Junction Tree Decomposition for Parallel Exact Inference*

Yinglong Xia[1] and Viktor K. Prasanna[2,1]
Computer Science Department[1]
Ming Hsieh Department of Electrical Engineering[2]
University of Southern California
Los Angeles, CA 90089, U.S.A.
{yinglonx, prasanna}@usc.edu

Abstract

We present a junction tree decomposition based algorithm for parallel exact inference. This is a novel parallel exact inference method for evidence propagation in an arbitrary junction tree. If multiple cliques contain evidence, the performance of any state-of-the-art parallel inference algorithm achieving logarithmic time performance is adversely affected. In this paper, we propose a new approach to overcome this problem. We decompose a junction tree into a set of chains. Cliques in each chain are partially updated after the evidence propagation. These partially updated cliques are then merged in parallel to obtain fully updated cliques. We derive the formula for merging partially updated cliques and estimate the computation workload of each step. Experiments conducted using MPI on state-of-the-art clusters showed that the proposed algorithm exhibits linear scalability and superior performance compared with other parallel inference methods.

1. Introduction

A full joint probability distribution for any real-world system can be used for inference. However, such a distribution increases intractably large as the number of variables used to model the system grows. It is known that independence and conditional independence relationships can greatly reduce the size of the joint probability distributions. This property is utilized by *Bayesian networks* [8]. Bayesian networks have been used in artificial intelligence since the 1960s. Now they have found applications in a number of domains, including medical diagnosis, consumer help desks, pattern recognition, credit assessment, data mining, genetics, etc. [3, 12, 16].

Inference on a Bayesian network is the computation of the conditional probability of the *query* variables, given a set of *evidence* variables as the knowledge to the network. Inference on a Bayesian network can be *exact* or *approximate*. Exact inference is NP hard [11]. The most popular exact inference algorithm for multiple connected networks was proposed by Lauritzen and Speigelhalter [8], which converts a Bayesian network into a *junction tree*, then performs exact inference on the junction tree.

The complexity of the exact inference algorithms increases dramatically with the density of the network, the clique width and the number of states of the random variables. Each potential table of a junction tree represents a joint probability distribution. A joint probability distribution assigns a probability to every possible combination of states of these random variables, and thus consists of r^w probabilities, where r is the number of states of random variables and w is the clique width. The potential table is quite large even for moderately large r and w. Therefore, although the problems considered in Section 7 can be solved on a single processor, problems with larger scales can not be solved on a single processor in reasonable time. In many cases exact inference must be performed in real time. In order to accelerate exact inference and solve large scale inference problems, we need to develop parallel techniques.

Several parallel implementations of exact inference have been presented, such as Pennock [11], Kozlov and Singh [7], and Szolovits [17]. However, some of those methods, such as [7], are dependent upon the structure of the Bayesian network. Their performance can be poor when applied to an arbitrary network. Others, such as [11], exhibit limited performance for multiple evidence inputs, since the evidence is assumed to be only in the root of the junction tree. If there are multiple evi-

*This research was partially supported by the National Science Foundation under grant number CNS-0613376 and utilized the DataStar at the San Diego Supercomputer Center. NSF equipment grant CNS-0454407 is gratefully acknowledged.

dences existing in several cliques, these works *reroot* the junction tree for each of these cliques. These works then perform exact inference in each of the rerooted trees.

The paper is organized as follows: Section 2 discusses the background of Bayesian networks and junction trees. Section 3 discusses the related work on parallel exact inference. Section 4 explores the exact inference algorithm in junction trees. Section 5 presents the junction tree decomposition algorithm for parallel exact inference. Section 6 presents the parallel algorithm and analyzes its cost. Experimental results are shown in Section 7. Section 8 concludes the paper.

2. Background

Consider a set of n random variables $\mathcal{W} = \{A_1, A_2, \cdots, A_n\}$. The probability that random variable A_j takes the value a is $P(A_j = a)$ where $a \in \{0, 1, \cdots, r-1\}$ and r is the number of states of variable A_j. A *joint distribution* $P(\mathcal{W}) = P(A_1, A_2, \cdots, A_n)$ assigns a probability to every possible combination of states of these random variables. Thus, the joint probability consists of r^n probabilities, a large number even for moderately large n.

A *Bayesian network* exploits conditional independence to represent a joint distribution more compactly. Figure 1 (a) shows a sample Bayesian network. A Bayesian network is defined as $\mathrm{B} = (\mathbb{G}, \mathbb{P})$ where $\mathbb{G}$ is a *directed acyclic graph* (DAG) and $\mathbb{P}$ is the parameter of the network. The graph $\mathbb{G}$ is denoted $\mathbb{G} = (\mathcal{V}, \mathcal{E})$ where $\mathcal{V} = \{A_1, A_2, \ldots, A_n\}$ is the node set and $\mathcal{E}$ is the edge set. Each node A_i represents a random variable. If there is an edge from A_i to A_j i.e. $(A_i, A_j) \in \mathcal{E}$, A_i is called a *parent* of A_j. $pa(A_j)$ denotes the set of all parents of A_j. Given the value of $pa(A_j)$, A_j is conditionally independent of all other preceding variables. The parameter $\mathbb{P}$ represents a group of *conditional probability tables* which are defined as the conditional probability $P(A_j|pa(A_j))$ for each random variable A_j. Given the Bayesian network, a joint distribution $P(\mathcal{V})$ can be given as [8]:

$$\begin{aligned} P(\mathcal{W}) &= P(A_1, A_2, \cdots, A_n) \\ &= \prod_{j=1}^{n} Pr(A_j|pa(A_j)) \end{aligned} \tag{1}$$

The *evidence* in a Bayesian network are the variables that have been instantiated with values e.g. $\mathrm{E} = \{A_{e_1} = a_{e_1}, \cdots, A_{e_c} = a_{e_c}\}$, $e_k \in \{1, 2, \ldots, n\}$. Given the evidence, we can inquire the distribution of any other variables. The variables to be inquired are called *query* variables. The process of *exact inference* involves propagating the evidence throughout the network and then computing the updated probability of the query variables.

It is known that traditional exact inference using Bayes' rule fails for networks with undirected cycles [8]. Most inference methods for networks with undirected cycles convert a network to a cycle-free hypergraph called a *junction tree*. A junction tree is defined as $\mathrm{J} = (\mathbb{T}, \hat{\mathbb{P}})$ where $\mathbb{T}$ represents a tree and $\hat{\mathbb{P}}$ denotes the parameter of the tree. Each vertex $\mathcal{C}_i$, known as a clique of J, is a set of random variables. Assuming $\mathcal{C}_i$ and $\mathcal{C}_j$ are adjacent, the *separator* between them is defined as $\mathcal{C}_i \cap \mathcal{C}_j$. All junction trees satisfy the *running intersection property* (RIP). $\hat{\mathbb{P}}$ is a group of *potential tables*. The potential table of $\mathcal{C}_i$, denoted $\psi_{\mathcal{C}_i}$, can be viewed as the joint distribution of the random variables in $\mathcal{C}_i$. For a clique with w variables, each taking r different values, the number of entries in the potential table is r^w. Figure 1 (b) shows a junction tree.

