A parallel algorithm for Bayesian network structure learning from large data sets

Madsen, Anders Læsø; Jensen, Frank; Salmerón, Antonio; Langseth, Helge; Nielsen, Thomas Dyhre

[Link to publication from Aalborg University](#)

A parallel algorithm for Bayesian network structure learning from large data sets

Anders L. Madsen [a,b,*], Frank Jensen [a], Antonio Salmerón [d], Helge Langseth [c], Thomas D. Nielsen [b]

[a] *HUGIN EXPERT A/S, DK-9000 Aalborg, Denmark*
[b] *Aalborg University, DK-9220 Aalborg, Denmark*
[c] *Norwegian University of Science and Technology, NO-7491 Trondheim, Norway*
[d] *University of Almería, ES-04120 Almería, Spain*

ABSTRACT

This paper considers a parallel algorithm for Bayesian network structure learning from large data sets. The parallel algorithm is a variant of the well known PC algorithm. 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 algorithm is also compared empirically with a process-based approach where each process manages a subset of the data over all the variables on the Bayesian network. The results demonstrate that significant time performance improvements are possible using both approaches.

1. Introduction

A *Bayesian network* (BN) [1–5] 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 a variant of the PC algorithm [6] exploiting parallel computations.

Large data sets both in terms of the number 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 the performance of learning algorithms. A number of different approaches to parallel structure learning have been considered in the literature. In [7] the authors describe a MapReduce-based method for learning Bayesian networks from massive data using a search & score algorithm while [8] describes a MapReduce-based method for machine learning on multi-core computers. Also, [9] presents the R package **bnlearn** which provides implementations of some structure learning algorithms including support for parallel computing. [10] introduces a method for accelerating Bayesian network parameter learning using Hadoop and MapReduce. Other relevant work on parallelization of learning Bayesian networks from data include [11–15].

In this paper, we consider two different approaches to parallelization of the PC algorithm. First, we describe a new 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 in [16] on vertical parallelization of TAN learning using Balanced Incomplete Block (BIB) designs [17]. Second, we consider

---

* Corresponding author.
E-mail addresses: alm@hugin.com (A.L. Madsen), fj@hugin.com (F. Jensen), antonio.salmeron@ual.es (A. Salmerón), helgel@idi.ntnu.no (H. Langseth), tdn@cs.aau.dk (T.D. Nielsen).

an embarrassingly parallel version of the PC algorithm. This approach uses processes where each process manages a subset of the data over all variables. In order to distinguish between the two approaches, the latter approach is referred to as the *horizontal PC algorithm*. The horizontal PC algorithm is developed for distributed memory concurrent computers using the standardized and portable message-passing system referred to as the *Message Passing Interface* (MPI) [18]. The horizontal PC algorithm also takes advantage of BIB designs to improve efficiency. The results of an empirical evaluation show a significant improvement in time performance over a purely sequential implementation for both approaches.

This paper is organized as follows. Section 2 presents preliminaries and notation, including an introduction to BIB designs and the PC algorithm. Section 3 describes the details of both methods for parallel structure learning while Section 4 presents the results of an empirical evaluation of the algorithms on both real-world Bayesian networks and examples from literature. Finally, Section 5 gives a discussion of the results and Section 6 conclusions.

2. Material and methods

Let $\mathcal{X} = \{X_1, \ldots, X_n\}$ be a set of random variables such that dom(X) is the state space of X when X is discrete. The state space size is $||X|| = |\text{dom}(X)|$. A BN $\mathcal{N} = (\mathcal{X}, G, \mathcal{P})$ over the set $\mathcal{X}$ consists of an acyclic directed graph (DAG) $G = (V, E)$ with vertices V and edges E and a set of CPDs $\mathcal{P} = \{P(X \mid \text{pa}(X)) : X \in \mathcal{X}\}$, where pa($X$) denotes the parents of X in G. The BN $\mathcal{N}$ specifies a joint probability distribution over $\mathcal{X}$:

$$P(\mathcal{X}) = \prod_{i=1}^{n} P(X_i \mid \text{pa}(X_i)).$$

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 $\mathcal{S}$. In this paper, we only consider discrete variables.

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 $\mathcal{D}$, we use a test statistic to test the hypothesis $I(X, Y; \mathcal{S})$ based on counts in $\mathcal{D}$. That is, to test the conditional independence hypothesis $I(X, Y; \mathcal{S})$ between two discrete variables X and Y conditional on $\mathcal{S}$ based on counts in $\mathcal{D}$, we use the test statistic $G^2 = \sum_{\mathcal{S}=s} G_s^2$ where

$$G_s^2 = 2 \sum_{x,y} O_{xy|s} \log \frac{O_{xy|s}}{E_{xy|s}}, \tag{1}$$

where $O_{xy|s}$ is the observed count for x and y given s and $E_{xy|s}$ is the expected count for x and y given s under the null-hypothesis.

2.1. 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 [6] consists of five steps:

1. Determine pairwise (conditional) independence $I(X, Y; \mathcal{S})$.
2. Identify the 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 to contain only potential neighbors of either X or Y, i.e., a variable Z is excluded from $\mathcal{S}$, if the independence hypothesis between X (or Y) and Z was previously not rejected. This is referred to as the PC* algorithm by [6], but we will refer to it as the PC algorithm.

Steps 2–5 use the results of Step 1 to determine the DAG G. We will not consider Step 2–5 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. This is clearly demonstrated in the empirical evaluation. The interested reader is referred to, e.g., [6] for more details.

Hence, our proposal for scaling up the PC algorithm is based on parallelizing Step 1, which involve the calculation of the G^2 score (see Eq. (1)) between each pair of variables. An immediate approach for scaling up the algorithm could be to simply generate one computing thread for each pair of variables and then process the threads in parallel. However, with n variables this approach would require accessing the underlying database $\binom{n}{2}$ times, inducing a significant overhead in terms of disk/network access. Alternatively, one might group the variables in blocks so that each block only accesses the data a single time in order to calculate the sufficient statistics required for computing the G^2 score for all pairs of variables within the block. A key issue here is finding an appropriate block size and at the same time ensuring that the blocks, in combination, guarantee that all pairs of variables are considered exactly once.

