Distributed Evidence Propagation in Junction Trees on Clusters

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

Abstract—Evidence propagation is a major step in exact inference, a key problem in exploring probabilistic graphical models. In this paper, we propose a novel approach for parallelizing evidence propagation in junction trees on clusters. Our proposed method explores structural parallelism in a given junction tree. We decompose a junction tree into a set of subtrees, each consisting of one or multiple leaf-root paths in the junction tree. In evidence propagation, we first perform evidence collection in these subtrees concurrently. Then, the partially updated subtrees exchange data for junction tree merging, so that all the cliques in the junction tree can be fully updated for evidence collection. Finally, evidence distribution is performed in all the subtrees to complete evidence propagation. Since merging subtrees requires communication across processors, we propose a technique called bitmap partitioning to explore the tradeoff between bandwidth utilization efficiency and the overhead due to the startup latency of message passing. We implemented the proposed method using Message Passing Interface (MPI) on a state-of-the-art Myrinet cluster consisting of 128 processors. Compared with a baseline method, our technique results in improved scalability.

Index Terms—Junction tree, exact inference, decomposition, message passing.

1 Introduction

Many real-world systems can be modeled using a set of random variables, each representing a parameter of the system, such as temperature or humidity. These variables can be dependent on each other and the joint probability distribution of the variables describes the state of the system. However, such a joint distribution increases dramatically in size as the number of variables used for modeling the system increases. *Bayesian networks* greatly reduce the size of the joint probability distribution by utilizing conditional independence relationships among the variables. Bayesian networks have found applications in a number of domains, including medical diagnosis, consumer help desks, data mining, genetics, etc., [3], [10], [20].

Evidence propagation in a Bayesian network is the computation of the updated conditional distribution of the variables in the Bayesian network, given a set of *evidence* variables as the knowledge to the network. The most popular exact inference algorithm for multiply connected networks converts a Bayesian network into a *junction tree*, and then performs evidence propagation in the junction tree [6], [9]. The complexity of exact inference increases dramatically with the various parameters of junction trees, such as the number of cliques, the clique width, and the number of states of the random variables. In many cases exact inference must be performed in real time.

Designing efficient parallel algorithms for evidence propagation must take into account the characteristics of the platform. Most supercomputers are clusters of processors, where the communication time consists of the startup time known as *latency* and the data transfer time [2]. On a cluster with dual quadcore AMD 2,335 processors connected by Infiniband (IB) cables, we observed that the startup latency for a single message passing is 3.31 microseconds, equivalent to completing 211×10^3 floating point operations on a compute node. On the other hand, the aggregate memory of a cluster increases linearly with the number of processors. Thus, for algorithm design on clusters, we must minimize the overhead due to startup latency in communication even if it requires duplicating some data.

Our contributions include

1. a heuristic to decompose a junction tree into subtrees with approximately equal weights so as to perform evidence propagation in the subtrees in parallel,
2. a technique to merge partially updated subtrees,
3. a metric for checking if the granularity of a junction tree decomposition can lead to improved performance,
4. a method to explore the tradeoff between overhead due to startup latency and bandwidth utilization efficiency in communication, and
5. implementation details and experimental evaluation on state-of-the-art clusters.

The rest of the paper is organized as follows: in Section 2, we review the background. Section 3 discusses related works. Our proposed method and the experimental results are shown in Sections 4 and 5, respectively. Section 6 concludes the paper.

2 Background

2.1 Bayesian Network and Junction Tree

A *Bayesian network* is a probabilistic graphical model that exploits conditional independence to compactly represent a

- *Y. Xia is with the IBM T.J. Watson Research Center, 1101 Kitchawan Road, Route 134, Yorktown Heights, NY 10598. E-mail: yxia@us.ibm.com.*
- *V.K. Prasanna is with the Ming Hsieh Department of Electrical Engineering, University of Southern California, 3740 McClintock Avenue, EEB 200C, Los Angeles, CA 90089-2562. E-mail: prasanna@usc.edu.*

Manuscript received 1 Oct. 2010; revised 15 Oct. 2011; accepted 30 Oct. 2011; published online 29 Nov. 2011.
Recommended for acceptance by X.-H. Sun.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number TPDS-2010-10-0576.
Digital Object Identifier no. 10.1109/TPDS.2011.278.

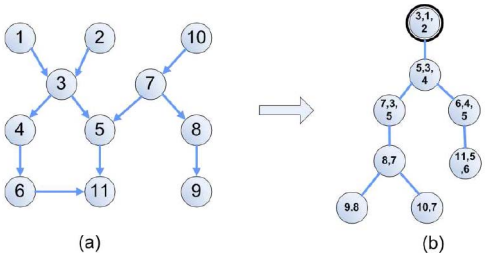


Fig. 1. (a) A sample Bayesian network. (b) Its junction tree.

joint distribution. Fig. 1a shows a sample Bayesian network, where each node represents a random variable. Each edge indicates the probabilistic dependence relationships between two random variables. Notice that these edges *cannot* form directed cycles. Thus, the structure of a Bayesian network is a *Directed Acyclic Graph* (DAG). The *evidence* in a Bayesian network is the variables that have been instantiated [6], [9].