In a junction tree, exact inference proceeds as follows: Assuming evidence is $\mathrm{E} = \{A_i = a\}$ and $A_i \in \mathcal{C}_j$, E is *absorbed* at $\mathcal{C}_j$ by instantiating the variable A_i, then renormalizing the remaining constituents of the clique. The effect of the updated $\psi_{\mathcal{C}_j}$ is propagated to all other cliques by iteratively setting $\psi^*_{\mathcal{C}_x} = \psi_{\mathcal{C}_x}\psi^*_{\mathcal{S}}/\psi_{\mathcal{S}}$ where $\mathcal{C}_x$ is the clique to be updated; $\mathcal{S}$ is the separator between $\mathcal{C}_x$ and its neighbor that has been updated; ψ^* denotes the updated potential table. Mathematically the evidence propagation is represented as [8]:

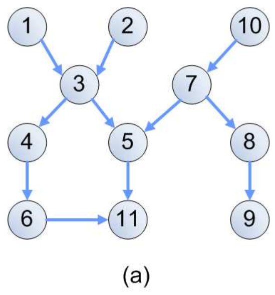
$$\psi^*_{\mathcal{S}} = \sum_{\mathcal{Y}\backslash\mathcal{S}} \psi^*_{\mathcal{Y}} \tag{2}$$

$$\psi^*_{\mathcal{X}} = \frac{\psi_{\mathcal{X}}}{\psi_{\mathcal{S}}}\psi^*_{\mathcal{S}} \tag{3}$$

After all cliques are updated, the distribution of a query variable $Q \in \mathcal{C}_y$ is obtained by summing up all entries with respect to $Q = q$ for all possible q in $\psi_{\mathcal{C}_y}$.

3. Related Work on Parallelizing Exact Inference

There are several works on parallel exact inference, such as Pennock [11], Kozlov and Singh [7] and Szolovits [17]. However, some of those methods, such as [7], are dependent upon the structure of the Bayesian network. The performance of this method also depends upon the structure of the network. Others, such as [11] and [10], exhibit limited performance for multiple evidence inputs, since the evidence is assumed to be in the root of the junction tree. Rerooting techniques are employed to deal with the case where the evidence appears at more than one clique. In this paper, for the sake of comparison, we simply call these rerooting based parallel exact inference methods *traditional methods*. Some other works address certain individual steps of exact inference. Reference [9] discusses the structure conver-



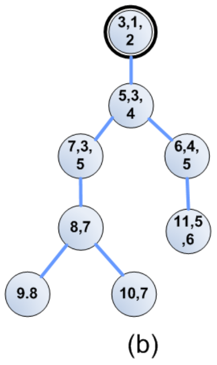


Figure 1. An example of (a) Bayesian network and its (b) Junction tree. The bold circle indicates the root of the junction tree.

sion of the junction tree from Bayesian networks. In [18] the node level primitives are parallelized.

In this paper, we deviate from all these and present a dramatically different method for exploring parallelization in evidence collection. Our algorithm starts with the junction tree and converts it to a set of chains. The performance of our algorithm is not sensitive to the location and number of evidence variables. Our method can also be used to load balance in exact inference and cooperate with other parallel techniques such as node level primitives.

4. Exact Inference on Junction Trees

The computation of exact inference on a junction tree consists of three steps: First, the clique potential tables that include evidence variables are updated. This is called *local evidence absorption*. Second, the evidence is propagated throughout the entire junction tree. This is called *evidence propagation*. Third, the probability distribution of the query variables is computed from the potential table of the cliques that include query variables. This step is simply called *query*.

In evidence absorption, we need to update potential tables by introducing evidence. Assuming evidence variable $E \in \{0, 1, \cdots, r_E\}$ takes state e in an observation, then $P(E = e)$ is the evidence. Suppose clique $\mathcal{C}$ includes E, and $\psi_\mathcal{C}$ is the potential table of $\mathcal{C}$. Since E takes only state e, $\psi_\mathcal{C}$ can be adjusted by yielding 0 for all entries where E does not take e:

$$\psi^*_\mathcal{C} = \psi_\mathcal{C}\delta(E = e) \tag{4}$$

$\psi^*_\mathcal{C}$ is the updated version of $\psi_\mathcal{C}$. Because the local evidence absorption is performed on each clique independently, this step can be implemented in parallel.

Evidence propagation is the major step of exact inference in junction trees. It brings the effect of evidence to every clique of a junction tree. The evidence propagation algorithm should ensure *local consistence* of a junction tree [8], which is implemented by evidence propagation: *evidence collection* and *evidence distribution*. Evidence collection starts at the leaf cliques of a junction tree. These cliques update themselves and propagate beliefs to other neighbors. In evidence collection, each clique absorbs the evidence in the subtree rooted at the clique. The direction of belief flow in evidence distribution is opposite to that in evidence collection. It broadcasts the evidence to all cliques in the junction tree.

The query step is straightforward: As the joint distribution of the random variables in a clique can be obtained from the potential table of the clique, the probability distribution of the query variables can be computed by marginalizing the potential tables. Equation (5) provides the joint distribution of the random variables in clique $\mathcal{C}$, where $Z = \sum \psi_\mathcal{C}$ is a normalization factor. In the formula derivation and algorithm analysis of this paper, we simply assume $Z = 1$. Equation (6) gives the probability distribution of the query variable Q from $P(\mathcal{C})$. As Equations (5) and (6) do not involve interaction between cliques, this step can be executed in parallel.

$$P(\mathcal{C}) = \frac{1}{Z}\psi_\mathcal{C} \tag{5}$$

$$P(Q) = \sum_{\mathcal{C}\setminus Q} P(\mathcal{C}) \tag{6}$$

5. Junction Tree Decomposition

5.1. Parallel Evidence Propagation in Chains

A *chain* is a special tree without any branches. Parallel evidence propagation in chains is a part of the parallel exact inference in junction trees. Figure 2 gives an example of a chain of cliques. Assume clique $\mathcal{A}_1$ contains evidence variables and has absorbed the evidence. We use $\psi^*_{\mathcal{A}_1}$ to denote the updated $\psi_{\mathcal{A}_1}$. Other cliques in this chain need to update their potential table using

$\psi^*_{\mathcal{A}_1}$. The sequential updating process iteratively utilizes Equations (2) and (3): $\psi_{\mathcal{A}_2}$ is updated using $\psi_{\mathcal{A}_1}$, and $\psi_{\mathcal{A}_3}$ is updated using $\psi_{\mathcal{A}_2}$, so on and so forth. Pennock [11] proposed a pointer jumping technique for updating a chain of random variables. This technique can be generalized for evidence propagation in a chain of cliques. Assume there are n processors and processor i is assigned to clique $\mathcal{A}_i$ in Figure 2. In step 0, each processor updates its potential table with respect to its parent. Mathematically, the processor handling clique i computes:

$$\begin{aligned}\psi^*_{\mathcal{A}_{i+1}\leftarrow\mathcal{A}_i} &= \frac{\psi_{\mathcal{A}_{i+1}}}{\psi_{\mathcal{A}_{i+1}\cap\mathcal{A}_i}}\psi^*_{\mathcal{A}_{i+1}\cap\mathcal{A}_i}\\ &= \frac{\psi_{\mathcal{A}_{i+1}}}{\psi_{\mathcal{A}_{i+1}\cap\mathcal{A}_i}}\sum_{\mathcal{A}_i\backslash\mathcal{A}_{i+1}}\psi_{\mathcal{A}_i} \qquad (7)\end{aligned}$$

where $\psi^*_{\mathcal{A}_{i+1}\leftarrow\mathcal{A}_i}$ indicates updating $\psi_{\mathcal{A}_{i+1}}$ using $\psi_{\mathcal{A}_i}$. $\mathcal{A}_{i+1}\cap\mathcal{A}_i$ is the separator of the two cliques and $\psi_{\mathcal{A}_{i+1}\cap\mathcal{A}_i} = \sum_{\mathcal{A}_{i+1}\backslash\mathcal{A}_i}\psi_{\mathcal{A}_{i+1}}$; $\mathcal{A}_i\backslash\mathcal{A}_{i+1}$ denotes the variables that belong to $\mathcal{A}_i$ but not to $\mathcal{A}_{i+1}$. After this step, the $\psi_{\mathcal{A}_2}$ has been fully updated.

