Title Page

Parallel Learning of Belief Networks in Large and Difficult Domains

Y. Xiang and T. Chu

Contact author:

Y. Xiang
Associate Professor
Department of Computer Science
University of Regina
Regina, Saskatchewan, Canada S4S 0A2
Phone: (306) 585-4088
Fax: (306) 585-4745
E-mail: yxiang@cs.uregina.ca

Parallel Learning of Belief Networks in Large and Difficult Domains

Y. Xiang
Department of Computer Science, University of Regina
Regina, Saskatchewan, Canada

T. Chu
Avant Corporation, Sunnyvale, CA, USA

Abstract

Learning belief networks from large domains can be expensive even with single-link lookahead search (SLLS). Since a SLLS cannot learn correctly in a class of problem domains, multi-link lookahead search (MLLS) is needed which further increases the computational complexity. In our experiment, learning in some difficult domains over more than a dozen variables took days. In this paper, we study how to use parallelism to speed up SLLS for learning in large domains and to tackle the increased complexity of MLLS for learning in difficult domains.

We propose a natural decomposition of the learning task for parallel processing. We investigate two strategies for job allocation among processors to further improve load balancing and efficiency of the parallel system. For learning from very large datasets, we present a regrouping of the available processors such that slow data access through the file system can be replaced by fast memory access. Experimental results in a distributed memory MIMD computer demonstrate the effectiveness of the proposed algorithms.

`Keywords:` belief networks, parallel implementation of data mining

1 Introduction

Probabilistic belief networks [12, 7] have been widely used for inference with uncertain knowledge in artificial intelligence. As an alternative to elicitation from domain experts, learning belief networks from data has been actively studied [3, 4, 5, 9, 13, 16]. Since the task is NP-hard in general [2], it is justified to use heuristics in learning. Many algorithms developed use a scoring metric combined with a single-link lookahead search (SLLS), where alternative network structures differing from the current structure by *one* link are evaluated exhaustively before one of them is adopted. Although the complexity is polynomial on the number of variables of the problem domain, the computation is still expensive for large domains. Furthermore, a class of domain models termed pseudo-independent (PI) models

cannot be learned correctly by a SLLS [16]. One alternative is to use a multi-link lookahead search (MLLS) [16], where consecutive structures differ by multiple links. However, the complexity of a MLLS is higher. In our experiment (Section 11), learning a 35 variable PI domain model (containing two small PI submodels) took about two and half days, and learning a 16 variable PI domain model (containing a slightly larger PI submodel) took about 25 days.

In this paper, we study parallel learning to speed up computation during SLLS in large domains and to tackle the increased complexity during MLLS in potential PI domains. We focus on learning decomposable Markov networks (DMNs) [16] and show that the lessons we learned are applicable to learning Bayesian networks (BNs) [12]. To the best of our knowledge, this is the first investigation on parallel learning of belief networks. As learning graphical probabilistic models has become an important subarea in data mining and knowledge discovery, this work extends parallel data mining to learning these models. We focus on multiple instruction multiple data (MIMD) distributed-memory architecture for it is available to us, and we discuss the generalization of our lessons to other architectures.

The paper is organized as follows: To make it self-contained, we briefly introduce PI models and MLLS in Sections 2 and 3. In Sections 4 through 9, we propose parallel algorithms for learning DMNs and their refinements. We present experimental results in Sections 10 and 11. Graph-theoretic terms unfamiliar to some readers and a list of frequently used acronyms are included in Appendix.

2 Pseudo-independent models

Let N be a set of discrete variables in a problem domain. A *tuple* of $N' \subseteq N$ is an assignment of values to every variable in N'. A *probabilistic domain model* (PDM) over N determines the probability of every tuple of N' for each $N' \subseteq N$. For disjoint sets X, Y and Z of variables, X and Y are *conditionally independent* given Z if $P(X|Y,Z) = P(X|Z)$ *whenever* $P(Y,Z) > 0$, which we shall denote by $I(X,Z,Y)$. If $Z = \phi$, X and Y are *marginally independent*, denoted by $I(X,\phi,Y)$.

(u,v,x,y)	$P(N)$	(u,v,x,y)	$P(N)$	(u,v,x,y)	$P(N)$	(u,v,x,y)	$P(N)$
(0,0,0,0)	0.0225	(0,1,0,0)	0.0175	(1,0,0,0)	0.02	(1,1,0,0)	0.035
(0,0,0,1)	0.2025	(0,1,0,1)	0.0075	(1,0,0,1)	0.18	(1,1,0,1)	0.015
(0,0,1,0)	0.005	(0,1,1,0)	0.135	(1,0,1,0)	0.01	(1,1,1,0)	0.12
(0,0,1,1)	0.02	(0,1,1,1)	0.09	(1,0,1,1)	0.04	(1,1,1,1)	0.08

Table 1: A PI model

Table 1 shows a PDM over four binary variables. The PDM satisfies $I(u,\{v,x\},y)$. In the subset $\{v,x,y\}$, each pair is marginally dependent, e.g., $P(v,x) \neq P(v)P(x)$, and is dependent given the third, e.g., $P(v|x,y) \neq P(v|y)$. However in the subset $\{u,v,x\}$, although each pair is dependent given the third, e.g., $P(u|v,x) \neq P(u|v)$, we have $I(u,\phi,v)$ and $I(u,\phi,x)$. Hence u, v and x are said to be *collectively dependent* even though u and v are marginally independent (so are u and x). This PDM is a PI model. In general, a PI model is a PDM where proper subsets of a set of collectively dependent variables display marginal

independence [16]. Example PI models include *parity* and *modulus addition* problems [14]. PI models have also been found in real datasets. Analysis of data[1] from 1993 General Social Survey (conducted by Statistics Canada) on Personal Risk has discovered two PI models, one on *harmful drinking* and the other on *accident prevention* [6].

For disjoint subsets X, Y and Z of nodes in an undirected graph G, we use $< X|Z|Y >_G$ to denote that nodes in Z intercept all paths between X and Y. A graph G is an *I-map* of a PDM over N if there is an one-to-one correspondence between nodes of G and variables in N such that for all disjoint subsets X, Y and Z of N, $< X|Z|Y >_G \implies I(X, Z, Y)$. G is a *minimal* I-map if no link can be removed such that the resultant graph is still an I-map. The minimal I-map of the above PDM is shown in Figure 1 (a).

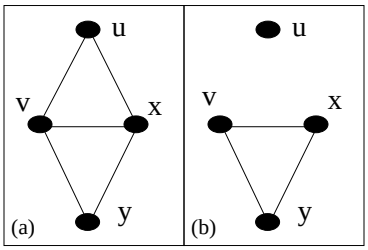

Figure 1: (a) Minimal I-map of PDM in Table 1. (b) Network structure learned by a SLLS.

Several algorithms for learning belief networks have been shown being unable to learn correctly when the underlying PDM is PI [15]. Suppose learning starts with an empty graph (with all nodes but without any link). A SLLS will not connect u and v since $I(u, \phi, v)$. Neither will u and x be connected. This results in the learned structure in Figure 1 (b), which is incorrect. On the other hand, if we perform a double link search after the single-link search, which can effectively test whether $P(u|v, x) = P(u|v)$ holds, then the answer will be negative and the two links (u, v) and (u, x) will be added. The structure in Figure 1 (a) will be learned.

3 A sequential MLLS algorithm

The parallel learning algorithms presented in the paper are based on the sequential MLLS algorithm Seq [16], which learns the structure (a chordal graph) of a DMN using K-L cross entropy [8] as scoring metric. Once the structure is learned, numerical parameters can be easily estimated from the same dataset. Search is organized into *levels* (the outer *for* loop) and the number of lookahead links is identical in the same level. Each level consists of multiple *passes* (the *repeat* loop). In each pass at the same level, alternative structures that differ from the current structure by the same number i of links are evaluated. Search at each pass selects i links that decrease the cross entropy maximally after evaluating all distinct and valid combinations of i links. If the corresponding entropy decrement is significant, the i links will be adopted and the next pass at the same level starts. Otherwise, the first pass at the next higher level starts.

