Scalable Node Level Computation Kernels for Parallel Exact Inference

Yinglong Xia, and Viktor K. Prasanna, *Fellow, IEEE*

Abstract—In this paper, we investigate data parallelism in exact inference with respect to arbitrary junction trees. Exact inference is a key problem in exploring probabilistic graphical models, where the computation complexity increases dramatically with clique width and the number of states of random variables. We study potential table representation and scalable algorithms for node level primitives. Based on such node level primitives, we propose computation kernels for evidence collection and evidence distribution. A data parallel algorithm for exact inference is presented using the proposed computation kernels. We analyze the scalability of node level primitives, computation kernels and the exact inference algorithm using the coarse grained multicomputer (CGM) model. According to the analysis, we achieve $O\left(Nd_{\mathcal{C}}w_{\mathcal{C}}\prod_{j=1}^{w_{\mathcal{C}}}r_{\mathcal{C},j}/P\right)$ local computation time and $O(N)$ global communication rounds using P processors, $1 \leq P \leq \max_{\mathcal{C}}\prod_{j=1}^{w_{\mathcal{C}}}r_{\mathcal{C},j}$, where N is the number of cliques in the junction tree; $d_{\mathcal{C}}$ is the clique degree; $r_{\mathcal{C},j}$ is the number of states of the j^{th} random variable in $\mathcal{C}$; $w_{\mathcal{C}}$ is the clique width; and w_s is the separator width. We implemented the proposed algorithm on state-of-the-art clusters. Experimental results show that the proposed algorithm exhibits almost linear scalability over a wide range.

Index Terms—Exact inference, Node level primitives, Junction tree, Bayesian network, Message passing.

1 Introduction

A full joint probability distribution for any real-world systems can be used for inference. However, such a distribution grows intractably with the number of variables used to model the system. Bayesian networks [1] are used to represent joint probability distributions compactly by exploiting conditional independence relationships. Bayesian networks have found applications in a number of domains, including medical diagnosis, credit assessment, data mining, image analysis, and genetics [2][3][4][5].

Inference on a Bayesian network is the computation of the conditional probability of certain variables, given a set of *evidence variables* as knowledge to the network. Such knowledge is also known as *belief*. Inference on Bayesian networks can be *exact* or *approximate*. Exact inference is NP hard [1]. The most popular exact inference algorithm converts a given Bayesian network into a junction tree, and then performs exact inference in the junction tree [6]. Huang and Darwiche synthesized various optimizations of sequential exact inference using junction trees [7]. 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. In many cases exact inference must be performed in real time. Therefore, in order to accelerate the exact inference, parallel techniques must be developed.

Several parallel algorithms and implementations of exact inference have been presented. Early work on parallel algorithms for exact inference appeared in [1], [8] and [9], which formed the basis of scalable parallel implementations discussed in [10]. In [10], the authors present the parallelization of exact inference using pointer jumping, focusing on the structure level parallelism.

The structure level parallelism can not offer large speedups when the size of the cliques or the number of states of the variables in a given junction tree increases, making the operations with respect to potential tables the dominant part of the problem. Unlike [10], we start with a junction tree and explore the *data parallelism*, including potential table representation and the parallelization of the operations with respect to potential tables. In this paper, we refer to the operations with respect to potential tables as *node level primitives*. We present scalable algorithms for the node level primitives. A composition of the node level primitives in a certain order (see Section 5.4) can be used to update the potential table in a clique. Such a composition is called a scalable *computation kernel* for evidence propagation. We present a data parallel algorithm for exact inference by traversing the cliques in a junction tree and updating each clique using the proposed computation kernels.

For the sake of illustrating the performance of parallel algorithms, several models of computation have been well studied. The parallel random access machine (PRAM) model is very straightforward. However, speedup results for the theoretical PRAM model do not necessarily match the speedups observed on real machines [11]. The PRAM model assumes *implicit*

This research was partially supported by the National Science Foundation under grant number CNS-0613376. NSF equipment grant CNS-0454407 is gratefully acknowledged.

- *Y. Xia is with the Computer Science Department, University of Southern California, Los Angeles, CA, 90089. E-mail: yinglonx@usc.edu*
- *V. K. Prasanna is with the Ming Hsieh Department of Electrical Engineering and the Computer Science Department, University of Southern California, Los Angeles, CA, 90089. E-mail: prasanna@usc.edu*

communication and ignores the communication cost. Since our target platform is a cluster with distributed memory, where the communication cost is relatively expensive, the PRAM model is *not* suitable for us. The bulk synchronous parallel (BSP) model requires a low or fixed gap g [12]. The coarse grained multicomputer (CGM) model generalizes the BSP model and consists of alternating local computation and global communication rounds [13]. The CGM model is essentially the BSP model with additional packing and coarse grained requirements, which captures the behavior of explicit message passing. Therefore, we utilize the CGM model to analyze the proposed algorithms in this paper.

We summarize our key contributions in this paper: 1) We explore data parallelism and present scalable algorithms for the node level primitives. 2) We propose two scalable computation kernels using the node level primitives, one for evidence collection and the other for evidence distribution. The computation kernels are constructed by assembling the node level primitives in a certain order, so that a potential table can be updated using these primitives. 3) We present a data parallel algorithm for exact inference using the computation kernels. 4) We analyze the node level primitives, computation kernels and the parallel exact inference algorithm using the CGM model. The parallel exact inference algorithm achieves $O\left(Nd_{\mathcal{C}}w_{\mathcal{C}}\prod_{j=1}^{w_{\mathcal{C}}} r_{\mathcal{C},j}/P\right)$ local computation time and $O(N)$ global communication rounds with respect to P processors, where N is the number of cliques in the junction tree; $d_{\mathcal{C}}$ is the clique degree; $r_{\mathcal{C},j}$ is the number of states of the j^{th} random variable in $\mathcal{C}$; $w_{\mathcal{C}}$ is the clique width; and w_s is the separator width. The exact inference algorithm scales linearly over the range $1 \leq P \leq \max_{\mathcal{C}} \prod_{j=1}^{w_{\mathcal{C}}} r_{\mathcal{C},j}$, compared to $1 \leq P \leq n$ for most structure level parallel methods [1], where n is the number of nodes in the Bayesian network. 5) We implement the parallel exact inference algorithm on state-of-the-art clusters using message passing interface (MPI). 6) We experimentally evaluate the data parallel algorithm for exact inference using various junction trees and show linear scalability.

The paper is organized as follows: Section 2 discusses the background of Bayesian networks, junction trees and the CGM model. Section 3 addresses related work on parallel exact inference. Section 4 presents the representation for potential tables in a given junction tree. Section 5 discusses evidence propagation, node level primitives and computation kernels for exact inference. Section 6 presents the parallel exact inference algorithm based on the node level primitives. Experiments are shown in Section 7. Section 8 concludes the paper.

2 Background

2.1 Exact Inference

A *Bayesian network* is a probabilistic graphical model that exploits conditional independence to represent a joint distribution compactly. 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 exists an edge from A_i to A_j i.e. $(A_i, A_j) \in \mathcal{E}$, then 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 probabilities $P(A_j|pa(A_j))$ for each random variable A_j. Given the Bayesian network, a joint distribution $P(\mathcal{V})$ can be given as [6]:

$$P(\mathcal{V}) = P(A_1, A_2, \cdots, A_n) = \prod_{j=1}^{n} P(A_j|pa(A_j))$$

.

The *evidence* in a Bayesian network is the variables that have been instantiated with values, for example, $\mathrm{E} = \{A_{e_1} = a_{e_1}, \cdots, A_{e_c} = a_{e_c}\}$, $e_k \in \{1, 2, \ldots, n\}$. Given the evidence, the new distribution of any other variable can be inquired. The variables to be inquired are called *query* variables, i.e. the random variables in which we are interested. *Exact inference* propagates the evidence throughout the entire network so that the conditional distribution of the query variables can be computed.

Traditional exact inference using Bayes' Theorem fails for networks with directed cycles [6]. 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) [6]. $\hat{\mathbb{P}}$ is a group of *potential tables* (POTs). 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.

In a junction tree, exact inference proceeds as follows: Assuming evidence $\mathrm{E} = \{A_i = a\}$ and $A_i \in \mathcal{C}_j$, E is *absorbed* at $\mathcal{C}_j$ by instantiating the variable A_i and renormalizing the remaining constituents of the clique. Each updated clique continues on propagating the evidence to all non-updated adjacent cliques through separators, until all cliques are updated. Mathematically, the evidence propagation between two adjacent cliques is represented as [6]:

$$\psi^*_{\mathcal{S}} = \sum_{\mathcal{Y}\setminus\mathcal{S}} \psi^*_{\mathcal{Y}}, \psi^*_{\mathcal{X}} = \psi_{\mathcal{X}} \frac{\psi^*_{\mathcal{S}}}{\psi_{\mathcal{S}}} \tag{1}$$

where $\mathcal{S}$ is a separator between cliques $\mathcal{X}$ and $\mathcal{Y}$; ψ^* denotes the updated potential table; ψ is the stale table. 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 $\psi_{\mathcal{C}_y}$.

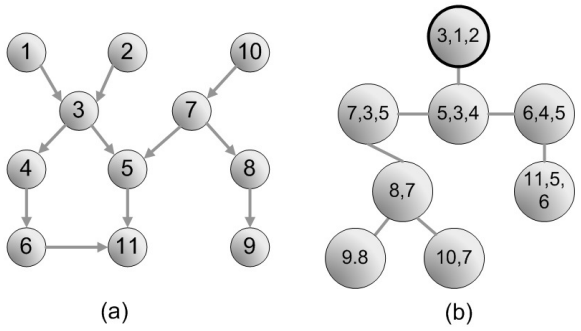


Fig. 1. An example of (a) Bayesian network and its (b) junction tree. The bold circle indicates the root of the junction tree.

2.2 CGM Model

The *coarse grained multicomputer* (CGM) model, as a practical version of the BSP model [12], was proposed by Dehne *et al*. [13]. The CGM model assumes a system consisting of P processors, $P_1, P_2, \cdots, P_p$, with $O(m/P)$ local memory per processor and an arbitrary communication network. Here m refers to the total memory capacity. Using an arbitrary communication network, the processors can communicate with each other by sending and receiving messages over the network. All algorithms based on the CGM model consist of alternating *local computation* and *global communication* rounds. In every global communication round, each processor can send $O(m/P)$ data and each processor can receive $O(m/P)$ data. The CGM model requires that all data sent from any processor to another processor in one global communication round be packed into one long message, therefore minimizing the overhead.

