A Parallel Algorithm for Exact Structure Learning of Bayesian Networks

Olga Nikolova, Jaroslaw Zola, and Srinivas Aluru
Department of Computer Engineering
Iowa State University
Ames, IA 50010
{olia,zola,aluru}@iastate.edu

Abstract

Given n random variables and a set of m observations of each of the n variables, the Bayesian network (BN) structure learning problem is to infer a directed acyclic graph on the n variables such that the implied joint probability distribution best explains the set of observations. In this paper, we present a parallel algorithm for exact BN structure learning that is work-optimal and communication-efficient[1]. We demonstrate the applicability of our method by an implementation on the IBM Blue Gene/L and an AMD Opteron cluster, and report extended experimental results that exhibit near perfect scaling.

1 Exact Estimation of a Bayesian Network Structure

Bayesian networks (BNs) are probabilistic graphical models which allow for a compact representation of the joint probability distribution (JPD) of a set of interacting variables of a given domain. The pair (N, P) of a directed acyclic graph (DAG) $N = (\mathcal{X}, E)$ and a JPD P over $\mathcal{X}$ defines a BN if each variable $X_i \in \mathcal{X}$ is independent of its non-descendants, given its parents. To evaluate the posterior probability of a considered graph given a set of observations the Bayesian approach utilizes a statistically motivated scoring function. Decomposable scoring functions have been widely used, which can be defined as the sum of the individual score-contributions $s\left(X_i, Pa\left(X_i\right)\right)$ of each variable $X_i \in \mathcal{X}$ given its parents, i.e. $Score\left(N\right) = \sum_{X_i \in \mathcal{X}} s\left(X_i, Pa\left(X_i\right)\right)$. Optimizing such a scoring criterion facilitates the discovery of a structure that best represents the observed data. The problem of exact structure learning quickly becomes intractable due to its combinatorial search-space, and has been shown to be NP-hard even for the case of bounded node in-degree. Parallel algorithms have been developed for heuristics-based BN learning, particularly using meta-heuristics and sampling techniques.[2] Such methods trade off optimality for the ability to learn larger networks. Constructing exact BNs without additional assumptions nevertheless remains valuable. In this paper, we present a parallel algorithm for exact BN learning with nearly perfect load-balancing and optimal parallel efficiency. To our knowledge, this is the first such algorithm for exact structure learning. The presented work in exact Bayesian learning is intended to push the scale of networks for which optimal structures can be estimated.

Given a set of observations $D_{n \times m}$ and a decomposable scoring function, the BN structure learning problem is to find a DAG N on the n random variables that optimizes $Score\left(N\right)$, or equivalently, finding the parents of each node in the DAG. Adopting a canonical representation of the DAGs in conjunction with a decomposable scoring function permits a dynamic programming

[1] This algorithm was originally presented at *HiPC* (see footnote 4). Here, novel experimental results are reported.

[2] O. Nikolova and S. Aluru. Parallel Discovery of Direct Causal Relations and Markov Blankets. Under review.

