

3.1 Test for Marginal Independence

The tests for pairwise marginal independence $I(X, Y; \emptyset)$ for all pairs X, Y should be divided into tasks of equal size such that we test exactly all pairs X, Y for marginal independence. This is achieved using BIB designs on the form $(q, 6, 1)$ where q is at least the number of variables. The blocks of the BIB design are generated using a difference family (e.g., Table 1). Blocks are assigned to threads

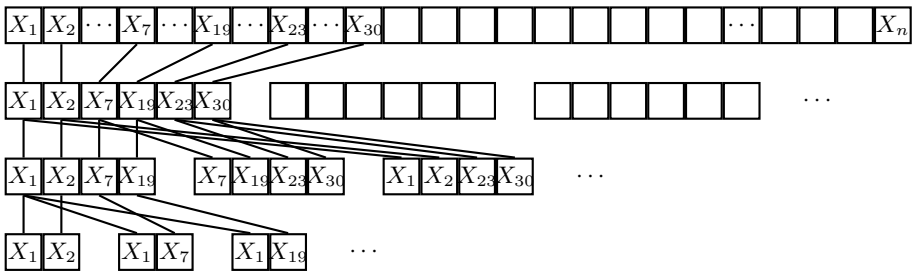


Fig. 1. Example illustrating the use of $(q, 6, 1)$ and $(3, 2, 1)$ designs.

using the unique rank of each thread. A thread with rank r iterates over the

block array and considers only blocks where the array index modulus t equals r where t is the number of threads (the uniqueness means that there is no need for synchronization). When a thread has selected a block, it performs all pairwise independence tests using a $(3, 2, 1)$ BIB design (all pairs) where the 6-block is marginalized to three blocks with four variables each (in this case each point corresponds to two variables). The table of four variables are marginalized down to all pairs for testing where the first pair is ignored producing a total of $\binom{6}{2} = 15$ tests. Figure 1 illustrates this principle. Notice that $q = 6$ represents 15 pairs and we should, in principle, obtain a speed-up of a factor 15 compared to just computing all pairs individually.

3.2 Extra Heuristics

Once the testing for marginal independence is completed, a new step compared to the traditional PC algorithm is performed. This step performs for each edge a set of the most promising tests, i.e., tests with high likelihood of not rejecting the independence hypothesis. At this and the following steps of the conditional independence testing we do not know in advance which tests we are going to perform (since we are using previous results to reduce the number of tests performed).

For each edge (X, Y) the set of *best candidate variables* to include in the conditioning set $\mathcal{S}$ are identified using the weight of a candidate variable Z. The *weight* $w(Z \,|\, (X, Y))$ is equal to the sum of the test scores for (X, Z) and (Y, Z). We create an array of best candidates. This array contains up to five variables, which are all neighbours of X (or Y), i.e., the independence hypothesis has been rejected so far. The main reason for limiting the size to five variables is to make sure that the count table fits in memory. If variables have many states, then the number of candidates is reduced. This array is sorted by the sum of the edge weights. The threads iterate over the sorted edge index array. A thread performs all tests for a selected edge (with the size of $\mathcal{S}$ running from one to three) from the table of up to seven variables. For the table of counts all possible tests are performed generating subsets using the combinatorial number system [12].

The extra heuristics step is responsible for finding a significant number of the independence relations. In combination marginal independence testing and the extra heuristics step usually find by far the highest number of independence relations meaning that higher order tests mainly ensure that no further independence relations can be found. The tests performed for each edge are stored.

3.3 Higher Order Testing

Once testing for marginal independence and the testing based on extra heuristics are completed, the remaining higher order tests for each edge are performed (unless independence has been established at a previous step). The algorithm iterates over $|\mathcal{S}|$ from one to three stopping when an independence hypothesis $I(X, Y; \mathcal{S})$ is not rejected. The threads iterate over the sorted edge index array. Candidate variables to be included in the conditioning set $\mathcal{S}$ are determined as potential neighbours of either X or Y. This list of edges (the candidate and its

potential neighbour X or Y) is sorted as described above and all possible subsets are generated again using the combinatorial number system in order to perform the most promising tests first.

In an iteration, each thread selects an edge and performs all conditional independence test for $|\mathcal{S}| = i$ and writes the result to the edge index array. There is only synchronization on the edge index array when a thread decides which edge to test and when writing to the array as we need to ensure that two threads do not select the same edge to test and that a thread does not try to read the array when another thread is writing its results to the array. This synchronization is also performed in the previous step.