Finding an optimal algorithm in the CGM model is equivalent to minimizing the number of global communication rounds, the size of data transferred in each global communication round and the local computation time. Chan [13] indicates the importance of reducing the number of global communication rounds to a *constant* or to a slowly growing function of P that is independent of m, e.g. $\log P$ and $\log^2 P$. As shown in [14], a number of global communication rounds independent of m leads to parallel algorithms with good speedup in theory and practice because of a good amortization of communication overhead: When m increases, the number of messages remains constant and *only* the message size grows. Thus, the total message overhead remains unchanged, but the message overhead per data unit decreases.

3 Related Work

There are several works on parallel exact inference, such as Pennock [1], Kozlov and Singh [8], and Szolovits [9]. However, the performance of some of those methods, such as [8], is dependent upon the structure of the Bayesian network. Others, such as [1], exhibit limited performance for multiple evidence inputs, since the evidence is assumed to be at the root of the junction tree. Rerooting techniques are employed to deal with the case where the evidence appears at more than one clique. Some other works address certain individual steps of exact inference. The algorithm given in [10] also explores structural parallelism, although the authors used an OpenMP clause to accelerate the execution of loops in the implementation. The algorithm in [10] is suitable for junction trees with high topological parallelism, but it does not parallelize the node level primitives across compute nodes in a cluster. Note that structure level parallelism can not offer large speedups when the size of the cliques in a junction tree or the number of states of the random variables increases, making node level operations the dominant part of the problem. Thus, we focus on data parallelism. Reference [15] introduces parallel node level primitives. However, in [15], all the random variables must have exactly the same number of states. Such a constraint is ignored in this paper. Unlike [15], we optimize the node level primitives with respect to evidence collection and evidence distribution, respectively. We propose computation kernels in Section 5.4 using the optimized node level primitives. Reference [15] analyzes node level primitives using the PRAM model, where the communication cost is ignored. This paper uses the CGM model to capture both the computation and communication costs.

4 Potential Table Organization

4.1 Potential Table Representation

Each node of a junction tree denotes a clique, which is a set of random variables. For each clique $\mathcal{C}$, there is a *potential function*, $\psi_{\mathcal{C}}$, which describes the joint distribution of the random variables in the clique [6]. The discrete form of the potential function is called a *potential table*. A potential table is a list of non-negative real numbers, where each number corresponds to a probability of the joint distribution. The straightforward representation for potential tables stores the state string along with each entry [10]. In such a representation, a potential value can be stored in any entry of the potential table. However, for large potential tables, such a representation occupies a large amount of memory. In addition, frequently checking the state string of an entry adversely affects the performance. For the sake of enhancing the performance of computation, we carefully organize the potential tables. The advantages of our representation include: 1) reduced memory requirement; 2) direct access to potential values based on the mapping relationship.

We define some terms to explain the potential table organization. We assign an order to the random variables in a clique to form a *variable vector*. We will discuss how to determine the order in Section 4.2. For a clique $\mathcal{C}$, the variable vector is denoted $V_{\mathcal{C}}$. Accordingly, the combination of states of the variables in the variable vector forms *state strings*. A state string is denoted $S = (s_1 s_2 \cdots s_w)$, where w is the clique width and $s_i \in \{0, 1, \cdots, r_i - 1\}$ is

the state of the i^{th} variable in the variable vector. Thus, there are $\prod_{i=1}^{w} r_i$ state strings with respect to $V_{\mathcal{C}}$. Since for each state string there is a corresponding potential (probability), we need an array of $\prod_{i=1}^{w} r_i$ entries to store the corresponding potential table.

Traditionally, state strings are stored with a potential table, because both the state strings and potentials are required in the computation of evidence propagation. As each potential corresponds to a state string, we need $O(w\prod_{i=1}^{w} r_i)$ memory to store the state strings, w times larger than that for storing a potential table. In addition, this approach leads to large message size in communication among processors.

We optimize the potential table representation by finding the relationship between array indices and state strings. That is, we encode a given state string $S = (s_1 s_2 \cdots s_w)$ as an integer number t, where $s_i \in \{0, 1, \cdots, r_i - 1\}$ and $t \in \{1, 2, \cdots, \prod_{i=1}^{w} r_i\}$:

$$t = 1 + \sum_{i=1}^{w} s_i \prod_{j=1}^{i-1} r_j \tag{2}$$

The formula that maps an index t to a state string $S = (s_1 s_2 \cdots s_w$, where $s_i, 1 \le i \le w$, is given by:

$$s_i = \left\lfloor \frac{t-1}{\prod_{j=1}^{i-1} r_j} \right\rfloor \%r_i \tag{3}$$

where $\%$ is the modulo operator.

To demonstrate the correctness of Eq. (2), we briefly prove that, for an arbitrary state string S, t is a legal index of the potential table. That is, t is an integer and $1 \le t \le \prod_{i=1}^{w} r_i$. Because s_i and r_j are integers for any i, it is apparent that t is also an integer. Since s_i and r_j are nonnegative, $\sum_{i=1}^{w} s_i \prod_{j=1}^{i-1} r_j$ is also nonnegative. Thus, $t \ge 1$ is satisfied. To prove $t \le \prod_{i=1}^{w} r_i$, we notice that $s_i \in \{0, 1, \cdots, r_i - 1\}$ i.e. $s_i \le r_i - 1$. Therefore, Eq. (3) is given by:

$$\begin{aligned} t &= 1 + \sum_{i=1}^{w} s_i \prod_{j=1}^{i-1} r_j \le 1 + \sum_{i=1}^{w} \left((r_i - 1) \prod_{j=1}^{i-1} r_j \right) \\ &= 1 + \sum_{i=1}^{w} \left(\prod_{j=1}^{i} r_j - \prod_{j=1}^{i-1} r_j \right) = \prod_{i=1}^{w} r_i \end{aligned} \tag{4}$$

Therefore, for a given state string S, Eq. (2) always maps S to an index of the potential table.

To demonstrate the correctness of Eq. (3), we prove that, using the expression of t given in Eq. (2), $\left\lfloor (t-1)/(\prod_{j=1}^{i-1} r_j) \right\rfloor \%r_i$ in Eq. (3) gives s_i, the i-th element of S. Notice that $\lfloor x \rfloor$ rounds x, and $\%$ is the modulo operator. We have:

$$\begin{aligned} \left\lfloor \frac{t-1}{\prod_{j=1}^{i-1} r_j} \right\rfloor \%r_i &= \left\lfloor \frac{\left(1 + \sum_{k=1}^{w} s_k \prod_{j=1}^{k-1} r_j\right) - 1}{\prod_{j=1}^{i-1} r_j} \right\rfloor \%r_i \\ &= \left\lfloor \sum_{k=i}^{w} s_k \prod_{j=i}^{k-1} r_j + \sum_{k=1}^{i-1} \frac{s_k}{\prod_{j=k}^{i-1} r_j} \right\rfloor \%r_i \\ &= \left(\sum_{k=i+1}^{w} s_k \prod_{j=i}^{k-1} r_j + s_i \right) \%r_i = s_i \end{aligned} \tag{5}$$

In Eq. (5), note that $\sum_{k=i}^{w} s_k \prod_{j=i}^{k-1} r_j$ is an integer and $0 \le \sum_{k=1}^{i-1} s_k / \prod_{j=k}^{i-1} r_j < 1$. Therefore, according to Eq. (5) we have proven that Eq. (3) correctly produces s_i for any $i \in \{1, 2, \cdots, w\}$.

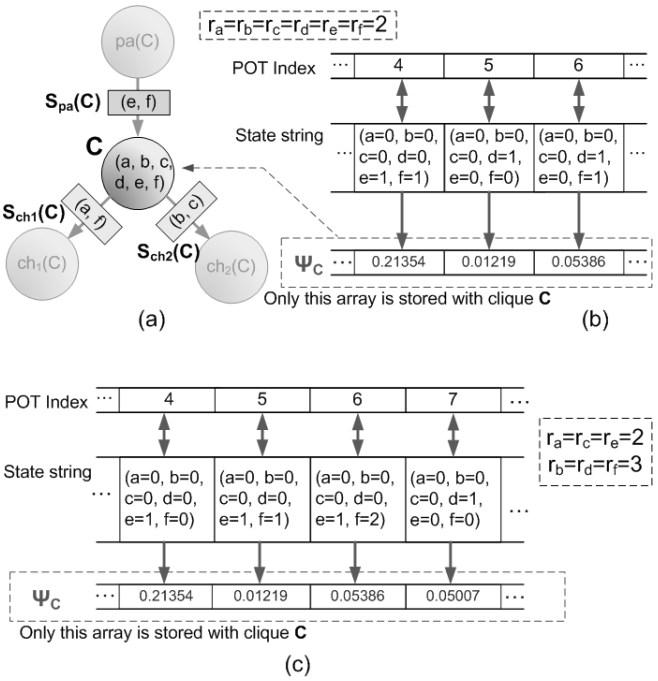


Fig. 2. (a) A sample clique and its variable vector; (b) The relationship among the potential table (POT) index, state string and potential table, assuming all random variables are binary variables; (c) The relationship among the array index, state string and potential table, assuming a, c, e are binary while b, d, f are ternary.

We organize the potential table using Eq. (2) and Eq. (3). For each entry index t ($t = 1, 2, \cdots, \prod_i r_i$), we convert t to a state string S using Eq. (2). For a given state string S, we store the corresponding potential in the t-th entry of the potential table, where t is obtained from S according to Eq. (2). For example, we show a segment of a sample potential table in Figure 2. In Figure 2 (a), we show a sample clique $\mathcal{C}$ with variable vector (a, b, c, d, e, f). A segment of the potential table (POT) for $\mathcal{C}$, i.e. $\psi_{\mathcal{C}}$, is given in Figure 2 (b). For each entry of $\psi_{\mathcal{C}}$, the corresponding state string and index are also presented. For the sake of illustration, we assume all ran-

dom variables are binary in Figure 2 (b). In Figure 2 (c), however, we assume random variables a, c, e are binary, while b, d, f are ternary. Using Equations (2) and (3), we obtain the relationship among indices, state strings and entries of the potential table shown in Figure 2 (c).

4.2 Separators Between Cliques

A separator is the intersection of two adjacent cliques. Therefore, for each edge in a junction tree, there is a separator with respect to the two cliques connected to the edge. In Figure 2 (a), we show three separators related to clique $\mathcal{C}$. $S_{pa}(\mathcal{C})$ is the separator between $\mathcal{C}$ and its parent, $pa(\mathcal{C})$. $S_{ch_1}(\mathcal{C})$ and $S_{ch_2}(\mathcal{C})$ are two separators between $\mathcal{C}$ and its two children, $ch_1(\mathcal{C})$ and $ch_2(\mathcal{C})$. The terms defined in the previous section, such as variable vector and state string, are also applied to separators. Using Eq. (2) and Eq. (3), we organize the potential tables for separators as we do for cliques.

