Wait-Free Primitives for Initializing Bayesian Network Structure Learning on Multicore Processors

Hsuan-Yi Chu
University of Southern California
Los Angeles, CA 90089
Email: hsuanyi@usc.edu

Yinglong Xia
IBM T.J. Watson Research Center
Yorktown Heights, NY 10598
Email: yxia@us.ibm.com

Anand Panangadan and Viktor K. Prasanna
University of Southern California
Los Angeles, CA 90089
Email: {anandvp, prasanna}@usc.edu

***Abstract*—Structure learning is a key problem in using Bayesian networks for data mining tasks but its computation complexity increases dramatically with the number of features in the dataset. Thus, it is computationally intractable to extend structure learning to large networks without using a scalable parallel approach. This work explores computation primitives to parallelize the first phase of Cheng et al.'s (Artificial Intelligence, 137(1-2):43-90, 2002) Bayesian network structure learning algorithm. The proposed primitives are highly suitable for multithreading architectures. Firstly, we propose a wait-free table construction primitive for building potential tables from the training data in parallel. Notably, this primitive allows multiple cores to update a potential table simultaneously without appealing to any lock operation, allowing all cores to be fully utilized. Secondly, the marginalization primitive is proposed to enable efficient statistics tests to be performed on all pairs of variables in the learning algorithm. These primitives are quantitatively evaluated on a 32-core platform and the experiment results show $23.5\times$ speedup compared to a single thread implementation.**

I. Introduction

Bayesian Networks are a class of probabilistic graphical models, which encodes the relationship among interacting random variables in a domain as a directed acyclic graph (DAG) [19]. Bayesian Networks have been applied extensively to model causal relationships under uncertainty in various fields such as bioinformatics, finance, medical diagnosis and signal processing.

One of the key problems in Bayesian networks is *structure learning*: given training data where each record consists of a set of observations of the random variables, the objective is to learn a DAG which best describes the probability distribution underlying the training data. The computation complexity of structure learning algorithms increase dramatically with the number of variables in the dataset and thus parallel implementations of such algorithms are valuable. This work explores the concept of parallel algorithm primitives for Bayesian network structure learning and presents a detailed design for two such primitives and which can be used as components in a scalable parallel structure learning implementation.

This material is based upon work supported by the National Science Foundation under Grant No. ACI-1339756.

The first primitive, *wait-free table construction*, is used for converting the training data into a potential table in parallel. We have developed a table representation based on a set of distributed hash tables. We show that this representation both improves space efficiency and also facilitates parallel marginalization of the potential table. We have developed a novel lock-free and wait-free data structure to enable multiple processor cores to access and modify this table representation. Note that in a naïve implementation, where each processor core is in charge of a disjoint subset of the training data, conflicts can be incurred while updating the potential table and can lead to a race condition. Though locks can be employed to prevent such conditions, lock-based solutions introduce waiting among processor cores and leave some cores idle, which reduces the speedup achievable from multi-core processor implementations. Our wait-free primitive addresses this challenge by a two-stage procedure, where all cores can update the potential table simultaneously in both stages, and only one synchronization step is needed between the two stages.

The second primitive implements marginalization of the potential table in parallel. This primitive uses data parallelism; each core accesses only a disjoint subset of the potential table during runtime. This approach helps in avoiding cache misses in a multi-core implementation.

These primitives are used to implement the first phase of Cheng et al.'s constraint satisfaction-based approach to Bayesian network structure learning [4]. This algorithm is one of the most commonly used approaches for structure learning. While the efficiency of their algorithm was shown for networks with a relatively small number of nodes, it is intractable to extend this algorithm to networks with hundreds of nodes without a scalable parallel implementation. To the best of our knowledge, our work is the first which leverages multi-core processors to parallelize constraint satisfaction-based approaches to Bayesian network structure learning.

We quantitatively evaluate these primitives using simulated training data on a 32-core AMD system. The evaluation results indicate that the proposed primitives are scalable

on such a system. We also compare our approach with a hashtable-based implementation using Intel's Threading Building Blocks (Intel TBB) [2]. Our approach presents consistent improvement over the Intel TBB method with different sizes of input data. The wait-free operation of the table construction primitive ensures that the speedup from using multiple cores is linear with the number of used cores.

In summary, we make the following contributions in this paper:

- We propose a wait-free table construction primitive to build potential tables from training data. This primitive divides table construction into two stages and all cores can update the potential table simultaneously. The computational complexity of this primitive is $O\left(\frac{m \times n}{P}\right)$, where m is the number of samples, n is the number of random variables, and P is the number of cores.
- We propose the parallel marginalization primitive to facilitate statistics tests on all pairs of variables. Using data parallelism, each core accesses only a disjoint subset of the potential. The computational complexity of this primitive is also $O\left(\frac{m \times n}{P}\right)$.
- The primitives are quantitatively evaluated on a 32-core processor platform, and the results demonstrate $23.5\times$ speedup. Notably, compared to a hashtable-based implementation using Intel TBB, our wait-free table construction primitive shows consistent improvement and higher scalability with the number of used cores.

The rest of this paper is organized as follows: In Section II, a brief introduction to Bayesian networks structure learning is presented. In Section III, the related work is discussed. In Section IV, the proposed approach and analysis are shown. Experiment results are presented in Section V. In Section VI, we conclude this paper.

II. Background

A. Overview of Bayesian Network

A probability distribution function over a set of random variables offers a quantitative measure for the likelihood of event occurrences and represents the (in)dependence between the random variables. Given the underlying probability distribution f, the conditional (in)dependence relationship can be encoded compactly in a Bayesian network. Formally, the Bayesian network is a directed acyclic graph G where the nodes represent random variables and the edges represent the (in)dependence relationships. To be more specific, the relationships are interpreted with the language of *active paths* and *d-separation*. Let $\mathbf{X}$, $\mathbf{Y}$ and $\mathbf{Z}$ be three sets of nodes in the Bayesian network. When the influence can flow between $\mathbf{X}$ and $\mathbf{Y}$ via $\mathbf{Z}$, the path $\mathbf{X} \Leftrightarrow \mathbf{Z} \Leftrightarrow \mathbf{Y}$ is active[1]. If there is no active path between $X \in \mathbf{X}$ and $Y \in \mathbf{Y}$ given $\mathbf{Z}$, $\mathbf{X}$ and $\mathbf{Y}$ are independent given $\mathbf{Z}$.