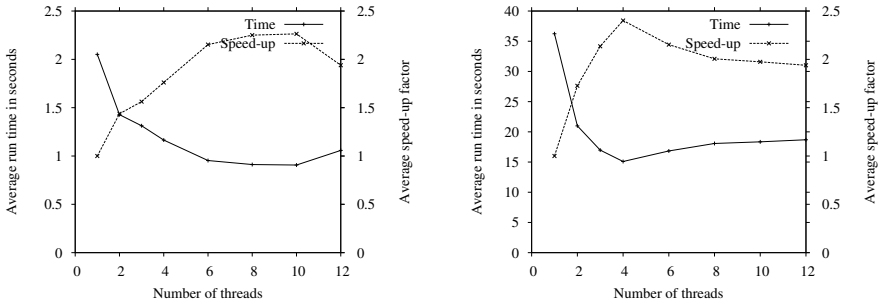
Table 2. Networks from which data sets used in the experiments are generated.

data set	$\|\mathcal{X}\|$	Total CPT size
ship-ship [15]	50	130,478
Munin1 [1]	189	19,466
Diabetes [2]	413	461,069
Munin2 [1]	1,003	83,920
sacso [8]	2,371	44,274

4 Empirical Evaluation

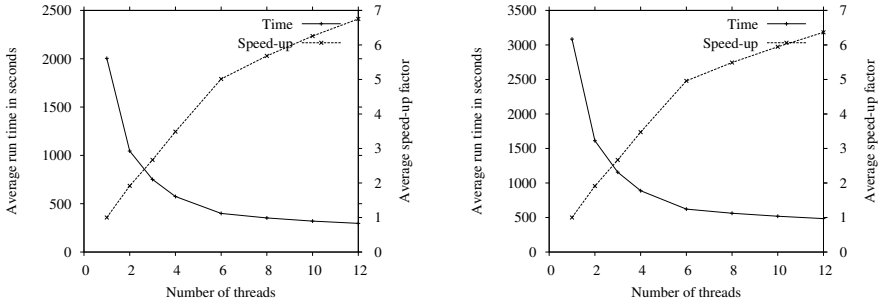
Random samples of data have been generated from the five networks of different size listed in Table 2. Three data sets are generated at random for each network with 100,000, 250,000, and 500,000 cases. All data sets used in the empirical evaluation are complete, i.e., there are no missing values in the data. The empirical evaluation is performed on a Linux computer running Red Hat Enterprise Linux 7 with a six-core Intel (TM) i7-5820K 3.3GHz processor and 64 GB RAM. The computer has six physical cores and twelve logical cores. The parallel PC algorithm is implemented employing a shared memory multicore architecture. All data is loaded into the main shared memory of the computer where the process of the program is responsible for creating a set of POSIX threads to achieve parallelisation. In the experiments, the number of threads used by the program is in the set $\{1, 2, 3, 4, 6, 8, 10, 12\}$ where the case of one thread is considered the baseline and corresponds to a sequential program.

The average computation time is calculated over five runs with the same data set. The computation time is measured as the elapsed (wall-clock) time of the different steps of the parallel PC algorithm. We measure the computation time of the entire algorithm in addition to the time for identifying the skeleton (Step 2), identifying v-structures (Step 3) as well as identifying derived directions (Step 4) and completing the orientation of edges (Step 5) combined.



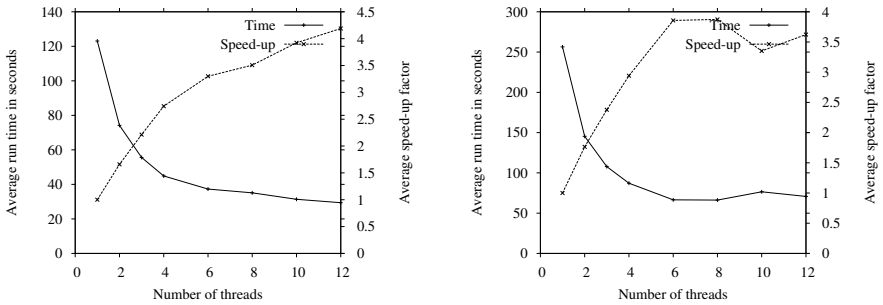
(a) ship-ship 500,000 (b) Munin1 250,000

Fig. 2. Average run times for ship-ship with 500,000 cases and Munin1 250,000 cases.



(a) Diabetes 250,000 (b) Diabetes 500,000

Fig. 3. Average run times for Diabetes with 250,000 and 500,000 cases, respectively.