[1]The survey is over 469 variables. Analysis was performed only on data about some subtopics due to limited time. More PI models may be found if analysis is applied to the entire data.

Algorithm 1 (Seq)

Input: A dataset D over a set N of variables, a maximum size η of clique, a maximum number $\kappa \leq \eta(\eta-1)/2$ of lookahead links, and a threshold δh.
begin
  initialize an empty graph $G = (N, E)$, $G' := G$;
  for $i = 1$ to κ, do
    repeat
      initialize the entropy decrement $dh' := 0$;
      for each set L of i links ($L \cap E = \phi$), do
        if $G^* = (N, E \cup L)$ is chordal and
        L is implied by a single clique of size $\leq \eta$, then
          compute the entropy decrement dh^*;
          if $dh^* > dh'$, then $dh' := dh^*$, $G' := G^*$;
      if $dh' > \delta h$, then $G := G'$,done $:= false$; else done $:= true$;
    until done = true;
  return G;
end

Note that each intermediate graph is chordal as indicated by the *if* statement in the innermost loop. The condition that L is implied by a single clique C means that all links in L are contained in the subgraph induced by C. It helps reduce search space. Note also that the algorithm is greedy while the learning problem is NP-hard. Hence, a link committed early in the search is not necessarily contained in a corresponding minimal I-map.

Figure 2 illustrates Seq with a dataset over variables $\{u, v, x, y\}$. A SLLS is performed for simplicity. Search starts with an empty graph in (a). Six alternative graphs in (b) through (g) are evaluated before, say, (b) is selected. The next pass starts with (b) as the current strucuture (redrawn as (h)) and graphs in (i) through (m) are evaluated. Repeating the above process, suppose eventually the graph in (n) is obtained. In the last pass, suppose none of the graphs in (o) and (p) decreases the cross entropy significantly. Then the graph in (n) will be the final result.

4 Task decomposition for parallel learning

In algorithm Seq, for each pass at level 1, $O(|N|^2)$ structures are evaluated before a link is added. $O(|N|^{2m})$ structures are evaluated before m links are added in a pass at level m. To tackle the complexity of MLLS and to speed up SLLS in large domains, we explore parallelism. To this end, we decompose the learning task based on the following observation: At each pass of search, the exploration of alternative structures are coupled only through the current structure. Given the current structure, evaluation of alternative structures are independent, and hence the evaluation can be performed in parallel.

As mentioned earlier, this study is performed using an architecture where processors communicate through message passing (vs. shared memory) only. We partition the pro-

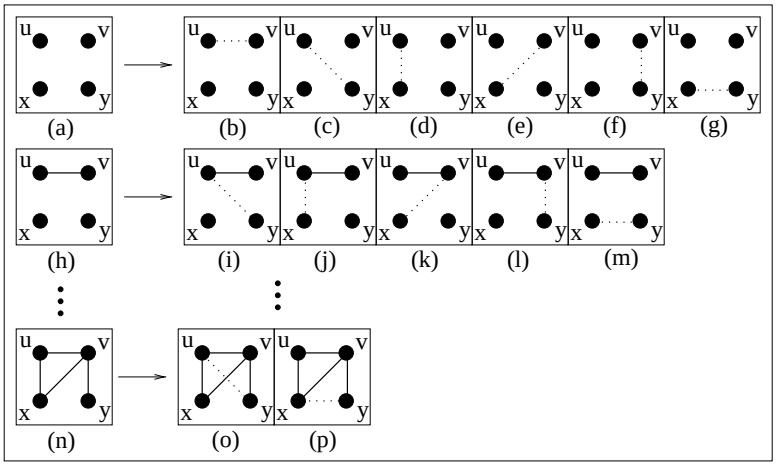

Figure 2: An example of sequential learning

cessors as follows: One processor is designated as the search *manager* and the others are structure *explorers*. The manager executes Mgr1 (Algorithm 2). For each pass, it generates alternative graphs based on the current graph. It then partitions them into n sets and distributes one set to each explorer.

Algorithm 2 (Mgr1)

Input: N, D, η, κ, δh *as algorithm Seq, and the total number* n *of explorers.*
begin
  send N, D *and* η *to each explorer;*
  initialize an empty graph $G = (N, E)$, $G' := G$*;*
  for $i = 1$ *to* κ*, do*
    repeat
      initialize the cross entropy decrement $dh' := 0$*;*
      partition all graphs that differ from G *by* i *links into* n *sets;*
      send one set of graphs and G *to each explorer;*
      for each explorer
        receive dh^* *and* G^**;*
        if $dh^* > dh'$ *then* $dh' := dh^*$, $G' := G^*$*;*
      if $dh' > \delta h$*, then* $G := G'$, *done* $:= false$*; else done* $:= true$*;*
    until done $= true$*;*
  send a halt signal to each explorer;
  return G*;*
end

Each explorer executes Epr1. It checks chordality for each graph received and computes dh^* for each chordal graph. It then chooses the best graph G^* and reports dh^* and G^* to manager. Manager collects the reported graphs from all explorers, selects the global best, and then starts the next pass of search.

Figure 3 illustrates the parallel learning with two explorers and a dataset over variables $\{u, v, x, y\}$. A SLLS is performed for simplicity. Manager starts with an empty graph in (a).

Algorithm 3 (Epr1)

begin
  receive N, D and η from the manager;
  repeat
    receive $G = (N, E)$ and a set of graphs from the manager;
    initialize $dh^* := 0$ and $G^* := G$;
    for each received graph $G' = (N, L \cup E)$, do
      if G' is chordal and L is implied by a single clique of size $\leq \eta$, then compute dh';
      if $dh' > dh^*$, then $dh^* := dh'$, $G^* := G'$;
    send dh^* and G^* to the manager;
  until halt signal is received;
end

It sends six alternative graphs in (b) through (g) to explorers 1 and 2. Explorer 1 checks graphs in (b), (c) and (d). Suppose the one in (b) is selected and reported to manager. Suppose explorer 2 reports the one in (e). After collecting the two graphs, manager chooses the one in (b) as the new current graph. It then sends graphs in (i) through (m). Repeating the above process, manager finally gets the graph in (n) and sends graphs in (o) and (p) to explorers. Suppose none of them decreases the cross entropy significantly. Then manager chooses the graph in (n) as the final result and terminates explorers.