Traditional exact inference using Bayes' theorem fails for networks with undirected cycles [6]. Most inference methods for networks with undirected cycles convert a network into a cycle-free hypergraph called a *junction tree*. In Fig. 1, we illustrate a junction tree converted from a Bayesian network. Note that all undirected cycles are eliminated in the given Bayesian network. Each vertex in Fig. 1b contains multiple random variables from the Bayesian network. For the sake of exploring evidence propagation in a junction tree, we use the following notations. 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. Given two adjacent cliques $\mathcal{C}_i$ and $\mathcal{C}_j$, the *separator* between them is defined as $\mathcal{C}_i \cap \mathcal{C}_j$. $\hat{\mathbb{P}}$ is a set 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 $\mathcal{C}_i$. Note that each separator also has a potential table describing the joint distribution of the common random variables between adjacent cliques. Since the separator potential tables can be computed from the clique potential tables, we do not include them into $\hat{\mathbb{P}}$. For a clique with w variables, each having r states, the number of entries in $\psi_{\mathcal{C}_i}$ is r^w.

2.2 Evidence Propagation

In evidence propagation, we update a junction tree in two stages, i.e., *evidence collection* and *evidence distribution*. In evidence collection, the evidence is propagated from the leaves to the root in topological order, where each clique $\mathcal{C}$ updates $\psi_{\mathcal{C}}$ using the separators between $\mathcal{C}$ and its children. Then, $\mathcal{C}$ updates the separator between $\mathcal{C}$ and its parent using the updated potential table $\psi^*_{\mathcal{C}}$. Evidence distribution is as the same as collection, except the evidence propagation direction is from the root to the leaves.

In a junction tree, evidence is propagated from a clique to its neighbors as follows: assuming that the evidence is $\mathrm{E} = \{A_i = a\}$ and $A_i \in \mathcal{C}_{\mathcal{Y}}$, E is *absorbed* at $\mathcal{C}_{\mathcal{Y}}$ by instantiating the variable A_i and renormalizing the remaining variables of the clique. The evidence is then propagated from $\mathcal{C}_{\mathcal{Y}}$ to all adjacent cliques $\mathcal{C}_{\mathcal{X}}$. Let $\psi^*_{\mathcal{Y}}$ denote the potential table of $\mathcal{C}_{\mathcal{Y}}$ after E is absorbed, and $\psi_{\mathcal{X}}$ the potential table of $\mathcal{C}_{\mathcal{X}}$. Mathematically, evidence propagation is represented as [6]

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

where $\mathcal{S}$ is the separator between cliques $\mathcal{X}$ and $\mathcal{Y}$; $\psi_{\mathcal{S}}(\psi^*_{\mathcal{S}})$ denotes the original (updated) potential table of $\mathcal{S}$; $\psi^*_{\mathcal{X}}$ is the updated potential table of $\mathcal{C}_{\mathcal{X}}$. For the sake of understanding (1), we intuitively describe the process of evidence propagation from $\mathcal{Y}$ to $\mathcal{X}$ as follows: since the joint distribution of random variables in $\mathcal{Y}$ is updated after evidence absorption, the distribution of each single variable in $\mathcal{Y}$ is changed because of the dependency among the random variables, even though some variables in $\mathcal{Y}$ are not evidence variables. Thus, $\psi^*_{\mathcal{S}}$ computed from $\psi^*_{\mathcal{Y}}$ can be different from $\psi_{\mathcal{S}}$. The difference is due to the evidence. Equation (1) uses $\psi^*_{\mathcal{S}}$, the updated distribution of the separator, to updated $\psi_{\mathcal{X}}$, since all the variables in $\mathcal{S}$ are also in $\mathcal{X}$. Because of the dependency among the random variables in $\mathcal{X}$, the distribution of all variables in $\mathcal{X}$ is updated. Therefore, evidence is propagated from $\mathcal{Y}$ to $\mathcal{X}$.

The computations among clique tables in (1) are called *node level primitives*. There include potential table multiplication, division, extension, and marginalization [18]. We discuss a complete sequential exact inference algorithm and its complexity analysis in Section 1 of the Supplemental Document which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TPDS.2011.277.

3 Related Work

There are several works on parallel exact inference, such as Pennock [9], Kozlov and Singh [5], and Shachter et al. [11]. However, some of these methods, such as [5], are developed for a class of Bayesian networks, such as polytrees. Our proposed method can be used for junction trees converted from arbitrary Bayesian networks. The execution time of some other existing methods, such as [9], is proportional to the number of cliques with evidence variables. When multiple cliques in a junction tree contain evidence variables, the algorithm in [9] reroots the junction tree with respect to each evidence clique sequentially and performs evidence propagation in each rerooted junction tree. In contrast, our proposed method does not depend on the number of evidence variables. A high-throughput method for inference in a Bayesian network is proposed in [7]. The proposed method is designed for reconfigurable hardware, where a Bayesian network is expanded to a polytree with virtual nodes. Due to the different programming models for cluster and reconfigurable hardware, the performance of the algorithm discussed in [7] can be adversely affected on clusters. In [18] and [15], the node level primitives are parallelized using message passing to explore data parallelism. Thus, the techniques proposed in [18] and [15] are suitable for junction trees with large potential tables. In this paper, we investigate task level parallelism so as to handle junction trees with limited data parallelism. In [14], an algorithm is developed for scheduling the cliques without precedence relationship to the cores of a multicore processor. Note that the cores in a multicore processor share the global memory. However, for clusters, the memory is not shared among the

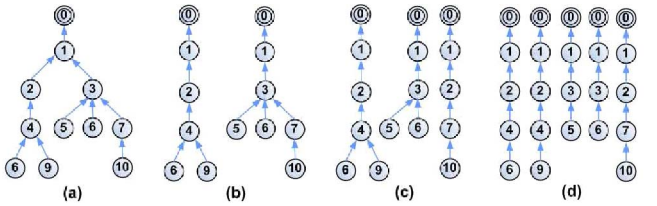


Fig. 2. A junction tree in (a) can be decomposed into subtrees of various granularities shown in (b), (c), and (d). The arrows in the graph indicate evidence collection direction.

processors. Using the technique in [14] on clusters leads to significant data transferring among the processors, which decreases the overall performance. Therefore, novel techniques must be studied for a cluster of processors without shared memory.