In Step 1, each clique updates its potential table with respect to its grandparent, which is given by:

$$\begin{aligned}\psi^*_{\mathcal{A}_{i+2}\leftarrow\mathcal{A}_i} &= \frac{\psi_{\mathcal{A}_{i+2}}\sum_{\mathcal{A}_{i+1}\backslash\mathcal{A}_{i+2}}\psi_{\mathcal{A}_{i+1}}}{\psi_{\mathcal{A}_{i+2}\cap\mathcal{A}_{i+1}}\psi_{\mathcal{A}_{i+1}\cap\mathcal{A}_i}}\sum_{\mathcal{A}_i\backslash\mathcal{A}_{i+1}}\psi_{\mathcal{A}_i}\\ &= \frac{\psi^0_{\mathcal{A}_{i+2}}}{\psi^0_{\mathcal{A}_{i+2}\cap\mathcal{A}_i}}\sum_{\mathcal{A}_i\backslash\mathcal{A}_{i+2}}\psi^0_{\mathcal{A}_i} \qquad (8)\end{aligned}$$

where $\psi_{\mathcal{A}_{i+2}\cap\mathcal{A}_{i+1}} = \sum_{\mathcal{A}_{i+2}\backslash\mathcal{A}_{i+1}}\psi_{\mathcal{A}_{i+2}}$ and $\psi_{\mathcal{A}_{i+1}\cap\mathcal{A}_i} = \sum_{\mathcal{A}_{i+1}\backslash\mathcal{A}_i}\psi_{\mathcal{A}_{i+1}}$. $\psi^0_{\mathcal{A}}$ denotes the resultant potential table of $\mathcal{A}$ after step 0.

Step $k(k = 2, 3, \cdots)$ proceeds exactly as Step 1, however, $\psi^*_{\mathcal{A}_{i+2}}$, is replaced by $\psi^*_{\mathcal{A}_{i+2^k}}$ in Formula (8). The processor handling clique i computes:

$$\psi^*_{\mathcal{A}_{i+2^k}\leftarrow\mathcal{A}_i} = \frac{\psi^{k-1}_{\mathcal{A}_{i+2^k}}}{\psi^{k-1}_{\mathcal{A}_{i+2^k}\cap\mathcal{A}_i}}\sum_{\mathcal{A}_i\backslash\mathcal{A}_{i+2^k}}\psi^{k-1}_{\mathcal{A}_i} \qquad (9)$$

where $\psi^{k-1}_{\mathcal{A}}$ denotes the resultant potential table of $\mathcal{A}$ after step k-1. After Step k, $1 \le k \le \log n$, the potential tables of the first 2^k cliques have been updated. After $\log n$ steps[1], all n processors are done, and all the potential tables are updated. Figure 2 illustrates the steps of evidence propagation in a chain. Evidence propagation in a chain using pointer jumping can be completed in $O(\log n)$ parallel time for a chain with n cliques.

[1]All log functions in this paper are of base 2.

5.2. Parallel Evidence Propagation in Trees

Evidence propagation in a tree includes evidence collection and evidence distribution. For the sake of illustration, we consider evidence distribution in this subsection. Assuming the evidence initially exists in the root clique only, evidence distribution ensures that the evidence can be propagated to all other cliques. The pointer jumping technique mentioned in the previous subsection can be generalized to evidence distribution in trees. Pennock [11] proposed a method which applies a pointer jumping technique to the evidence distribution in a tree. Although the algorithm presented in [11] is based on the poly tree where each node denotes a single random variable, it is straightforward to apply the pointer jumping technique to junction trees where each node is a clique consisting of several random variables.

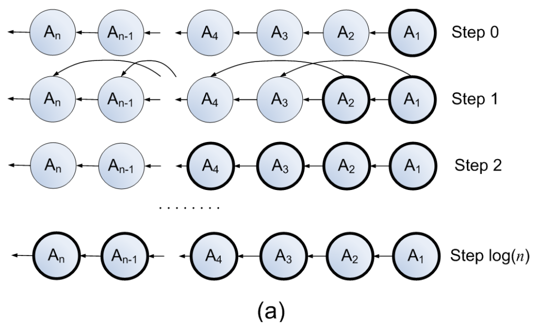
Evidence distribution using pointer jumping propagates the evidence from the root to all cliques in $\log(D+1)$ steps, where D the *maximum depth* of the junction tree. The *depth* of a clique in a junction tree is defined as the length of the path from the root to the clique. We denote the parent of clique $\mathcal{A}_i$ in the given junction tree $pa(\mathcal{A}_i)$, $pa^2(\mathcal{A}_i) = pa(pa(\mathcal{A}_i))$ and $pa^k(\mathcal{A}_i) = pa(pa^{k-1}(\mathcal{A}_i))$ where $k = 1, 2, \cdots, \log(D+1)$. In the initial step, each clique updates its potential table with respect to its parent $pa(\mathcal{A}_i)$, then in Step 1 each clique updates its potential table again according to its grandparent $pa^2(\mathcal{A}_i)$. In Step k, the potential table $\psi_{\mathcal{A}_i}$ is updated with respect to $pa^k(\mathcal{A}_i)$. As there are $(D+1)$ cliques in the path of depth D, after $\log(D+1)$ steps, the evidence at the root has been propagated to all the cliques in this junction tree. This is illustrated in Figure 2.

The formula for pointer jumping based evidence distribution in a junction tree is similar to that in chains. The propagation flow is from root to leaves, so each clique has a unique ancestor. According to formula (9), updating evidence the distribution in junction tree is given by:

$$\psi^*_{\mathcal{A}_i\leftarrow pa^k(\mathcal{A}_i)} = \frac{\psi^{k-1}_{\mathcal{A}_i}}{\psi^{k-1}_{\mathcal{A}_i\cap pa^k(\mathcal{A}_i)}}\sum_{pa^k(\mathcal{A}_i)\backslash\mathcal{A}_i}\psi^{k-1}_{pa^k(\mathcal{A}_i)} \qquad (10)$$

In this case, as some cliques share parents or other ancestors, they receive the evidence from ancestors simultaneously and update their potential tables in parallel using Formula (10).

However, the pointer jumping technique can not be applied to the evidence collection in a junction tree directly. In evidence collection, the evidence flow starts at the leaf cliques and terminates at the root. Each clique



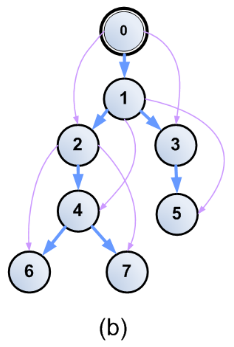


Figure 2. Illustration of parallel evidence propagation in a chain with n cliques and an arbitrary junction tree using pointer jumping. The cliques with bold circles have absorbed the propagated evidence.

gets fully updated only if it receives evidence from all its parents. Note that if $\mathcal{A}_i$ is a parent of $\mathcal{A}_j$ in evidence collection, then $\mathcal{A}_i$ become a child of $\mathcal{A}_j$ in evidence distribution, because the evidence flow direction is reversed. As shown in Figure 3 (a), clique 1 is fully updated after it gets the evidence from 5, 6 and 7. Clique 1 gets evidence from node 5 after two steps and from 6 and 7 after three steps. Therefore, it takes longer time for clique 1 to get fully updated so that it can send complete evidence to its ancestors. If a clique has a large number of descendants (or it has a large number of ancestors when the edges are reversed), such delays reduce the performance of pointer jumping. In order to take advantage of pointer jumping, we execute pointer jumping in chains decomposed from a junction tree, instead of performing pointer jumping in the junction tree directly.