For each clique $\mathcal{C}$ in the given junction tree, we impose the following requirement on the variable vector of $\mathcal{C}$ and the variable vector of $S_{pa}(\mathcal{C})$: the variable vector of $\mathcal{C}$ is ordered by $V_{\mathcal{C}} = (V_{\mathcal{C}\backslash S_{pa}(\mathcal{C})}, V_{S_{pa}(\mathcal{C})})$, where $V_{S_{pa}(\mathcal{C})}$ is the variable vector of $S_{pa}(\mathcal{C})$, and $V_{\mathcal{C}\backslash S_{pa}(\mathcal{C})}$ consists of random variables existing in $V_{\mathcal{C}}$ but not in $V_{S_{pa}(\mathcal{C})}$. The order of variables inside of $V_{\mathcal{C}}$ and $V_{S_{pa}(\mathcal{C})}$ is arbitrary. The advantage of such a requirement is the simplification of the computation in evidence propagation. Details are given in Section 5.4.

In evidence propagation, we propagate evidence from the *input separators* to $\mathcal{C}$, and then utilize the updated $\psi_{\mathcal{C}}$ to renew the *output separators*. We define *input separators* with respect to clique $\mathcal{C}$ as the separators which carry evidence information before updating $\mathcal{C}$. The *output separators* are defined as the separators to be updated. Taking Figure 3 as an example, in evidence collection, $S_{ch_1}(\mathcal{C})$ and $S_{ch_2}(\mathcal{C})$ are the input separators while $S_{pa}(\mathcal{C})$ is the output separator. In evidence distribution, $S_{pa}(\mathcal{C})$ becomes the input separator while $S_{ch_1}(\mathcal{C})$ and $S_{ch_2}(\mathcal{C})$ become the output separators.

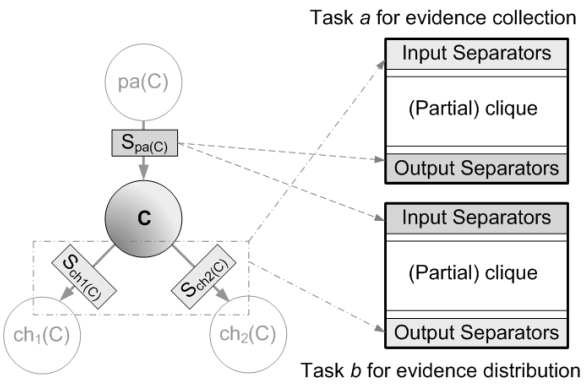


Fig. 3. Illustration of input and output separators of clique $\mathcal{C}$ with respect to evidence collection and evidence distribution.

5 Node Level Primitives

Evidence propagation is a series of computations on the clique potential tables and separators. Based on the node level probability representation addressed in Section 4, we introduce four node level primitives for evidence propagation: *table marginalization*, *table extension*, *table multiplication*, and *table division*.

5.1 Table Marginalization

Table marginalization is used to obtain separator potential tables from a given clique potential table. For example, in Figure 4, we marginalize $\psi_{\mathcal{C}}$ to obtain $\psi_{\mathcal{S}_{ch_1(\mathcal{C})}}$, $\psi_{\mathcal{S}_{ch_2(\mathcal{C})}}$ and $\psi_{\mathcal{S}_{pa(\mathcal{C})}}$.

To obtain the potential table for the i^{th} child of a given clique $\mathcal{C}$, i.e. $\psi_{\mathcal{S}_{ch_i(\mathcal{C})}}$, we marginalize the potential table $\psi_{\mathcal{C}}$. The marginalization requires identifying the mapping relationship between $V_{\mathcal{S}_{ch_i(\mathcal{C})}}$ and $V_{\mathcal{C}}$. We define the *mapping vector* to represent the mapping relationship from $V_{\mathcal{C}}$ to $V_{\mathcal{S}_{ch_i(\mathcal{C})}}$. The mapping vector is defined as $M_{ch_i(\mathcal{C})} = (m_1 m_2 \cdots m_{w_{\mathcal{S}_{ch_i}}} | m_j \in \{1, \cdots, w\})$, where w is the width of clique $\mathcal{C}$ and $w_{\mathcal{S}_{ch_i}}$ is the length of $V_{ch_i(\mathcal{C})}$. Notice that $V_{\mathcal{S}_{ch_i(\mathcal{C})}} \subset V_{\mathcal{C}}$. The value of m_j is determined if the m_j^{th} variable in $V_{\mathcal{C}}$ is the same as the j^{th} variable in $V_{\mathcal{S}_{ch_i(\mathcal{C})}}$. Using the mapping vector $M_{ch_i(\mathcal{C})}$, we identify the relationship between $\psi_{\mathcal{C}}$ and $\psi_{\mathcal{S}_{ch_i(\mathcal{C})}}$. Given an entry $\psi_{\mathcal{C}}(t)$ in a potential table $\psi_{\mathcal{C}}$, we convert the index t to a state string $S = (s_1 s_2 \cdots s_w)$. Then, we construct a new state string $\tilde{S} = (\tilde{s}_1 \tilde{s}_2 \cdots \tilde{s}_{ch_i(\mathcal{C})})$ by letting $\tilde{s}_i = s_{m_i}$. The new state string $\tilde{S}$ is then converted back to an index $\tilde{t}$. Therefore, we show that $\psi_{\mathcal{C}}(t)$ corresponds to $\psi_{\mathcal{S}_{ch_i(\mathcal{C})}}(\tilde{t})$. To compute $\psi_{\mathcal{S}_{ch_i(\mathcal{C})}}$ from $\psi_{\mathcal{C}}$, we identify the relationship for each t and accumulate $\psi_{\mathcal{C}}(t)$ to $\psi_{\mathcal{S}_{ch_i(\mathcal{C})}}(\tilde{t})$. We illustrate table marginalization for obtaining a separator $\psi_{\mathcal{S}_{ch_i(\mathcal{C})}}$ from a given clique potential table $\psi_{\mathcal{C}}$ in Figure 4.

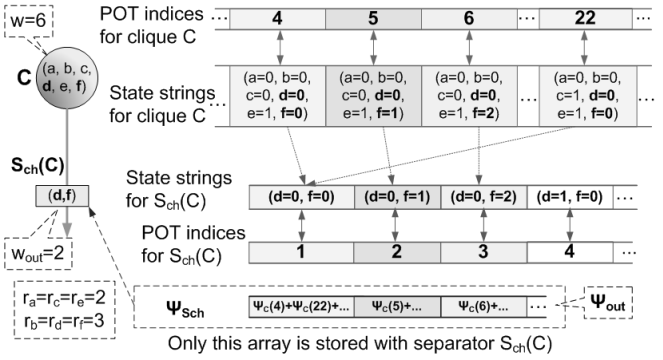


Fig. 4. Illustration of using table marginalization to obtain separator $\psi_{\mathcal{S}_{ch_i}(\mathcal{C})}$ from clique potential table $\psi_{\mathcal{C}}$.

Algorithm 1 describes table marginalization for obtaining $\psi_{\mathcal{S}_{ch_i(\mathcal{C})}}$ from $\psi_{\mathcal{C}}$. Let w and w_{out} denote the clique width and the separator width, respectively. The input to table marginalization includes clique potential table $\psi_{\mathcal{C}}$, a mapping vector $M_{\mathcal{C}} = (m_1, m_2, \cdots, m_{w_{out}})$, and the number of processors P. The output is separator potential table ψ_{out} i.e. $\psi_{\mathcal{S}_{ch_i(\mathcal{C})}}$. Each processor

is in charge of a segment of $\psi_{\mathcal{C}}$, consisting of $|\psi_{\mathcal{C}}|/P$ contiguous entries. Line 1 in Algorithm 1 launches P-way parallelism and assigns an ID, p, to each processor. Lines 2-8 form a local computation round. Line 2 in Algorithm 1 initializes the output on the local processor, denoted $\psi_{out}^{(p)}$. In Lines 4-7, each processor updates $\psi_{out}^{(p)}$. Notice that Line 3 does not reallocate the potential table, but converts the indices of the partitioned table into the corresponding indices of the original potential table. Then, in Line 4, the index scalar is converted into a state string using Eq. (3). Line 5 transforms the state string by assigning $\tilde{s}_i = s_{m_i}$ for $i = 1, 2, \cdots, w_{out}$. The resulting state string is converted back to an index in Line 6. Line 9 is a global communication round, where the all-to-all communication is performed to broadcast the local result $\psi_{out}^{(p)}$. Each processor receives $\psi_{out}^{(j)}$ from the j-th processor ($j = 1, 2, \cdots, P$ and $j \neq p$) and accumulates (Line 10) $\psi_{out}^{(j)}$ to its local result. The sum calculated in Line 10 gives the updated separator potential table $\psi_{\mathcal{S}_{ch_i(\mathcal{C})}}$.

Algorithm 1 Marginalization for separator $\psi_{\mathcal{S}_{ch_i(\mathcal{C})}}$

Input: clique potential table $\psi_{\mathcal{C}}$, mapping vector $M_{\mathcal{C}}$, number of processors P
Output: separator potential table ψ_{out}

1: **for** $p = 1$ to P **pardo**
   {local computation}
2: $\psi_{out}^{(p)}(1 : |\psi_{out}|) = \vec{0}$
3: **for** $t = \frac{|\psi_{\mathcal{C}}|}{P}(p-1)$ to $\frac{|\psi_{\mathcal{C}}|}{P}p - 1$ **do**
4: convert t to $S = (s_1 s_2 \cdots s_w)$
5: construct $\tilde{S} = (\tilde{s}_1 \tilde{s}_2 \cdots \tilde{s}_{w_{out}})$ from S using mapping vector $M_{\mathcal{C}}$
6: convert $\tilde{S}$ to $\tilde{t}$
7: $\psi_{out}^{(p)}(\tilde{t}) = \psi_{out}^{(p)}(\tilde{t}) + \psi_{\mathcal{C}}(t)$
8: **end for**
   {global communication}
9: broadcast $\psi_{out}^{(p)}$ and receive $\psi_{out}^{(j)}$ from Processor j ($j = 1, \cdots, P$ and $j \neq p$)
   {local computation}
10: $\psi_{out} = \sum_{j=1}^{P} \psi_{out}^{(j)}$
11: **end for**

For the sake of illustrating the scalability of the algorithm, we analyze Algorithm 1 using the CGM model introduced in Section 2.2. Line 1 takes constant time to initialize. Let $w_{\mathcal{C}}$ denote the clique width of $\mathcal{C}$. Line 4 takes $3w_{\mathcal{C}}$ time, since Eq. (3) computes $w_{\mathcal{C}}$ elements, each involving three scalar operations: a division, a modulo and the increase of the product of r_j. Line 5 takes w_{out} time and Line 6 takes $3w_{out}$ time, since Eq. (2) also involves three scalar operations. Note that $w_{out} < w_{\mathcal{C}}$. Line 7 takes constant time. Line 9 performs all-to-all communication, which is the only global communication round for Algorithm 1. The operation of Lines 9 and 10 is known as *all-reduce* [16]. By organizing the processors into a spanning tree with logarithmic depth, all-reduce takes $O(|\psi_{out}| \log P)$ time [17]. Since $|\psi_{out}| \ll |\psi_{\mathcal{C}}|$ in our context, the local computation time is $O(|\psi_{\mathcal{C}}| w_{\mathcal{C}}/P)$, $1 \leq P \leq |\psi_{\mathcal{C}}|$.