To get an intuitive understanding of this process we can as an analogy consider the organization of the Speedway World Championship (SWC). After the initial pre-qualifying rounds for the SWC, the remaining 16 highest ranked riders should be compared to each other to obtain a final ranking of the riders. One approach to achieve this would be to pair-up the riders so that each rider will participate in 15 races, yielding a total of 120 rounds with two riders competing in each round. This setup would put a strain on the riders and not use the full capacity of the speedway track, which is designed to accommodate four riders simultaneously. Instead, the SWC employs a heat-system ensuring that each of the 16 riders will meet each of the other riders at some time during the competition. Specifically, the heat-system consists of 20 heats with four riders in a heat. Each rider participates in only five heats, and within a single heat all riders compete jointly, thereby meeting each other. After completing the 20 heats, all pairs of riders will have met exactly once. This can also be seen by labeling the riders $\{0, \ldots, 15\}$ and constructing these heats: $H_1 = \{3, 6, 12, 15\}$, $H_2 = \{4, 5, 10, 13\}$, $H_3 = \{0, 4, 6, 7\}$, $H_4 = \{0, 10, 11, 15\}$, $H_5 = \{7, 10, 12, 14\}$, $H_6 = \{0, 8, 9, 14\}$, $H_7 = \{0, 1, 3, 13\}$, $H_8 = \{1, 6, 8, 10\}$, $H_9 = \{7, 9, 13, 15\}$, $H_{10} = \{1, 5, 14, 15\}$, $H_{11} = \{8, 11, 12, 13\}$, $H_{12} = \{5, 6, 9, 11\}$, $H_{13} = \{1, 4, 9, 12\}$, $H_{14} = \{3, 5, 7, 8\}$, $H_{15} = \{3, 4, 11, 14\}$, $H_{16} = \{2, 6, 13, 14\}$, $H_{17} = \{1, 2, 7, 11\}$, $H_{18} = \{0, 2, 5, 12\}$, $H_{19} = \{2, 4, 8, 15\}$, and $H_{20} = \{2, 3, 9, 10\}$.

When it comes to computing the G^2 scores, the 16 riders correspond to variables and each heat represents a block consisting of four variables to be pairwise compared. Thus, rather than handling pairs of variables independently and having to make data access $\binom{16}{2} = 120$ times, we can instead make 20 blocks/heats of four variables each and thereby only having to access the full dataset 20 times. Note that with the particular setup above, we are guaranteed not to make redundant calculations as the G^2 score is computed exactly once for each pair X_i, X_j, $1 \leq i, j \leq n$.

This approach of distributing variables/riders into blocks/heats is an instance of a so-called *balanced incomplete block (BIB) design*; in fact the heat-system configuration employed by the Speedway World Championship corresponds to a (16, 4, 1)-BIB design (see Definition 2).

2.2. Balanced incomplete block designs

The use of block designs dates back to the statistical theory of design of experiments [19], motivated in its origin by agricultural experiments. In this context the goal was to compare the yield of different plant varieties, considering that the yield could be significantly affected by the environment, i.e., the conditions under which the plants are grown. The idea was to compensate for the effect of the environment by setting up blocks of land small enough to assume uniform environmental conditions inside a block, and distribute the plant varieties among them. With space limitations inside each block, one may not be able to fit sufficient replications of all plant varieties inside a single block, and therefore rather required that each pair of plant varieties would be allocated at least once to the same block to facilitate a fair comparison between them. The relation to both the SWC and our calculation of the G^2 scores is evident.

BIB designs [17] can be applied to efficiently divide the statistical tests for independence among a set of, for instance, threads or processes. In particular, [16] 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. In this paper, we will use BIB designs to control the process of testing for marginal independence on a shared memory computer using threads and on a distributed memory system using processes.

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

Definition 1 (Design [17])**.** 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 non-empty subsets of X called *blocks*.

In this paper, we only exploit cases where each block is a set (and not a multiset, i.e., we do not allow multiple instances of the same element in the set). Nevertheless, some definitions will consider multi-sets. A BIB design is defined as:

Definition 2 (BIB design [17])**.** 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 will exploit. In Step 1 of the PC algorithm, we want to test each pair of variables for marginal independence 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):

$$\{0, 1, 2\}, \{0, 3, 4\}, \{0, 5, 6\}, \{1, 3, 5\}, \{1, 4, 6\}, \{2, 3, 6\}, \{2, 4, 5\}.$$
(2)

This BIB design is symmetric as $b = v$.

There is no single efficient method to construct all BIB designs. First, it is important to know that they do not exist for all combinations of v, k, and λ. Second, the problem of finding a BIB design is NP-complete [20]. To efficiently utilize them we have therefore pre-calculated a number of BIB designs, and utilize those at run-time. Instead of storing the full designs, it is sufficient to store *difference sets* that can be used to generate some symmetric BIB designs:

Definition 3 (Difference Set [17])**.** 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 [17, page 42].

Theorem 1 ([17], Theorem 3.8 p. 43)**.** *Let D be a (v, k, λ)-difference set in an Abelian group $(G, +)$. Then $(G, Dev(D))$ is a symmetric (v, k, λ)-BIB design.*

Example 2. The set $D = \{0, 1, 3\}$ is a (7, 3, 1)-difference set in $(\mathbb{Z}_7, +)$. The blocks constructed by iteratively adding one to each element of D (modulo 7) are:

$$\{0, 1, 3\}, \{1, 2, 4\}, \{2, 3, 5\}, \{3, 4, 6\}, \{4, 5, 0\}, \{5, 6, 1\}, \{6, 0, 2\}.$$

Notice that the *i*th element of each block is unique across all blocks. This property will be used to assign blocks to threads in Section 3. This was not the case for the blocks presented in Example 1.