(DP) approach first explored by Ott *et al.*[3] Our parallel algorithm is based on the sequential method of Ott *et al.*, which we briefly describe here. Let $A \subset \mathcal{X}$ and $X_i \in \mathcal{X} - A$. Using $D_{n\times m}$, a scoring function $s(X_i, A)$ determines the score of choosing A to be the set of parents for X_i given $D_{n\times m}$. Let $F(X_i, A)$ denote the highest possible score that can be obtained by choosing parents of X_i from A; i.e., $F(X_i, A) = \max_{B\subseteq A} s(X_i, B)$. A set B which maximizes the preceding equation is an optimal parents set for X_i from the candidate parents set A. While $F(X_i, A)$ can be computed by directly evaluating all $2^{|A|}$ subsets of A, it is advantageous to compute it based on the following recursive formulation. For any non-empty set A: $F(X_i, A) = \max\{s(X_i, A), \max_{X_j\in A} F(X_i, A - \{X_j\})\}$. An ordering of a set $A \subseteq \mathcal{X}$ is a permutation $\pi(A)$ of elements in A. A network on A is said to be consistent with a given order $\pi(A)$ if and only if $\forall X_i \in A$, the parents $Pa(X_i)$ precede X_i in π. An optimal order $\pi^*(A)$ is an order with which an optimal network on A is consistent. Let X_i be the last element in $\pi^*(A)$. Then, the permutation $\pi(A - \{X_i\})$ obtained by leaving out the last element X_i is consistent with an optimal network on $A - \{X_i\}$. We write this as $\pi^*(A) = \pi^*(A - \{X_i\})X_i$. Given $A \subseteq \mathcal{X}$ and an order $\pi(A)$, let $Q(A, \pi(A))$ denote the optimal score of a network on A that is consistent with $\pi(A)$: $Q(A, \pi(A)) = \sum_{X_i\in A} F(X_i, \{X_j \mid X_j \text{ precedes } X_i \text{ in } \pi(A)\})$. Optimal score of a network on $A \subseteq \mathcal{X}$ is then given by $Q^*(A) = Q(A, \pi^*(A))$. Our goal is to find $\pi^*(\mathcal{X})$ and the optimal network score $Q^*(\mathcal{X})$, and estimate the corresponding DAG. Like the F function, functions π^* and Q can be computed using a recursive formulation. Consider a subset A. To find $\pi^*(A)$, we consider all possible choices for its last element: $X_i^* = argmax_{X_i\in A} Q(A, \pi^*(A - \{X_i\})X_i) = argmax_{X_i\in A}(F(X_i, A - \{X_i\}) + Q^*(A - \{X_i\}))$. Then: $Q^*(A) = F(X_i^*, A - \{X_i^*\}) + Q^*(A - \{X_i^*\})$, and $\pi^*(A) = \pi^*(A - X_i^*)X_i^*$. By keeping track of the optimal parents sets for each of the variables in $\mathcal{X}$, an optimal network is easily reconstructed. Using these recursive formulations, a DP algorithm to compute an optimal BN can be easily derived. The algorithm considers all subsets of $\mathcal{X}$ in increasing size order, starting from the empty subset. When considering A, the goal is to compute $F(X_i, A)$ for each $X_i \notin A$ and $Q(A, \pi^*(A - \{X_i\})X_i)$ for each $X_i \in A$. All the F and Q^* values required for computing these have already been computed when considering subsets $A - \{X_i\}$ $\forall X_i \in A$.

2 Parallel Algorithm

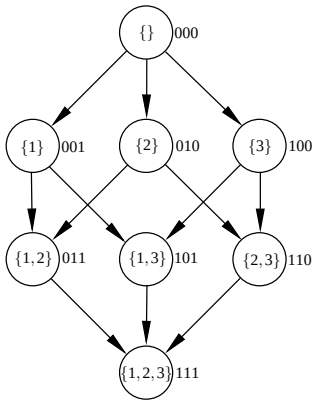


Figure 1: A lattice for 3 variables and its partitioning into 4 levels. The correspondence with a 3-dimensional hypercube is also shown.

To develop the parallel algorithm, it is helpful to visualize the DP algorithm as operating on the lattice L formed by the partial order "set inclusion" on the power set of $\mathcal{X}$. The lattice L is a directed graph (V, E), where $V = 2^{\mathcal{X}}$ and $(B, A) \in E$ if $B \subset A$ and $|A| = |B| + 1$. It is naturally partitioned into levels, where level $l \in [0, n]$ contains all subsets of size l (Fig. 1). A parallel algorithm can be derived by mapping the nodes to processors and letting edges represent

[3]S. Ott, S. Imoto, and S. Miyano. Finding optimal models for small gene networks. In *Pacific Symposium on Biocomputing*, pages $557 - 567$.

communication if the incident nodes are assigned to different processors. A node A at level l has l incoming edges from nodes $A-\{X_i\}$ for each $X_i \in A$, and $n-l$ outgoing edges to nodes $A\cup\{X_i\}$ for each $X_i \notin A$. There are $(n-l)$ F functions, and $Q^*(A)$ and $\pi^*(A)$ computed at node A. All of these values need to be sent along each of the outgoing edges. On an outgoing edge to node $A \cup \{X_i\}$, the $F(X_i, A)$ value is used in computing $Q^*(A \cup \{X_i\})$, and the remaining $F(X_j, A)$ ($X_j \notin A$ and $X_j \neq X_i$) values are used in computing $F(X_j, A \cup \{X_i\})$ values at node $A \cup \{X_i\}$. Note that each of the $(n-l)$ F values at A are used in computing the Q^* value at one of the $n-l$ nodes connected to A by outgoing edges. Each level in the lattice can be computed concurrently, with data flowing from one level to the next.