[1]Due to the limited space, the criteria for paths to be active are not present in this paper. Interested readers are directed to [13].

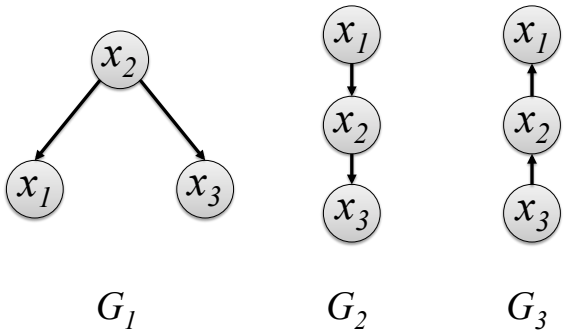


Figure 1: In order to encode the independence assertion where, given X_2, X_1 and X_3 are independent, the information flow between X_1 and X_3 should be separated by X_2. The graphs G_1, G_2 and G_3 all encode the same set of assertions and probability distribution.

Before we formally define the relationship between the Bayesian network G and underlying probability distribution f, we present an example to illustrate the concept. Assume that a distribution function $f(x_1, x_2, x_3)$ is factorized as $f(x_1, x_2, x_3) = f(x_1) f(x_2|x_1) f(x_3|x_2)$ which implies that X_1 and X_3 are independent given X_2. Following the procedure in [13], we can encode the independence assertion into Bayesian networks, as shown in Figure 1. Note that the set of independence assertions can be encoded by different Bayesian network representations; the graphs G_1, G_2 and G_3 in Figure 1 state the identical independent assertion, forming an *I-equivalence class* [13]. The relationship between the Bayesian network G and underlying probability distribution f is formally defined below [19], [13].

Definition 1. A Bayesian network G is a *dependence map* (*D-map*) of a distribution f if every dependence relationship derived from G holds for f. On the other hand, G is an *independence map* (*I-map*) of f if every independence relationship derived from G is true in f. Furthermore, if G is both a D-map and I-map of f, G is termed as a *perfect map* (*P-map*) of f.

However, the underlying probability distribution is not always accessible in advance. Thus, it is not feasible to derive (in)dependence assertions from probability distributions and impose them on a Bayesian network. In the next subsection, we introduce Bayesian network structure learning, which constructs graphs directly from training data (instead of from an explicit probability distribution).

B. Overview of Bayesian Network Structure Learning

The training data for structure learning is a set of observations of the random variables. A Bayesian network structure learning algorithm computes a DAG given a training data set

that best fits the training data. The data set is represented as an $m \times n$ matrix $\mathbf{D}$, where m denotes the number of samples, n denotes the number of random variables, and D_i denotes the i-th row of $\mathbf{D}$ (i.e., the i-th observation), $i = 1, 2, ..., m$. Since $D_i = \{s_{i1}, s_{i2}, ..., s_{in}\}$ is an enumeration of the states of the random variables in this observation, it is also called a *state string*. For a random variable X_j, the value r_j refers to the number of states it can take ($j \in \{1, 2, ..., n\}$). For a concise notation, we assume in this paper that all the random variables have the same number of states, i.e., the number of states for all random variables can be denoted by r. However, the techniques developed in this paper can be applied to varying number of states.

C. An Information-theoretic Approach

Cheng et al.'s [4] algorithm for Bayesian network structure learning is divided into three phases, *drafting*, *thickening* and *thinning*. In the first phase, *drafting*, an approximate network is derived by computing a statistics test of influence of all pairs of random variables. Essentially, this phase serves as a pre-processing step to yield a set of candidate edges. The first phase of the algorithm requires that the training data be mapped into a *potential table* representing the joint probability distribution. Statistics tests on all pairs of variables are computed after *probability marginalization* on this potential table. The algorithm uses *mutual information* as the statistics test to evaluate the influence between variables and determine the existence of edges. Mutual information is evaluated on a pair of random variables and if the value is greater than a pre-defined threshold, the two random variables are considered to be dependent and an edge could possibly exist between them. In the second phase, edges are added to the network (*thickening*) by inspecting the conditional mutual information. In the third phase, the intermediate network is trimmed by removing redundant edges (*thinning*) to produce the final Bayesian network. This algorithm requires $O(n^4)$ statistics tests to learn an n-feature Bayesian network.

The mutual dependence of two random variables (i.e., two nodes in the Bayesian network) is evaluated with mutual information. This quantity can be considered as the reduction in the uncertainty of one random variable with the knowledge of the other [8].

Definition 2. The mutual information of random variables X and Y is

$$I(X;Y) = \sum_{(X,Y)} \left[P(x,y) \log \frac{P(x,y)}{P(x)\,P(y)} \right] \quad (1)$$

However, there might be a set of evidence random variables which can change the relationship between two random variables. Thus, the *conditional mutual information* with respect to the set of evidence random variables is defined as below.

Definition 3. The conditional mutual information of random variables X and Y, given a set of random variables $\mathbf{Z}$, is

$$I(X;Y|\mathbf{Z}) = \sum_{(X,Y,\mathbf{Z})} \left[P(x,y,\mathbf{z}) \log \frac{P(x,y|\mathbf{z})}{P(x|\mathbf{z})\,P(y|\mathbf{z})} \right] \quad (2)$$

Note that $\mathbf{Z}$ is a set, and if it is empty, the conditional mutual information reduces to Equation 1.