A junction tree decomposition method is provided in [16] to decompose a junction tree into chains. Unlike [16], an improved algorithm is proposed in [17] for decomposing a junction tree into subtrees so as to reduce clique duplication. A distributed algorithm proposed in [17] is used in this paper. Compared with [17], we propose an algorithm for implementing bitmap partitioning. We also provide analysis of the proposed algorithm. The bitmap partitioning method investigates the tradeoff between the number of communication steps and bandwidth utilization efficiency. We conduct additional experiments to thoroughly evaluate the proposed algorithms.

4 Distributed Evidence Propagation Based on Decomposition

4.1 Overview

We decompose a junction tree into a series of *subtrees*, each consisting of one or more root-leaf paths in the input junction tree (see Fig. 2). Note that the decomposition is different from conventional tree *partition* that divides a given tree without duplicating any node [4]. A junction tree can be decomposed at various granularity levels to generate different number of subtrees. Given the number of processors P in a cluster, we decompose a given junction tree into P subtrees of approximately equal task weights, so that each processor can host a distinct subtree. In this paper, we assume that the number of leaf cliques of the input junction tree is greater than the number of available processors.

Since cliques may be duplicated in several subtrees, we must *merge* partially updated cliques in the subtrees to obtain fully updated cliques. For example, in Fig. 2c, the root exists in all the three subtrees. After evidence collection, the root in each subtree is partially updated, since the root in the first subtree cannot collect evidence from Cliques 3, 5, 6, 7, and 10. Thus, we must merge the duplicated cliques. Merging cliques requires communication among the processors. Our proposed method guarantees all the duplicated cliques can be merged in parallel. After clique merging, we perform evidence distribution in each subtree in parallel and obtain the final output of evidence propagation. Note that we do not need to merge cliques after evidence distribution.

4.2 Junction Tree Decomposition

We show our proposed heuristic for junction tree decomposition in Algorithm 1. This algorithm decomposes the input junction tree into P, $P \geq 1$, subtrees of approximately equal workload. By workload, we mean the overall execution time for processing the cliques in a subtree.

Algorithm 1. Junction Tree Decomposition into Subtrees
Input: Junction tree J, clique weight $V_{\mathcal{C}}$, number of processors P, tolerance factor Δ
Output: Subtrees $\mathrm{J}_i, i = 0, 1, \ldots, P-1$

1: let $R_{\mathcal{C}} = 0, S_{\mathcal{C}} = 0, \forall \mathcal{C} \in \mathrm{J}, \tilde{r}$ be the root of J
{Find weight of path from $\tilde{r}$ to parent of $S_{\mathcal{C}}$, $\forall \mathcal{C} \in \mathrm{J}$}
2: $\mathrm{Q} = \{\tilde{r}\}$
3: **while** $\mathrm{Q} \neq \emptyset$ **do**
4: **for all** $\mathcal{C} \in \mathrm{Q}$ **do**
5: $R_{\mathcal{C}'} = R_{\mathcal{C}} + V_{\mathcal{C}}, \forall \mathcal{C}' \in ch(\mathcal{C})$
6: **end for**
7: $\mathrm{Q} = \{\mathcal{C}' : \mathcal{C}' \in ch(\mathcal{C}) \text{ and } \mathcal{C} \in \mathrm{Q}\}$
8: **end while**
{Find weight of subtree rooted at $S_{\mathcal{C}}$, $\forall \mathcal{C} \in \mathrm{J}$}
9: $\mathrm{Q} = \{\text{leaf cliques in J}\}, S_{\mathcal{C}} = V_{\mathcal{C}}, \forall \mathcal{C} \in \mathrm{Q}$
10: **while** $\mathrm{Q} \neq \emptyset$ **do**
11: **for all** $\mathcal{C} \in \mathrm{Q}$ **do**
12: $S_{\mathcal{C}} = V_{\mathcal{C}} + \sum_{\mathcal{C}' \in ch(\mathcal{C})} S_{\mathcal{C}'}$
13: **end for**
14: $\mathrm{Q} = \{\mathcal{C} : S_{\mathcal{C}} = 0 \text{ and } S_{\mathcal{C}'} > 0, \forall \mathcal{C}' \in ch(\mathcal{C})\}$
15: **end while**
{Decomposition}
16: $\mathrm{G} = \{\tilde{r}\}$
17: **repeat**
18: $\mathcal{C}_m = \arg\max_{\mathcal{C} \in \mathrm{G}}(R_{\mathcal{C}} + S_{\mathcal{C}})$ where $ch(\mathcal{C}) \neq \emptyset$
19: $\mathrm{G} = \mathrm{G} \cup \{\mathcal{C}' : \mathcal{C}' \in ch(\mathcal{C}_m) \neq \emptyset\} \backslash \{\mathcal{C}_m\}$
20: let $\mathrm{G}' = \mathrm{G}$, $\mathrm{K}_0 = \mathrm{K}_1 = \cdots = \mathrm{K}_{P-1} = \emptyset$
21: **while** $\mathrm{G}' \neq \emptyset$ **do**
22: let $j = \arg\min_{i \in [0, P-1]}(\sum_{\mathcal{C}' \in \mathrm{K}_i}(S_{\mathcal{C}'} + R_{\mathcal{C}'}))$
23: $\mathrm{K}_j = \mathrm{K}_j \cup \{\arg\max_{\mathcal{C} \in \mathrm{G}'}(S_{\mathcal{C}} + R_{\mathcal{C}})\}$, $\mathrm{G}' = \mathrm{G}' \backslash \{\mathcal{C}\}$
24: **end while**
25: $\mathrm{K}_{max} = \max_{i \in [0, P-1]}(\sum_{\mathcal{C}' \in \mathrm{K}_i}(S_{\mathcal{C}'} + R_{\mathcal{C}'}))$,
$\mathrm{K}_{min} = \min_{i \in [0, P-1]}(\sum_{\mathcal{C}' \in \mathrm{K}_i}(S_{\mathcal{C}'} + R_{\mathcal{C}'}))$
26: **until** $\mathrm{G} = \{\text{leaf cliques in J}\}$ or
$(\mathrm{K}_{max} - \mathrm{K}_{min})/(\mathrm{K}_{max} + \mathrm{K}_{min}) < \Delta$
27: **for all** $i = 0, 1, \ldots, P-1$ **do**
28: $\mathrm{J}_i = \mathrm{J} \cap (\mathrm{Path}(\tilde{r}, \mathcal{C}) \cup \mathrm{Subtree}(\mathcal{C}))$, $\forall \mathcal{C} \in \mathrm{K}_i$
29: **end for**