5.2 Table Extension

Table extension identifies the mapping relationship between two potential tables and equalizes the size of the two tables. Table extension is an inverse mapping of table marginalization, since the former expands a small table (separator potential table) to a large table (clique potential table), while the latter shrinks a large table to a small table. Table extension is utilized to simplify table multiplication and table division.

Figure 5 illustrates table extension using the sample clique and separator given in Figure 4. The clique consists of 6 random variables where the numbers of states are $r_a = r_c = r_e = 2$ and $r_b = r_d = r_f = 3$. The separator consists of 2 random variables r_d and r_f. In Figure 5 (a), we illustrate the state strings of $\psi_{\mathcal{S}(\mathcal{C})}$, the corresponding state strings of $\psi_{\mathcal{C}}$, and the indices. We can see from Figure 5 (a) that an entry in $\psi_{\mathcal{S}(\mathcal{C})}$ corresponds to multiple entries in $\psi_{\mathcal{C}}$. In addition, the corresponding entries in $\psi_{\mathcal{C}}$ may not be contiguous. However, after applying table extension, each entry in $\psi_{\mathcal{S}(\mathcal{C})}$ corresponds to the entry with the same index in $\psi_{\mathcal{C}}$ (see Figure 5 (b)).

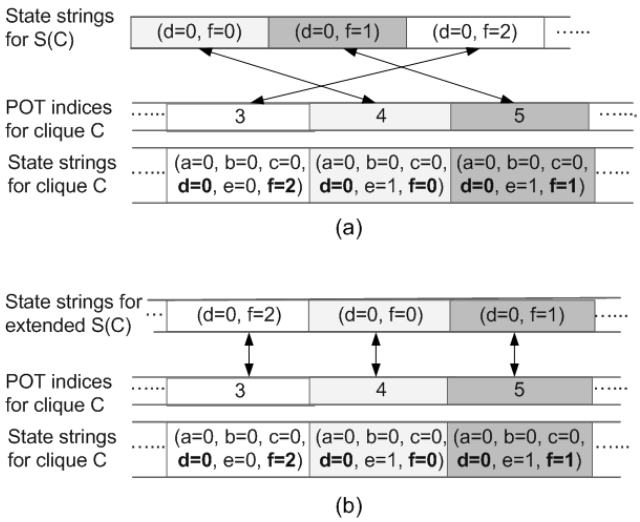


Fig. 5. (a) The mapping relationship between separator $\psi_{\mathcal{S}(\mathcal{C})}$ and clique potential table $\psi_{\mathcal{C}}$; (b) The mapping relationship between extended $\psi_{\mathcal{S}(\mathcal{C})}$ and $\psi_{\mathcal{C}}$, where every entry in $\psi_{\mathcal{S}(\mathcal{C})}$ corresponds to the entry in $\psi_{\mathcal{C}}$ with the same index.

The parallel algorithm for the primitive of table extension is shown in Algorithm 2. The input to table extension includes the original separator $\psi_{\mathcal{S}_{(\mathcal{C})}}$, the size of the corresponding clique potential table $|\psi_{\mathcal{C}}|$, the mapping vector $M_{\mathcal{C}} = (m_1, m_2, \cdots, m_{w_{out}})$, and the number of processors P. The output is an extended separator table ψ_{ext}. Regarding the data layout, we assume that a

segment of ψ_{ext} ($|\psi_{\mathcal{C}}|/P$ entries) is maintained in local memory. Other inputs are stored in every local memory. Line 2 in Algorithm 2 initializes the output by allocating memory for $|\psi_{\mathcal{C}}|$ entries. In Line 3-7, each processor assigns values to ψ_{ext} in parallel. Line 4 converts an index scalar to a state string using Eq. (3). Line 5 transforms the state string by assigning $\tilde{s}_i = s_{m_i}$ for $i = 1, 2, \cdots, w_{out}$. The resulting state string is converted back to an index in Line 6. Line 7 copies the value in the identified entry to $\psi_{ext}(t)$. No communication is needed in table extension.

Using the CGM model, we analyze the complexity of Algorithm 2. Line 2 takes constant time for memory allocation. As with the analysis for Algorithm 1, we know Line 4 in Algorithm 2 takes $3w_{\mathcal{C}}$ time for local computation. Line 5 takes $w_{\mathcal{S}}$ time, where $w_{\mathcal{S}}$ is the width of separator $\mathcal{S}$. Line 6 requires $3w_{\mathcal{S}}$ time since we need three scalar operations to obtain each element of $\tilde{S}$. Therefore, the local computation time is $O(|\psi_{\mathcal{C}}|w_{\mathcal{C}}/P)$, where $1 \leq P \leq |\psi_{\mathcal{C}}|$.

Algorithm 2 Table extension for separator $\psi_{\mathcal{S}_{(\mathcal{C})}}$

Input: separator potential table $\psi_{\mathcal{S}}$, the size of clique potential table $|\psi_{\mathcal{C}}|$, mapping vector $M_{\mathcal{C}}$, number of processor P
Output: extended separator potential table ψ_{ext}

1: **for** $p = 1$ to P **pardo**
   {local computation}
2: allocate memory of $|\psi_{\mathcal{C}}|/P$ entries for ψ_{ext}
3: **for** $t = \frac{|\psi_{\mathcal{C}}|}{P}(p-1)$ to $\frac{|\psi_{\mathcal{C}}|}{P}p - 1$ **do**
4: convert t to $S = (s_1 s_2 \cdots s_w)$
5: construct $\tilde{S} = (\tilde{s}_1 \tilde{s}_2 \cdots \tilde{s}_{w_{out}})$ from S using $M_{\mathcal{C}}$
6: convert $\tilde{S}$ to $\tilde{t}$
7: $\psi_{ext}(t) = \psi_{\mathcal{S}}(\tilde{t})$
8: **end for**
9: **end for**

5.3 Table Multiplication and Table Division

In exact inference, table multiplication occurs between a clique potential table and its separators. For each entry in a separator, table multiplication multiplies the potential $\psi_{\mathcal{S}}(t)$ in the separator with another potential $\psi_{\mathcal{C}}(\tilde{t})$ in the clique potential table, where the random variables shared by the separator $\mathcal{S}$ and the clique $\mathcal{C}$ have identical states. Table multiplication requires the identification of the relationship between entries in the separator and those in the clique potential table. We use table extension to identify this relationship.

We present the algorithm for table multiplication in Algorithm 3. The input includes the separator potential table $\psi_{\mathcal{S}}$, the clique potential table $\psi_{\mathcal{C}}$, the mapping vector $M_{\mathcal{C}}$, and the number of processors P. The output is the updated potential table $\psi^*_{\mathcal{C}}$. Line 1 extends the separator with respect to $\psi_{\mathcal{S}}$ using Algorithm 2. Lines 3-5 update $\psi_{\mathcal{C}}$ by multiplying the extended separator and the clique potential table. Each processor is in charge of a segment of the $\psi^*_{\mathcal{C}}$.

Algorithm 3 Table multiplication

Input: separator potential table $\psi_{\mathcal{S}}$, clique potential table $\psi_{\mathcal{C}}$, mapping vector $M_{\mathcal{C}}$, number of processor P
Output: resulting potential table $\psi^*_{\mathcal{C}}$

1: ψ_{ext} = extend $\psi_{\mathcal{S}}$ to size $|\psi_{\mathcal{C}}|$ using Algorithm 2
2: **for** $p = 1$ to P **pardo**
   {local computation}
3: **for** $t = \frac{|\psi_{\mathcal{C}}|}{P}(p-1)$ to $\frac{|\psi_{\mathcal{C}}|}{P}p - 1$ **do**
4: $\psi^*_{\mathcal{C}}(t) = \psi_{\mathcal{C}}(t) * \psi_{ext}(t)$
5: **end for**
6: **end for**

Since Line 1 utilizes Algorithm 2, the local computation time is $O(|\psi_{\mathcal{C}}|w_{\mathcal{C}}/P)$. Lines 3-5 consist of $|\psi_{\mathcal{C}}|/P$ iterations, where each iteration takes $O(1)$ time for local computation. Therefore, the total computation time for Algorithm 3 is also $O(|\psi_{\mathcal{C}}|w_{\mathcal{C}}/P)$, where $1 \leq P \leq |\psi_{\mathcal{C}}|$. Notice that $|\psi_{\mathcal{C}}| = \prod_{i=1}^{w_{\mathcal{C}}} r_i$.

Table division is very similar to table multiplication, as shown in Algorithm 4. According to Eq. (1), table division occurs between two separator potential tables $\psi^*_{\mathcal{S}}$ and $\psi_{\mathcal{S}}$. Algorithm 4 shows the primitive of parallel table division. Similar to the analysis for table multiplication, we obtain the total computation time for Algorithm 3 as $O(|\psi_{\mathcal{C}}|w_{\mathcal{C}}/P)$, where $1 \leq P \leq |\psi_{\mathcal{C}}|$.

Algorithm 4 Table division

Input: separator potential tables $\psi^*_{\mathcal{S}}$ and $\psi_{\mathcal{S}}$, mapping vector $M_{\mathcal{S}}$, number of processor P
Output: resulting potential table $\psi^\dagger_{\mathcal{S}}$

1: ψ_{ext} = extend $\psi_{\mathcal{S}}$ to size $|\psi_{\mathcal{S}}|$ using Algorithm 2
2: **for** $p = 1$ to P **pardo**
   {local computation}
3: **for** $t = \frac{|\psi^*_{\mathcal{S}}|}{P}(p-1)$ to $\frac{|\psi^*_{\mathcal{S}}|}{P}p - 1$ **do**
4: **if** $\psi_{\mathcal{S}}(t) \neq 0$ **then**
5: $\psi^\dagger_{\mathcal{S}}(t) = \psi^*_{\mathcal{S}}(t)/\psi_{ext}(t)$
6: **else**
7: $\psi^\dagger_{\mathcal{S}}(t) = 0$
8: **end if**
9: **end for**
10: **end for**

5.4 Optimized Computation Kernels for Evidence Propagation

The node level primitives discussed above can be utilized to implement exact inference. In this section, we optimize the primitives with respect to evidence collection and evidence distribution separately. The optimized primitives form the *computation kernels* for evidence collection and distribution.