In Bayesian network structure learning, mutual information serves as the criterion for the existence of an edge between two nodes. Thus, mutual information of various combinations of random variables would be evaluated repeatedly throughout the learning process. Thus, an algorithm for efficient evaluation of mutual information is necessary for large-scale Bayesian network structure learning.

III. Related Work

There are two main paradigms for Bayesian network structure learning. The first one formulates structure learning as an *optimization problem*. To be more specific, a statistically motivated *score* is given to each possible network structure to indicate the fitness of this structure to the input training data. Proposed scores include likelihood [6], ratio of posterior joint probabilities [7] and Bayesian metric with Dirichlet priors [12]. Ott et al. [18] apply dynamic programming to develop an $O\left(n^2 2^n\right)$ algorithm to compute the exact scores. Although methods to parallelize Ott et al.'s algorithm have been proposed [23], [17], it is intrinsically difficult to scale to large networks due to the computational complexity of computing a global score. Henceforth, heuristics are applied to scale to networks with large number of random variables [9], [14], [24]. For instance, Friedman et al. [9] restrict the search space by examining only a small number of candidate parents for each random variable via mutual information and lead to significant reduction in the learning time.

The second paradigm for Bayesian network structure learning formulates structure learning as a *constraint satisfaction problem*. In particular, statistical tests are conducted to verify conditional independence between the random variables in the training data [20], [22], [4]. Commonly used statistical tests include χ^2-test and mutual information. Due to the computational efficiency of this approach, it is preferred when the number of random variables is large (e.g., in bioinformatics applications) [25], [3], [29]. Notably, the three-phase algorithm proposed by Cheng et al. [4] demonstrated computational efficiency for Bayesian networks with a relatively small number of nodes.

Note that regardless of which paradigm is followed, it is infeasible to exhaustively search the entire space since the search space is exponential (the problem has been proved to be in NP-hard [5]). Therefore, all approaches prune the search space to a tractably small space with various statistics tests computed on the data set. Following are two pruning

methods, corresponding to each paradigm, to illustrate the concept. Friedman et al. [9] compute the mutual information between a node with all other nodes and find a set of random variables to be the candidate parents of the considered node in their optimization-based method. For the constraint satisfaction paradigm, Cheng et al. [4] compute mutual information between all pairs of random variables to yield a set of candidate edges in the first phase as a pre-processing step. Hence, mutual information is intensively evaluated throughout the learning procedure for both paradigms. In view of this, the primitives developed in this paper not only parallelize the first phase of Cheng et al.'s [4] algorithm but also yield a parallel and efficient tool to help reduce the search space of other structure learning algorithms.

A complementary problem to Bayesian network structure learning is Bayesian network *inference*. In inference, the underlying network is given and the problem is to evaluate marginal or conditional distributions efficiently to facilitate prediction of an interesting phenomenon. This is also an NP-hard problem and methods to parallelize Bayesian network inference have been proposed [26], [10], [11], [28]. For instance, [26] propose an approach to decompose a junction tree into chains to perform exact inference in parallel.

IV. Parallel Primitives

The parallel architecture that we consider in this work is modeled as a *parallel random access machine* (*PRAM*) [21]. In this model, a shared global memory that multiple processors are able to access simultaneously is assumed. Throughout, we use P to denote the number of processor cores. Given the training data set **D**, we first convert it into a potential table.

We first describe the abstract representation of the potential table and then present the data structure to enable for wait-free construction of the table.

A. Representation of Probability Distribution

The probability distribution is to be extracted from the training data. A potential table representation records the number of occurrences for each state string[2]. A straightforward representation is to store the state string along with each entry [15]. However, this representation not only occupies a large amount of memory $O(n \times r^n)$ but also needs to perform state string comparisons at every access which is a computationally expensive operation. Thus, we encode each state string which appears in **D** as a *key* [27]. The advantages of this approach are: 1) since state strings are no longer stored, the memory requirement is reduced to $O\left(\prod_{k=1}^{n} r_k\right)$, 2) direct access to the targeted entry is possible with the *1-to-1 mapping* between state strings and keys,

[2]It is not necessary to divide the counts by the total number of samples to obtain probabilities at this point. The normalization can be performed later when marginalization is needed. In fact, this approach helps reduce memory usage.

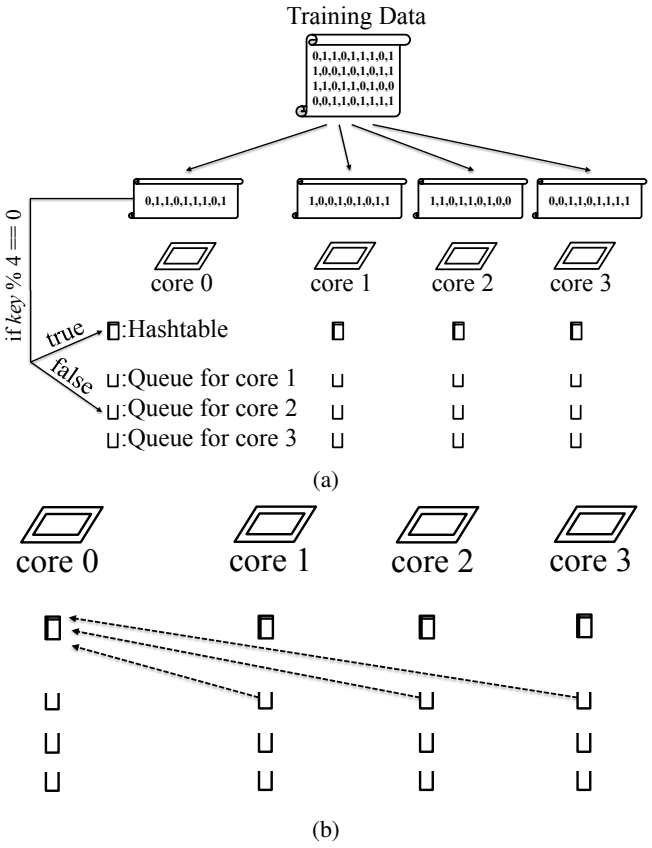


Figure 2: The two stages of the wait-free table construction primitive. (a) Keys are classified in parallel by each core in the first stage, and only the keys, falling into the range which that core is in charge of, are used to update the hashtable. (b) Each core fetches the keys assigned to it by the other cores and stored in their queues and uses them to update its hashtable in parallel. Note that this figure only shows the operation of core 0.