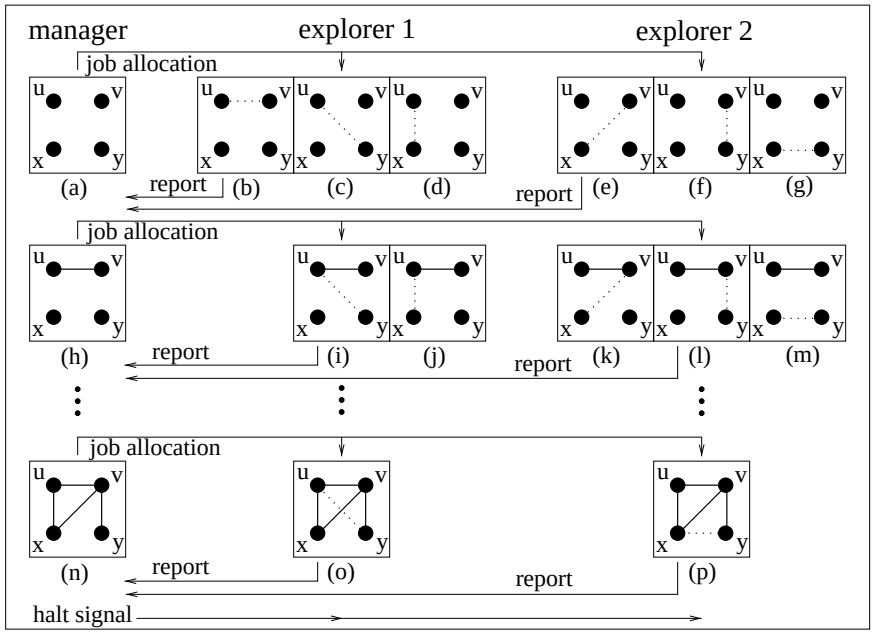

Figure 3: An example of parallel learning

5 Issue of Load Balancing

In algorithm Mgr1, alternative graphs are *evenly* allocated to explorers. However, the amount of computation in evaluating each graph tends to swing between two extremes. If a graph is non-chordal, it is discarded immediately without further computation. On the other hand, if a graph is chordal, its cross entropy decrement will be computed. Figure 4 (a) shows an example graph. There are six supergraphs (graphs with *more* links) that differ by one link. If any of the dotted links in (b) is added to (a), the resultant graph is non-chordal. If any of the dashed links in (c) is added to (a), the resultant graph is chordal. Since the complexity of checking chordality is $O(|N|+|E|)$, where $|E|$ is the number of links in the graph, the amount of computation is very small. Since the complexity of computing cross entropy decrement is $O(|D|+\eta\ (\eta \log \eta + 2^{\eta}))$ [16], where $|D|$ is the number of distinct tuples appearing in the dataset, the amount of computation is much greater. As a result, even job allocation may cause significant fluctuation among explorers in the amount of computation. As manager must collect reports from all explorers before the new current graph can be selected, some explorers will be idle while others are completing their jobs.

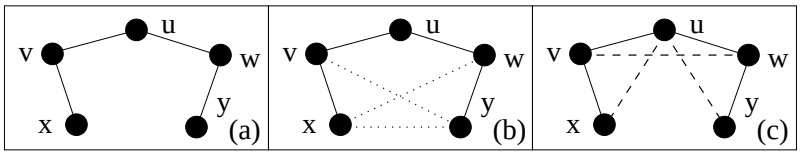

Figure 4: Chordal and nonchordal alternative structures

Figure 5 shows the time taken by each of six explorers in a particular pass in learning from a dataset over 37 variables, where a distributed memory MIMD computer was used. Explorer 1 took much longer than others did.

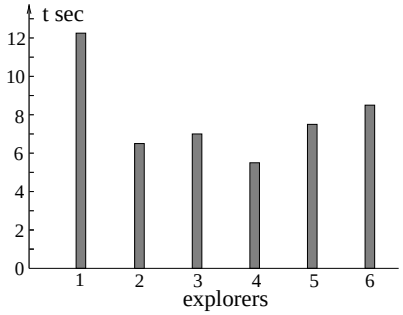

Figure 5: Job completion time of six explorers

The above analysis implies that more sophisticated job allocation strategy is needed to improve the efficiency of the parallel system. In the following sections, we propose two strategies: multi-batch allocation and two-stage allocation.

6 Multi-batch allocation

Multi-batch allocation is based on the idea of keeping some jobs unallocated in the initial allocation and allocating them later to explorers who finish early. The multi-batch

allocation problem can be abstracted as follows:

Let L_0 be the total number of job units, each of which corresponds to a graph to be evaluated. A job unit is either of type 0 (non-chordal) or of type 1 (chordal). It takes time T_0 to process a unit of type 0 job and T_1 for that of type 1. After an explorer has finished a given batch of job units, it takes time T_c to send another batch of job units (by one message) to the explorer. We shall refer to any batch sent to an explorer after the first batch as an *additional* batch. The goal is to find the proper size of each batch such that the sum of idle time of all explorers is reduced during the completion of L_0 job units.

In deriving the batch sizes, we make the following assumptions:

Assumption 1 *T_0 and T_c are constants in a pass.*

T_0 is the computation time to test the chordality of a graph. Since the complexity of checking chordality is $O(|N| + |E|)$, and each graph in the same pass has the identical number of nodes and links, T_0 can be treated as a constant.

T_c is the time for manager to send an additional batch to an explorer. An additional batch is much smaller (as will be seen) than the first batch. A message for an additional batch is thus very short. Messages are sent through communication channels ($> 10M$ bps) within the parallel computer, and the actual data transfer is very fast. Consequently, T_c consists mainly of handshaking time and only varies slightly from message to message.

Assumption 2 *T_1 is a constant in a pass and is much larger than T_0 and T_c.*

T_1 is the computation time to process one unit of type 1 job which involves checking the chordality of a given graph and computing the cross entropy decrement of a chordal graph. It is much larger than T_0 and T_c. For example, in learning from a database with 37 variables, we found T_0 to be between 0.007 to 0.009 seconds and T_c about 0.017 seconds in our parallel computing environment. T_1 was at least 0.06 seconds. However, the assumption that T_1 is a constant is less accurate. When the variation of clique sizes in a chordal graph is small, T_1 tends to be close to a constant. When the variation is large, T_1 tends to vary depending on specific job unit. Still, we found the assumption to be a useful approximation in deriving a simple method to determine the batch size.

Suppose the first batch allocated to each explorer has J_0 ($< L_0/n$) units. Let Q_i (B_i) denote the number of type 1 (0) units in the batch assigned to explorer i. Let Q denote the total number of type 1 units in the n batches. Let $\beta_i = Q_i/J_0$ be the percentage of type 1 units in the batch to explorer i. Let $\beta = Q/(n\ J_0)$ be the percentage of type 1 units in the n batches. Without losing generality, suppose $\beta_1 = \max_{i=1}^{n}(\beta_i)$ and we alternatively denote β_1 by β_{max}.

The time t_i taken by explorer i to process its first batch is

$$t_i = Q_i T_1 + B_i T_0 = \beta_i J_0 T_1 + (1 - \beta_i) J_0 T_0 = J_0(\beta_i(T_1 - T_0) + T_0). \tag{1}$$

Let T be the sum of the idle time of explorers 2 through n while explorer 1 is processing its first batch. We can derive

$$\begin{aligned} T &= \sum_{i=2}^{n}(t_1 - t_i) = \sum_{i=2}^{n} J_0((\beta_{max}(T_1 - T_0) + T_0) - (\beta_i(T_1 - T_0) + T_0)) \\ &= \sum_{i=2}^{n} J_0\beta_{max}(T_1 - T_0) - \sum_{i=2}^{n} J_0\beta_i(T_1 - T_0). \end{aligned} \tag{2}$$

Substituting $\sum_{i=2}^{n} J_0\beta_i = Q - Q_1 = nJ_0\beta - J_0\beta_{max}$ in equation (2), we have

$$\begin{aligned} T &= (n-1)J_0\beta_{max}(T_1 - T_0) - (nJ_0\beta - J_0\beta_{max})(T_1 - T_0) \\ &= nJ_0(\beta_{max} - \beta)(T_1 - T_0). \end{aligned} \tag{3}$$

To make use of the idle time T, we allocate the $L_0 - nJ_0$ (denoted by L_1) reserved job units in additional batches to explorers who finish their first batches before explorer 1. Denote the percentage of type 1 jobs in the L_1 units by β_r. Ideally, the L_1 units should be allocated to explorers 2 through n such that they will be fully engaged during the $[0, t_1]$ time period and all L_1 units will be completed at time t_1. Using the result in equation (1), this condition can be expressed as

$$T = L_1(\beta_r(T_1 - T_0) + T_0) + MT_c \tag{4}$$

where M is the total number of additional batches to allocate the L_1 units. The value of M depends on the actual size of each batch (including J_0) and its estimation will be discussed shortly.