Mapping to an n-dimensional Hypercube
Observe that the undirected version of L is equivalent to an n-dimensional (n-D) hypercube. Let $(X_1, X_2, \ldots, X_n)$ be an arbitrary ordering of the nodes in the BN. A subset A can be represented by an n-bit string ω, where $\omega[i] = 1$ if $X_i \in A$, and $\omega[i] = 0$ otherwise. As lattice edges connect pairs of nodes that differ in the existence of one element, they naturally correspond to hypercube edges (Fig. 1). This suggests an obvious parallelization on an n-D hypercube. While we expect the number of processors $p \ll 2^n$, we describe this parallelization in slightly greater detail due to its use as a module in the development of our parallel algorithm. The n-D hypercube algorithm runs in $n+1$ steps. Let ω denote the id of a processor and let $\mu(\omega)$ denote the number of 1's in ω. Each processor is active in only one time step – processor ω is active in time step $\mu(\omega)$. It receives $(n-\mu(\omega)+1)$ F values and one Q^* value from each of its $\mu(\omega)$ neighbors obtained by inverting one of its 1 bits to zero. It then computes its own F and Q^* values, and sends them to each of its $n-\mu(\omega)$ neighbors obtained by inverting one of its zero bits to 1.

Partitioning into k-dimensional Hypercubes
Let $p = 2^k$ be the number of processors, where $k < n$. We assume that the processors can communicate as in a hypercube. This is true of parallel computers connected as a hypercube, hypercubic networks such as the butterfly, multistage interconnection networks such as the Omega, and point-to-point communication models such as the permutation network model or the MPI programming model. Our strategy is to decompose the n-D hypercube into 2^{n-k} k-D hypercubes and map each k-D hypercube to the $p = 2^k$ processors as described previously. Although the efficiency of such a mapping by itself is suboptimal ($\Theta(1/k)$), note that each processor is active in only one time step and $p \ll 2^n$. Therefore, we pipeline the execution of the k-D hypercubes to complete the parallel execution in $2^{n-k}+k$ time steps such that all processors are active except for the first k and last k time steps during the build up and finishing off of the pipeline. For convenience of presentation, we use the lattice L and the n-D hypercube interchangeably and use A to denote the lattice node for subset A, and use ω_A to denote the binary string denoting the corresponding hypercube node. We number the positions of a binary string using $1 \ldots n$, and use $\omega_A[i, j]$ to denote the substring of ω_A between and including positions i and j. We partition the n-D hypercube into 2^{n-k} k-D hypercubes based on the first $n-k$ bits of node ids. For a lattice node ω_A, $\omega_A[1, n-k]$ specifies the k-D hypercube it is part of and $\omega_A[n-k+1, n]$ specifies the processor it is assigned to. Conversely, a processor with id r computes for all lattice nodes ω_A such that $r = \omega_A[n-k+1, n]$.

Pipelining Hypercubes
Each k-D hypercube is specified by an $(n-k)$ bit string, which is the common prefix to the 2^k lattice/k-D hypercube nodes that are part of this k-D sub-hypercube. The k-D hypercubes are processed in the increasing order of the number of 1's in their bit string specifications, and in lexicographic order within the group of hypercubes with the same number of 1's. This total order is used to initiate the 2^{n-k} k-D hypercube evaluations, starting from time step 0. If T denotes the time step in which H_i is initiated and A is a lattice node mapped to H_i, then processor with id $\omega_A[n-k+1, n]$ computes for the lattice node A in time step $T + \mu(\omega_A[n-k+1, n])$. This algorithm is correct, work-optimal, and space efficient. (For proofs and details see Nikolova *et al.*[4])