5.3. Decomposing a Junction Tree into Chains

We decompose a tree into a set of chains so that the parallel evidence propagation discussed in Section 5.1 can be applied. Assuming there are l leaves out of N cliques in a given junction tree, we initialize l chains for this junction tree. Each chain is a path from a leaf clique to the root. As the computation time for path search is much less than that for potential table computation, we simply assign N cliques of the given junction tree to p processors, where $1 \leq p \leq N$. Each processor is in charge of N/p cliques and checks if there exists a leaf clique. For each leaf clique, the processor performs a path search starting at this clique. Figure 3 shows an example of tree decomposition.

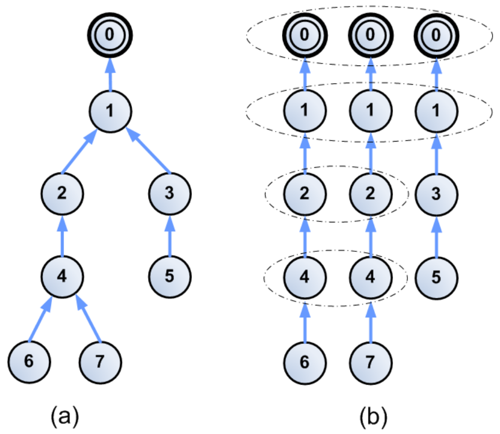


Figure 3. An example of tree decomposition. (a) A junction tree. (b) The chains decomposed from the junction tree. The cliques in a dash circle should be merged after evidence propagation.

After tree decomposition, the parallel evidence propagation is performed on these chains from the leaves to the root. The evidence propagation using pointer jumping is performed in these chains in parallel. It is not necessary to synchronize the pointer jumping in these chains during evidence propagation. After the propagation, the evidence on each chain is collected at the root. However, if a clique is duplicated into more than one chain, then each copy of this clique is only partially updated. These partially updated copies should be merged to obtain a fully updated clique. This is discussed in the next subsection.

5.4. Merging Partially Updated Cliques

Some cliques of a junction tree are duplicated several times in tree decomposition. For example, clique 0 in Figure 3 is duplicated to three chains. These copies get partially updated independently in each chain. *Clique merging* is the fusion of partially updated cliques to obtain a fully updated clique. For example, in Figure 3, merging clique 4 can be viewed as merging clique 4 in subchain $4-6$ and subchain $4-7$.

Since the evidence propagates from leaf to root in evidence collection, we can partially update a clique in any chain by using the potential table of its parent. For example, in Figure 3 (b), the evidence propagated to clique 1 is from three chains: clique 2 that appears in two chains, and clique 3 in another chain. Therefore, instead of directly merging all copies of clique 1 in these chains, we can obtain the fully updated potential table $\psi^*_{\mathcal{A}_1}$ by using the potential tables of the separators between $\mathcal{A}_1$ and its parent in each chain. $\psi_{S_{ij}}$ is the potential table of the separator between $\mathcal{A}_i$ and $\mathcal{A}_j$, where $S_{ij} = \mathcal{A}_i \cap \mathcal{A}_j$. The potential table $\psi_{\mathcal{A}_i}$ can be fully updated by *clique merge function* $\mathbb{M}$:

$$\psi^*_{\mathcal{A}_i} = \mathbb{M}\Big(\psi_{\mathcal{A}_i}, \psi^*_{\mathcal{A}_i \cap pa_1(\mathcal{A}_i)}, \psi^*_{\mathcal{A}_i \cap pa_2(\mathcal{A}_i)}, \cdots, \psi^*_{\mathcal{A}_i \cap pa_l(\mathcal{A}_i)}\Big) \tag{11}$$

where $\psi_{\mathcal{A}_i}$ is the potential table of $\mathcal{A}_i$ before evidence propagation; l is the number of chains that contains $\mathcal{A}_i$; $pa_k(\mathcal{A}_i)$ is the parent of $\mathcal{A}_i$ on the kth chain of the l chains; $\psi_{\mathcal{A}_i \cap pa_k(\mathcal{A}_i)}$ is the potential table of the intersection between $\mathcal{A}_i$ and $pa_k(\mathcal{A}_i)$.

Equation (11) can be simplified: During the evidence collection in the first chain, evidence is propagated from $pa_1(\mathcal{A}_i)$ to $\mathcal{A}_i$ via the separator $\mathcal{A}_i \cap pa_1(\mathcal{A}_i)$. Therefore, $\psi_{\mathcal{A}_i}$ has been updated by $\psi_{\mathcal{A}_i \cap pa_1(\mathcal{A}_i)}$. According to Equations (2) and (3), the updated potential table of $\mathcal{A}_i$ in the 1st chain is denoted $\psi^*_{(1)\mathcal{A}_i} = \psi_{\mathcal{A}_i}\psi^*_{\mathcal{A}_i \cap pa_1(\mathcal{A}_i)}/\psi_{\mathcal{A}_i \cap pa_1(\mathcal{A}_i)}$, where $\psi_{\mathcal{A}_i \cap pa_1(\mathcal{A}_i)}$ is obtained by marginalizing $\psi_{\mathcal{A}_i}$ and $\psi^*_{\mathcal{A}_i \cap pa_1(\mathcal{A}_i)}$ is obtained by marginalizing the updated potential function $\psi^*_{pa_1(\mathcal{A}_i)}$. Therefore, we can use $\psi^*_{(1)\mathcal{A}_i}$, the potential table of $\mathcal{A}_i$ updated in the first chain, to substitute $\psi_{\mathcal{A}_i}$ and $\psi^*_{\mathcal{A}_i \cap pa_1(\mathcal{A}_i)}$ in Equation (11):

$$\psi^*_{\mathcal{A}_i} = \mathbb{M}\Big(\psi^*_{(1)\mathcal{A}_i}, \psi^*_{\mathcal{A}_i \cap pa_2(\mathcal{A}_i)}, \cdots, \psi^*_{\mathcal{A}_i \cap pa_l(\mathcal{A}_i)}\Big) \tag{12}$$

Comparing Equations (11) and (12), we see that using Equation (12) has two merits: first, it reduces the number of parameters; second, as all parameters in Equation (12) are updated potential tables, we can perform *in-place* rewriting on the original potential table during evidence propagation and therefore save local memory. As each copy of a clique keeps its own potential table, we do not have to synchronize the potential table updating. Assuming each clique is handled by a unique processor, each processor executes Equation (12) to merge all cliques in parallel.

5.5. Derivation of The Partially Updated Clique Merge Function $\mathbb{M}$

In order to obtain the expression for clique merge function $\mathbb{M}$ in Equation (12), we first derive $\mathbb{M}$ for a simple case and then calculate the general form of $\mathbb{M}$.

Consider the simplest case shown in Figure 4 (a) where evidence flows from clique $\mathcal{A}_2$ to $\mathcal{A}_1$ and clique $\mathcal{A}_3$ to $\mathcal{A}_1$, respectively. Assume $\mathcal{A}_1 = \{A, B, C\}$, $\mathcal{A}_2 = \{B, E\}$ and $\mathcal{A}_3 = \{D, E\}$, where $A - E$ are random variables. We intend to merge partially updated cliques corresponding to $\mathcal{A}_1$. According to Equation (3), the partially updated potential table for clique $\mathcal{A}_1$ in the left chain is given by:

$$\begin{aligned}\psi^*_{(1)\mathcal{A}_1} &= \psi^*_{(1)}(\{A,B,C\}) \\ &= \frac{\psi(\{A,B,C\})\psi^*(B)}{\psi(B)}\end{aligned} \tag{13}$$

If we update $\psi_{\mathcal{A}_1}$ from both chains, we obtain the fully updated potential table $\psi^*_{\mathcal{A}_1}$ by using Equation (3) twice:

$$\begin{aligned}\psi^*_{\mathcal{A}_1} &= \psi^*_{(2)}(\{A,B,C\}) \\ &= \frac{\psi^*_{(1)}(\{A,B,C\})\psi^*(C)}{\psi^*_{(1)}(C)}\end{aligned} \tag{14}$$

where $\psi^*_{(1)}(C)$ is obtained by marginalizing $\psi^*_{(1)}(\{A,B,C\})$; $\psi^*_{(2)}(\{A,B,C\})$ is the partially updated potential table of clique $\mathcal{A}_1 = \{A,B,C\}$, based on the first two chains. As there are only two chains in this case, Equation (14) is equal to the fully updated potential table $\psi^*_{\mathcal{A}_1}$.