and 3) marginal probability distributions can be computed without comparing each character in the state string. We illustrate the conversion between state strings and keys next. For the i-th observation D_i, the state string $(s_1, s_2, ..., s_n)$, where $s_j \in \{0, 1, ..., (r-1)\}$ and $j \in \{1, 2, ..., m\}$, is mapped into an integer key as

$$key = f_{\text{enc}}(s_1, s_2, ..., s_n) = \sum_{j=1}^{n} s_j \times r^{(j-1)} \tag{3}$$

where f_{enc} denotes the *encoding function* from the state string to the corresponding key. Inversely, key is decoded to s_j with

$$s_j = f_{\text{dec}}(key, j) = \left\lfloor \frac{key}{r^{(j-1)}} \right\rfloor \% \; r_j \tag{4}$$

where f_{dec} denotes the *decoding function*. (% is the modulo operation.) Note that the complexities of encoding and

decoding are both $O(n)$. Since there is a unique key for each state string, the pair $(key, value)$ can be used to store the count of the occurrences of key. Hence, the probability distribution can be represented as a table, called the *probability table*, where each row consists of a $(key, value)$ pair.

In most real-world cases, the number of state strings will be sparse in **D** since the size of **D** is exponential with respect to the number of variables. In such situations, a *hashtable* is used to store and lookup keys. Otherwise, an array can be used with its index corresponding to the key of each state string. In this paper, we focus on sparse state strings and use hashtables to represent the distribution table.

Since the training data is typically very large, it is partitioned and divided between the available cores for conversion to the potential table representation. Nevertheless, a shared data structure for the potential table is not a viable solution since multiple processor cores can update the counts of the same keys, resulting in a race condition [16]. We next propose a wait-free primitive that addresses this challenge by a two-stage procedure. The primitive avoids using locks and enables all processor cores to update the potential table simultaneously with only a single synchronization step between the two stages.

B. Wait-free Table Construction Primitive

The integer representation of state strings enables this primitive to be defined using integer primitives. Given the number of random variables and the state sizes, we can bound the key range (from 0 to $(r^n - 1)$). This range is divided into P disjoint partitions and each processor core is in charge of a distinct part of the key space. A separate hashtable, whose keys are in one subspace, is assigned to each processor core. In addition, each processor core is equipped with $(P-1)$ queues. Each such queue is indexed with a number to associate it with one of the other processors.

The wait-free primitive consists of two stages and one synchronization step between them, as schematically shown in Figure 2. In the first stage, the training data **D** is divided into P partitions and delegated to P cores to convert state strings to keys. At each core, if a key falls into its key range, this core simply updates its hashtable. Otherwise, this key is assigned to the queue whose index corresponds to the processor core in charge of this key. The complexity of this stage is $O\left(\frac{m \times n}{P}\right)$. In the second stage, each processor core accesses the queues on other processors that correspond to its index. Since the first stage ensures that all the keys are already assigned to the appropriate queues, each processor core only pops the keys and updates its hash map. Assuming the training data is uniformly divided between the cores, the complexity of this step is $O\left(\frac{m}{P}\right)$. Conceptually, the primitive assigns keys in the first stage to avoid updating the "foreign keys" in the second stage. This method of dividing the key space not only makes updating straightforward but also divides the memory accessed by multiple cores into P disjoint parts. Thus, there is no conflict between the processor cores throughout table construction. The total computational complexity of the wait-free table construction primitive is $O\left(\frac{m \times n}{P}\right)$. The two stages of the algorithm are formally presented in Algorithm 1 and 2 respectively.

Algorithm 1 Wait-free Primitive – Stage 1

Input: Dataset **D**, number of processor cores P
Output: Hashtables and queues $\mathbb{H}$, $\mathbb{Q}$
Initialize P hashtables $\mathbb{H} = (\mathbb{H}_0, ..., \mathbb{H}_{P-1})$
Initialize $P \times (P-1)$ queues and represent them as follows: $\mathbb{Q} = (\mathbb{Q}_0, ..., \mathbb{Q}_p, ..., \mathbb{Q}_{P-1})$, where $\mathbb{Q}_p = (\mathbb{Q}_{p,0}, ..., \mathbb{Q}_{p,p-1}, \mathbb{Q}_{p,p+1}, ..., \mathbb{Q}_{p,P-1}), p = 0, ..., (P-1)$, where each $\mathbb{Q}_{i,j}$ is a queue
for $p = 0$ to $(P-1)$ **in parallel do**
    for $i = 1 + (p-1) \times \frac{m}{P}$ to $p \times \frac{m}{P}$ **do**
        $(s_1, s_2, ..., s_n) \longleftarrow D_i$
        $key \longleftarrow f_{en}(s_1, s_2, ..., s_n)$
        $index \longleftarrow key \% P$
        if $index == p$ **then**
            Update $\mathbb{H}_p(key)$
        else
            Add key onto $\mathbb{Q}_{p,index}$
        end if
    end for
end for
Return $(\mathbb{H}, \mathbb{Q})$

C. Marginalization Primitive

Marginalization is the process of computing the probability distribution of a set of variables from a larger probability table by summing over the state space of all random variables not in the set of interest [13]. For instance, given the probability distribution over random variables $X_1, X_2, ..., X_n$, the marginal distribution of X_i is represented as

$$P(x_i) = \sum_{(X_1, X_2, ..., X_{i-1}, X_{i+1}, ..., X_n)} P(x_1, x_2, ..., x_n) \quad (5)$$

A straightforward approach based on this equation decodes each key stored in the data structure, filters and sums over those values which correspond to the right hand side of Equation (5). This implementation results in computationally expensive decoding operations, the complexity of which is $O(n)$ for each key. Thus, the complexity of this approach is $O(n \times r^n)$. The complexity is dominated by an exponential term which makes this approach unscalable with respect to the size of the networks (i.e., the number of random variables).