The concept of a difference set can be generalized to the concept of 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. Table 1 shows a set of difference families for BIB designs on the form $(q, 6, 1)$, which we will use later. Base blocks for generating BIB-designs are tabulated, e.g., [21], but can also be found computationally. The base blocks in Table 1 have been generated using SageMath[1]. The value $k = 6$ is chosen for practical reasons: First, difference families for generating the blocks need to be known to exist; second, we need to be able to store the count tables representing the joint distribution of the variables in a block in memory, required to compute the G^2 scores. The main idea for parallelization considered in this paper is to use the $(q, 6, 1)$ design to distribute the computations of the scores over a set of computing units such that each score is computed exactly once from a smaller intermediate table over six variables.

3. Theory

There are two obvious approaches to parallelize the testing step of the PC algorithm. One approach is to assign the same number of cases to each thread. For a specific statistical test, each thread would then be responsible for computing the necessary counts over its data. The counts from all threads are combined and used to perform the statistical test. We refer to this as *horizontal* parallelization. This approach is *embarrassingly* parallel, i.e., it requires little effort to separate the problem into a number of parallel tasks.

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

BIB design	Difference family	#(base blocks)	b = q · #(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), ...}	5	755
(211,6,1)	{(0, 1, 107, 55, 188, 71), ...}	7	1477
(271,6,1)	{(1, 242, 28, 9, 10, 232), ...}	9	2439

Horizontal parallelization mainly addresses learning from data sets, where N is large, i.e., many cases. Another approach is referred to as vertical parallelization as used by [16] for parallelization of TAN learning. In vertical parallelization, processes read all data for a subset of variables and the pairwise conditional independence tests between a pair of features conditional on the target variable are distributed using BIB designs. Vertical parallelization mainly addresses learning from data sets where $|\mathcal{X}|$ is large, i.e., many variables. Each process reads all data over the variables assigned to it.

Improving the performance of the PC algorithm on large data sets can be achieved in a number of ways, see, for instance, [9,11,13]. We consider one approach where the counting of sufficient statistics for a specific conditional independence test is performed in parallel and an approach where the tests for (conditional) independence are performed in parallel.

For the case where we use threads to perform tests in parallel, two different approaches are considered. When testing for marginal independence the set of tests to be performed are known in advance and we use BIB designs to obtain parallelization. For the higher order tests we do not know which tests to perform as this depends on the results of previous tests. Therefore, 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 have not been removed at an earlier step and it is sorted in decreasing order of the test score as explained below. 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 for conditional independence $(X, Y; \mathcal{S})$ where $|\mathcal{S}| = 1, 2, 3$.

In [6] 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). Step 2 is explained in more detail below. This implementation of the PC algorithm was described in [22], which also reports on an empirical evaluation of its performance.

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 of the form $(q, 6, 1)$ where q is at least the number of variables. That is, q is selected as the smallest value larger than the number of variables such that a $(q, 6, 1)$-BIB design is known to exist. This means that some points will not represent

any variable and tests involving points not representing a variable are not performed. The blocks of the BIB design are generated using a difference family (e.g., Table 1). Each block is used to compute the marginal counts of the variables represented in the block. If all the variables have the same state space size, then the count tables will be of equal size.

The computation of the G^2 scores is parallelized assigning blocks to threads as each thread can compute the scores corresponding to a block in parallel with other threads. Blocks are assigned to threads 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 where the 6-block is reduced to three blocks with four variables each (in this case each point corresponds to two variables). The operation of reducing a count table to a lower dimension by adding the counts for a specific configuration of the remaining variables is referred to as marginalization. The table of four variables is marginalized down to all pairs for testing where the first pair is ignored producing a total of $\binom{6}{2} = 15$ tests.

Fig. 1 illustrates this principle, assuming an example with $q = 31$ variables labelled as $X_0, \ldots, X_{30}$. The first block (second row in the figure) is $\{X_1, X_2, X_7, X_{19}, X_{23}, X_{30}\}$, corresponding to the difference family for design $(31, 6, 1)$, as given in Table 1. The second block would be obtained by adding 1 to the index of the variable in each coordinate, modulo 31, i.e. $\{X_2, X_3, X_8, X_{20}, X_{24}, X_0\}$. According to the same procedure, the third block would be $\{X_3, X_4, X_9, X_{21}, X_{25}, X_1\}$ and so on.

Taking the first block, we form three pairs of variables, $P_1 = \{X_1, X_2\}$, $P_2 = \{X_7, X_{19}\}$ and $P_3 = \{X_{23}, X_{30}\}$ and compute the blocks of a $(3, 2, 1)$-BIB design, where each block has two pairs. These blocks are actually all the possible pairings of P_1, P_2 and P_3, namely $\{P_1, P_2\}$, $\{P_2, P_3\}$ and $\{P_3, P_1\}$, placed on the third row of Fig. 1. It can be seen that every three pairings we come up with $5 \times 3 = 15$ pairs of features for which the G^2 score is computed. In fact, each block corresponding to a pairing $\{P_i, P_j\}$ yields 6 pairs of variables, but the first one is discarded in order to avoid repetitions. In Fig. 1 it is indicated by marking both variables in red on the lower row.

Notice that $k = 6$ represents 15 pairs and the number of times we count is reduced by a factor of 15, but each count is a factor three more expensive (as we are counting six variables instead of two variables). In addition, there is the task of marginalizing the count tables to pairs. If the number of states for some variables is high, then it may be more efficient to compute the score directly from the data set instead of creating an intermediate table.

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 a set of the most promising tests for each edge,
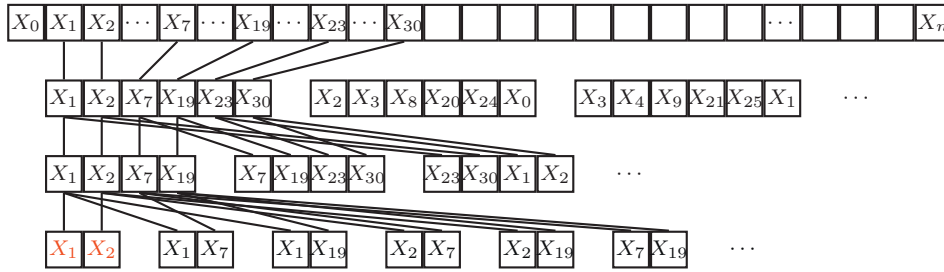
Fig. 1. Example illustrating the use of $(q, 6, 1)$ and $(3, 2, 1)$ designs.
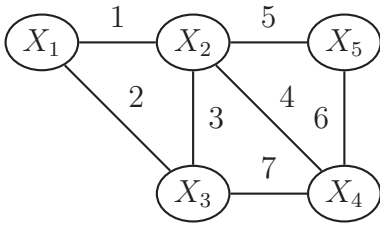

Fig. 2. Example illustrating the use of the heuristic weights.