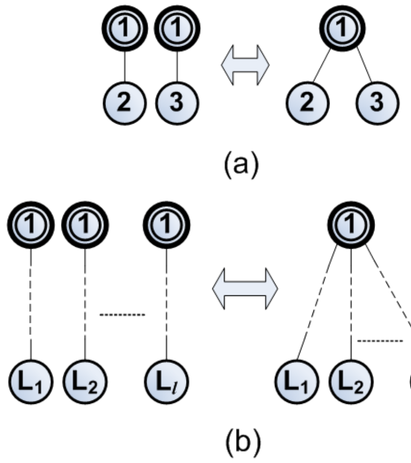


Figure 4. Illustration of clique merging. (a) the simplest case; (b) the general case.

Equation (14) gives the expression of the clique merge function $\mathbb{M}$ for merging partially updated cliques from two chains. Assume Equation (14) holds for merging cliques from k chains, that is:

$$\psi^*_{\mathcal{A}} = \psi^*_{(k)\mathcal{A}} = \frac{\psi^*_{(k-1)\mathcal{A}} \psi^*_{S_k}}{\psi^*_{(k-1)S_k}} \tag{15}$$

where $\psi^*_{(k-1)\mathcal{A}}$ is the partially updated potential table of $\mathcal{A}$ based on the first $(k-1)$ chains; S_k is the intersection between $\mathcal{A}$ and its parent in the kth chain; $\psi^*_{(k-1)S_k}$ is the potential table of S_k obtained by marginalizing $\psi^*_{(k-1)\mathcal{A}}$. Now, we prove that the expression for $\mathbb{M}$ in Equation (15) also holds for those nodes duplicated in $(k+1)$ chains.

If we need to merge one more clique from the $(k+1)$th chain, using Equation (3) and the previous results iteratively, we have:

$$\begin{aligned} \frac{\psi^*_{(k)\mathcal{A}} \psi^*_{S_{k+1}}}{\psi^*_{(k)S_{k+1}}} &= \frac{\psi^*_{(k-1)\mathcal{A}} \psi^*_{S_k} \psi^*_{S_{k+1}}}{\psi^*_{(k-1)S_k} \psi^*_{(k)S_{k+1}}} \\ &= \cdots = \frac{\psi^*_{(1)\mathcal{A}} \psi^*_{S_2} \psi^*_{S_3} \cdots \psi^*_{S_{k+1}}}{\psi^*_{(1)S_1} \psi^*_{(2)S_2} \cdots \psi^*_{(k)S_{k+1}}} \end{aligned} \tag{16}$$

Equation (16) is exactly the expression for fully updating $\psi_{\mathcal{A}}$ by absorbing evidence propagation from k parents. Therefore, by induction, Equation (15) can be used to merge cliques from an arbitrary number of chains, which gives the expression for clique merge function $\mathbb{M}$ in Equation (12):

$$\begin{aligned} &\mathbb{M}\left(\psi^*_{(1)\mathcal{A}_i}, \psi^*_{\mathcal{A}_i \cap pa_2(\mathcal{A}_i)}, \cdots, \psi^*_{\mathcal{A}_i \cap pa_k(\mathcal{A}_i)}\right) \\ &= \frac{\psi^*_{(1)\mathcal{A}} \psi^*_{S_2} \psi^*_{S_3} \cdots \psi^*_{S_{k+1}}}{\psi^*_{(1)S_1} \psi^*_{(2)S_2} \cdots \psi^*_{(k)S_{k+1}}} \end{aligned} \tag{17}$$

6. Parallel Exact Inference Algorithm

6.1. Algorithm Steps

The proposed method provides several opportunities to parallelize exact inference. For example, decomposition leads to independent parallel activities; each chain can be parallelized using pointer jumping. In this section, we present an implementation using pointer jumping. A high level description of the algorithm flow is given in Algorithm 1.

The input to the algorithm is an arbitrary junction tree JT, evidence E and query variables Q. In the initial stage (Lines 1-3), the data layout is as follows: we arbitrarily assign the cliques in JT to p processors $(1 \leq p \leq N)$, ensuring that each processor is assigned approximately N/p cliques. A processor is called the *owner* of a clique if the clique is assigned to it. Each processor keeps the potential tables of the cliques assigned to it, as well as the evidence variable set E. The information regarding each clique's parent and children is also stored in the same processor. This is used later in Lines 14-16 of the algorithm for evidence distribution. In this stage, each processor works in parallel to identify if any cliques assigned to it contain evidence variables in E. If a clique contains evidence variables, the potential table of the clique is updated by using Equation (4).

In Lines 4-6, for leaf cliques assigned to the processor, each processor identifies chains (paths) from these leaf cliques to the root. In this stage, we have the same data layout as in the previous stage. In addition, the structure of the junction tree is stored as an adjacency list in each processor. Each processor uses the adjacency list to identify the chains for the leaf cliques it contains. Notice that we use a straightforward method to identify chains in each processor. The processors do not collaborate in this stage.

After the chain identification stage, we perform data remapping before we move on to the next stage. The purpose of data remapping is to improve the parallel performance in successive stages. The data remapping here includes two steps: first, as some cliques are assigned to multiple chains after chain identification, each processor duplicates such cliques so that each chain has a copy; second, we redistribute the cliques on the chains to processors, so that we can improve the performance in the following stages where pointer jumping is used. We allocate the cliques in the chains in a round robin fashion. In order to perform this, each processor has all the information needed to identify the clique-to-processor mapping function. Each processor locally creates the message to be sent to other processors to perform the remapping. For each clique in a chain, a pointer to its parent (if any) is appended so that pointer jumping can be performed. For each clique that appears on a chain, a pointer to the owner of the clique is also maintained. This information is used in the merge step.

In Lines 8-10, we use pointer jumping to perform parallel evidence collection in the chains decomposed from the junction tree. Each processor works for $\log d$ iterations, where d is the maximum length of chains containing cliques allocated to the processor. In each iteration, the processor updates all cliques on it using separator potential tables received from the children of these cliques. The processor also sends updated separator potential tables to the corresponding parent of each clique. In addition, the processor updates the pointer of each clique so that it points to the grandparent of the clique.

The processors work in parallel and coordinate to merge partially updated potential tables corresponding to the cliques handled by it (Lines 11-13). In this stage, if a processor is the owner of $\mathcal{A}$, it collects all the separator potential tables related to $\mathcal{A}$ from all the chains in which $\mathcal{A}$ appears. We use these data to fully update the potential table of $\mathcal{A}$. The data layout in clique merge is the same as in the previous stage. Using Equation (15), each processor updates the potential tables of cliques it owns. After this stage, we operate on the original junction tree, though all potential tables have been fully updated with respect to evidence collection.