In Algorithm 1, we use the following notations. Weight $V_{\mathcal{C}}$ is the estimated execution time for updating a clique $\mathcal{C}$ in junction tree J. Given clique width $w_{\mathcal{C}}$, the number of children $d_{\mathcal{C}}$ and the number of states of random variables r, we have $V_{\mathcal{C}} = d_{\mathcal{C}} w_{\mathcal{C}}^2 r^{w_{\mathcal{C}}+1}$ [18]. The tolerance factor $\Delta \in [0, 1]$ is a threshold that controls the load balance among the subtrees. Small Δ results in better load balance, but can lead to longer execution time for Algorithm 1. $ch(\mathcal{C})$ represents the children of $\mathcal{C}$. We use Path $(\tilde{r}, \mathcal{C})$ to denote the path from root $\tilde{r}$ to the parent of a clique $\mathcal{C}$, and Subtree $(\mathcal{C})$ the subtree rooted at $\mathcal{C}$. We use $R_{\mathcal{C}}$ to represent the overall weight of cliques in Path $(\tilde{r}, \mathcal{C})$ and $S_{\mathcal{C}}$ the overall weight of cliques in Subtree $(\mathcal{C})$. Thus, $(S_{\mathcal{C}} + R_{\mathcal{C}})$ gives the estimated execution time for updating a decomposed junction tree rooted at $\tilde{r}$.

We assume there are P processors and the decomposed junction tree hosted by the ith processor is J_i.

Algorithm 1 consists of three parts: the first part (Lines 2-8) populates $R_{\mathcal{C}}$ for each clique; Lines 9-15 populates $S_{\mathcal{C}}$. The third part uses a heuristic to decompose the input junction tree into P subtrees. Note that $(S_{\mathcal{C}} + R_{\mathcal{C}})$ is the estimated execution time for updating subtree $(\mathrm{Path}(\tilde{r}, \mathcal{C}) \cup \mathrm{Subtree}\ (\mathcal{C}))$. In each iteration of the loop (Lines 17-26), we partition the heaviest subtree into a set of lighter subtrees, each corresponding to a child of $\mathcal{C}$. Note that $(S_{\mathcal{C}} + R_{\mathcal{C}})$ is monotonic nonincreasing from the root to leaf cliques. Thus, the variance of the weights of the subtrees is reduced. This helps load balance. In Lines 20-24, we assign the cliques to P groups using a heuristic, so that the overall weight of the decomposed junction trees are approximately equal. If the load balance is acceptable (Lines 25-26), the decomposed junction trees are generated using the cliques in Path $(\tilde{r}, \mathcal{C})$ and Subtree $(\mathcal{C})$. Let N denote the number of cliques in the junction tree. The complexity is $O(1)$ for Lines 1-2, and $O(N)$ for Lines 3-8 and 9-15. The time taken by Lines 16-26 depends on Δ and the input junction tree. In the worst case, each nonleaf clique is inserted into set G once. Thus, the complexity is also $O(N)$. Lines 27-28 take $O(P)$ time. Therefore, the serial execution time of Algorithm 1 is $O(N)$.

4.3 Junction Tree Merging

A straightforward approach for junction tree merging is as follows: for each clique $\mathcal{C}$, access partially updated potential tables $\psi_{\mathcal{C}}$ from all the processors where $\mathcal{C}$ exists, and then combine them through a series of computations. Note that the potential table of cliques is generally much larger than the potential table of separators. Thus, although the above approach is doable, it requires intensive data transfer among the processors. This can adversely affect the performance.