Table marginalization for obtaining the separator between clique $\mathcal{C}$ and its parent $pa(\mathcal{C})$ can be simplified by avoiding conversions between indices and state strings.

To obtain the potential table $\psi_{\mathcal{S}_{pa(\mathcal{C})}}$, we also marginalize the potential table $\psi_{\mathcal{C}}$. However, notice that in Section 4.2 we require the variable vector of clique $\mathcal{C}$ to be ordered by $V_{\mathcal{C}} = (V_{\mathcal{C}\setminus S_{pa}(\mathcal{C})}, V_{S_{pa}(\mathcal{C})})$, where $V_{S_{pa}(\mathcal{C})}$ is the variable vector of $S_{pa}(\mathcal{C})$ and $V_{\mathcal{C}\setminus S_{pa}(\mathcal{C})}$ consists of random variables existing in $V_{\mathcal{C}}$ but not in $V_{S_{pa}(\mathcal{C})}$. Using the relationship between $V_{\mathcal{C}}$ and $V_{S_{pa}(\mathcal{C})}$, we simplify the table marginalization for obtaining $\psi_{\mathcal{S}_{pa(\mathcal{C})}}$ from $\psi_{\mathcal{C}}$. Since $V_{S_{pa(\mathcal{C})}}$ is the lower part of $V_{\mathcal{C}}$, the relationship between entries in $\psi_{\mathcal{S}_{pa(\mathcal{C})}}$ and entries in $\psi_{\mathcal{C}}$ is straightforward (see Figure 6). Segment i of $\psi_{\mathcal{C}}$ denotes $\psi_{\mathcal{C}}\left((i-1)|\psi_{pa(\mathcal{C})}| : (i|\psi_{pa(\mathcal{C})}|-1)\right)$, i.e. an array consists of entries from the $(i-1)|\psi_{pa(\mathcal{C})}|^{th}$ to the $(i|\psi_{pa(\mathcal{C})}|-1)^{th}$ in $\psi_{\mathcal{C}}$. Thus, this marginalization can be implemented by accumulating all segments, without checking the variable states for each entry of the potential table.

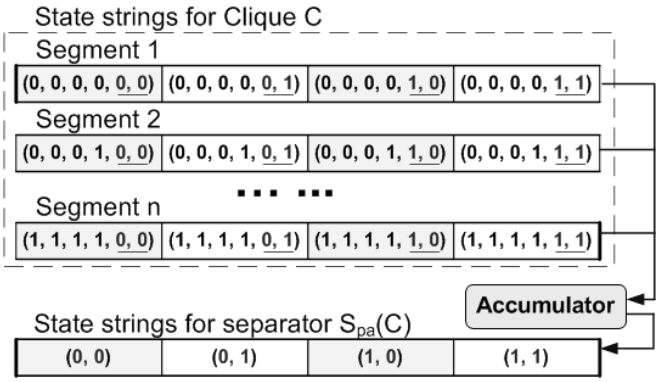


Fig. 6. Illustration of using the primitive of table marginalization to obtain separator $\psi_{\mathcal{S}_{pa(\mathcal{C})}}$ from clique potential table $\psi_{\mathcal{C}}$.

Since multiplication and division have the same priority in Eq. (1), we can either perform either first. However, if we perform multiplication first, we must perform table extension twice (for multiplication and division respectively). If we perform division first, we need to perform table extension only once ($\psi^*_{\mathcal{S}}$ and $\psi_{\mathcal{S}}$ in Eq. (1) have the same size, so table extension is not needed by division). In this paper, we always perform the division between two separators first. Notice that the variable vectors of the two separators are the same. Therefore, we eliminate table extension (Line 1) from Algorithm 4 and the table division becomes the element-wise division between two arrays.

Table multiplication can also be optimized by combining table extension and multiplication. According to Algorithm 3, the extended table has the same size as $\psi_{\mathcal{C}}$. Since the size of the clique potential tables can be very large, allocation of memory for the extended table is expensive. To avoid allocating memory for the extended table, we combine table extension and multiplication. Once we obtain the value of $\psi_{ext}(t)$, where ψ_{ext} is the extended table and t is the index, we multiply $\psi_{ext}(t)$ by $\psi_{\mathcal{C}}(t)$ instead of storing $\psi_{ext}(t)$ in memory. The product of the multiplication replaces the original data in $\psi_{\mathcal{C}}(t)$.

We analyze the complexity of Algorithms 5 and 6 based on the analysis of the node level primitives. Line 1 in Algorithm 5 launches P-way parallelism. Lines 2-11 plus Line 13 perform *table marginalization* for each child of $\mathcal{C}$. Lines 2-11 take $O(d_{\mathcal{C}}|\psi_{\mathcal{C}}|w_{\mathcal{C}}/P)$ time for local computation, where $d_{\mathcal{C}}$ is the number of children of clique $\mathcal{C}$. Line 11 is a global communication round. Line 14 takes $O(|\psi^*_{in_i}|)$ local computation time to performs element-wise *table division*. Notice that we do not parallelize the computation of table division. Considering $|\psi_{in_i}|$ is very small in our context, the communication overhead due to the parallelization of table division is larger than the local computation time. Without loss of generality, we assume $|\psi_{out}|/P \geq |\psi_{in_i}| \approx |\psi_{out}|$ in this analysis. Lines 15-17 perform *table multiplication* on $d_{\mathcal{C}}|\psi_{\mathcal{C}}|/P$ pairs of entries. Using $\tilde{t}_j$ computed in Lines 5-7, Lines 19-22 perform simplified marginalization, for which the local computation time is $O(|\psi_{\mathcal{C}}|/P)$. Line 23 is the second global communication round. Therefore, the local computation time for Algorithm 5 is $O(d_{\mathcal{C}}|\psi_{\mathcal{C}}|w_{\mathcal{C}}/P)$, $1 \leq P \leq |\psi_{out}|$. The number of global communication rounds is 2. Similarly, the local computation time for Algorithm 6 is $O(d_{\mathcal{C}}|\psi_{\mathcal{C}}|w_{\mathcal{C}}/P)$, and the number of global communication rounds is also 2.

6 Exact Inference with Node Level Primitives

6.1 Data Layout

For an arbitrary junction tree, we distribute the data as follows: Let P denote the number of processors. Instead of simply splitting every potential table $\psi_{\mathcal{C}}$ into P segments, we compare $|\psi_{\mathcal{C}}|/P$ with $|\psi_{\mathcal{S}}|$, where $\psi_{\mathcal{S}}$ is the largest potential table of the separators adjacent to $\mathcal{C}$. If $|\psi_{\mathcal{C}}|/P \geq |\psi_{\mathcal{S}}|$, each processor stores $|\psi_{\mathcal{C}}|/P$ entries of the potential table. Otherwise, we find another clique $\tilde{\mathcal{C}}$, which can be processed in parallel with $\mathcal{C}$ (e.g. $\mathcal{C}$ and $\tilde{\mathcal{C}}$ share the same parent clique). We distribute $\psi_{\mathcal{C}}$ to $\lfloor P|\psi_{\mathcal{C}}|/(|\psi_{\mathcal{C}}| + |\psi_{\tilde{\mathcal{C}}}|)\rfloor$ processors and $\psi_{\tilde{\mathcal{C}}}$ to the rest (see Figure 7). Similarly, if $(|\psi_{\mathcal{C}}| + |\psi_{\tilde{\mathcal{C}}}|)/P \leq |\psi_{\mathcal{S}}|$ and k parallel cliques exist ($k > 2$), each having a small potential table, then we allocate the processors to the k cliques. Let the k cliques be denoted by $\mathcal{C}_1, \mathcal{C}_2, \cdots, \mathcal{C}_k$, then $\psi_{\mathcal{C}_i}$, $\forall i \in [1, k]$, is partitioned into $P_i = P|\psi_{\mathcal{C}_i}|/\sum_{j=1}^{k}|\psi_{\mathcal{C}_j}|$ segments, each being stored by a processor. The allocation information of potential tables is maintained in each processor using a queue called *clique queue*: Let $\mathbb{C}_i$ denote the clique queue maintained in Processor i. $\mathbb{C}_i$ gives the order by which Processor i updates the cliques. Such an order must be consistent with the breadth first search (BFS) order of the junction tree. Each element in $\mathbb{C}_i$ consists of a clique ID denoted $\mathcal{C}$, the index of a segment of $\psi_{\mathcal{C}}$ stored in Processor i and the processor IDs where other segments of $\psi_{\mathcal{C}}$ are processed. In Figure 7 (b), each column corresponds to a clique queue. In addition to the potential tables, each processor maintains $O(d_{\mathcal{C}}|\psi_{\mathcal{S}}|)$ memory to store the adjacent separators during evidence propagation.

Algorithm 5 Computation kernel of evidence collection

Input: input separators $\psi^*_{in_i}$, clique potential table $\psi_{\mathcal{C}}$, mapping vector M_{in_i} $(i = 1, 2, \cdots, d_{\mathcal{C}})$

Output: updated clique potential table $\psi_{\mathcal{C}}$ and the output separator ψ_{out}

1: **for** $p = 1$ to P **pardo**

{local computation}

2: **for** $i = 1$ to $d_{\mathcal{C}}$ **do**

3: $\psi^{(p)}_{in_i}(1 : |\psi^*_{in_i}|) = \vec{0}$

4: **for** $j = \frac{|\psi_{\mathcal{C}}|}{P}(p-1)$ to $\frac{|\psi_{\mathcal{C}}|}{P}p - 1$ **do**

5: Convert j to $S = (s_1 s_2 \cdots s_w)$

6: Construct $\tilde{S} = (\tilde{s}_1 \tilde{s}_2 \cdots \tilde{s}_{|in|})$ from S using M_{in_i}

7: Convert $\tilde{S}$ to $\tilde{t}_j$

8: $\psi^{(p)}_{in_i}(\tilde{t}_j) = \psi^{(p)}_{in_i}(\tilde{t}_j) + \psi_{\mathcal{C}}(j)$

9: **end for**

10: **end for**

{global communication}

11: broadcast $\psi^{(p)}_{in_i}$ and receive $\psi^{(k)}_{in_i}$ from Processor k $(i = 1, \cdots, d_{\mathcal{C}};\ k = 1, \cdots, P \text{ and } k \neq p)$

{local computation}

12: **for** $i = 1$ to $d_{\mathcal{C}}$ **do**

13: $\psi_{in_i} = \sum_{j=1}^{P} \psi^{(j)}_{in_i}$

14: $\psi_{in_i}(1 : |\psi^*_{in_i}|) = \psi^*_{in_i}(1 : |\psi^*_{in_i}|)/\psi_{in_i}(1 : |\psi^*_{in_i}|)$ for non-zero elements of ψ_{in_i}