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 need 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). The idea is to condition on candidate variables that have a strongest association with both X and Y.

We create an array of best candidates. This array contains up to five variables, which are all neighbours of X (or Y) in the current graph. The main reason for limiting the number of candidate variables to five is to make sure that the count table fits in memory. If variables have many states, then the number of candidates is reduced as follows. First, the combined state space size of X and Y is computed. Next, candidate variables are selected until the combined state space size reaches the number of cases in the data set or all five candidates are selected. The objective is to perform as many tests where the null hypothesis is not rejected as quickly as possible. There is a balance between increasing the number of candidate variables and the time and space required to perform the tests. Since the size of the count table increases exponentially with the number of candidate variables included, there is an upper limit on the number of candidate variables. The limit of five candidate variables has been set based on experience with simple tests. 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 by marginalising down to the appropriate number of variables. From the table of counts all possible tests are performed generating subsets using the combinatorial number system [23] as we want to generate the most promising subsets first.

Example 3 (Candidates). Assume Fig. 2 shows the graph after completing the marginal independence tests where the score for marginal independence is shown above each edge and assume all other scores are zero.

The edge with the highest score is (X_3, X_4) and it is the first edge in the edge index array. For the edge (X_3, X_4), variable X_2 is the only candidate variable with weight $w(X_2 | (X_3, X_4)) = 3 + 4$. This means that a table over X_2, X_3, X_4 is created. From this table the three conditional independence tests $I(X_2, X_3|X_4)$, $I(X_2, X_4|X_3)$, and $I(X_3, X_4|X_2)$ are performed by one thread.

The three tests performed based on edge (X_3, X_4) may lead to removal of up to three edges (in the case the null hypothesis is not rejected for any of the tests). The aim of sorting the edges and selecting candidate variables based on a score is to remove edges from the graph as quickly as possible in order to reduce the number of later tests.

Assuming independence assumptions are rejected for the tests associated with (X_3, X_4), (X_4, X_5), and (X_2, X_5), we reach edge (X_2, X_4) which has two candidates X_3 and X_5 with weights $w(X_3 | (X_2, X_4)) = 3 + 7 = 10$ and $w(X_5 | (X_2, X_4)) = 5 + 6 = 11$. If the number of candidate variables is limited to one, then only X_5 is considered producing the count table over X_2, X_4, X_5. Using an upper limit of five candidates (and assuming their joint state space is less than the number of cases), the count table over X_2, X_3, X_4, X_5 is created. From this we can perform a total of seven conditional independence tests.

The extra heuristics step is responsible for finding a significant number of the independence relations. In combination, the step testing for marginal independence and the step performing the most promising higher-order independence tests based on heuristics 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. This also suggests putting an upper limit on the size of the conditioning set. The tests performed for each edge are stored.

3.3. Higher order independence testing

Once testing for marginal independence and the testing based on 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. The 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, i.e., a heuristic is used to identify the conditional independence test where the independence hypothesis is least likely to be rejected.

In an iteration, each thread selects an edge and performs all conditional independence test for $|\mathcal{S}| = i$ and writes the results to the edge index array. There is only synchronization on the edge index array when a thread decides which edge to test and when

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

Data set	$|\mathcal{X}|$	$|E|$	Total CPT size
Ship-Ship [24]	50	75	130,478
Munin1 [25]	189	282	19,466
Diabetes [26]	413	602	461,069
Munin2 [25]	1003	1244	83,920
SACSO [27]	2371	3521	44,274

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 results from the edge index array when another thread is writing its results to the array. This synchronization is also performed in the previous step.

3.4. Horizontal parallel PC

The horizontal parallel PC algorithm is designed for a distributed memory architecture. The basic idea of the horizontal parallel PC algorithm is to divide the data set $\mathcal{D}$ into subsets such that each process manages a proper subset of the cases over all variables in the data. That is, given a data set $\mathcal{D} = \{c_1, \ldots, c_N\}$ and p processes, the data $\mathcal{D}$ is divided into p disjoint subsets $\mathcal{D}_1, \ldots, \mathcal{D}_p$ of (approximately) equal size such that $\bigcup_i \mathcal{D}_i = \mathcal{D}$.

The structure learning process is controlled by a *master* process m, which is responsible for creating a set of p *worker* processes. The process m performs all steps of the PC algorithm as described in Section 2.1, whereas the computation of the required sufficient statistics to perform the conditional independence testing in Step 1 is divided among the p worker processes. That is, each time a test for (conditional) independence $I(X, Y; \mathcal{S})$ is to be performed the process m asks each process p to compute and return the marginal count table over $X, Y, \mathcal{S}$ computed from the data set $\mathcal{D}_p$. When count tables over subsets of variables are communicated, all possible tests are performed from these count tables. That is, if a table over, for instance, X_1, X_2, X_3 is communicated, then all tests for marginal independence and conditional independence on a single variable are performed from the table over X_1, X_2, X_3.

When data is complete, it is possible to exploit BIB designs to further improve the efficiency of the testing for marginal independence. BIB designs are used in the same way as described in Section 3.1. That is, when data is complete we use a $(q, 6, 1)$-BIB design to speed up the testing for marginal independence. The benefit is twofold; we reduce the number of times each worker process has to make a parse over the data and we reduce the number of times the master process has to communicate with each worker process. On the other hand, we are in some cases increasing the amount of data transmitted for each communication. We will evaluate the impact of using BIB designs in horizontal parallel PC algorithm.

This approach is most naturally used for learning tasks where the number of cases is large. Thus, the implementation used in the experimental analysis is based on the use of processes.