An alternative approach for merging cliques is to utilize the separator potential tables, instead of the large clique potential tables. Consider the root clique in Fig. 2c as an example. Note that evidence flows into the root through the separators between the root (i.e., Clique 0) and Clique 1 only. Thus, if we want to fully update $\psi_{\mathcal{C}}$, we simply transfer the potential tables of all the separators adjacent to $\mathcal{C}$. Note that the transferred data are much smaller than $\psi_{\mathcal{C}}$. Let $\psi^*_{\mathcal{C}}$ denote the fully updated potential table for $\mathcal{C}$ and $\psi^i_{\mathcal{C}}$ the partially updated potential table in the ith subtree. $\psi^i_{S_{\mathcal{C}}(j)}$ represents the jth separator of $\mathcal{C}$ in the ith subtree. Let $\tilde{\mathrm{S}}^k_{\mathcal{C}}$ denote all the separator potential tables required for obtaining $\psi^*_{\mathcal{C}}$ on the kth processor

$$\tilde{\mathrm{S}}^k_{\mathcal{C}} = \{\psi^i_{S_{\mathcal{C}}(j)}\}, \quad 0 \le i < P, \quad i \ne k, 0 \le j < d_{\mathcal{C}}(i), \tag{2}$$

where $d_{\mathcal{C}}(i)$ is the number of children of $\mathcal{C}$ in subtree J_k. Let $\hat{\psi}^i_{S_{\mathcal{C}}(j)}$ represent the potential table of separator $S_{\mathcal{C}}(j)$ obtained by marginalizing $\psi_{\mathcal{C}}$ before $\psi^i_{S_{\mathcal{C}}(j)}$ is used to update $\psi_{\mathcal{C}}$. We have the following formula $\mathcal{M}(\cdot)$ for merging any duplicated clique $\mathcal{C}$ in any subtree J_k

$$\psi^*_{\mathcal{C}} = \mathcal{M}\big(\psi^k_{\mathcal{C}}, \tilde{\mathrm{S}}^k_{\mathcal{C}}\big) = \psi^k_{\mathcal{C}} \prod_{i,j} \frac{\psi^i_{S_{\mathcal{C}}(j)}}{\hat{\psi}^i_{S_{\mathcal{C}}(j)}}, \tag{3}$$

where $0 \le i < P, i \ne k, 0 \le j < d_{\mathcal{C}}(i)$.

During evidence collection, each processor checks if any local clique has been duplicated onto other processors. For each duplicated clique, we send the local separators corresponding to the clique to other processors and also receive separators from other processors as well. Then, we use (3) to obtain the fully updated cliques. Note that we do not need to merge junction trees after evidence distribution.

4.4 Granularity of Junction Tree Decomposition

Given a junction tree with L leaf cliques, Algorithm 1 decomposes the junction tree into P subtrees, where $1 \le P \le L$. As shown in Fig. 2, greater P leads to finer granularity in junction tree decomposition, i.e., smaller but more subtrees, which results in less execution time for evidence propagation in subtrees, but higher execution time for junction tree merging. To avoid the situation that junction tree merging dominates overall execution time, we must control the granularity for junction tree decomposition. We propose a metric to determine if a granularity (a given P) can make the overall execution time dominated by junction tree merging or not.

As discussed in Section 4.2, an estimate of execution time for updating a clique $\mathcal{C}$ is $V_{\mathcal{C}} = d_{\mathcal{C}} w^2_{\mathcal{C}} r^{w_{\mathcal{C}}+1}$. Let J_i, $0 \le i < P$, denote the ith subtree decomposed from a given junction tree J. Assuming each subtree is hosted by a separate processor, the execution time for propagating evidence in subtrees is given by

$$t_{local} = \max_{0 \le i < P} \sum_{j=0}^{|\mathrm{J}_i|-1} 2 d^i_j {w^i_j}^2 r^{w^i_j+1}, \tag{4}$$

where the constant 2 is introduced because each clique must be updated twice in both evidence collection and distribution. Let D^i_j denote the number of duplicates of the jth clique in J_i. When merging partially updated cliques, a clique receives messages from all the remote $(D^i_j - 1)$ duplicates of the clique. Each received message is used as a separator to update the clique. Thus, the time taken by merging partially updated cliques is

$$t_{merge} = \max_{0 \le i < P} \sum_{j=0}^{|\mathrm{J}_i|-1} \big(D^i_j - 1\big) {w^i_j}^2 r^{w^i_j+1}. \tag{5}$$

Note that D^i_j in (5) is also related to the granularity of junction tree decomposition. Greater P leads to greater D^i_j. Based on (4) and (5), we present the following *metric*:

$$t_{local} \ge t_{merge}. \tag{6}$$

Given the number of processors P, if the inequality in (6) is held, then we claim that the overall execution time is not dominated by junction tree merging; otherwise, we must reduce P until the inequality in (6) is satisfied.

4.5 Tradeoff between Startup Latency and Bandwidth Utilization Efficiency

During junction tree merging, the processors exchange locally updated separators by message passing. Assume the total amount of data transferred for junction tree merging is s and there are k communication steps executed in sequence. Given the aggregate network *bandwidth* B and the startup

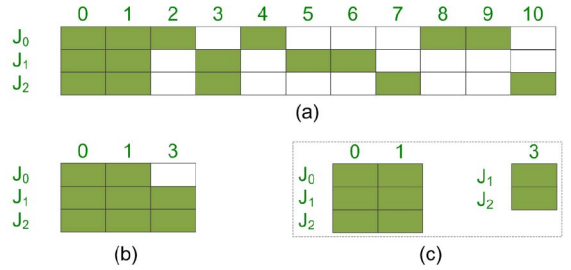


Fig. 3. (a) A sample bitmap. (b) A reduced bitmap. (c) Partitioned bitmaps.

latency l for sending a message, an optimal approach for exchanging separators must minimize t_{comm}, where

$$t_{comm} = k \cdot l + s/B. \tag{7}$$

Since l and B are constants for a given system, we must minimize k and s. However, our analyze (see Section 2 available in the online Supplemental Document) shows that decreasing k results in increased s, vice versa. Thus, k and s cannot be minimized simultaneously.