Equations (3) and (4) imply

$$(L_0 - nJ_0)(\beta_r(T_1 - T_0) + T_0) + MT_c = nJ_0(\beta_{max} - \beta)(T_1 - T_0). \tag{5}$$

Solving equation (5), J_0 can be expressed as

$$J_0 = \frac{L_0(\beta_r(T_1 - T_0) + T_0) + MT_c}{n((\beta_{max} - \beta + \beta_r)(T_1 - T_0) + T_0)}. \tag{6}$$

To compute J_0, we need the values for β, β_{max}, β_r and M. However, they are unknown at the beginning of the search pass when J_0 is to be computed. The estimation of these values is discussed below:

The values of β and β_{max} can be estimated based on the following assumption:

Assumption 3 *The difference between the values of β (β_{max}) in successive search passes is small.*

Assumption 3 usually holds since the graphs involved in successive passes differ by only i links. Figure 6 shows the values of β and β_{max} from search pass 5 to 75 in learning from a dataset of 37 variables, which provides an empirical justification of the assumption.

The value of β_r usually varies from $\beta_{min} = \min_{i=1}^{n}(\beta_i)$ to β_{max}. We can approximate β_r of equation (6) by the average $\beta_{avg} = 0.5(\beta_{min} + \beta_{max})$.

By equation (6), estimation errors in β, β_{max} and β_r can make J_0 smaller or larger than the ideal value. If J_0 is smaller, more units will be reserved, resulting in more additional batches. On the other hand, if J_0 is larger, less units will be reserved and some explorers will be idle after all units have been allocated.

Finally, we consider the estimation of M. From the numerator of equation (6), the effect of estimation error in M is small because $\beta_r(T_1 - T_0) + T_0$ is larger than T_c and L_0 is much larger than M.

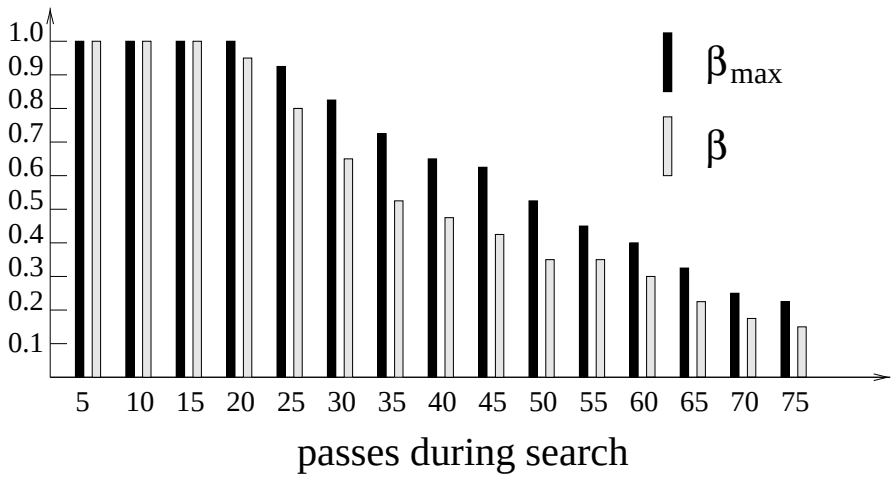

Figure 6: β and β_{max} values obtained with eight explorers

Based on Assumption 3 and the above analysis, manager can collect the values β', β'_{avg}, β'_{max} and M' from the previous pass of search to calculate the value of J_0 as follows:

$$J_0 \approx \frac{L_0(\beta'_{avg}(T_1 - T_0) + T_0) + M'T_c}{n((\beta'_{max} + \beta'_{avg} - \beta')(T_1 - T_0) + T_0)}. \tag{7}$$

We have now determined the size of the first batch to each explorer.

Next, we determine the size for additional batches. As an example, consider a situation illustrated by Figure 5. Suppose that the histogram depicts the computation time of the first batch by each explorer. Explorer 4 finishes the first. Let J_1 be the size of the second batch allocated to explorer 4. The most conservative batch size is $J_1 = L_1/(n-1)$, which effectively assumes that every explorer (other than explorer 1) finishes at this moment. Usually other explorers will finish later and hence this size will under-allocate for explorer 4. However, the under-allocation will only slightly increases the number M of additional batches. Since T_c is very small, a few more additional batches will not affect the overall efficiency significantly. We have therefore adopted this conservative batch size.

In general, let L_2 be the remaining job units after the allocation of a batch of J_1 units to the explorer that finishes the first, L_3 be the remaining job units after the allocation of a batch of J_2 units to the explorer that finishes the second, and so on. The batch size allocated to the explorer that finishes the ith place will be

$$J_i = \begin{cases} \frac{L_i}{n-1} & \text{when } L_i \geq 2(n-1) \\ 1 & \text{when } L_i < 2(n-1) \end{cases} \tag{8}$$

where $i = 1, 2, \ldots$, and $L_{i+1} = L_i - J_i$. Note that after the number of remaining units drops below $2(n-1)$, jobs are allocated unit by unit to achieve high degree of load balancing.

Based on equations (7) and (8), we modify Mgr1/Epr1 into algorithms Mgr2/Epr2.

Manager executes Mgr2. For each pass, it computes J_0 according to equation (7), and then sends the current graph G and a batch of J_0 graphs to each explorer. Each explorer executes Epr2. It checks chordality for each graph received and computes the entropy decrement for each chordal graph. The explorer then sends a signal to manager indicating

Algorithm 4 (Mgr2)

Input: N, D, η, κ, δh *and* n.
begin
    send N, D *and* η *to each explorer;*
    initialize an empty graph $G = (N, E)$, $G' := G$;
    set initial values for β, β_{max}, β_{avg}, T_0, T_1, T_c *and* M;
    for $i = 1$ *to* κ, *do*
        repeat
            initialize the cross entropy decrement $dh' := 0$; $j := 0$;
            send current graph G *and* J_j *graphs to each explorer;* j++;
            repeat
                receive a completion signal from an explorer;
                if $L_j > 0$, *then send* J_j *graphs to the explorer;* j++;
                else send a report signal to the explorer;
            until report signal has been sent to each explorer;
            for each explorer x, *do*
                receive dh^*, β_x, T_0, T_1 *and* G^*;
                if $dh^* > dh'$, *then* $dh' := dh^*$, $G' := G^*$;
            if $dh' > \delta h$, *then* $G := G'$, $done := false$; *else* $done := true$;
            if $done = true$, *then update* β, β_{max}, β_{avg}, T_0, T_1 *and* $M = j$;
        until $done = true$;
    send a halt signal to each explorer;
    return G;
end

its completion of the batch. Upon receiving the signal, manager computes size J_j for an additional batch and sends the batch to the explorer. If no job units are left for this pass, manager will signal the explorer for report. After reports are collected from all explorers, manager updates the relevant search parameters and starts the next pass. Note that both T_0 and T_1 are updated to account for the inaccuracy of Assumptions 1 and 2.

7 Two-stage allocation

The two-stage allocation is based on the fact that a chordal structure and a non-chordal one require significantly different amount of computation in evaluation, and the difference is the major source of unbalanced load amount processors in even allocation.

To improve load balancing, we modify even job allocation of Mgr1/Epr1 by allocating jobs in two stages as shown in algorithms Mgr3/Epr3. In the first stage, manager (see Mgr3) partitions alternative graphs *evenly* and distributes one set to each explorer. Each explorer (see Epr3) checks the chordality for each graph received and reports to manager valid candidates (chordal graphs). Since the complexity of checking chordality is $O(|N| + |E|)$, and each graph has the identical number of nodes and links, the computation among

Algorithm 5 (Epr2)

begin
    receive N, D *and* η *from manager;*
    repeat
        receive $G = (N, E)$ *from manager;*
        initialize $dh^* := 0$ *and* $G^* := G$*;*
        repeat
            receive a set of graphs from manager;
            for each received graph $G' = (N, L \cup E)$*, do*
                if G' *is chordal and* L *is implied by a single clique of size* $\leq \eta$*,*
                    then compute the entropy decrement dh'*;*
                if $dh' > dh^*$*, then* $dh^* := dh'$*,* $G^* := G'$*;*
            send a completion signal to manager;
        until report signal is received;
        send dh^**,* β_x*,* T_0*,* T_1 *and* G^* *to manager;*
    until halt signal is received;
end

explorers is *even*.