(a) Munin2 250,000 (b) Munin2 500,000

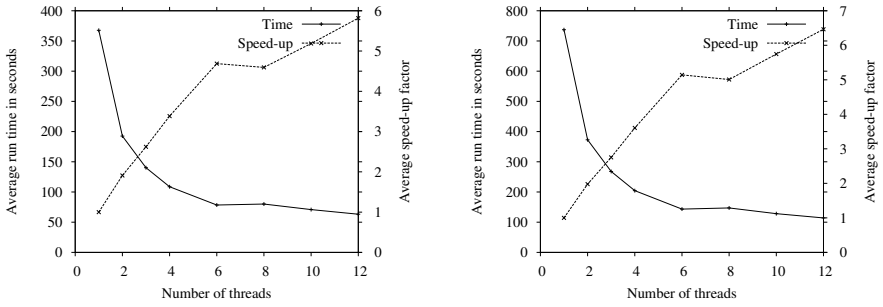
Fig. 4. Average run times for Munin2 with 250,000 and 500,000 cases, respectively.

Figure 2 (left) shows the average run time in seconds (left axis) and speed-up factor (right axis) for ship-ship using 500,000 cases. Notice that the computation time is low for the ship-ship network even with one thread meaning that the potential improvement from parallelization is limited as the evaluation shows. Figure 2 (right) shows the average run time and speed-up factor for Munin1 using 250,000 cases where the speed up deteriorates for six or more threads illustrating the principle of diminishing returns. The additional threads adds overhead to the process and we expect that the increase in time cost is due to the synchronisation on the edge index array.

Figure 3 (left) and Figure 3 (right) show the average run time and speed-up factor for Diabetes using 250,000 and 500,000 cases, respectively. The speed up factor increases smoothly for both 250,000 and 500,000 cases.

Figure 4 (left) and Figure 4 (right) show the average run time and speed-up factor for Munin2 using 250,000 and 500,000 cases, respectively. For 250,000 cases there is a smooth improvement in speed-up whereas for 500,000 cases the speed up factor drops slightly using ten or twelve threads.

Figure 5 (left) and Figure 5 (right) show the average run time and speed-up factor for sacso using 250,000 and 500,000 cases, respectively. The experiment on sacso using 500,000 cases is the task with the highest number of variables and cases considered in the evaluation. This task produces an average speed-up of a factor 6.46 with average run time dropping from 737 to 114 seconds. The experiment on Diabetes using 500,000 cases is the task taking the longest time to complete. This task produces an average speed-up of a factor 6.36 with average run time dropping from 3084.65 to 484.65 seconds. Table 3 shows the



(a) sacso 250,000 (b) sacso 500,000

Fig. 5. Average run times for sacso with 250,000 and 500,000 cases, respectively.

average time cost of identifying the skeleton (Step 2), identifying the v-structures (Step 4), identifying derived directions (Step 4) and completing the orientation to obtain a DAG (Step 5). It is clear from Table 3 that the costs of Step 2 to Step 5 are negligible compared to the total cost.

Table 3. Average run times in seconds for Step 2 to Step 5.

Data set	Skeleton (Step 2)	v-structures (Step 3)	Orientation (Steps 4 and 5)
ship-ship	0	0	0
Munin1	0.005	0	0.001
Diabetes	0.001	0.004	0.002
Munin2	0.006	0.002	0.034
sacso	0.051	5.692	0.502

5 Discussion

This paper introduces a new approach to parallelisation of learning the structure of a Bayesian network from data using the PC algorithm. The approach is based on the use of threads with all data cases stored in shared memory. The PC algorithm consists of five main steps where the focus of this paper has been on performing the independence tests in parallel as the results in Sect. 4 clearly demonstrate that the total time cost of Step 2 to Step 5 are negligible compared to the time cost of Step 1.

Step 1 of the PC algorithm consists, as presented in this paper, of three steps. In the first step the tests for marginal independence are performed. Parallelization of this step is based on the use of difference sets and families where the tests to be performed are known in advance as all pairs are to be tested for marginal independence. In the second step a set of the most promising higher order tests are performed whereas in the third step tests for conditional independence are performed using conditioning sets of size one, two and three, respectively.

The edge index array is the central bottleneck of the approach as it is the only element that requires synchronization. Synchronization is limited to selecting which edge to test and does not include synchronization of the counting and testing. The counting usually being the most time consuming element of testing for conditional pairwise independence.