As described earlier, state strings tend to become sparse as the number of random variables increase. For instance, the number of state strings of a network with 40 random

Algorithm 2 Wait-free Primitive – Stage 2

Input: Hashtables and queues $\mathbb{H}$, $\mathbb{Q}$, number of processor cores P
Output: Hashtables $\mathbb{H}$
for $p = 0$ to $(P-1)$ **in parallel do**
  for $i = 0, ..., (p-1), (p+1), ..., (P-1)$ **do**
    while $\mathbb{Q}_{i,p}.size() != 0$ **do**
      $key \longleftarrow \mathbb{Q}_{i,p}.pop()$
      Update $\mathbb{H}_p(key)$
    end while
  end for
end for
Return $\mathbb{H}$

variables, each with 2 states, is 2^{40}. Such a large number of samples is not observed in real-world datasets[3]. Thus, as the number of random variables grow, most of the potential state strings would not be observed and it is not necessary to sum over these strings in the operand of summation in Equation 5). In our proposed marginalization primitive, this observation is exploited to avoid the exponential complexity of the straightforward approach.

After the wait-free table construction primitive, the potential table is represented in P hashtables. For each marginalization, each processor iterates through all the keys of its hashtable. With the decoding function (Equation 4), the count of each key is matched to the corresponding entry of the marginal table. Note that it only needs the values of the random variables which appear in the marginal table to determine the entry. We do not need to recover the entire state string from each key. First, each processor performs this procedure on its own hashtable to obtain a partial marginal table. Second, a merging step is performed to output the final marginal table. The algorithm is formally shown in Algorithm 3.

If the sizes of all the hashtables are equal, the workload on each core would be equal and the complexity is $O\left(\frac{m \times n}{P}\right)$. If the hashtables are unbalanced, entries can be moved between hashtables to make them balanced. The requirement that each hashtable has a range of keys is necessary only in the wait-free table construction primitive; there is no such constraint for the marginalization primitive. Thus, rearranging the hashtables to make them balanced does not affect the correctness of the marginalization primitive.

With the Wait-free Table Construction and Marginalization primitives, the first phase of Cheng et al.'s [4] structure learning algorithm can be parallelized. First, the Wait-free Table Construction primitive is provided with the training data as input to generate the potential table. Second, mutual information is computed for all pairs of random variables according to Equation 1. Evaluating this equation

[3]Some common data sets for Bayesian networks can be found in [1].

Algorithm 3 Parallel Marginalization

Input: Hashtable $\mathbb{H}$, a set of random variables $\mathbf{V}$, number of processor cores P
Output: Marginal distribution table M
Initialize a marginal distribution table M
for $p = 1$ to P **in parallel do**
  $it \longleftarrow$ get iterator from $\mathbb{H}_p$
  Initialize a partial marginal distribution table M_p
  while $it != end$ **do**
    // $\mathbf{s}$ refers to a marginal state string
    // $\mathbf{V}$ refers to the RVs of interest which should be decoded
    $\mathbf{s} \longleftarrow$ decode $it.key$ for RVs $\in \mathbf{V}$ via (Eq. 4)
    $c \longleftarrow \mathbb{H}_p(it.key)$
    $M_p(\mathbf{s}) \longleftarrow M_p(\mathbf{s}) + c$
    $it \longleftarrow it + 1$
  end while
end for
Merge partial marginal distribution tables M_p, where $p = 1, ..., P$
Divide each entry of M by m
Return M

requires computing marginal probability distributions. There are three marginal probability distributions in Eqution 1. Note that instead of computing each of the three distributions independently, we can first compute $P(x, y)$ and obtain $P(x)$ and $P(y)$ from $P(x, y)$ (by summing over $P(y)$ and $P(x)$ respectively). This approach eliminate two computationally expensive operations of marginalization. Since the complexity for computing the mutual information for a pair of variables is $O\left(\frac{m \times n}{P}\right)$, the complexity of all-pairs mutual information is $O\left(\frac{m \times n^3}{P}\right)$. The algorithm is formally presented in Algorithm 4. In the next section, the experiment results demonstrate the scalability of our approach.

V. Experiments

A. Experiment Platform

The experiments are conducted on an AMD Opteron 6278 platform which has 2×16 cores with 64 GB DDR3 main memory. Each core runs at 2.4GHz with 2 MB L2 cache, and the operating system is Linux OpenSUSE 12.2. The proposed primitives are implemented using *POSIX Threads* on C++. The number of variables, the number of observations and the number of cores are varied to demonstrate the scalability of our approach. Variable instances (training data) that are the input for Bayesian network structure learning are synthesized from uniform and independent distributions for each variable. Note that independently distributed training data implies that each core would process approximately the same number of instances during the table construction phase. The specific distribution does not affect the running

Algorithm 4 Paralleling the First Phase of Cheng et al.'s [4] Structure Learning Algorithm

Input: Dataset **D**, number of processor cores P
Output: Mutual information of all pairs of RVs **I**
Initialize an array **I**
// The first stage of the table construction primitive
$(\mathbb{H}, \mathbb{Q}) = Alg1\,(\mathbf{D}, P)$
// The second stage of the table construction primitive
$\mathbb{H} = Alg2\,(\mathbb{H}, \mathbb{Q}, P)$
// compute the mutual information for all pairs of RVs
for $i = (n-1)$ to 1 **do**
    $j \leftarrow (n - i - 1)$
    while $p \leq i$ **do**
        $I[j \times n + (j+p)] \longleftarrow Ent^a(Alg3\,(\mathbb{H}, \{X_i, X_j\}, P))$
        $p \leftarrow p + P$
    end while
    $p \leftarrow p - i$
end for
Return I

[a] Ent refers to the function which computes mutual information by Equation 1 from a given marginal table.