4. Results

Random samples of data were generated from the five networks of different sizes listed in Table 2. Three data sets are generated at random for each network with 100,000, 250,000, and 500,000 cases. All generated data sets used are complete, i.e., there are no missing values in the data. In cases where data is not complete it is not possible to use BIB designs to the full extent described above. Therefore, we consider an example where data is made incomplete by adding an empty case to the data.

The empirical evaluation is performed on a desktop computer named Odin and a computer cluster named Fyrkat. Odin runs Red Hat Enterprise Linux 7 with a six-core Intel (TM) i7-5820K 3.3 GHz processor and has 64 GB RAM. Odin has six physical and twelve logical cores. Fyrkat is a computer cluster where each worker node used has two Intel Xeon (TM) X5260 processors and 16 GB RAM. It has a total of 80 such nodes. This cluster system uses SLURM (Simple Linux Utility for Resource Management) for resource management. Odin is used to evaluate both approaches on shared memory while Fyrkat is used to evaluate the horizontal parallel PC on distributed memory. All test programs are implemented using the C programming language and HUGIN API version 8.3. On Odin parallelization is achieved using POSIX threads and on Fyrkat parallelization is achieved using MPI.

4.1. Parallel PC

The parallel PC algorithm is implemented employing a shared memory multi-core 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 parallelization. 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.

Fig. 3 (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. Fig. 3 (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 add overhead to the process and we expect that the increase in time cost is due to the synchronization on the edge index array.

Fig. 4 (left) and (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.

Fig. 5 (left) and (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.

Fig. 6 (left) and (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 s. 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 s.
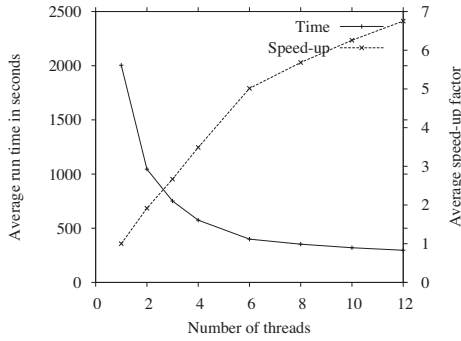
Step 1 of the PC algorithm consists of marginal independence tests, extra heuristics and higher order conditional independence tests. Fig. 7 shows the time costs for the marginal independence tests and extra heuristics.
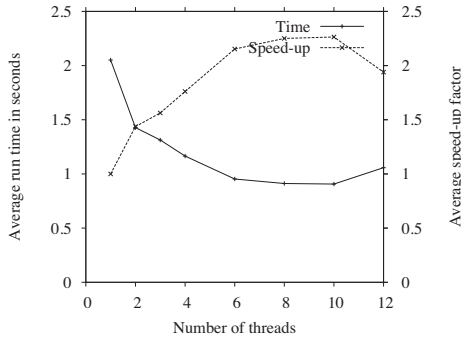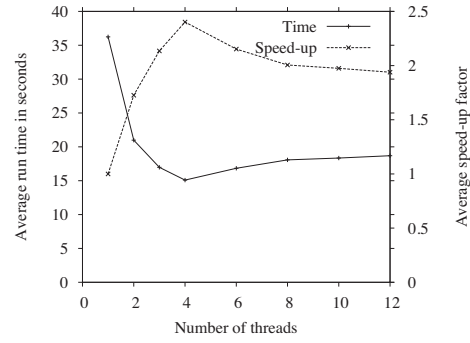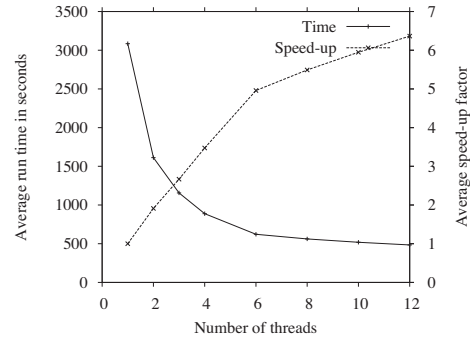
(a) Ship-Ship 500,000   (b) Munin1 250,000
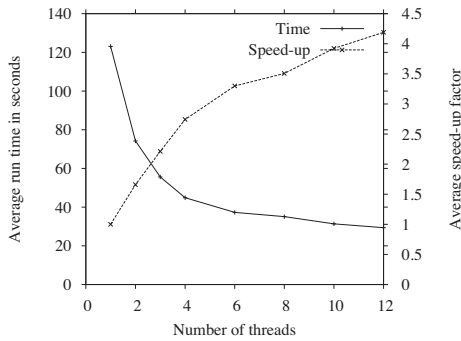
Fig. 3. Average run times for Ship-Ship with 500,000 cases and Munin1 250,000 cases.
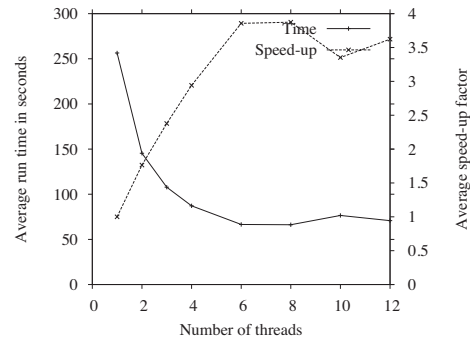

(a) Diabetes 250,000   (b) Diabetes 500,000

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

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

Fig. 5. Average run times for Munin2 with 250,000 and 500,000 cases, respectively.

Figs. 8 and 9 show the time costs for higher order tests for each size of the conditioning set. It is clear from Figs. 7–9 that the most time consuming step is the marginal independence tests where a large number of edges are excluded from the graph.