In the second stage, manager partitions all received graphs *evenly* and distributes one set to each explorer. Each explorer computes entropy decrement for each graph received. It then chooses the best graph and reports it and its entropy decrement to manager. Manager collects the reported graphs, selects the best, and then starts the next pass. Since all graphs are chordal in the second stage, the degree of load balance mainly depends on the variability of the sizes of the largest cliques.

8 Comparison of allocation strategies

Compared with multi-batch allocation, two-stage allocation is much simpler. It only needs to partition and distribute job units twice. With the multi-batch allocation, multiple batches are sent to each explorer, resulting higher communication overhead. For example, in learning from a database of 37 variables with 12 explorers, we found that on average six batches are sent to each explorer. The data collection and computation involved in multi-batch allocation are also more expensive.

However, two-stage allocation suffers from variation in the amount of computation for calculating entropy decrements as each set L of new links forms new cliques whose sizes may vary significantly. On the other hand, the multi-batch allocation has the resistance to the variation in clique size since allocation is dynamically adapted to the *actual* amount of computation used for each batch.

We present the experimental comparison of the two strategies in Section 11.

Algorithm 6 (Mgr3)

Input: N, D, η, κ, δh *and* n.
begin
    send N, D *and* η *to each explorer;*
    initialize an empty graph $G = (N, E)$, $G' := G$;
    for $i = 1$ *to* κ, *do*
        repeat
            initialize $dh' := 0$;
            partition all graphs that differ from G *by* i *links into* n *sets;*
            send one set of graphs and G *to each explorer;*
            receive a set of valid graphs from each explorer;
            partition all received graphs into n *sets;*
            send one set of graphs to each explorer;
            for each explorer, do
                receive dh^* *and* G^*;
                if $dh^* > dh'$, *then* $dh' := dh^*$, $G' := G^*$;
            if $dh' > \delta h$, *then* $G := G'$, *done* $:= false$; *else done* $:= true$;
        until done $= true$;
    send a halt signal to each explorer;
    return G;
end

Algorithm 7 (Epr3)

begin
    receive N, D *and* η *from manager;*
    repeat
        receive current graph $G = (N, E)$ *and a set of graphs from manager;*
        initialize $dh^* := 0$ *and* $G^* := G$;
        for each received graph $G' = (N, L \cup E)$, *do*
            if G' *is chordal and* L *is implied by a single clique of size* $\leq \eta$,
                then mark it as valid;
        send all valid graphs to manager;
        receive a set of graphs from manager;
        for each received graph G', *do*
            compute the entropy decrement dh';
            if $dh' > dh^*$, *then* $dh^* := dh'$, $G^* := G'$;
        send dh^* *and* G^* *to manager;*
    until halt signal is received;;
end

9 Marginal Servers

In order to learn a belief network with satisfactory accuracy, a dataset of large number of cases is preferred. During learning, the data will be frequently accessed by each explorer to obtain marginal probability distributions (marginals) of subsets of variables (for computing entropy decrements). Using a distributed-memory architecture, the available local memory to each processor is limited. If the dataset (with a proper compression) can be fit into the local memory such that each processor has one copy of the dataset, then data can be accessed effectively during learning. Otherwise, special measure has to be taken for data access.

One obvious solution is to access data through the file system. However file access is much slower than memory access. Even worse, many parallel computers have limited channels for file access, making it a bottleneck. For example, in the computer available to us, file access by all processors must be performed through a single host computer.

To achieve efficient data access, we propose an alternative using so called *marginal servers* to avoid file access completely during learning. The idea is to split the dataset so that each subset can be stored into the local memory of a processor. A group of (say m) such processors is then given the task of serving explorers in computing partial marginals from their local data.

In particular, the m servers are connected *logically* into a pipeline. The dataset is partitioned into $m + 1$ sets, where the size of each set may not be identical as we will discuss shortly. Each server stores one *distinct* set of data and each explorer *duplicates* one copy of the remaining set.

As an example, consider the computation of the marginal over two binary variables $\{x, y\} \subset N$. Suppose $|D| = 10000$ and there are one explorer and two marginal servers. We store 5000 tuples in the explorer and 2500 in each server. Table 2 shows one possible scenario of how the tuples might be distributed according to $\{x, y\}$.

(x, y)	tuples in explorer	tuples in server 1	tuples in server 2
(0, 0)	2000	1000	500
(0, 1)	1500	500	1000
(1, 0)	1000	500	500
(1, 1)	500	500	500

Table 2: Data storage using servers

When the explorer needs to compute the marginal over $\{x, y\}$, it first sends $\{x, y\}$ to servers, and then computes locally the *potential* (non-normalized distribution) $(2000, 1500, 1000, 500)$. Requested by the explorer, server 1 computes the local potential $(1000, 500, 500, 500)$ and sends to server 2. Server 2 computes its local potential, adds to the result from server 1 to obtain the sum $(1500, 1500, 1000, 1000)$, and sends the sum to the explorer. The explorer adds the sum to its local potential to obtain $(3500, 3000, 2000, 1500)$ and normalizes to get the marginal $(0.35, 0.3, 0.2, 0.15)$.

Two-stage allocation enhanced by marginal servers is implemented in Mgr4, Epr4 and Svr. Multi-batch allocation can be enhanced accordingly.

Algorithm 8 (Mgr4)

Input: N, D, η, κ, δh, n *and* m.
begin
    partition D *into* $m+1$ *sets;*
    send one set to each server and broadcast the last set to explorers;
    initialize an empty graph $G = (N, E)$, $G' := G$;
    for $i = 1$ *to* κ, *do*
        repeat
            initialize $dh' := 0$;
            partition all graphs that differ from G *by* i *links into* $m + n$ *sets;*
            send one set of graphs and G *to each explorer and each server;*
            receive a set of valid graphs from each explorer and each server;
            partition all received graphs into n *sets;*
            send one set of graphs to each explorer;
            for each explorer, do
                receive dh^* *and* G^*;
                if $dh^* > dh'$ *then* $dh' := dh^*$, $G' := G^*$;
            if $dh' > \delta h$, *then* $G := G'$, $done := false$; *else* $done := true$;
            send an end-of-pass signal to each server;
        until $done = true$;
    send a halt signal to each explorer and each server;
    return G;
end

Manager executes Mgr4. It partitions data into $m+1$ sets, distributes to explorers and servers, and starts the search process. In the first stage of each pass, manager generates alternative graphs based on the current graph. It partitions them into $m+n$ sets, distributes to explorers and servers, and receives reported valid graphs. In the second stage, manager partitions valid graphs into n sets and sends one set to each explorer.

Each explorer executes Epr4. In the first stage of each pass, it checks the chordality of each received graph and reports valid graphs to manager. In the second stage, the explorer receives a set of valid graphs from manager. For each graph received, it identifies the marginals (each over a subset $C \subset N$) necessary in computing entropy decrement. For each marginal, it sends a request to servers, computes a local potential, receives a potential from a server (to be specified below), sums them up and obtains the marginal. After evaluating all valid graphs received, the explorer chooses the best graph and reports to manager. Manager collects reported graphs from all explorers, selects the best as the new current graph, sends a signal to each server to notify the end of the current pass, and then starts the next pass.