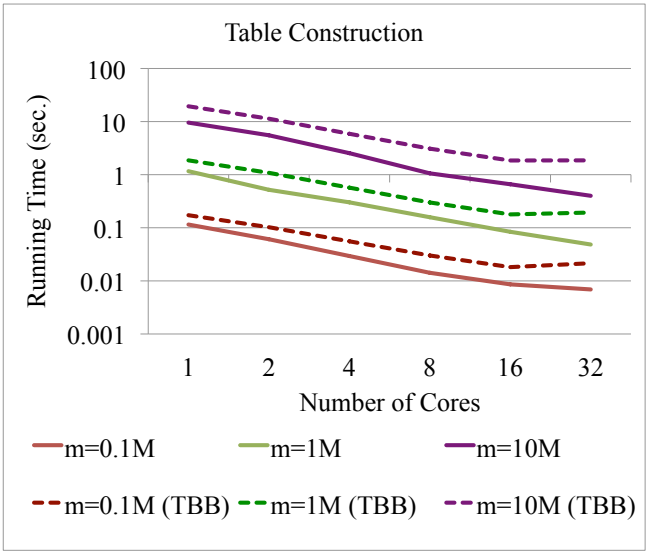
time of Bayesian network structure learning after the potential table is constructed.

We compare the scalability of the wait-free table construction primitive with the concurrent hashtable implementation in the *Intel Threading Building Blocks* (Intel TBB) [2] multi-threaded C++ library which ensures thread safe operations with the aid of a lock operation. In all the plots in this section, the *x-axis* indicates the number of used cores on a logarithmic scale to show the scalability of our approach with the number of used cores.

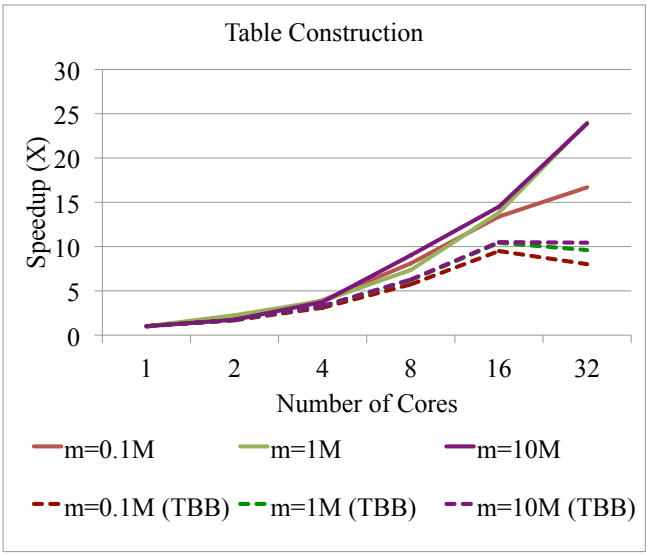
B. Table Construction

The performance of the wait-free table construction primitive is evaluated by varying two parameters. First, we investigate the scalability of the table construction primitive with the number of samples in the training dataset. While the number of random variables is set as 30, the numbers of samples are varied from 0.1, 1, to 10 million samples. As is shown in Figure 3, the running time of our approach increases linearly with the number of samples (i.e., the equal gaps between two neighboring curves). This observation aligns with the analysis and presents a scalable primitive. Moreover, with the number of used cores increasing, the running time presents a consistent speedup, which holds regardless of how many samples are input[4]. On the other hand, the running time for Intel TBB is shown by the dash

[4]When 32 cores are used, the speedup appears to be less desirable. This is because other system overheads can dominate when the input size is very small.



(a)



(b)

Figure 3: The scalability of the Wait-free Table Construction primitive and the TBB concurrent hashtable with the size of training data (m). (a) Running times. The y-axis is in logarithmic scale. (b) Speedup.

lines in Fig. 3. For the same number of samples, there is a gap between our wait-free primitive and the TBB concurrent hashtable. Notably, the gap widens with the number of used cores increasing. This implies that while one uses more cores to update the hashtable, the speedup becomes less significant or even worse. This observation is emphasized by Fig. 3b, where the slopes of the dash lines decrease from the number of cores equal to 4 and even become negative after the number of cores equal to 16.

Secondly, the scalability with another input dimension, the

number of random variables, is investigated. As the number of samples is fixed at 10 million, the numbers of random variables is varied from 30 to 50 with 10 increments. As is shown in Figure 4, the running time increases linearly with the numbers of samples which agrees with the analysis in Section IV. The speedup shown in Figure 4b presents the scalability with the number of cores. For the same number of used cores, there is a gap between our wait-free primitive and the TBB concurrent hashtable. This gap widens with the number of used cores. Thus, our primitive is more scalable than the TBB concurrent hashtable with the number of used cores.