15: **for** $j = \frac{|\psi_{\mathcal{C}}|}{P}(p-1)$ to $\frac{|\psi^*_{\mathcal{C}}|}{P}p - 1$ **do**

16: $\psi_{\mathcal{C}}(j) = \psi_{\mathcal{C}}(j) * \psi_{in_i}(\tilde{t}_j)$

17: **end for**

18: **end for**

19: $\psi^{(p)}_{out}(1 : |\psi_{out}|) = \vec{0}$

20: **for** $j = \frac{|\psi_{\mathcal{C}}|}{P}(p-1)$ to $\frac{|\psi_{\mathcal{C}}|}{P}p - 1$ **do**

21: $\psi^{(p)}_{out}(j\%|\psi_{out}|) = \psi^{(p)}_{out}(j\%|\psi_{out}|) + \psi_{\mathcal{C}}(j)$

22: **end for**

{global communication}

23: broadcast $\psi^{(p)}_{out}$ and receive $\psi^{(k)}_{out}$ from Processor k $(k = 1, \cdots, P \text{ and } k \neq p)$

{local computation}

24: $\psi_{out} = \sum_{j=1}^{P} \psi^{(j)}_{out}$

25: **end for**

Algorithm 6 Computation kernel of evidence distribution

Input: input separator ψ^*_{in}, clique potential table $\psi_{\mathcal{C}}$, mapping vector M_{out_i} $(i = 1, 2, \cdots, d_{\mathcal{C}})$

Output: updated clique potential table $\psi_{\mathcal{C}}$ and the output separator ψ_{out_i}

1: **for** $p = 1$ to P **pardo**

{local computation}

2: $\psi^{(p)}_{in}(1 : |\psi^*_{in}|) = \vec{0}$

3: **for** $i = \frac{|\psi_{\mathcal{C}}|}{P}(p-1)$ to $\frac{|\psi_{\mathcal{C}}|}{P}p - 1$ **do**

4: $\psi^{(p)}_{in}(i\%|\psi^*_{in}|) = \psi^{(p)}_{in}(i\%|\psi^*_{in}|) + \psi_{\mathcal{C}}(i)$

5: **end for**

{global communication}

6: broadcast $\psi^{(p)}_{in}$ and receive $\psi^{(k)}_{out}$ from Processor k $(k = 1, \cdots, P \text{ and } k \neq p)$

{local computation}

7: $\psi_{in} = \sum_{j=1}^{P} \psi^{(j)}_{in}$

8: $\psi_{in}(1 : |\psi_{in}|) = \psi^*_{in}(1 : |\psi^*_{in}|)/\psi_{in}(1 : |\psi_{in}|)$ for non-zero elements of ψ_{in_i}

9: **for** $i = \frac{|\psi_{\mathcal{C}}|}{P}(p-1)$ to $\frac{|\psi_{\mathcal{C}}|}{P}p - 1$ **do**

10: $\psi_{\mathcal{C}}(i) = \psi_{\mathcal{C}}(i) * \psi_{in}(i\%|\psi_{in}|)$

11: **end for**

12: **for** $i = 1$ to $d_{\mathcal{C}}$ **do**

13: $\psi^{(p)}_{out_i}(1 : |\psi_{out_i}|) = \vec{0}$

14: **for** $t = \frac{|\psi_{\mathcal{C}}|}{P}(p-1)$ to $\frac{|\psi_{\mathcal{C}}|}{P}p - 1$ **do**

15: Convert $t + t_\delta$ to $S = (s_1 s_2 \cdots s_w)$

16: Construct $\tilde{S} = (\tilde{s}_1 \tilde{s}_2 \cdots \tilde{s}_{|out_i|})$ from S using M_{out_i}

17: Convert $\tilde{S}$ to $\tilde{t}$

18: $\psi^{(p)}_{out_i}(\tilde{t}) = \psi^{(p)}_{out_i}(\tilde{t}) + \psi_{\mathcal{C}}(t)$

19: **end for**

20: **end for**

{global communication}

21: broadcast $\psi^{(p)}_{out_i}$ and receive $\psi^{(k)}_{out_i}$ from Processor k $(i = 1, \cdots, d_{\mathcal{C}};\ k = 1, \cdots, P \text{ and } k \neq p)$

{local computation}

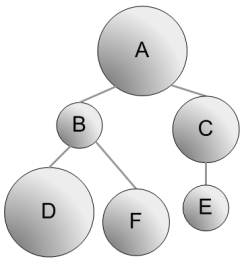
22: $\psi_{out_i} = \sum_{j=1}^{P} \psi^{(j)}_{out_i}$ $(i = 1, \cdots, d_{\mathcal{C}})$

23: **end for**

6.2 Complete Algorithm

The process of exact inference with node level primitives is given in Algorithm 7. The input to this algorithm includes the number of processors P, the clique queue $\mathbb{C}_i$ for each processor, the structure of a given junction tree $\mathbb{J}$, and the mapping vectors for all cliques with respect to their adjacent separators. The output is updated potential tables. Notice that the complete process of exact inference also includes local evidence absorption and query computation. The parallelization of these two steps is intuitive [15]. Therefore, Algorithm 7 focuses only on evidence propagation, including evidence collection and evidence distribution.

In Algorithm 7, Lines 2-7 perform evidence collection in the given junction tree. In evidence collection, each processor updates cliques in the reverse order given in $\mathbb{C}_i$. Therefore, a processor first processes leaf cliques. Line 3 defines some notations. Since input separators are *always* empty for leaf cliques during evidence collection, Line 4 is not executed for the leaf cliques. Thus, Line 4 can *not* suspend all processors. For non-leaf cliques, Line 4 ensures that all input separators are updated before we process $\mathcal{C}$. Line 5 applies Algorithm 5 to perform evidence collection. Line 6 declares that ψ_{out} has been updated in evidence propagation. Lines 8-13 in Algorithm 7 perform evidence distribution. Line 10 ensures that all the inputs are ready. Line 11 performs



(a)

P_1	P_2	P_3	P_4	P_5
A_1	A_2	A_3	A_4	A_5
B_1	B_2	C_1	C_2	C_3
D_1	D_2	D_3	D_4	D_5
F_1	F_2	F_3	E_1	E_2

(b)

Fig. 7. (a) Sample junction tree where the size of the nodes indicates the size of potential tables. (b) The layout of potential tables. Each row consists of segments of one or more potential tables, while each column corresponds to a processor.

Algorithm 7 Exact inference using computation kernels

Input: number of processors P, clique queues $\mathbb{C}_1, \cdots, \mathbb{C}_P$, junction tree $\mathbb{J}$, mapping vectors $M_{\mathcal{S}}$ for all cliques with respect to their adjacent separators

Output: updated clique potential tables for all cliques

1: **for** $p = 1$ to P **pardo**

{Evidence collection}

2: **for** $i = |\mathbb{C}_p|$ downto 1 **do**

3: Let $\mathcal{C} = \mathbb{C}_p(i)$; $\psi^*_{in_j} = \psi_{ch_j(\mathcal{C})}; M_{in_j} = M_{ch_j(\mathcal{C})}; \psi_{out} = \psi_{pa(\mathcal{C})}, \forall j = 1, \cdots, d_{\mathcal{C}}$

4: wait until nonempty $\psi^*_{in_j}$, $\forall j = 1, \cdots, d_{\mathcal{C}}$, are updated

5: EvidenceCollect$(\mathcal{C}, \psi^*_{in_j}, \psi_{out}, M_{in_j})$

6: set ψ_{out} as a updated separator with respect to evidence collection

7: **end for**

{Evidence distribution}

8: **for** $i = 1$ to $|\mathbb{C}_p|$ **do**

9: Let $\mathcal{C} = \mathbb{C}_p(i)$; $\psi^*_{in} = \psi_{pa(\mathcal{C})}$; $M_{out_j} = M_{ch_j(\mathcal{C})}$; $\psi_{out_j} = \psi_{ch_j(\mathcal{C})}, \forall j = 1, \cdots, d_{\mathcal{C}}$

10: wait until nonempty ψ^*_{in} is updated

11: EvidenceDistribute$(\psi_{\mathcal{C}}, \psi^*_{in}, \psi_{out_j}, M_{out_j})$

12: set ψ_{out_j}, $\forall j = 1, \cdots, d_{\mathcal{C}}$, as updated separators with respect to evidence distribution

13: **end for**

14: **end for**

evidence distribution in $\mathcal{C}$ using Algorithm 6. The output separator is updated in Line 14.

The computation in Algorithm 7 occurs in Lines 5 and 11. The local computation time and the number of global communication rounds have been discussed in Section 5.4. However, Lines 4 and 10 introduce two more communication rounds. Based on the analysis in Section 5.4, the local computation time for Algorithm 7 is $O(Nd_{\mathcal{C}}|\psi_{\mathcal{C}}|w_{\mathcal{C}}/P)$. Since we perform six communications per clique, the number of global communication rounds is $O(N)$.

7 Experiments

7.1 Computing Facilities

We implemented the node level primitives using the Message Passing Interface (MPI) on the linux cluster in High-Performance Computing and Communications (HPCC) at the University of Southern California (USC) [18].

The HPCC cluster at USC employs a diverse mix of computing and data resources, including 256 Dual Quadcore AMD Opteron quad-core processor nodes running at 2.3 GHz with 16 GB memory. This machine uses a 10 GB low-latency Myrinet backbone to connect the compute nodes. The cluster achieved 44.19 teraflops in fall 2008, ranked at the 7th among supercomputers in an academic setting in the United States. The cluster runs USCLinux, a customized distribution of the RedHat Enterprise Advanced Server 3.0 (RHE3) Linux distribution. The Portable Batch System (PBS) is used to allocate nodes for a job.

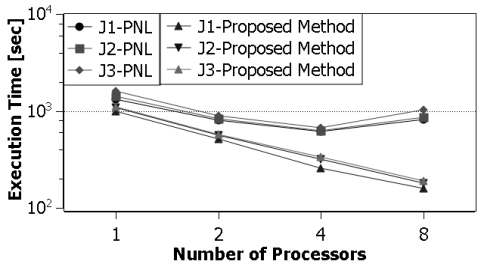
7.2 Experiments using PNL

Intel Open Source Probabilistic Networks Library (PNL) is a full function, free, graphical model library [19]. A parallel version of PNL is now available, which is developed by Intel Russia Research Center and University of Nizhni Novgorod. Intel China Research Center and Intel Architecture Laboratory were also involved in the development process.