Table 3 shows the average time cost of identifying the skeleton (Step 2), identifying the *v*-structures (Step 3) and identifying derived directions as well as completing the orientation to obtain a DAG (Step 4 and Step 5).
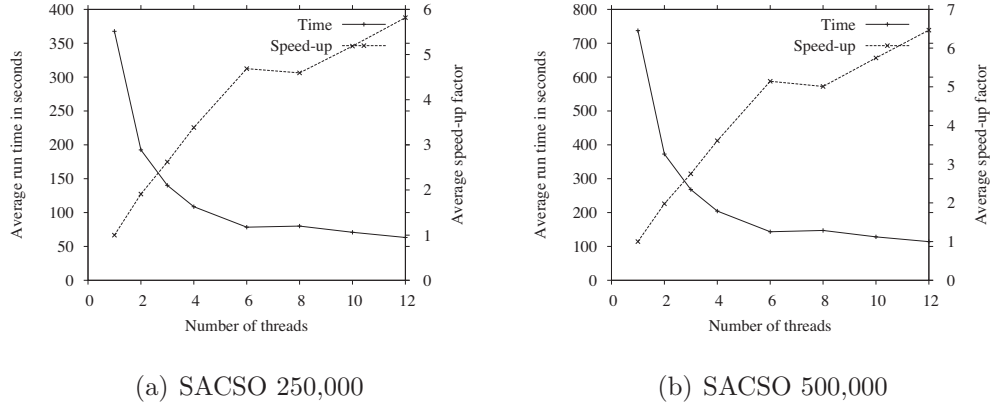
It is clear from Table 3 that the costs of Step 2–5 are negligible compared to the total cost.
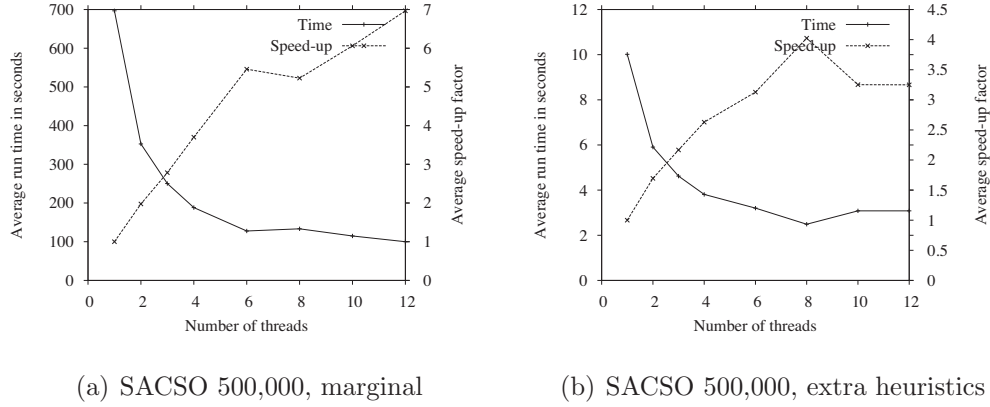
Table 3
Average run times in seconds for Steps 2–5.

Data set	Skeleton (Step 2)	*v*-structures (Step 3)	Orientation (Step 4 & 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

(a) SACSO 250,000

(b) SACSO 500,000

Fig. 6. Average run times for SACSO with 250,000 and 500,000 cases, respectively.

(a) SACSO 500,000, marginal

(b) SACSO 500,000, extra heuristics

Fig. 7. Average run times for SACSO with 500,000 cases for marginal independence testing and extra heuristics, respectively.

(a) SACSO 500,000, $|\mathcal{S}| = 1$

(b) SACSO 500,000, $|\mathcal{S}| = 2$
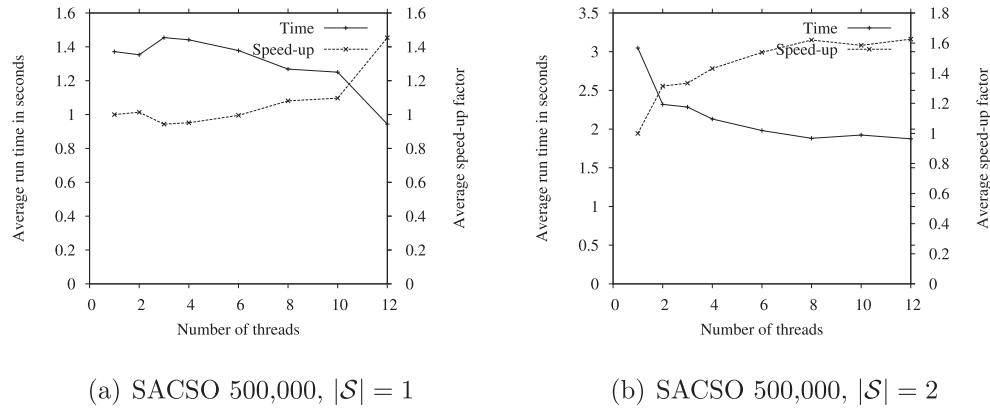
Fig. 8. Average run times for SACSO with 500,000 cases for higher order tests using $|\mathcal{S}| = 1$ and $|\mathcal{S}| = 2$, respectively.

4.2. Horizontal parallel PC

The horizontal parallel PC algorithm is implemented employing a distributed memory multi-processor architecture. The implementation is based on MPI where a master process is responsible for performing all steps of the PC algorithm using a set of worker processes to compute sufficient statistics for subsets of the data in parallel. The communication between the master and worker processes is performed using MPI. In the experimental evaluation of the horizontal parallel PC algorithm, we will consider the effect of using $(q, 6, 1)$-BIB designs to improve performance. BIB designs can only be used for the set of variables with complete data. Thus, in order to evaluate the impact of BIB designs on performance, we add a single empty case to each data set considered in the evalua-

tion. Incomplete data is handled at the level of each independence test $I(X, Y; \mathcal{S})$ where a configuration over X, Y and $\mathcal{S}$ with a missing value is ignored. Since data is made incomplete by adding a single empty case, we are in practice using the same data in the evaluation (just without exploiting the fact that data is complete).

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 entire program.