C. All-pairs Mutual Information

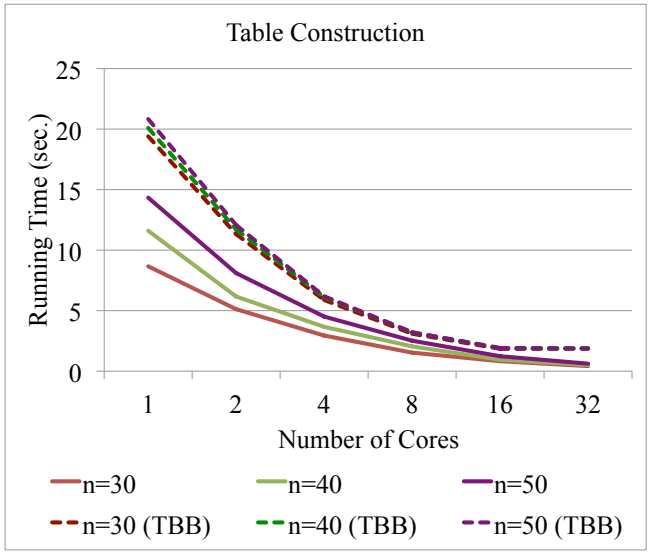
In this experiment, we show the running time of computing the mutual information between all pairs of random variables. The potential table is constructed using the wait-free table construction primitive from the training data. The number of observations is 10 million, and the number of random variables is varied from 30 to 50 with 10 increments.

As is shown in Figure 5a, the running times decrease consistently with the numbers of used cores for three cases $n = 30, 40, 50$. The theoretical running time of computing the mutual information between all pairs of random variables is $O\left(\frac{m \times n^3}{P}\right)$ (Section IV) which agrees with the decrease in the experimental running times. The speedup achievable using our approach is shown in Figure 5b.

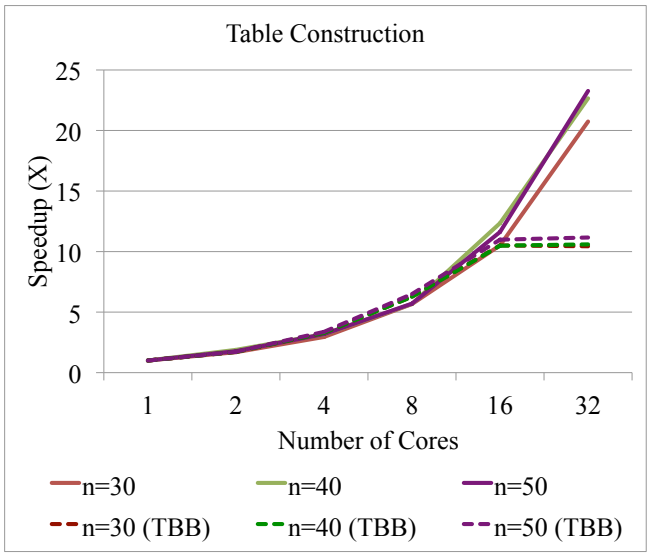
VI. Conclusion

We developed two primitives as the first step in parallelizing constraint satisfaction-based approaches to Bayesian network structure learning. The wait-free table construction primitive is designed to build the potential table from the training data in parallel. This primitive allows multiple processors to update the potential table simultaneously without using any lock operation, allowing all processors to be fully utilized. The marginalization primitive is designed to enable parallelization of statistics tests between random variables; such tests are performed between all pairs of variables in the structure learning algorithm that we consider. The marginalization primitive uses data parallelism to ensure that each processor needs to access only a disjoint subset.

In experiments with synthetic training data, the wait-free table construction primitive is compared with a hashtable-based implementation using Intel TBB. Our approach demonstrated consistent improvement with different sizes of input data. Due to its wait-free operation, the speedup is linear with the number of used cores and thus demonstrates a scalable primitive. The marginalization primitive is also evaluated in the experiments and shows scalability with the number of used cores. The parallel marginalization primitive avoids cache misses by letting each core access only a subset of the potential table that is disjoint from other cores.



(a)



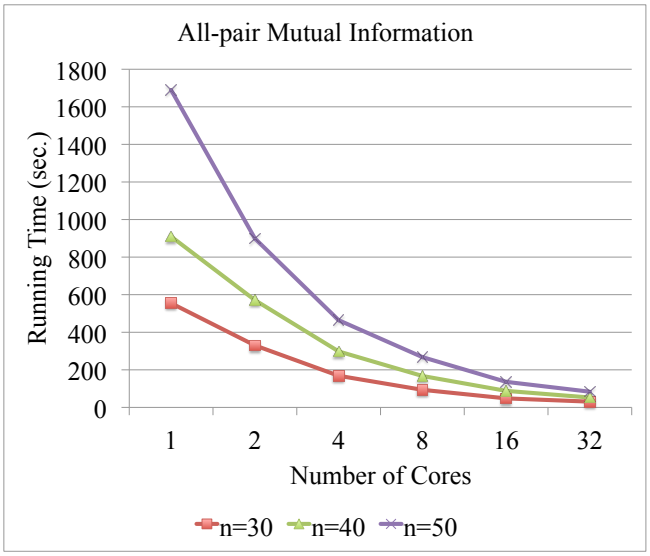
(b)

Figure 4: The scalability of the wait-free table construction primitive (solid lines) compared with the Intel TBB library primitives (dashed lines) with the number of random variables. (a) Running times. The running time of the wait-free primitive increases linearly with the number of variables (equal gaps between two neighboring curves). (b) Speedup.

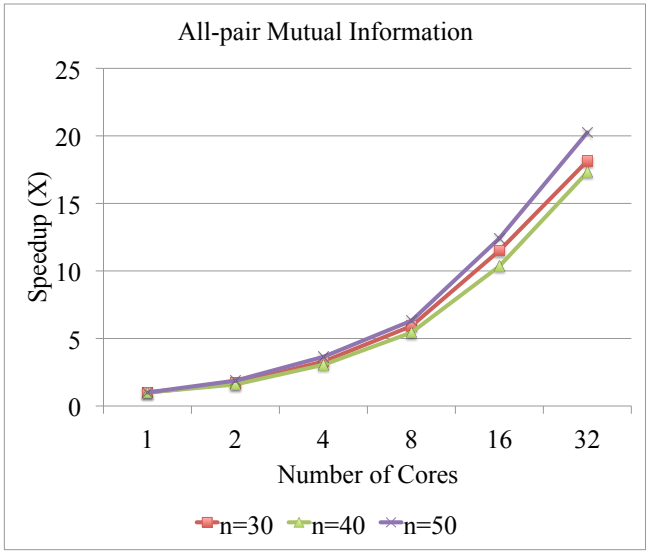
For future work, we will develop primitives that encompass all stages of a parallel implementation of Bayesian structure learning algorithm. We will design task-parallel and data-parallel optimization methods for these primitives and compare their relative advantages.