Each marginal server executes Svr. In the first stage of each pass, each server functions as an explorer (testing chordality). In the second stage, a server processes requests repeatedly until it receives a signal to end the current pass. For each request (a marginal over a subset $C \subset N$), a server computes a local potential, adds to the potential from its

Algorithm 9 (Epr4)

begin
  receive η *and a subset of data over* N*;*
  repeat
    receive $G = (N, E)$ *and a set of graphs from manager;*
    initialize $dh^* := 0$ *and* $G^* := G$*;*
    for each received graph $G' = (N, L \cup E)$*, do*
      if G' *is chordal and* L *is implied by a clique of size* $\leq \eta$*, then mark* G' *valid;*
    send all valid graphs to manager;
    receive a set of graphs from manager;
    for each received graph $G' = (N, L \cup E)$*, do*
      for each set C *of variables involved in computing* dh'*, do*
        send C *to each marginal server;*
        compute local potential over C*;*
        receive a potential over C *from a server;*
        compute marginal over C*;*
      compute the entropy decrement dh'*;*
      if $dh' > dh^*$*, then* $dh^* := dh'$*,* $G^* := G'$*;*
    send dh^* *and* G^* *to manager;*
  until halt signal is received;;
end

predecessor if it is not the head of the pipeline, and sends the sum to the next server or the requesting explorer depending on whether it is the end of the pipeline.

To keep all processors fully engaged, the dataset D must be properly partitioned among explorers and servers. Since each server serves n explorers, the processing of one request by a server must be n times as fast as the local processing of a requesting explorer. This implies $nT_s = T_e$, where T_s and T_e are the time to process one marginal request by a server and an explorer, respectively. Let $|D_s|$ and $|D_e|$ be the number of tuples stored locally in each server and each explorer, respectively. T_s and T_e can be expressed as $T_s = k_d|D_s|$ and $T_e = k_g|N| + k_d|D_e|$, where k_d and k_g are coefficients, and $k_g|N|$ is the computation time to identify the marginals necessary in computing entropy decrement. Therefore, we have

$$nk_d|D_s| = k_g|N| + k_d|D_e|. \tag{9}$$

In algorithm Mgr4, D is partitioned into $m + 1$ sets and hence

$$|D| = m|D_s| + |D_e|. \tag{10}$$

Denoting $p = m + n$ and solving equations (9) and (10), we obtain

$$n = \frac{p(\alpha|N| + |D_e|)}{\alpha|N| + |D|}, \qquad m = \frac{p(|D| - |D_e|)}{\alpha|N| + |D|}, \qquad |D_s| = \frac{\alpha|N| + |D|}{p},$$

where $\alpha = k_g/k_d$ with its value between 0.003 to 0.006 in our experimental environment. In practice, m, n, $|D_s|$ and $|D_e|$ must be rounded to integers, and $|D_e|$ must be upper bounded

Algorithm 10 (Svr)

begin
    receive η *and a subset of data over* N*;*
    repeat
        receive $G = (N, E)$ *and a set of graphs from manager;*
        for each received graph $G' = (N, L \cup E)$*, do*
            if G' *is chordal and* L *is implied by a clique of size* $\leq \eta$*, then mark* G' *valid;*
        send all valid graphs to manager;
        repeat
            receive a set C *of variables from an explorer;*
            compute local potential over C*;*
            if this server is not head of server pipeline, then
                receive a potential over C *from predecessor server;*
                sum local potential with received potential;
            if this server is not tail of server pipeline, then send sum to the next server;
            else send sum to the requesting explorer;
        until end-of-pass signal is received;;
    until halt signal is received;;
end

by the available local memory for data storage. As an example, suppose $|D| = 100k$, $|D_e| = 20k$, $|N| = 1000$, $p = 30$ and $\alpha = 0.005$. We have $n = 6$, $m = 24$ and $|D_s| \approx 3.334k$.

10 Experimental environment

The parallel algorithms presented have been implemented on an ALEX AVX Series 2 distributed memory MIMD computer. It contains 8 root nodes and 64 compute nodes, which may be partitioned among and used by multiple users at any time. Each root node is a $T805$ processor, which can be used to control the topology of compute nodes. Each compute node consists of an $i860$ processor ($40Mhz$) for computation and a $T805$ processor for message passing with other nodes through four channels at each node. Data transmission rate is $10Mbps$ in simplex mode and $20Mbps$ in duplex mode. The $i860$ and $T805$ processors at each node share $32MB$ memory and the latter has its own additional $8MB$ memory. All access to the file system is through a root node and a host computer.

We configure the available processors into a ternary tree (Figure 7) to reduce the length of message passing path. The root is manager and non-root nodes are explorers/servers. Servers cooperate *logically* as a pipeline.

We tested our implementation using the $ALARM$ network [1] and four randomly generated networks $PIMi$ ($i = 1, ..., 4$) each of which is a PI model. $ALARM$ has 37 variables. $PIM1$ has 26 variables and contains an embedded PI submodel over three variables. $PIM2$ has 30 variables and contains two embedded PI submodels each of which is over three variables. $PIM3$ has 35 variables and contains two embedded PI submodels similar to those

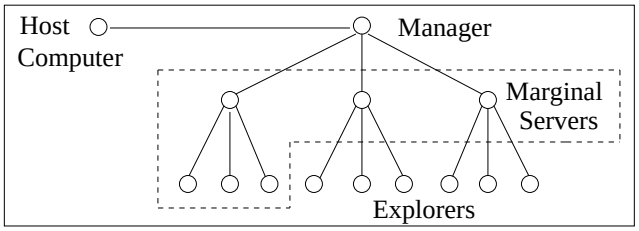

Figure 7: Ternary tree topology

of $PIM2$. $PIM4$ has 16 variables and contains one embedded PI submodel over four variables. Five datasets are generated by sampling the five control networks with 10000, 20000, 25000, 30000 and 10000 cases, respectively.

We measure the performance of our programs by *speed-up* (S) and *efficiency* (E). Given a task, let $T(1)$ be the execution time of a sequential program and $T(n)$ be that of a parallel program with n processors. Then $S = T(1)/T(n)$ and $E = S/n$.

11 Experimental results

We demonstrate the performance of multi-batch and two-stage allocation strategies and the benefit of using marginal servers.

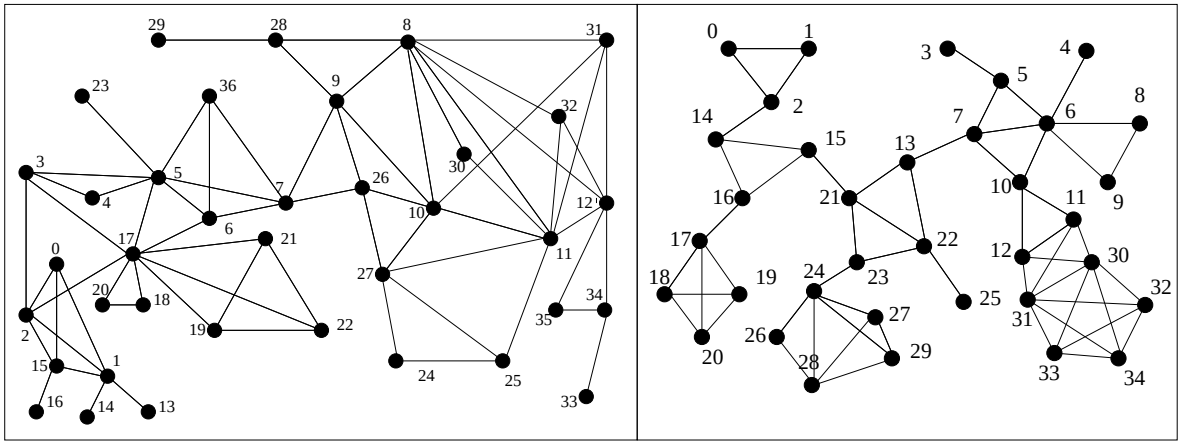

Figure 8: DMNs learned from data obtained from ALARM (left) and PIM3 (right)

The DMN learned from $ALARM$ dataset is shown in Figure 8 (left). Since the task decomposition that we used for parallelism does not introduce errors, the learning outcome is *identical* to what is obtained by Seq with the same learning parameters. Figure 8 (right) shows the DMN learned from $PIM3$ dataset. Nodes labeled 6, 8 and 9 form a PI submodel in $PIM3$ and so do nodes labeled 14, 15 and 16.