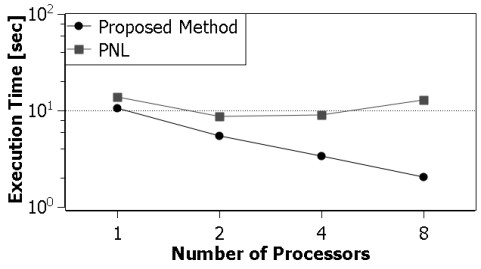
We conducted experiments to explore the scalability of the exact inference algorithm using the PNL library [10]. The input graphs were denoted $J1, J2$ and $J3$, where each clique in $J1$ had at most one child; all cliques except the leaves in $J2$ had equal numbers of children; the cliques in $J3$ had random numbers of children. All the graphs had 128 cliques, each random variable having 2 states. We show the execution time in Figure 8 (a). We can see from Figure 8 (a) that, although the execution time reduced when we used 2 and 4 processors, we failed to achieve reduced execution time when we used 8 processors. We measured the time taken by performing table operations and non-table operations in exact inference implemented using the PNL library. We observed that the time taken by performing table operations is the dominant part of the overall execution time. With clique width $w_{\mathcal{C}} = 20$ and the number of states $r = 2$, table operations take more than 99% of the overall execution time. This result demonstrates that parallelizing the node level primitives can lead to high performance.

We conducted an experiment using a Bayesian network from a real application. The Bayesian network is called the Quick Medical Reference decision theoretic version (QMR-DT), which is used in a microcomputer-based decision support tool for diagnosis in internal medicine [20]. There were 1000 nodes in this network. These nodes formed two layers, one representing diseases and the other symptoms. Each disease has one or more edges pointing to the corresponding symptoms.

All random variables (nodes) were binary. We converted the Bayesian network to a junction tree offline and then performed exact inference in the resulting junction tree. The resulting junction tree consists of 114 cliques while the average clique width is 10. In Figure 8 (b), we illustrate the experimental results using the PNL and our proposed method respectively. Our method illustrated almost linear speedup while the PNL based method did not show scalability using 8 processors.



(a) Scalability of exact inference using PNL



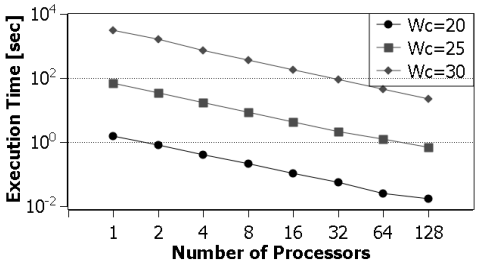
(b) Exact inference on QMR-DT network

Fig. 8. Scalability of PNL based parallel exact inference.

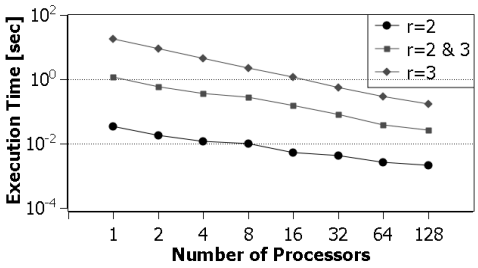
7.3 Experimental Results

We evaluated the performance of individual node level primitives, the computation kernels and the complete exact inference by using various junction trees generated by off-the-shelf software. We utilized Bayes Net Toolbox [21] to generate the input data including potential tables and junction tree structures. The implementation of the proposed methods was developed using the C language with MPI standard routines for message passing. Our implementation based on Algorithm 7 employed a fully distributed paradigm, where the code was executed by all the processors in parallel.

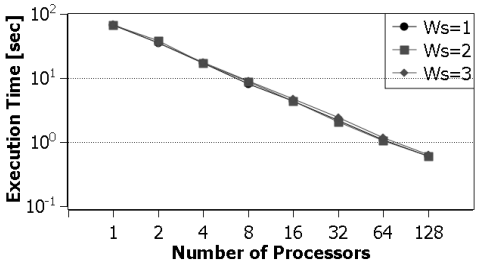
To evaluate the performance of individual node level primitives, we generated a set of single cliques as well as the adjacent separators to each clique. A potential table was generated for every clique and separator. The set contained potential tables with various parameters: The clique width w_C was selected to be $20, 25$, or 30. The number of states r was selected to be 2 for some random variables, and 3 for the others. As double precision floating point numbers were used, the size of the potential tables varied from 8 MB ($w_C = 20,\ r = 2$) to 8 GB ($w_C = 30,\ r = 2$). The widths of the separators w_S were chosen to be $1, 2$ or 4. The clique degree d (i.e. maximum number of children) was selected to be $1, 2$ or 4. The mapping vectors $\mathbb{M}$ for the junction trees were constructed offline. The construction of the mapping variables is explained in Section 5.1. We performed node level primitives on these potential tables using various numbers of processors (P was chosen from 1, 2, 4, 8, 32, 64 and 128). Each processor processed $1/P$ of the total entries in a given potential table.



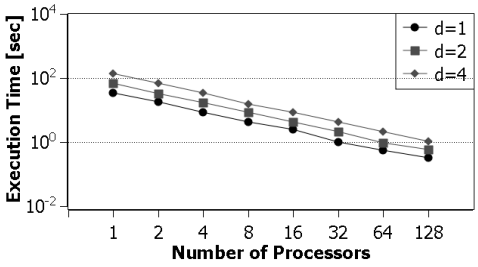
(a) Scalability with respect to clique width



(b) Scalability with respect to number of states



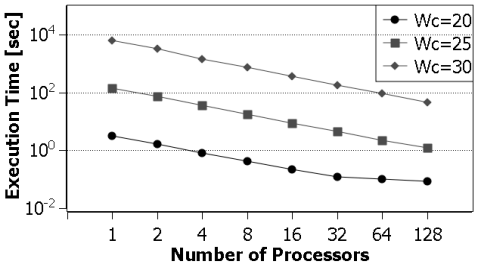
(c) Scalability with respect to separator width



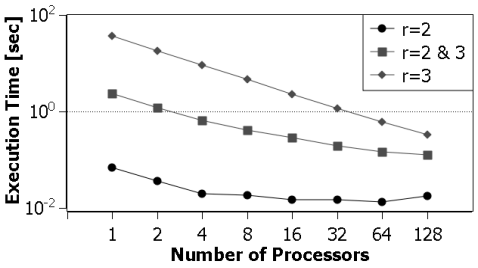
(d) Scalability with respect to clique degree

Fig. 9. Scalability of table marginalization with respect to various parameters.

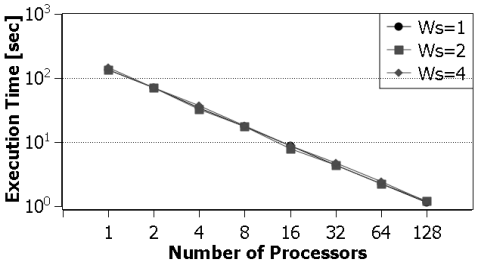
The experimental results of table marginalizaiton are shown in Figures 9. The other primitives showed similar results [22]. The default parameter values for the experiments were: $w_C = 25, w_S = 2, r = 2$ and clique



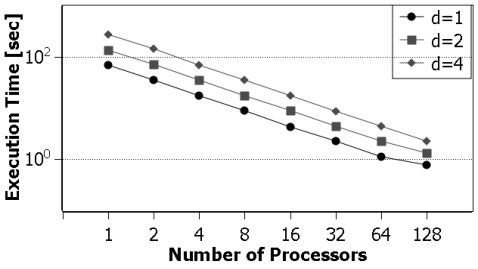
(a) Scalability with respect to clique width



(b) Scalability with respect to number of states

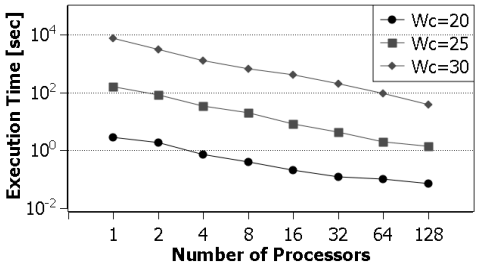


(c) Scalability with respect to separator width

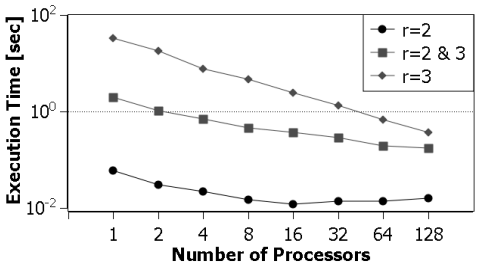


(d) Scalability with respect to clique degree

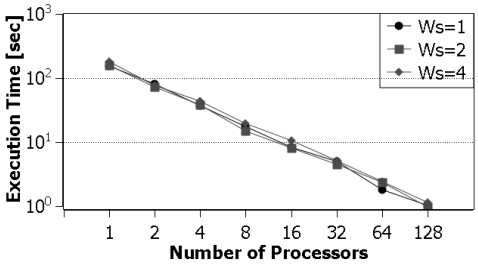
Fig. 10. Scalability of the computation kernel for evidence collection with respect to various parameters.



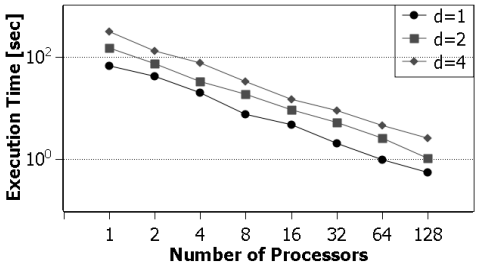
(a) Scalability with respect to clique width



(b) Scalability with respect to number of states



(c) Scalability with respect to separator width



(d) Scalability with respect to clique degree

Fig. 11. Scalability of the computation kernel for evidence distribution with respect to various parameters.

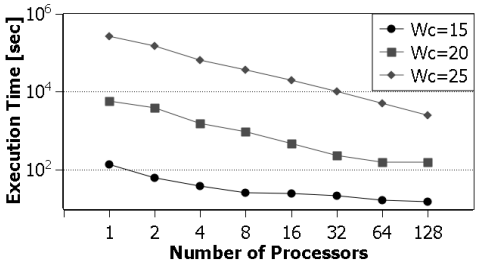
degree $d = 2$. However, we let $w_{\mathcal{C}} = 15$ for Panels (b), since the size of the potential table with $w_{\mathcal{C}} = 25$ and $r = 3$ is about 6 TB, which exceeds our quota. For Panel (b), the label $r = 2\ \&\ 3$ means that the number of states for half of the random variables is 2 while for the other half is 3. In each panel of the above figures, we varied one parameter at a time to show the influence of each specific parameter. From the experimental results, we can see that the primitives exhibited almost linear scalability using 128 processors, regardless of the input parameters.

We also evaluated the performance of the computation kernels for evidence collection and distribution. Notice that the computation kernels given in Algorithms 5 and 6 update a given clique potential table as well as its related separators. We also used the same input data used to evaluate node level primitives. We conducted experiments for the two computation kernels separately in Figures 10 and 11.