We develop a technique to explore the tradeoff between s and k. We represent the clique allocation and duplication using a *bitmap*, i.e., a Boolean matrix where each row corresponds to a subtree and each column corresponds to a clique. We label the cliques using a Breadth-First Search (BFS) order starting from the root of the junction tree. Fig. 3a illustrates a sample bitmap for the subtrees shown in Fig. 2c. Since we are only concerned with the duplicated cliques during junction tree merging, we remove the columns corresponding to the unduplicated cliques from the bitmap. The result is called *reduced bitmap* (see Fig. 3b). Note that 0s in a bitmap correspond to unused data transferred between processors. If all entries in a column (row) of a bitmap are 0s, this column (row) can be removed, since such a column (row) means the corresponding clique (subtree) does not exist. Therefore, we partition a sparse bitmap into a set of smaller bitmaps and eliminate the columns and rows with all entries equal to 0. Each resulting bitmap is called a *sub-bitmap* (Fig. 3c). For each sub-bitmap, we utilize an all-to-all communication step to exchange data among the corresponding cliques.

A constraint in partitioning a bitmap is that the duplicates of a clique must be assigned to the same sub-bitmap. Otherwise, the cliques across multiple sub-bitmaps cannot be fully updated. To the best of our knowledge, such a constraint is *not* considered by general graph partitioning methods [19]. Thus, we design a heuristic using the characteristics of the junction tree decomposition discussed in Section 4.2.

Our heuristic utilizes the following metric to determine if a sub-bitmap is dense enough. By *density*, we mean the percentage of the entries equal to 1 in a bitmap. Given the size of a separator potential table $|\psi_S|$ and the number of children of a clique d, the metric is

$$\frac{z \cdot d \cdot |\psi_S|}{\eta \cdot B} < l, \tag{8}$$

where z is the number of entries equal to 0, l is the startup latency for communication, B is bandwidth, and η is a platform-specific constant called *tradeoff factor*. We have $\eta = 0.1$ in our experiments.

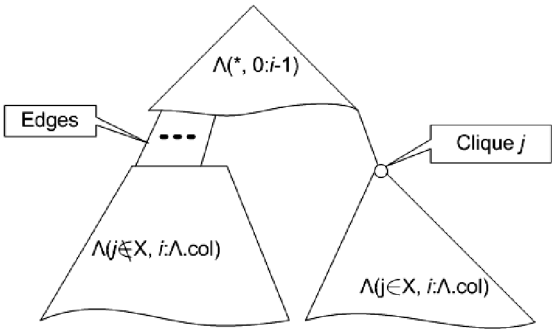


Fig. 4. Illustration of bitmap partitioning. The junction tree is partitioned into three parts, each represented by a polygon. The same notations are used in Algorithm 2.

We show the proposed heuristic in Algorithm 2. In Line 4, we eliminate cliques and subtrees not appearing in Λ. Line 5 uses (8) to identify the dense sub-bitmap of the maximum size. We identify a dense sub-bitmap by checking $\Lambda(*, 0:j)$ in Line 5. If the entire Λ is dense, Line 7 stores it into $\mathbb{M}$; otherwise, Λ is partitioned into three parts, a dense sub-bitmap $\Lambda(*, 0:i-1)$, the sub-bitmap $\Lambda(j \in X, i:\Lambda.col)$ for the subtree rooted at the clique corresponding to the ith column of Λ, and the sub-bitmap $\Lambda(j \notin X, i:\Lambda.col)$ corresponding to the rest of Λ (Line 9). The dense sub-bitmap is added to $\mathbb{M}$ and the other two are added to $\tilde{\mathbb{M}}$ for further partitioning. Algorithm 2 exits when all bitmaps in $\tilde{\mathbb{M}}$ are processed.

Algorithm 2. Bitmap partitioning
Input: Reduced bitmap $\tilde{\Lambda}$ for junction tree J, bandwidth B, startup latency l, tradeoff factor η
Output: A set of partitioned bitmaps $\mathbb{M}$
1: let $\mathbb{M} = \emptyset; \tilde{\mathbb{M}} = \{\tilde{\Lambda}\}$
2: **repeat**
3: **for all** $\Lambda \in \tilde{\mathbb{M}}$ **do**
4: remove columns and rows with all 0s from Λ
5: $i = \max\{j : 0 \le j < \Lambda.col, \frac{z \cdot d \cdot |\psi_S|}{\eta \cdot B \cdot l} < 1\}$,
where z is the number of 0s in $\Lambda(*, 0:j)$
6: **if** $i = \Lambda.col - 1$ **then**
7: $\mathbb{M} = \mathbb{M} \cup \{\Lambda\}$
8: **else**
9: $\mathbb{M} = \mathbb{M} \cup \{\Lambda(*, 0:i-1)\}$,
$\tilde{\mathbb{M}} = \tilde{\mathbb{M}} \cup \{\Lambda(j \in X, i:\Lambda.col)\} \cup$
$\{\Lambda(j \notin X, i:\Lambda.col)\}$,
where $X = \{j : \Lambda(j, i) = 1\}$
10: **end if**
11: $\tilde{\mathbb{M}} = \tilde{\mathbb{M}} \backslash \{\Lambda\}$
12: **end for**
13: **until** $\tilde{\mathbb{M}} = \emptyset$