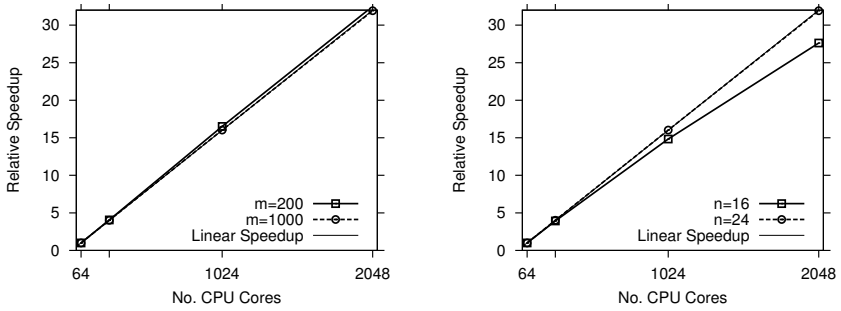
3 Experimental Validation

To assess the performance of the presented algorithm we developed a C++ and MPI implementation. Experiments were performed on a 1,024 dual-core CPU Blue Gene/L (BG/L) and up to 256 8-core

[4] O. Nikolova, J. Zola, and S. Aluru. A parallel algorithm for exact Bayesian network inference. In *HiPC*, pages $342-349$, 2009.

Table 1: Run-time results (in seconds) for: (i) varying number of variables n and fixed number of observations $m = 1000$, and (ii) fixed number of variables $n = 24$ and varying number of observations m. Experimental results are shown on both Blue Gene/L and AMD Opteron Cluster.

Blue Gene/L					AMD Opteron Cluster				
#CPU	$n = 24$		$m = 1000$		#CPU	$n = 24$		$m = 1000$	
Cores	$m = 200$	$m = 1000$	$n = 16$	$n = 24$	Cores	$m = 200$	$m = 1000$	$n = 16$	$n = 24$
					32	166	745	2	745
64	905	4237	12	4237	128	44	193	0.48	193
256	223	1054	3	1054	512	12	53	0.15	53
1024	55	264	0.79	264	1024	8	29	0.08	29
2048	28	133	0.42	133	2048	4	17	0.06	17



(a) Varying sample size with $n = 24$ (b) Varying domain size with $m = 1000$

Figure 2: Relative speedup on Blue Gene/L for $n = 24$ variables and varying number of observations m (a), and varying number of variables n and fixed number of observations $m = 1000$ (b).

nodes of an Infiniband AMD Opteron cluster (AMDO). Our focus is exclusively on performance obtained by the parallel algorithm and the resulting ability to learn larger size networks fast. We evaluated a gene regulatory network learning application on a synthetic experimental data generated using the SynTReN package[5] and our implementation of the MDL scoring criterion. We examine scalability with respect to (i) varying number of observations m for a fixed number of variables $n = 24$, and (ii) varying number of variables n for a fixed number of observations $m = 1000$, on both platforms. Results are summarized in Tab. 1 and the corresponding relative speedup graphs for BG/L is shown in Fig. 2. The number of observations required to derive a meaningful network grows exponentially in the number of variables. Therefore the ability to compute for datasets with large number of observations is important. We test our implementation on both platforms for $m = 200$ and $m = 1000$. The experimental results show that with increasing number of observations, execution time increases linearly on BG/L and approximately linearly on AMDO, while maintaining linear scalability (Fig. 2(a)). This indicates efficient communication as speedup is maintained even in the cases of small number of observations, as computational complexity decreases with decreasing number of observations while communication complexity remains unaffected. In the case of varying number of variables we tested our implementation for $n = 16$ and $n = 24$ while fixing $m = 1000$. The resulting run-times reflect the exponential complexity in the number of variables. As can be expected, better speedup is observed for the larger dataset of $n = 24$ (Fig. 2(b)). Note that memory requirements also grow exponentially in the numbed of variables, which indicates that going parallel is advantageous from the perspective of both run-time and the ability to solve larger problems. Our implementation was capable of learning a network of 33 variables in 1 hour and 14 minutes. Note that Ott *et al.* reported that sequentially a network of 24 variables took multiple days (see footnote 3).

[5]T. Van den Bulcke, K. Van Leemput, B. Naudts, et al. SynTReN: a generator of synthetic gene expression data for design and analysis of structure learning algorithms. BMC Bioinformatics, 7:43, 2006.

4 Conclusions

Heuristics-based BN learning methods trade off optimality for the ability to learn larger networks. This work is intended to push the scale of BN structure learning without additional assumptions. In this paper, we presented a parallel algorithm for exact structure learning that is work-optimal and scalable, which achieves near linear speedup in practice.