We use pointer jumping to perform parallel evidence distribution (Lines 14-16) in the original junction tree. We use the data layout for the original junction tree (Lines 1-6). Note that we perform pointer jumping on the junction tree instead of on the chains created in Lines 4-6. Using pointer jumping for evidence distribution takes $\log(D+1)$ iterations, where the depth of the junction tree is D. In each iteration, all the potential tables assigned to a processor need to be updated.

In Algorithm 1, the last stage is local computation for query variable set Q (Lines 17-19). We use the data layout for the original junction tree (Lines 1-6). Each processor checks cliques assigned to it in parallel. If a clique contains query variables in Q, the processor uses Equations (5) and (6) to obtain the distribution of the query variables.

6.2. Analysis of the Algorithm

For the sake of illustration of potential speedup, we analyze the complexity of our algorithm in terms of *computation cost* and *communication cost*. In this paper, computation cost is defined as the computations performed by each processor on data in local memory; communication cost is defined as the amount of data any processor communicated with other processors. In the following, we give bounds on these costs, even though different processors may incur different costs.

In Algorithm 1, the first stage is local evidence absorption (Lines 1-3). We use N to represent the number of cliques in the original junction tree; p denotes the number of processors. We assign approximately N' cliques to each processor, where $N' \approx N/p$. In order to absorb evidence using Equation (4), we need to visit each entry of a potential table, which has $O(r^w)$ entries. For each entry, absorbing the evidence requires $O(rw^2)$ time of local computation [18]. Therefore, the computation cost in local evidence absorption is $O(N' \cdot r^w \cdot rw^2) = O(Nw^2r^{w+1}/p)$. As no communication is needed in this stage, we have no communication cost.

In the chain identification stage (Lines 4-6), each processor uses adjacency list to identify the chains for the leaf cliques it contains. We identify if a clique $\mathcal{A}$ is a leaf by looking up in the adjacency list. This takes $O(1)$ time for each clique. The number of leaf cliques in a processor is bounded by N'. Assuming D is the depth of the junction tree, we identify a chain by using at most D hops. Therefore, for each processor, the computation cost of the chain identification stage is $O(N' \cdot D) = O(DN/p)$. There is no communication in this stage.

After the chain identification stage, we perform data remapping. In data remapping, the chains in each processor are allocated among p processors, so that we can use pointer jumping to accelerate the evidence propagation. Each processor works independently of other processors and sends the information of the chains it has uniformly to other processors. The computation cost in this stage is at most $O(DN/p)$, because we need to visit each element of the chains in a processor to allocate the chains. The number of chains is bounded by N/p, and the length of chains is bounded by $O(D)$. Note that a clique can be duplicated at most l times, and moving a potential table from one processor to another leads to $O(r^w)$ communication cost. The communication cost for data remapping is bounded by $O(lr^wN/p)$.

The evidence collection stage is presented in Lines 8-10. The computation cost of pointer jumping in these

Algorithm 1 Parallel Exact Inference

```
Input: Junction tree $JT$, Set of evidence variables $E$ and set of query variables $Q$
Output: Probability distribution of $Q$
 1: for each clique $\mathcal{A}_i$ in parallel do
 2:     Use Equation (4) for local evidence absorption
 3: end for
 4: for each leaf clique $\mathcal{L}_i$ in parallel do
 5:     Create a chain as the path from $\mathcal{L}_i$ to the root
 6: end for
 7: Data remapping: change from the distribution of $JT$ among processors to distribution of chains among processors
 8: for each processor in parallel do
 9:     Perform evidence collection (using pointer jumping. Use Equation (9))
10: end for
11: for each processor in parallel do
12:     Merge partially updated cliques using Equation (17) by collecting information at the owner of each clique
13: end for
14: for each processor in parallel do
15:     Perform evidence distribution (using pointer jumping. Use Equation (10)) on $JT$
16: end for
17: for each clique $\mathcal{A}_i$ in parallel do
18:     Use Equations (5) and (6) to obtain query response
19: end for
```

chains depends on the number of jumps and the cost of rewriting the potential tables. Although the actual performance of pointer jumping is sensitive to several factors, such as data layout and cache architecture, we focus on the influence of the data size in this analysis. The data layout in this stage is the same as the layout at the end of the previous stage. Because the longest chain has D edges, or $D+1$ cliques, according to the pointer jumping technique, the number of jumps needed is bounded by $O(\log(D+1))$. The size of a potential table is bounded by $O(r^w)$, where r is the number of states of these random variables involved in the junction tree and w is the maximum width of the cliques. For each entry of the potential table, a processor spends $O(rw^2)$ time for local computation. Each processor has at most lN/p cliques. The computation cost of these operations is bounded by $O(rw^2 \cdot r^w \cdot lN/p)$ for at most $\log(D+1)$ steps, i.e. $O((lNw^2r^{(w+1)}\log(D+1))/p)$. The scalability range is $1 \leq p \leq Dl$. In pointer jumping, each clique sends an updated separator potential table of size $O(r^{w_s})$ to its parent, where w_s is the maximum width of the separators. As there are $O(lN/p)$ cliques in each processor, and we need $\log(D+1)$ pointer jumping iterations, the communication cost to complete Lines 8-10 is $O((lNr^{w_s}\log(D+1))/p)$. Note that there is a barrier synchronization at the end of each pointer jumping iteration.

Next, we perform the clique merge operation (Lines 11-13), in which the chains are merged to get back the original junction tree layout. This is done by each processor sending information about each clique it processed, to the owner processor of the clique (see Figure 5). We assume that there are l chains. Therefore a clique can be duplicated at most $l-1$ times. So, for each clique $\mathcal{A}$, the processor which owns $\mathcal{A}$ updates $\psi_{\mathcal{A}}$ at most $(l-1)$ times. The computation cost to update $\psi_{\mathcal{A}}$ once is bounded by $O(w^2r^{(w+1)})$ [18]. Thus, the computation cost for clique merge is $O(Nlw^2r^{(w+1)}/p)$. In this stage, for each clique, the owner processor needs to obtain a separator potential table from all its children (l children at most). The size of a separator potential table is $O(r^{w_s})$. Therefore, the communication cost of this stage is $O(Nlr^{w_s}/p)$.

The evidence distribution is presented in Lines 14-16. Using pointer jumping for evidence distribution takes $\log(D+1)$ steps, as the depth of the junction tree is of length D. In each step, all the potential tables of cliques on their owner processor need to be updated. Therefore, there are approximately N/p potential tables for each processor to process in every pointer jumping iteration. The updating process is the same as the potential table updating in evidence collection, except that we operate on the original junction tree instead of chains. Thus, the computation cost of evidence distribution is similar to

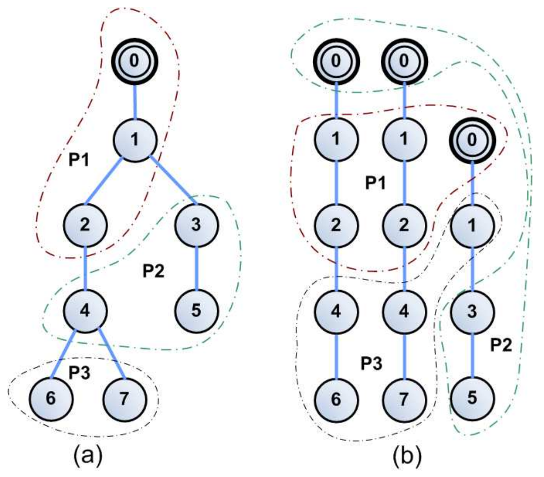


Figure 5. Illustration of (a) clique distribution of original junction tree and (b) clique distribution of chains decomposed from the junction tree.

that of evidence collection in chains, which is bounded by $O((Nw^2r^{(w+1)}\log(D+1))/p)$. Note that the scalability is $1 \leq p \leq N$, since we operate on the original junction tree layout. The pointer jumping in evidence distribution requires each clique to send an updated separator potential table to all its children. As a clique has at most l children, each clique sends a separator potential table of size $O(r^{w_s})$ to at most l processors. Thus, the communication cost is $O((lNr^{w_s}\log(D+1))/p)$.