In Fig. 4, we illustrate the relationship between the proposed bitmap partitioning method and the partitioning of a junction tree. As shown in Line 9 in Algorithm 2, if a bitmap Λ is not dense enough according to (8), we partitioning the junction tree into three parts: $\Lambda(*, 0:i-1)$, $\Lambda(j \in X, i:\Lambda.col)$, and $\Lambda(j \notin X, i:\Lambda.col)$, where $X = \{j : \Lambda(j, i) = 1\}$. The three parts are shown in Fig. 4. The first part $\Lambda(*, 0:i-1)$ is close to the root and all its cliques are duplicated during

junction tree decomposition. Thus, $\Lambda(*, 0 : i-1)$ is dense compared with the other two parts. The subtree rooted at clique j is another part as shown in the bottom right. The rest of the cliques form the third part $\Lambda(j \notin X, i : \Lambda.col)$. The latter two parts can be sparse since they are close to the leaf cliques, which are not duplicated by Algorithm 1. Therefore, the two parts can be further partitioned.

4.6 Distributed Evidence Propagation Algorithm

Based on the techniques discussed in Sections 4.2, 4.3, and 4.5, we present the complete algorithm for distributed evidence propagation in junction trees in Algorithm 3. The input includes an arbitrary junction tree J, clique weights (i.e., estimated execution time) $V_{\mathcal{C}}$, some system parameters such as bandwidth B and startup latency l and a user threshold $\Delta \in [0, 1]$ for controlling the junction tree decomposition. We let $\Delta = 0.1$ in our experiments. The output is the updated junction tree J, where the evidence originally on any clique has been propagated to the rest of the cliques.

Algorithm 3. Distributed Evidence Propagation
Input: Junction tree J, clique weight $V_{\mathcal{C}}, \forall \mathcal{C} \in$ J, number of processors P, tolerance factor Δ, bandwidth B, startup latency l, tradeoff factor η
Output: Updated junction tree J

```
   {Initialization}
 1: (J_0, ..., J_{P-1})=Decompose(J, V_C, P, Δ) // Algorithm 1
 2: Λ̃ = reduced bitmap for {J_0, J_1, ..., J_{P-1}}
 3: 𝕄 = BitmapPartition(Λ̃, B, l, η, J) // Algorithm 2
 4: assign J_i and 𝕄 to processor p_i, ∀0 ≤ i < P
    {Parallel Evidence Propagation}
 5: for processor p_i, i = 0, 1, ..., P − 1 pardo
 6:    EvidenceCollect(J_i)
 7:    for all Λ ∈ 𝕄 do
 8:       S^i = ∅
 9:       for all clique C in both J_i and Λ do
10:          S^i = S^i ∪ {ψ^i_{S_C(j)}}, 0 ≤ j < d_C(i)
11:       end for
12:       All-to-all send S^i to p_j and receive Ŝ^j,
          ∀0 ≤ j < P − 1, j ≠ i
13:       Obtain fully updated cliques using Ŝ^j //(3)
14:    end for
15:    EvidenceDistribute(J_i)
16: end for
```

Algorithm 3 consists of initialization and parallel evidence propagation. Initialization can be done offline. Thus, we only provide sequential algorithm for these steps. In Line 1, we invoke Algorithm 1 in Section 4.2 to decompose J into P subtrees. The corresponding bitmap is created in Line 2 and partitioned in Line 3. Line 4 assigns each subtree to a separate processor and broadcasts $\mathbb{M}$. In Lines 5-16, all processors run in parallel to process the subtrees. In Lines 6 and 15, each processor performs evidence collection and distribution in its local subtree using the sequential algorithm proposed in [6]. In Lines 7-14, for each bitmap in Λ, we perform all-to-all communication among the processors corresponding to the bitmap. In Line 14, each processor collects the relevant separators in S^i. The received data are used to obtain fully updated cliques in Line 13.

5 Experiments

5.1 Facilities

We implemented the proposed method using the Message Passing Interface (MPI) on the High-Performance Computing and Communications (HPCC) Linux cluster at the University of Southern California (USC) [13]. The HPCC cluster employs a diverse mix of computing and data resources, including 256 Dual Quadcore AMD Opteron 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 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.

5.2 Data Sets

To evaluate the performance of our proposed method, we generated various junction trees by mimicking the junction trees converted from a real Bayesian network called the Quick Medical Reference decision theoretic version (QMR-DT) [8]. The number of cliques N in these junction trees was in the range of 32,768 to 65,536. The clique degree d, i.e., the number of children of a clique, was in the range of 1 to 8, except for the leaf cliques. The clique width W_C, i.e., the number of random variables per clique was varied from 8 to 20. The separator width W_S was from 2 to 5. The random variables r in the junction trees were either binary or ternary. When binary variables were used, the number of entries per clique potential table was between 256 and 10,48,576; while the number of entries per separator potential table was between 4 and 32. We used single precision floating point data for the potential tables. All the potential tables were aligned in the memory to improve the performance. We conducted experiments with 1 to 128 processors. We checked the number of processors using (6) and ensured that 128 processors would not make the overall execution time dominated by junction tree merging. For the data layout and the potential table organization, we followed the approach used in [18].