Fig. 10 shows the average run times of the horizontal parallel PC algorithm as a function of the number of worker threads for SACSO with 500,000 cases of complete and incomplete data running on Fyrkat, respectively. As expected, the average run time for the complete case is significantly lower than for the incomplete case. The difference between having complete and incomplete data
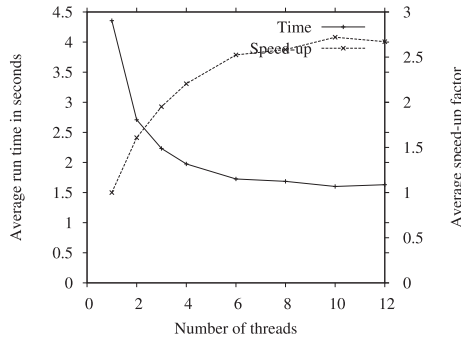
Fig. 9. Average run times for SACSO with 500,000 cases for higher order tests using $|\mathcal{S}| = 3$.

is twofold. First, in the incomplete data case there is no use of BIB designs in the marginal independence test. Second, no extra heuristic tests are performed involving variables with incomplete data. The difference between Fig. 10 (a) and (b) shows that these two optimizations produce a speed-up factor of more than two for the horizontal parallel PC algorithm.

Fig. 11 shows the average run times of horizontal parallel PC for SACSO with 500,000 cases of complete and incomplete data running on Odin, respectively. In comparison, Fig. 6 (right) shows the average run time of the parallel PC algorithm for the same network and data set.

Recall that Figs. 10 and 11 show the average time cost as a function of the number of worker processes (in addition to the master process). In the case of one worker process, this process still has to communicate the count tables to the master process (running on a different computer). This is the reason that there is a difference in time performance between parallel PC and horizontal PC for the value one.

Recall that Odin is a shared memory computer with a single CPU (six physical cores and 12 logical cores) whereas Fyrkat is a computer cluster with distributed memory. The significant difference in the average run time for the same task is probably due to different CPU performance.

5. Discussion

This paper considers parallel Bayesian network structure learning from data using a variant of the PC algorithm. Two approaches to parallelization have been considered in the paper. One approach is designed for a multi-core shared memory architecture whereas the other approach is designed for a computer cluster with dis-

tributed memory. The first 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 Section 4 clearly demonstrate that the total time cost of Steps 2–5 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 in both approaches 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.

In the statistical tests for marginal independence, BIB designs are used on the subset of variables with complete data. BIB designs on the form $(q, 6, 1)$ are used to produce counts tables over six variables. If variables have many states and there are only a few cases, then this table may be larger than the number of cases in the original data set. Therefore, the approach requires a minimum number of cases.

The edge index array is the central bottleneck of the approach as it is the only element that requires synchronization. There is no need for synchronization during the marginal independence testing. Synchronization is limited to selecting which edge to test and to determine which remaining tests need to be performed. There is no synchronization related to the counting. The counting usually being the most time consuming element of testing for conditional pairwise independence.

The horizontal parallel PC approach is based on distributing a subset of the data over all variables to a set of worker processes. This approach is embarrassingly parallel. Each process holds a distinct subset of the data cases over all variables and it is responsible for computing partial counts over this subset each time the master process needs to perform a test. When the horizontal parallel PC approach exploits the use of BIB designs (over variables with complete data), the tables communicated may become large. We have used a limit on the count tables equal to the number of cases in the original data set.

The results of the empirical evaluation show a significant time performance improvement over the pure sequential method for both approaches. For most cases considered there is a point where using additional threads or processes 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
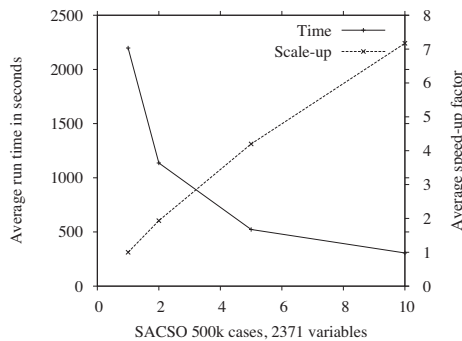
(a) SACSO 500,000, Fyrkat, complete
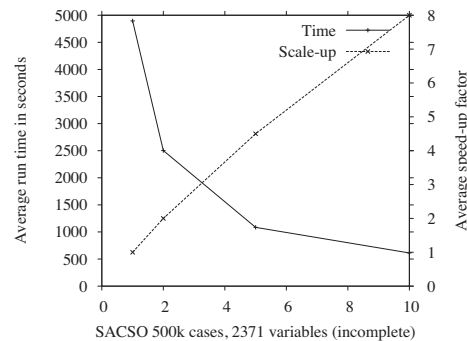
(b) SACSO 500,000, Fyrkat, incomplete

Fig. 10. Average run times for SACSO with 500,000 on Fyrkat using complete and incomplete data, respectively, as a function of the number of worker processes.
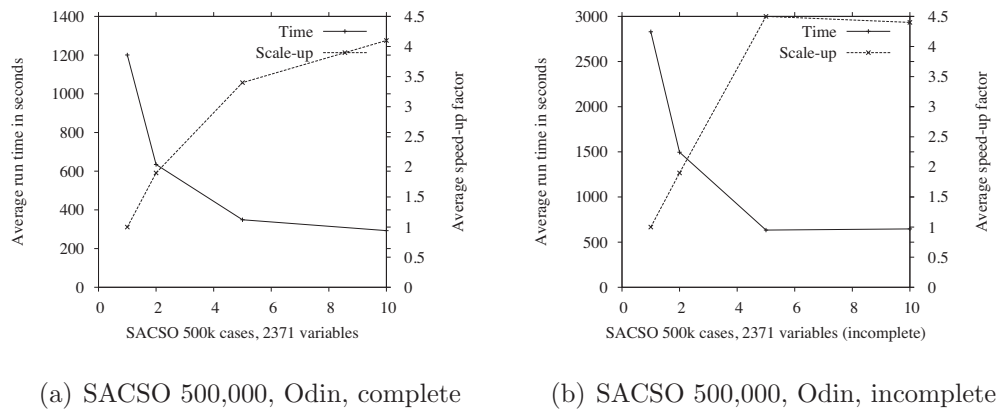
(a) SACSO 500,000, Odin, complete

(b) SACSO 500,000, Odin, incomplete

Fig. 11. Average run times for SACSO with 500,000 on Odin using complete and incomplete data, respectively, as a function of the number of worker processes.

time costs. Notice that on SACSO with complete data, the thread-based version is faster and offers a better speed-up factor than the process-based approach.