The PC algorithm is known to be sensitive to the order in which the conditional independence tests are performed. This means that the number of threads used by the algorithm may impact the result as the order of tests is not invariant under the number of threads used. This is a topic of future research.

The results of the empirical evaluation of the proposed method on a Linux server with six physical cores and twelve logical cores show a significant time performance improvement over the pure sequential method. For most cases considered there is a point where using additional threads does not improve performance illustrating the principle of diminishing returns. In a few cases, where the number of variables is low, the number of cases is low, or both increasing the number of threads used may increase time costs.

There is some variance in the run time measured. This should also be expected as the evaluation is performed on systems serving other users, i.e., the experiments have not been performed on an isolated system.

References

1. S. Andreassen, F. V. Jensen, S. K. Andersen, B. Falck, U. Kjærulff, M. Woldbye, A. R. Sørensen, A. Rosenfalck, and F. Jensen. MUNIN — an expert EMG assistant. In *Computer-Aided Electromyography and Expert Systems*, chapter 21. Elsevier Science, 1989.
2. Steen Andreassen, Roman Hovorka, Jonathan Benn, Kristian G. Olesen, and Ewart R. Carson. A model-based approach to insulin adjustment. In *Proc. of the Third Conference on Artificial Intelligence in Medicine*, pages 239–248, 1991.
3. A. Basak, I. Brinster, X. Ma, and O.J. Mengshoel. Accelerating Bayesian network parameter learning using Hadoop and MapReduce. In *Proceedings of the 1st International Workshop on Big Data, Streams a nd Heterogeneous Source Mining: Algorithms, Systems, Programming Models and Applications*, pages 101–108, 2012.
4. W. Chen, L. Zong, W. Huang, G. Ou, Y. Wang, and D. Yang. An empirical study of massively parallel bayesian networks learning for sentiment extraction from unstructured text. In *Web Technologies and Applications*, pages 424–435. Springer, 2011.
5. C.-T. Chu, S.K. Kim, Y.-A. Lin, Y. Yu, G. Bradski, A.Y. Ng, and K. Olukotun. Map-reduce for machine learning on multicore. In *NIPS*, pages 281–288, 2006.
6. M. de Jongh. *Algorithms for Constraint-Based Learning of Bayesian Network Structures With Large Numbers of Variables*. PhD thesis, Uniof Pittsburgh, 2014.
7. Q. Fang, K. Yue, X. Fu, H. Wu, and W. Liu. A MapReduce-Based Method for Learning Bayesian Network from Massive Data. In *Web Technologies and Applications*, volume 7808 of *Lecture Notes in Computer Science*, pages 697–708. 2013.
8. F. V. Jensen, C. Skaanning, and U. Kjærulff. The SACSO System for Troubleshooting of Printing Systems. In *In Proceedings of the Seventh Scandinavian Conference on Artificial Intelligence*, 2001.
9. F.V. Jensen and T.D. Nielsen. *Bayesian Networks and Decision Graphs*. Springer, 2nd edition, 2007.
10. M. Kalisch and P. Buhlmann. Estimating high-dimensional directed acyclic graphs with the pc-algorithm. *Journal of Machine Learning Research*, 8:613–636, 2008.
11. U.B. Kjærulff and A.L. Madsen. *Bayesian Networks and Influence Diagrams: A Guide to Construction and Analysis*. Springer, 2nd edition, 2013.
12. D. E. Knuth. *The Art of Computer Programming, 4, Fascicle 3*. Addison-Wesley, 2005.
13. A. L. Madsen, F. Jensen, A. Salmeron, M. Karlsen, H. Langseth, and T.D. Nielsen. A new method for vertical parallelisation of tan learning based on balanced incomplete block designs. In *Proc. of PGM*, pages 302–317, 2014.
14. Olga Nikolova and Srinivas Aluru. Parallel discovery of direct causal relations and markov boundaries with applications to gene networks. In *Parallel Processing (ICPP), 2011 International Conference IEEE*, pages 512–521, 2011.
15. A. Papanikolaou. *Presents modern risk-based methods and applications to ship design, operation, and regulations*. Springer, 2009.
16. M. Scutari. Learning Bayesian Networks with the bnlearn R Package. *Journal of Statistical Software*, 35(3):1–22, 2010.
17. P. Spirtes, C. Glymour, and R. Scheines. *Causation, Prediction, and Search*. Adaptive Computation and Machine Learning. MIT Press, second edition, 2000.
18. D. Stinson. *Combinatorial Designs — Constructions and Analysis*. Springer, 2003.