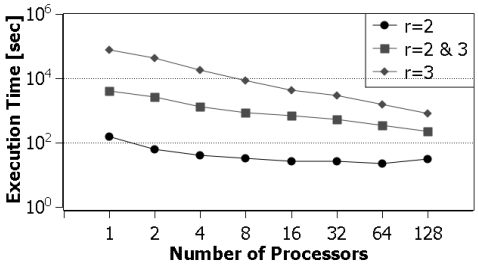
Finally, we evaluated the performance of the complete parallel exact inference (Algorithm 7 in Section 6.2). Unlike the experiments conducted above, the input to Algorithm 7 included an entire junction consisting of potential tables for each clique, the related separators,

and a clique queue for each processor. The data layout for the input junction tree is addressed in Section 6.1. The clique queues generated according to the layout were sent to each processor separately. That is, each processor kept partial potential potential tables, related separators and a clique in its local memory. In our experiments, we generated several junction trees with the number of cliques $N = 1024$. For cliques and separators in the junction tree, as with for evaluating primitives and kernels, we also selected various values for the clique width w_C, the number of random variables r, the separator width w_S, and the clique degree d_C. Notice that, by varying these parameters, we obtained various types of junction trees. For example, $d_C = 1$ implies a chain of cliques, while an arbitrary d_C gives a junction tree with random numbers of branches. The results are shown in Figure 12 (a)-(c). In each panel of Figure 12, we varied the value for one parameter and conducted experiments with respect to various numbers of processors.

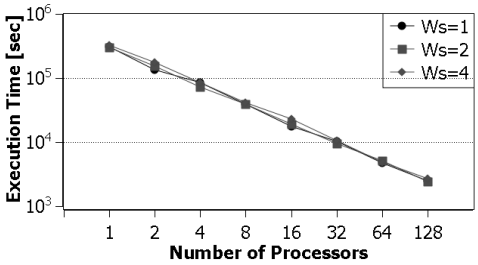
According to the results shown in Figure 12, the proposed method exhibits scalability using 128 processors for various input parameters. For the junction tree where $N = 1024, w_C = 25, r = 2$ and $d = 2$, the parallel exact inference achieved a speedup of 98.4 using 128 processors. Thus, the execution time was reduced to about two minutes from approximately three hours using a single processor. Compared to the experiment in Section 7.2, our method performed exact inference in junction trees with large potential tables. In addition, the proposed exact inference algorithm exhibited scalability over a much larger range. We observed almost linear speedup in our experiments. According to the algorithm analysis based on the CGM model in Section 6.2, the computation complexity is inversely proportional to the number of processors, which matches the experimental results.



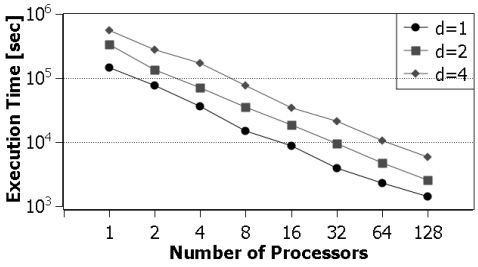
(a) Scalability with respect to clique width



(b) Scalability with respect to number of states



(c) Scalability with respect to separator width



(d) Scalability with respect to clique degree

Fig. 12. Scalability of exact inference with respect to various parameters.

8 Conclusions

In this article, we explored data parallelism for exact inference in junction trees. We proposed scalable algorithms for node level primitives and assembled the primitives into two computation kernels. A parallel exact inference algorithm was developed using the computation kernels. We analyzed the scalability of the node level primitives, computation kernels and the exact inference algorithm using the CGM model and implemented the exact inference algorithm on state-of-the-art clusters. The experimental results exhibited almost linear scalability over a much larger range compared to existing methods such as the parallel version of the Probabilistic Networks Library. As part of our future work, we are planning to study efficient data layout and scheduling policies for exact inference on multicore clusters where each compute node consists of multiple processors. We also intend to explore parallelism in exact inference at multiple levels, and map the parallelism to the architecture of the heterogeneous clusters. We plan to partition an arbitrary junction tree into subtrees and schedule the subtrees to various compute nodes in a cluster. Each subtree is further decomposed and processed by the processors or cores in the compute node. We expect higher performance for exact inference by integrating the parallelism at various levels.

References

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

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

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

[4] 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*, 2001, pp. 243–252.

[5] L. Yin, C.-H. Huang, and S. Rajasekaran, "Parallel data mining of bayesian networks from gene expression data," in *Poster Book of the 8th International Conference on Research in Computational Molecular Biology*, 2004, pp. 122–123.

[6] 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*, vol. 50, pp. 157–224, 1988.

[7] C. Huang and A. Darwiche, "Inference in belief networks: A procedural guide," *International Journal of Approximate Reasoning*, vol. 15, no. 3, pp. 225–263, 1996.

[8] A. V. Kozlov and J. P. Singh, "A parallel lauritzen-spiegelhalter algorithm for probabilistic inference," in *Supercomputing*, 1994, pp. 320–329.

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

[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*, 2006, pp. 143–150.

[11] R. Anderson and L. Snyder, "A comparison of shared and nonshared memory models of parallel computation," *Proceedings of the IEEE*, vol. 79, no. 4, pp. 480–487, 1991.

[12] L. G. Valiant, "General purpose parallel architectures," pp. 943–973, 1990.

[13] A. Chan, F. Dehne, P. Bose, and M. Latzel, "Coarse grained parallel algorithms for graph matching," *Parallel Computing*, vol. 34, no. 1, pp. 47–62, 2008.

[14] A. Chan, F. Dehne, and R. Taylor, "Implementing and testing CGM graph algorihtms on PC clusters and shared memory machines," *International Journal of High Performance Computing Applications*, vol. 19, no. 1, pp. 81–97, 2005.

[15] 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*, 2007, pp. 221–228.

[16] J. M. Bjøndalen, O. J. Anshus, B. Vinter, and T. Larsen, "Configurable collective communication in LAM-MPI," *Proceedings of Communicating Process Architectures*, pp. 133–143, 2002.

[17] A. Grama, G. Karypis, V. Kumar, and A. Gupta, *Introduction to Parallel Computing (2nd Edition)*. Addison Wesley, January 2003.

[18] "USC center for High-Performance Computing and Communications (HPCC)," http://www.usc.edu/hpcc/.

[19] "Intel's Probablistic Networks Library (PNL)," http://sourceforge.net/projects/openpnl/.

[20] B. Middleton, M. Shwe, D. Heckerman, H. Lehmann, and G. Cooper, "Probabilistic diagnosis using a reformulation of the INTERNIST-1/QMR knowledge base," *Medicine*, vol. 30, pp. 241–255, 1991.

[21] "The Bayesian Network Toolbox for Matlab (BNT)," http://bnt.sourceforge.net/.

[22] Y. Xia and V. K. Prasanna, "Node level computation kernels for parallel exact inference," University of Southern California, Tech. Rep., 2009.

[23] J. Jingxi, B. Veeravalli, and D. Ghose, "Adaptive load distribution strategies for divisible load processing on resource unaware multilevel tree networks," *IEEE Transactions on Computers*, 2007.

[24] I. Patel and J. R. Gilbert, "An empirical study of the performance and productivity of two parallel programming models," in *the 22th International Parallel and Distributed Processing Symposium*, 2008.

[25] K. Lu, R. Subrata, and A. Y. Zomaya, "On the performance-driven load distribution for heterogeneous computational grids," *Journal of Computer and System Sciences*, vol. 73, no. 8, pp. 1191–1206, 2007.

[26] G. Han and Y. Yang, "Scheduling and performance analysis of multicast interconnects," *Journal of Supercomputer*, vol. 40, no. 2, pp. 109–125, 2007.

[27] D. Bader, "Petascale computing for large-scale graph problems," in *the 21th International Parallel and Distributed Processing Symposium*, 2007.

[28] M. Tan, H. J. Siegel, J. K. Antonio, and Y. A. Li, "Minimizing the application execution time through scheduling of subtasks and communication traffic in a heterogeneous computing system," *IEEE Transactions on Parallel and Distributed Systems*, vol. 8, no. 8, pp. 857–871, 1997.

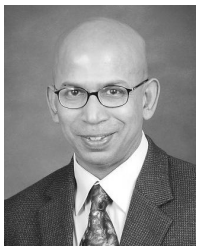
[29] T. Dean, "Scalable inference in hierarchical generative models," in *Proceedings of the 9th International Symposium on Artificial Intelligence and Mathematics*, 2006, pp. 1–9.

[30] I. Troxel and A. George, "Scheduling tradeoffs for heterogeneous computing on an advanced space processing platform," *12th International Conference on Parallel and Distributed Systems*, 2006.

[31] E. Grobelny, D. Bueno, I. Troxel, A. George, and J. Vetter, "Fase: A framework for scalable performance prediction of hpc systems and applications," *Simulation*, vol. 83, no. 10, pp. 721–745, 2007.

[32] S. Salleh, S. Olariu, A. Y. Zomaya, K. L. Yieng, and N. A. Aziz, "Single-row mapping and transformation of connected graphs," *Journal of Supercomputing*, vol. 39, no. 1, pp. 73–89, 2007.

Yinglong Xia received the BS degree from the University of Electronic Science and Technology of China in 2003, the MEng degree from the Tsinghua University in 2006. He is currently a PhD candidate in Computer Science Department at the University of Southern California. His research interests include parallel computing and application optimization techniques for high performance computing systems. He is a student member of the ACM.



Viktor K. Prasanna (V. K. Prasanna Kumar) (ceng.usc.edu/∼prasanna) is Charles Lee Powell Chair in Engineering in the Ming Hsieh Department of Electrical Engineering and Professor of Computer Science at the University of Southern California. He is the executive director of the USC-Infosys Center for Advanced Software Technologies (CAST). He is also an associate member of the Center for Applied Mathematical Sciences (CAMS) at USC, and a member of the USC-Chevron Center of Excellence for Research and Academic Training on Interactive Smart Oilfield Technologies (Cisoft). His research interests include parallel and distributed systems including networked sensor systems, embedded systems, configurable architectures and high performance computing. He is the Steering Co-chair of the International Parallel and Distributed Processing Symposium and is the Steering Chair of the International Conference on High Performance Computing (HiPC). He has served on the editorial boards of the Journal of Parallel and Distributed Computing, Proceedings of the IEEE, IEEE Transactions on VLSI Systems, and IEEE Transactions on Parallel and Distributed Systems. He served as the Editor-in-Chief of the IEEE Transactions on Computers during 2003-06. He was the founding Chair of the IEEE Computer Society Technical Committee on Parallel Processing. He is a Fellow of the IEEE and the ACM. He is a recipient of the 2005 Okawa Foundation Grant and 2009 Outstanding Engineering Alumnus Award from the Pennsylvania State University.