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.

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.

6. Conclusions

In this paper, we have considered two different approaches to parallelization of Bayesian network structure learning using the PC algorithm. The horizontal approach is embarrassingly parallel and shows that a significant speed-up is possible both on a shared memory system and a cluster system using processes. The other approach based on the use of BIB designs for marginal independence testing shows a significant speed-up on shared memory systems using threads. This makes it possible to take advantage of multi-core and multi-processor systems to improve time efficiency of structure learning.

References

[1] J. Pearl, Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference, Series in Representation and Reasoning, Morgan Kaufmann, 1988.
[2] R. Cowell, A. Dawid, S. Lauritzen, D. Spiegelhalter, Probabilistic Networks and Expert Systems, Springer, 1999.
[3] F.V. Jensen, T.D. Nielsen, Bayesian Networks and Decision Graphs, 2nd ed., Springer, 2007.
[4] D. Koller, N. Friedman, Probabilistic Graphical Models — Principles and Techniques, MITPress, 2009.
[5] U.B. Kjærulff, A.L. Madsen, Bayesian Networks and Influence Diagrams: A Guide to Construction and Analysis, 2nd ed., Springer, 2013.
[6] P. Spirtes, C. Glymour, R. Scheines, Causation, Prediction, and Search, Adaptive Computation and Machine Learning, second ed., MIT Press, 2000.

[7] Q. Fang, K. Yue, X. Fu, H. Wu, W. Liu, A MapReduce-based method for learning bayesian network from massive data, in: Web Technologies and Applications, in: Lecture Notes in Computer Science, vol. 7808, Springer, 2013, pp. 697–708, doi:10.1007/978-3-642-37401-2_68.
[8] C.-T. Chu, S. Kim, Y.-A. Lin, Y. Yu, G. Bradski, A. Ng, K. Olukotun, Map-Reduce for machine learning on multicore, in: NIPS, 2006, pp. 281–288.
[9] M. Scutari, Learning Bayesian networks with the bnlearn R package, J. Stat. Software 35 (3) (2010) 1–22.
[10] A. Basak, I. Brinster, X. Ma, O. Mengshoel, Accelerating Bayesian network parameter learning using Hadoop and MapReduce, in: Proceedings of the 1st International Workshop on Big Data, Streams and Heterogeneous Source Mining: Algorithms, Systems, Programming Models and Applications, 2012, pp. 101–108.
[11] M. Kalisch, P. Buhlmann, Estimating high-dimensional directed acyclic graphs with the PC-algorithm, J. Mach. Learn. Res. 8 (2008) 613–636.
[12] M. de Jongh, Algorithms for constraint-based learning of Bayesian network structures with large numbers of variables (Ph.D. thesis), University of Pittsburgh, 2014.
[13] O. Nikolova, S. Aluru, Parallel discovery of direct causal relations and Markov boundaries with applications to gene networks, in: Parallel Processing (ICPP), 2011 International Conference IEEE, 2011, pp. 512–521.
[14] W. Chen, L. Zong, W. Huang, G. Ou, Y. Wang, D. Yang, An empirical study of massively parallel Bayesian networks learning for sentiment extraction from unstructured text, in: Web Technologies and Applications, Springer, 2011, pp. 424–435.
[15] J. Arias, J. Gamez, J. Puerta, Learning distributed discrete Bayesian network classifiers under MapReduce with Apache spark, Knowledge Based Syst. (2016), doi:10.1016/j.knosys.2016.06.013. Available online 22 June 2016
[16] A.L. Madsen, F. Jensen, A. Salmeron, M. Karlsen, H. Langseth, T.D. Nielsen, A new method for vertical parallelisation of TAN learning based on balanced incomplete block designs, in: Proceedings of PGM, 2014, pp. 302–317.
[17] D. Stinson, Combinatorial Designs — Constructions and Analysis, Springer, 2003.
[18] T.M. Forum, MPI: A Message Passing Interface, in: Supercomputing '93, Portland, OR, 1993, pp. 878–883.
[19] R. Fisher, An examination of the different possible solutions of a problem in incomplete blocks, Ann. Eug. 10 (1940) 52–75.
[20] D. Corneil, R. Mathon, Algorithmic techniques for the generation and analysis of strongly regular graphs and other combinatorial configurations, Ann. Discrete Math. 2 (1978) 1–32.
[21] K. Takeuchi, A table of difference sets generating balanced incomplete block designs, Rev. Int. Stat. Inst. 30 (3) (1962) 361–366.
[22] A.L. Madsen, M. Lang, U.B. Kjærulff, F. Jensen, The Hugin tool for learning Bayesian networks, in: Proceedings of ECSQARU, 2003, pp. 549–605.
[23] D.E. Knuth, The Art of Computer Programming, 4, Fascicle 3, Addison-Wesley, 2005.
[24] A. Papanikolaou, Presents Modern Risk-Based Methods and Applications to Ship Design, Operation, and Regulations, Springer, 2009.
[25] S. Andreassen, F.V. Jensen, S.K. Andersen, B. Falck, U. Kjærulff, M. Woldbye, A.R. Sørensen, A. Rosenfalck, F. Jensen, MUNIN — an expert EMG assistant, Computer-Aided Electromyography and Expert Systems, Elsevier Science, 1989.
[26] S. Andreassen, R. Hovorka, J. Benn, K.G. Olesen, E.R. Carson, A model-based approach to insulin adjustment, in: Proceedings of the Third Conference on Artificial Intelligence in Medicine, 1991, pp. 239–248.
[27] F.V. Jensen, C. Skaanning, U. Kjærulff, The SACSO system for troubleshooting of printing systems, in: In Proceedings of the Seventh Scandinavian Conference on Artificial Intelligence, IOS Press, 2001, pp. 67–79.
[28] A.L. Madsen, F. Jensen, A. Salmeron, H. Langseth, T.D. Nielsen, Parallelization of the PC algorithm, in: The XVI Conference of the Spanish Association for Artificial Intelligence, 2015, pp. 14–24.