In learning from the $ALARM$ dataset, we compared even (Mgr1/Epr1), multi-batch (Mgr2/Epr2) and two-stage (Mgr3/Epr3) allocations. The dataset, after compression, was loaded into the local memory of each explorer. Table 3 shows experimental results for even and two-stage allocations as the number n of explorers increases from 1 to 12.

n	Even allocation			Two-stage allocation		
	time(s)	speed-up	efficiency	time(s)	speed-up	efficiency
1	3160	1.0	1.0	3160	1.0	1.0
2	1750	1.81	0.903	1617	1.95	0.977
4	957	3.30	0.825	850	3.72	0.929
6	712	4.44	0.740	609	5.19	0.865
8	558	5.66	0.708	472	6.69	0.837
10	486	6.50	0.650	393	8.04	0.804
12	454	6.96	0.580	351	9.00	0.750

Table 3: Experimental results for even and two-stage allocations

Columns 3 and 6 show that as n increases, speed-up increases as well when either allocation strategy is used. This demonstrates that the parallel algorithms can effectively reduce learning time and provides positive evidence that parallelism is an alternative to tackle the computational complexity in learning belief networks.

Comparing column 3 with 6 and column 4 with 7, it can be seen that two-stage allocation further speeds up learning and improves efficiency beyond that of even allocation. For example, when eight explorers are used, speed-up is 5.66 and efficiency is 0.708 for even allocation, and 6.69 and 0.837 for two-stage. Figure 9 plots the speed-up and efficiency for all three strategies for comparison.

Among the three strategies, even allocation has the lowest speed-up and efficiency, especially when n increases. There is no significant difference between multi-batch and two-stage allocations. For $n > 6$, multi-batch allocation is slightly better than two-stage allocation. As n increase beyond 9, two-stage performs better than multi-batch. This is because the overhead of multi-batch job allocation becomes more significant as the number of explorers increases.

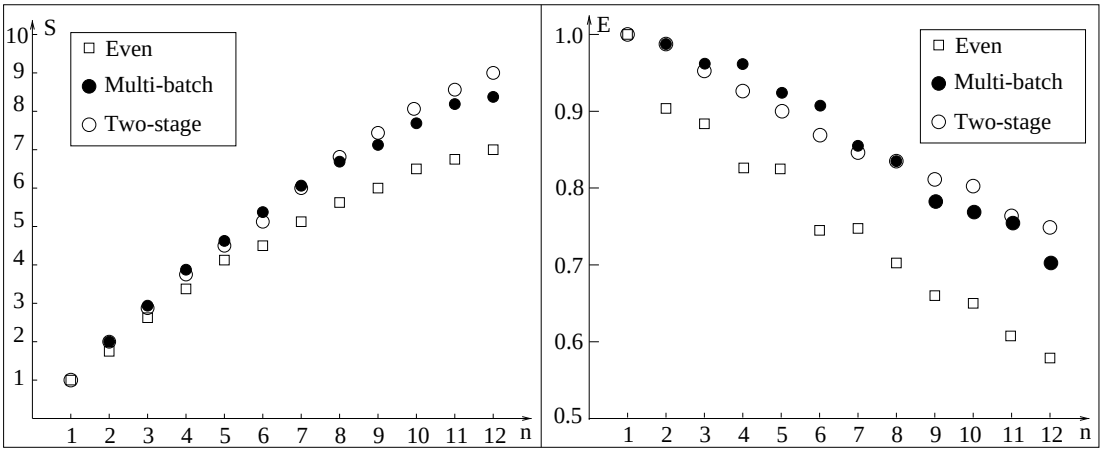

Figure 9: Speed-up (left) and efficiency (right) in learning from ALARM dataset

The results also show a gradual decrease in efficiency as n increases. This decrease is

due to allocation overhead. At the start of each pass, manager allocates jobs to explorers in sequence. Hence an explorer is idle between submission of its report in previous pass and receipt of the next batch of jobs. However, efficiency decrease will be less significant when learning is performed in large or PI domains as the proportion of message passing time in each pass will be much smaller than computation time. This is illustrated by our learning results in PI domains as follows:

n		PIM1	PIM2	PIM3	PIM4
1	Time (min)	262.4	868.6	3555.4	36584
12	Time (min)	26.8	89.3	352.2	3382
	Speed-up	9.8	9.7	10.1	10.8
	Efficiency	0.82	0.81	0.84	0.90
24	Time (min)	17.2	54.2	179.4	1735
	Speed-up	15.3	16.0	19.8	21.1
	Efficiency	0.64	0.67	0.83	0.88
36	Time (min)	12.5	37.7	124.5	1197
	Speed-up	21.0	23.0	28.6	30.6
	Efficiency	0.58	0.64	0.79	0.85

Table 4: Experimental results in learning PI models

Table 4 shows experimental results for learning PI models $PIMi$ $(i = 1, ..., 4)$, where triple-link lookahead is used for learning $PIMi$ $(i = 1, ..., 3)$ and six-link lookahead is used for learning $PIM4$. The first column indicates the number of explorers used. As expected, speed-up is shown to increase with n.

The third column shows results in learning $PIM1$. When 12 explorers are used, speed-up and efficiency are 9.8 and 0.82. The table shows rapid decrease of efficiency when 24 and 36 explorers are used. The similar trend can be seen in column 4 for learning $PIM2$. This is because the two domains are relatively small (with 20 and 30 variables, respectively) and less complex (sparse, and with one and two small PI submodels, respectively). Message passing time is significant compared with computation time in these cases.

Column 5 shows results for learning $PIM3$. The domain contains 35 variables and two PI submodels, and the control network is more densely connected. Significantly longer computation time (3555.4 min) was used by the sequential program. The last column shows results for learning $PIM4$. Although its domain is not large (16 variables), the fact that it contains a PI sub-model with 4 variables and a six-link lookahead is needed to identify the sub-model makes its computation expensive. It took the sequential program over 25 days (36584 min). Compared with $PIM1$ and $PIM2$, speed-up and efficiency in learning these two models are much better when larger number of explorers are used. Note that with 36 explorers, the time to learn $PIM4$ is reduced from over 25 days to less than one day (1197 min).

Finally, we demonstrate the use of marginal servers by learning the $ALARM$ network. Although $ALARM$ is not very large and the dataset can be loaded entirely into the local memory of each explorer, we choose to use it for two reasons: First, domain size does

not hinder demonstration of correctness of the server method. Second, if we decrease the available local memory below what we have, at some point, it would not be large enough to hold $ALARM$ dataset. In that case, data access by file system would be necessary if the server method were not used. Hence, generality is not compromised by using $ALARM$.

To demonstrate the effect of using servers, we assume that the dataset cannot be loaded into local memory of explorers. Using data access by file system, it took $12780sec$ for the sequential program to complete learning $ALARM$. Table 5 shows results of learning $ALARM$ by using $m = 4$ servers. The number n of explorers ranges from one to eight. The data size stored in each explorer was twice as large as that in each server. Note that since marginal servers replace slow file access by fast memory access, the efficiency can be larger than 1.0 as shown in the table.

n+m	5	6	7	8	9	10	11	12
Time (s)	2870	1616	1166	1015	910	819	762	737
Speed-up	4.45	7.91	10.96	12.59	14.04	15.60	16.77	17.34
Efficiency	0.891	1.318	1.566	1.574	1.560	1.560	1.525	1.445

Table 5: Experimental results by using four marginal servers

12 Looking beyond distributed memory MIMD

Flynn's taxonomy [11] classifies hardware into SISD, SIMD, MISD and MIMD. MIMD computers can be further classified into shared or distributed memory. The following discussion extends our lessons from using distributed memory MIMD to the suitability of other architectures for parallel learning of belief networks. As SISD is incapable of true parallelism [10], we discuss only SIMD, MISD and MIMD.