In Algorithm 1, the last stage is local computation for query variable set Q (Lines 17-19). Each processor processes approximately N/p cliques. Calculating the distribution of query variables from a given potential table takes $O(w^2r^{w+1})$ time, since this calculation needs to perform local computation of cost $O(rw^2)$ to each of the $O(r^w)$ entries of the potential table. Thus, the computation cost for this stage is $O(w^2r^{(w+1)}N/p)$. As all processors work separately in this stage, there is no communication.

Based on above analysis, we arrive at the upper bound on the total computation cost for junction tree decomposition based parallel exact inference: $O(DN/p + (\log(D+1))lw^2r^wN/p)$; the total communication cost is $O(lr^wN/p + Nl(\log(D+1))r^{w_s}/p)$.

6.3. Speedup Estimation

We compare the performance of *traditional* parallel exact inference methods with that of the proposed method. By traditional parallel exact inference, we mean pointer jumping with rerooting for each evidence clique. The computation cost of traditional parallel exact inference is given by $O((KNw^2r^w \log(D+1))/p), 1 \leq p \leq N$ where K is the number of cliques containing evidence variables. The *speedup* S_p is defined as the ratio of the two total cost expressions. l denotes the number of leaf cliques in the given junction tree. When the number of states of random variables r is large or the clique width w is large, r^w increases sharply compared with other expressions. In such a scenario, we can simplify the speedup by removing terms that do not contain r^w. As the maximum depth of a junction tree $D \geq 1$, we have $\log(D + 1) \geq 1$. Therefore, the speedup is given by:

$$\begin{aligned} S_p &= \frac{(KNw^2r^w \log(D+1))/p}{DN/p + (l \log(D+1))w^2r^wN/p} \\ &= \frac{KN(\log(D+1))w^2r^w}{DN + (l \log(D+1))w^2r^wN} \\ &\approx \frac{K \log(D+1)}{l \log(D+1)} \geq K/l \end{aligned} \tag{18}$$

In Equation (18), we have $S_p \propto K$ because traditional method needs to apply evidence collection and distribution K times for a junction tree where K cliques contain evidence. We have $S_p \propto 1/l$ because junction tree decomposition causes duplication of some cliques. This can happen for at most l times. According to Equation (18), for a junction tree with large r and w, if we have $K \geq l$, the proposed method is at least as good as the traditional method. However, since not all cliques need to be duplicated l times, the speedup can be better than the above estimate. Note that actual speedup depends on the overheads in the implementation and the input instance.

7. Experiments

We conducted experiments on a state-of-the-art cluster at the San Diego Supercomputer Center (SDSC) [14]. The DataStar Cluster at SDSC employs IBM P655 nodes running at 1.5 GHz with 2 GB of memory per processor. It uses a Federation interconnect, and has a theoretical peak performance of 15 Tera-Flops. Furthermore, each node is connected to a GPFS (parallel file system) through a fiber channel. The DataStar Cluster runs Unix with MPICH for message passing. IBM Loadleveler was used to submit jobs to batch queue.

We used two random junction trees of different sizes to analyze and evaluate the performance of our method. One junction tree had 100 cliques and 18 leaves; the other had 500 cliques and 83 leaves. Each clique consisted of several random variables. The smallest clique in our experiments contained only two variables, while the largest had 10. All variables were binary. Therefore, the potential tables for these cliques were of length

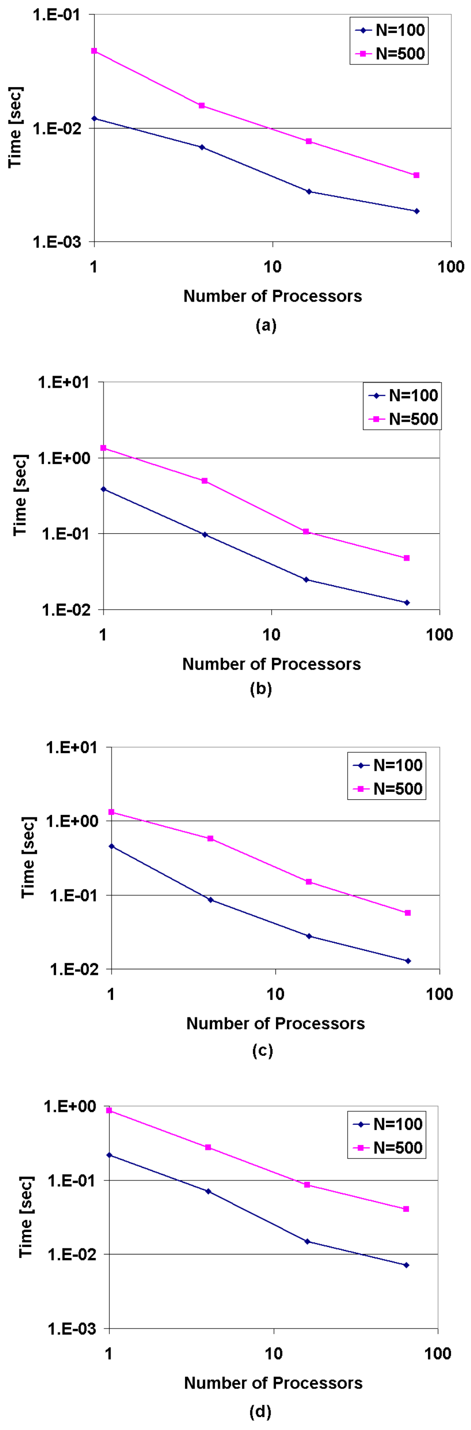


Figure 6. Computation time for individual steps of junction tree decomposition based exact inference. (a) Chain creation; (b) Evidence collection; (c) Evidence distribution; (d) Clique merging. The number of cliques are 100 and 500 respectively.

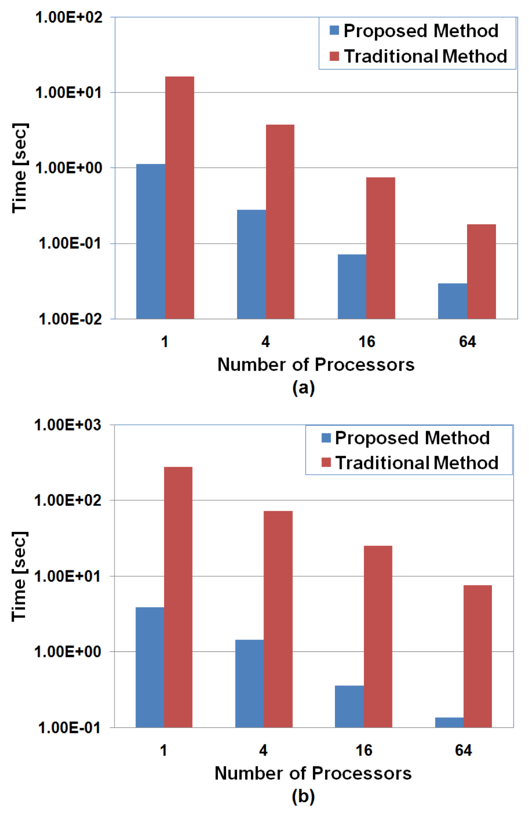


Figure 7. Computation time for junction tree decomposition based exact inference (proposed method) and rerooting based exact inference (traditional method) on junction trees with (a) 100 cliques and (b) 500 cliques.