References

[1] Bayesian network repository. http://www.cs.huji.ac.il/site/labs/compbio/Repository/



(a)



(b)

Figure 5: The scalability of computing the mutual information between all pairs of random variables with the number of random variables. The proposed marginalization primitive is used to compute the mutual information between every pair of variables. (a) running time (b) speedup

networks.html.

[2] Intel Threading Building Blocks (Intel TBB). https://www.threadingbuildingblocks.org/.

[3] C. Auliac, V. Frouin, X. Gidrol, and F. d'Alche Buc. Evolutionary approaches for the reverse-engineering of gene regulatory networks: A study on a biologically realistic dataset. *BMC Bioinformatics*, 9(1):91, 2008.

[4] J. Cheng, R. Greiner, J. Kelly, D. Bell, and W. Liu. Learning Bayesian networks from data: an information-theory based approach. *Artificial Intelligence*, 137(1-2):43–90, May 2002.

[5] D. M. Chickering. A transformational characterization of equivalent Bayesian network structures. In *Conference on Uncertainty in Artificial Intelligence (UAI)*, Aug. 1995.

[6] C. Chow and C. Liu. Approximating discrete probability distributions with dependence trees. *IEEE Transactions on Information Theory*, 14(3):462–467, May 1968.

[7] G. F. Cooper and E. Herskovits. A Bayesian method for the induction of probabilistic networks from data. *Machine Learning*, 9(4):309–347, Oct. 1992.

[8] T. Cover and J. Thomas. *Elements of Information Theory*. Wiley, 2006.

[9] N. Friedman, I. Nachman, and D. Peér. Learning Bayesian network structure from massive datasets: the "sparse candidate" algorithm. In *Conference on Uncertainty in Artificial Intelligence (UAI)*, Jul./Aug. 1999.

[10] J. Gonzalez, Y. Low, and C. Guestrin. Residual splash for optimally parallelizing belief propagation. In *Artificial Intelligence and Statistics (AISTATS)*, Apr. 2009.

[11] J. Gonzalez, Y. Low, C. Guestrin, and D. O'Hallaron. Distributed parallel inference on large factor graphs. In *Uncertainty in Artificial Intelligence (UAI)*, Jun. 2009.

[12] D. Heckerman, D. Geiger, and D. M. Chickering. Learning Bayesian networks: The combination of knowledge and statistical data. *Machine Learning*, 20(3):197–243, Sep. 1995.

[13] D. Koller and N. Friedman. *Probabilistic Graphical Models: Principles and Techniques*. MIT Press, 2009.

[14] A. Moore and W.-K. Wong. Optimal reinsertion: A new search operator for accelerated and more accurate Bayesian network structure learning. In *International Conference on Machine Learning (ICML)*, Aug. 2003.

[15] V. Namasivayam and V. Prasanna. Scalable parallel implementation of exact inference in Bayesian networks. In *International Conference on Parallel and Distributed Systems (ICPADS)*, Jul. 2006.

[16] R. H. B. Netzer and B. P. Miller. What are race conditions?: Some issues and formalizations. *Transactions on Programming Languages and Systems*, 1(1):74–88, Mar. 1992.

[17] O. Nikolova, J. Zola, and S. Aluru. Parallel globally optimal structure learning of Bayesian networks. *Journal of Parallel and Distributed Computing*, 73(8):1039–1048, Aug. 2013.

[18] S. Ott, S. Imoto, and S. Miyano. Finding optimal models for small gene networks. In *Pacific Symposium on Biocomputing (PSB)*, Jan. 2004.

[19] J. Pearl. *Probabilistic reasoning in intelligent systems: networks of plausible inference*. Morgan Kaufmann, 1988.

[20] J. Pearl and T. Verma. A theory of inferred causation, 1991.

[21] D. M. Pennock. Logarithmic time parallel Bayesian inference. In *Conference on Uncertainty in artificial intelligence (UAI)*, Jul. 1998.

[22] P. Spirtes, C. Glymour, and R. Scheines. *Causation, Prediction, and Search*. Adaptive Computation and Machine Learning. MIT Press, 2000.

[23] Y. Tamada, S. Imoto, and S. Miyano. Parallel algorithm for learning optimal Bayesian network structure. *Journal of Machine Learning Research*, 12:2437–2459, Jul. 2011.

[24] M. Teyssier. Ordering-based search: A simple and effective algorithm for learning Bayesian networks. In *Conference on Uncertainty in artificial intelligence (UAI)*, Jul. 2005.

[25] I. Tsamardinos, C. F. Aliferis, A. Statnikov, A. Statnikov, and L. E. Brown. Scaling-up Bayesian network learning to thousands of variables using local learning techniques. Technical report, Department of Biomedical Informatics, Vanderbilt University, 2003.

[26] Y. Xia and V. Prasanna. Junction tree decomposition for parallel exact inference. In *IEEE International Symposium*

on Parallel and Distributed Processing (IPDPS), Apr. 2008.

[27] Y. Xia and V. K. Prasanna. Scalable node-level computation kernels for parallel exact inference. *IEEE Transactions on Computers*, 59(1):103–115, Jan. 2010.

[28] Y. Xia and V. K. Prasanna. Distributed evidence propagation in junction trees on clusters. *IEEE Transactions on Parallel and Distributed Systems*, 23(7):1169–1177, Jul. 2012.

[29] Z. Yang. *Machine Learning Approaches to Bioinformatics*. Science, engineering, and biology informatics. World Scientific Publishing Company, 2010.