An MISD computer applies multiple instructions to a single data stream. For example, it can perform matrix operations $\mathbf{A} + \mathbf{B}$ and $\mathbf{A} - \mathbf{B}$ simultaneously. The task of learning belief networks decomposes naturally into evaluation of alternative network structures (multiple data streams) as we have investigated in this study. Therefore, the MISD architecture appears unsuitable for this task.

SIMD computers consist of multiple arithmetic logic units (ALUs) under the supervision of a single control unit (CU). CU synchronizes all the ALUs by broadcasting control signals to them. The ALUs perform the same instruction on different data that each of them fetches from its own memory. For instance, CM-2 connect machine has $64K$ processors each of which is an one-bit CPU with $256K$ one-bit memory. Normal instructions are executed by a host computer and the vector instructions are broadcast by the host computer and executed by all processors.

In learning belief networks, each alternative network structure has a unique graphical topology and requires a unique stream of instructions for its evaluation. Therefore, SIMD computers do not appear suitable if the learning task is decomposed at the level of network structures. In other words, it appears necessary to decompose the task at a much *lower*

abstraction level. One alternative is to partition the dataset into small subsets each of which is then loaded into the memory of one processor. Each marginal can then be computed by cooperation of multiple processors when requested by a host computer. However, the host computer must carry out all other major steps in evaluating each alternative structure. This is essentially the sequential learning (algorithm Seq) with parallelism applied to only marginal computation. The degree of parallelism is much reduced compared with what we have presented. Therefore, SIMD computers do not appear a better architecture than the MIMD that we have used.

In a MIMD computer, each processor can execute its own program upon its own data. Cooperation among processors is achieved by either shared memory or message passing (in distributed memory architectures). In a MIMD computer with shared memory, all programs and data are stored in k memories and are accessible by all processors with the restriction that each memory can be accessed by one processor at any time. This restriction tends to put an upper bound on the number of processors that can be effectively incorporated. Therefore, shared memory systems are efficient for small to medium number of processors.

For parallel learning of belief networks on a shared memory MIMD computer, our manager/explorer partition of processors can be used. Manager generates alternative structures and stores them in one memory. Each explorer can fetch one or more structures for evaluation at each time, which can be controlled by accessing a critical section. Hence job allocation can be performed similarly to our multi-batch or two-stage strategies. On the other hand, dataset access will become a bottleneck if a large number of processors want to access the same memory for data at the same time. The problem may be alleviated by duplicating the dataset in multiple memories. However, this may not be practical for large datasets due to limited total memory.

Based on our investigation using a distributed memory MIMD computer and the above analysis, we believe that this architecture is most suited to parallel learning of belief networks among the four architectures considered.

13 Conclusion

We have investigated parallel learning of belief networks as a way to tackle the computational complexity when learning in large and difficult (e.g., PI) problem domains. We have proposed parallel algorithms that decompose the learning task naturally for parallelism and they do not introduce errors compared with a corresponding sequential learning algorithm. We have studies multi-batch and two-stage job allocations which further improve the efficiency of the parallel system beyond the straightforward even allocation strategy. We found that multi-batch is more effective when the number of processors is small and two-stage is more effective when the number is large. We have proposed marginal server configuration to replace slow data access through file system by fast memory access. This allows parallel learning from very large datasets be performed effectively. We have implemented the algorithms in a distributed memory MIMD computer and our experimental results confirmed our analysis.

Our study has focused on learning DMNs. However, our results can be easily extended to learning Bayesian networks (BNs). This is because all known algorithms for learning

belief networks (whether they are DMNs or BNs) are based on evaluation of alternative network structures (often using local computations) relative to the given dataset. Therefore, our results on task decomposition, job allocation strategies and use of marginal servers are applicable to learning any type of belief networks.

We have extended the lessons we learned from using the distributed memory MIMD system to other architectures based on Flynn's taxonomy. Our analysis of the features of each architecture and the features of learning belief networks makes us believe that the distributed memory MIMD architecture is most suited to this task.

Acknowledgement

This work is supported by research grants from the Natural Sciences and Engineering Research Council of Canada and from the Institute for Robotics and Intelligent Systems in the Networks of Centres of Excellence Program of Canada.

Appendix A: Graph-theoretic terminology

Let G be an undirected graph. A set X of nodes in G is *complete* if each pair of nodes in X is adjacent. A set C of nodes is a *clique* if C is complete and no superset of C is complete. A *chord* is a link that connects two nonadjacent nodes. G is *chordal* if every cycle of length > 3 has a chord.

A decomposable Markov network (DMN) over a set N of variables is a pair (G, P) where G is a chordal graph and P is a probability distribution over N. Each node in $G = (N, E)$ is labeled by an element of N. Each link in G signifies the direct dependence of its end nodes. For disjoint subsets X, Y and Z of nodes, $< X|Z|Y >_G$ signifies $I(X, Z, Y)$, and hence P can be factorized into marginal distributions over cliques of G.

Appendix B: Frequently used acronyms

BN: Bayesian network
DMN: decomposable Markov network
PDM: probabilistic domain model
PI: pseudo-independent
MIMD: multiple instruction, multiple data
MISD: multiple instruction, single data
MLLS: multi-link lookahead search
SISD: single instruction, single data
SIMD: single instruction, multiple data
SLLS: single link lookahead search

References

[1] I.A. Beinlich, H.J. Suermondt, R.M. Chavez, and G.F. Cooper. The alarm monitoring system: a case study with two probabilistic inference techniques for belief networks. Technical Report KSL-88-84, Knowledge Systems Lab, Medical Computer Science, Stanford University, 1989.

[2] D. Chickering, D. Geiger, and D. Heckerman. Learning Bayesian networks: serach methods and experimental results. In *Proc. of 5th Conf. on Artificial Intelligence and Statistics*, pages 112–128, Ft. Lauderdale, 1995. Society for AI and Statistics.

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

[4] D. Heckerman, D. Geiger, and D.M. Chickering. Learning Bayesian networks: the combination of knowledge and statistical data. *Machine Learning*, 20:197–243, 1995.

[5] E.H. Herskovits and G.F. Cooper. Kutato: an entropy-driven system for construction of probabilistic expert systems from database. In *Proc. 6th Conf. on Uncertainty in Artificial Intelligence*, pages 54–62, Cambridge,, 1990.

[6] J. Hu. Learning belief networks in pseudo indeependent domains. Master's thesis, University of Regina, 1997.

[7] F.V. Jensen. *An introduction to Bayesian networks*. UCL Press, 1996.

[8] S. Kullback and R.A. Leibler. On information and sufficiency. *Annals of Mathematical Statistics*, 22:79–86, 1951.

[9] W. Lam and F. Bacchus. Learning Bayesian networks: an approach based on the MDL principle. *Computational Intelligence*, 10(3):269–293, 1994.

[10] T.G. Lewis and H. El-Rewini. *Introduction to Parallel Computing*. Prentice Hall, 1992.

[11] D.I. Moldovan. *Parallel Processing: From Applications To Systems*. Morgan Kaufman, 1993.

[12] J. Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann, 1988.

[13] P. Spirtes and C. Glymour. An algorithm for fast recovery of sparse causal graphs. *Social Science Computer Review*, 9(1):62–73, 1991.

[14] Y. Xiang. Towards understanding of pseudo-independent domains. In *Poster Proc. 10th Inter. Symposium on Methodologies for Intelligent Systems*, Oct 1997.

[15] Y. Xiang, S.K.M. Wong, and N. Cercone. Critical remarks on single link search in learning belief networks. In *Proc. 12th Conf. on Uncertainty in Artificial Intelligence*, pages 564–571, Portland, 1996.

[16] Y. Xiang, S.K.M. Wong, and N. Cercone. A 'microscopic' study of minimum entropy search in learning decomposable Markov networks. *Machine Learning*, 26(1):65–92, 1997.