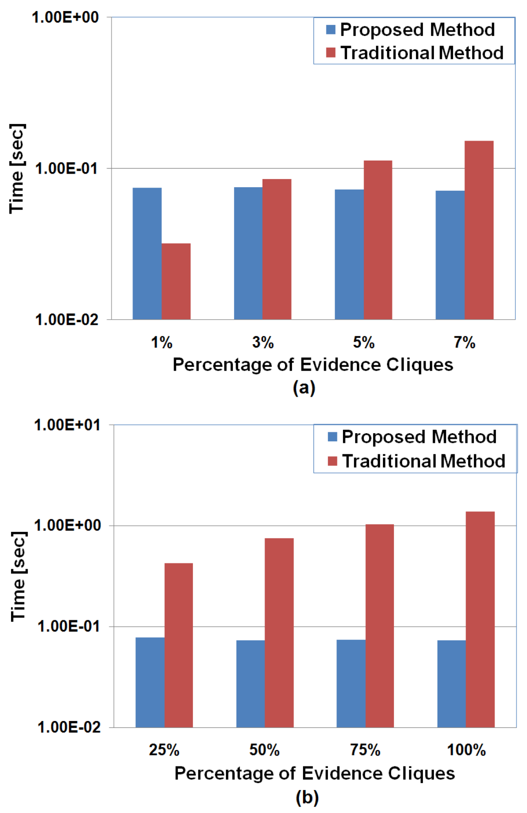


Figure 8. Illustration of computation time versus the percentage of cliques containing evidence variables.

4∼1024. These potential tables were stored as real number arrays. We randomly chose half of the cliques as evidence cliques, so that the two junction trees needed to absorb evidence from 50 cliques and 250 cliques, respectively. We conducted the experiments with 1, 4, 16 and 64 processors.

The data layout was as follows: We assigned the cliques to the processors so that each processor was in charge of approximate N/p cliques, where N is the total number of cliques and p is the number of processors. Specifically, in each processor, there were N/p clique potential tables and all separator potential tables related to these cliques. Also, each processor kept a partial processor mapping table for cliques adjacent to its own. In chain creation, each processor checked if there were any leaf cliques among the cliques assigned to it. Each processor identified the chains corresponding to the leaf cliques assigned to it. After evidence collection and distribution, each processor merged the cliques assigned to it, i.e. each processor was in charge of N/p cliques in this step.

In order to illustrate the scalability of our method, we recorded the computation time of the major steps, including chain creation, evidence collection, evidence distribution and clique merging (see Figure 6). Notice that the computation time for creating the chain was much less than that for the other three steps. Also notice that the actual performance of pointer jumping depended on data size, layout and the processor architecture features. The computation time decreased almost linearly in Figure 6 as the number of processors increased. The axes of these figures are logarithmic. We can see that the proposed algorithm exhibited scalability for the exact inference in junction trees.

We compared the total time for the proposed method in Figure 7 with the traditional method [10] on the junction tree with 100 cliques. Notice that we used logarithmic axes, so the speedup was linear with the number of evidence cliques. We can see that the proposed method exhibited scalability, and was superior to the traditional method. In Figure 8, we show that our method's constant computation time was independent of the number of cliques containing evidence, while the computation time for traditional methods increased with the number of cliques containing evidence. However, because of the overhead for decomposition, the proposed method took more time when only one clique contains evidence. From Figure 8, we see that the speedup of our proposed method was more obvious for the junction tree with larger numbers of evidence cliques. Our method maintained a constant computation time for different numbers of cliques containing evidence, while the computation time for the rerooting technique based exact inference method increased with the number of cliques containing evidence.

8. Conclusion

In this paper, we developed a novel parallel exact inference algorithm based on junction tree decomposition. We decomposed a junction tree into a group of chains and performed evidence propagation on those chains in parallel. Then, the cliques that were duplicated into multiple copies were merged to obtain the fully updated clique potential functions. Pointer jumping based techniques were used in both evidence collection and evidence propagation. The experimental results indicate the scalability of this method. As part of our future work, we intend to investigate the parallelization of the merge step, and load balancing of the computation. For load balancing, the complexity of each chain will be estimated, and the chains will be assigned to processors. We also intend to study minimizing the increase in the total work by reducing clique duplications.

Acknowledgment

We would like to thank Sudhir Vinjamuri for carefully reading the draft of this paper.

References

[1] D. Bader. High-performance algorithm engineering for large-scale graph problems and computational biology. In *4th International Workshop on Efficient and Experimental Algorithms*, pages 16–21, 2005.

[2] R. Biswas, B. Hendrickson, and G. Karypis. Graph partitioning and parallel computing. In *Parallel Computing*, volume 12, pages 1515–1517, 2000.

[3] D. Heckerman. Bayesian networks for data mining. In *In Data Mining and Knowledge Discovery*, 1997.

[4] Y.-S. Hwang and J. H. Saltz. Identifying parallelism in programs with cyclic graphs. *Journal of Parallel and Distributed Computing*, 63(3):337–355, 2003.

[5] Intel Open Source Probabilistic Networks Library. http://www.intel.com/technology/computing/pnl/.

[6] L. V. Kale, B. H. Richards, and T. D. Allen. Efficient parallel graph coloring with prioritization. In *Lecture Notes in Computer Science*, volume 1068, pages 190–208. 1995.

[7] A. V. Kozlov and J. P. Singh. A parallel Lauritzen-Spiegelhalter algorithm for probabilistic inference. In *Supercomputing*, pages 320–329, 1994.

[8] S. L. Lauritzen and D. J. Spiegelhalter. Local computation with probabilities and graphical structures and their application to expert systems. *J. Royal Statistical Society B*, 50:157–224, 1988.

[9] V. K. Namasivayam, A. Pathak, and V. K. Prasanna. Scalable parallel implementation of Bayesian network to junction tree conversion for exact inference. In *Proceedings of the 18th International Symposium on Computer Architecture and High Performance Computing*, pages 167–176, 2006.

[10] V. K. Namasivayam and V. K. Prasanna. Scalable parallel implementation of exact inference in Bayesian networks. In *Proceedings of the 12th International Conference on Parallel and Distributed Systems*, pages 143–150, 2006.

[11] D. Pennock. Logarithmic time parallel Bayesian inference. In *Proceedings of the 14th Annual Conference on Uncertainty in Artificial Intelligence*, pages 431–438, 1998.

[12] S. J. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach (2nd Edition).* Prentice Hall, December 2002.

[13] P. Sadayappan, F. Ercal, and J. Ramanujam. Partitioning graphs on message-passing machines by pairwise mincut. *Information Sciences – Informatics and Computer Science: An International Journal*, 111(1-4):223–237, 1998.

[14] San Diego Supercomputer Center. http://www.sdsc.edu.

[15] K. Schloegel, G. Karypis, and V. Kumar. Parallel multilevel algorithms for multi-constraint graph partitioning. In *Lecture Notes in Computer Science*, volume 1900, pages 296–310, 2001.

[16] E. Segal, B. Taskar, A. Gasch, N. Friedman, and D. Koller. Rich probabilistic models for gene expression. In *9th International Conference on Intelligent Systems for Molecular Biology*, pages 243–252, 2001.

[17] R. D. Shachter, S. K. Andersen, and P. Szolovits. Global conditioning for probabilistic inference in belief networks. In *Proceedings of the Tenth Conference on Uncertainty in Articial Intelligence*, pages 514–522, 1994.

[18] Y. Xia and V. K. Prasanna. Node level primitives for parallel exact inference. In *Proceedings of the 19th International Symposium on Computer Architecture and High Performance Computing*, pages 221–228, October 2007.