5.3 Baseline Methods

`Serial` is a sequential implementation of evidence propagation discussed in Section 2.2. For a given junction tree, we start from the root to obtain the breadth-first search order of the cliques. During evidence collection, we utilize (1) to update each clique according to the *reversed* BFS order. Then, during evidence distribution, the processor updates all of the cliques again according to the BFS order.

`Clique allocate` is an asynchronous message driven parallel implementation of evidence propagation. We mapped the cliques at each level of the input junction tree onto distinct processors so as to maximize the number of cliques that can be updated in parallel. We started with leaf cliques during evidence collection. Once a clique obtained messages from all its children, the clique was updated by the host processor. Similarly, we started with the root and updated all of the cliques again during evidence distribution.

`Data parallel` is a parallel baseline method exploring data parallelism in exact inference. We traversed the input

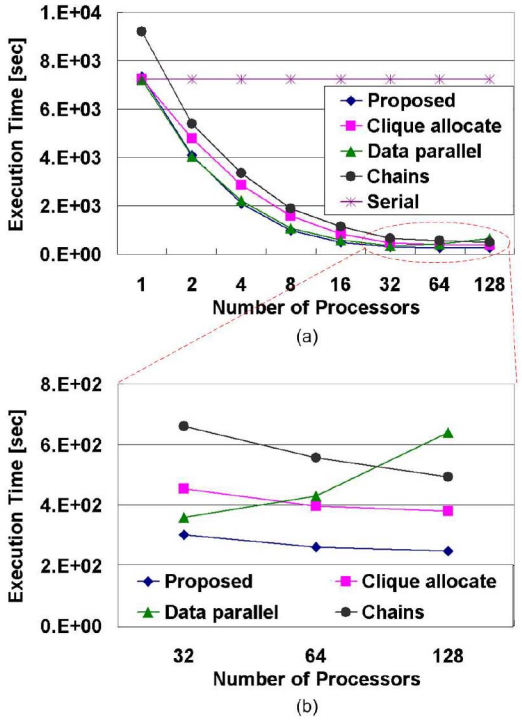


Fig. 5. Scalability of the proposed technique compared with baseline methods.

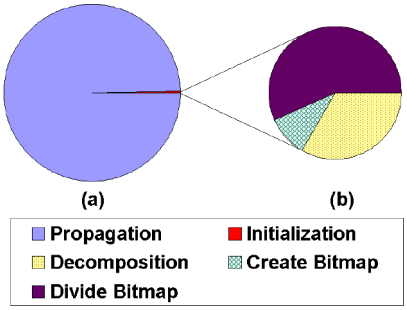


Fig. 6. Percentage of execution time for (a) initialization and evidence propagation; (b) each stage in initialization.

junction tree according to breadth-first search order and updated the cliques one by one. For each clique, the potential table was partitioned and each processor updated a part of the potential table using parallel node level primitives proposed in [18]. The processors were synchronized after a clique was updated.

`Chains` is a parallel baseline method which decomposes a junction tree into a set of chains, each corresponding a path from a leaf clique to the root [16]. We manually assigned the chains to the processors, so that workload was evenly distributed. The processors first updated the assigned chains in parallel and then exchanged duplicated cliques in a global communication step. The junction tree was fully updated after the duplicated cliques were merged.

5.4 Experimental Results

In Fig. 5, we illustrate the experimental results of the proposed distributed evidence propagation method and two baseline methods discussed in Section 5.3 with respect to a junction tree with the following parameters: $N = 65{,}536$, $W_C = 15, r = 2, d = 4, W_S = 4$. We had consistent results with respect to other junction trees. We ran each experiment 10 times and calculated the average execution time and its standard deviation. As shown in Fig. 5, the overhead due to parallelization was negligible for the proposed method, since the execution time with respect to a single processor was almost the same as the baseline called `serial`. In contrast, the baseline `chains` shows significant overhead. `Clique allocate` showed higher overhead than our proposed method, since it involves more coordination among the processors. Since the potential tables in the input junction tree are much smaller than those in [18], the `data parallel` baseline method showed limited scalability.

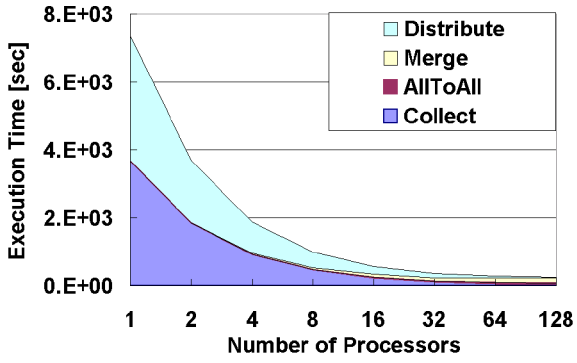
From Fig. 5a, the *improvement* of the proposed distributed evidence propagation method compared with all the baseline methods was more than 28 percent when 128 processors were used. In Fig. 5b, we illustrate the execution time when more than 32 processors were used, so that we can take a close look at the improvement of the proposed method.

In Fig. 6, we show the percentage of execution time for various steps in our proposed method. On a single processor, we performed our proposed method using the same junction tree used in the previous experiment. According to Fig. 6a, initialization (Lines 1-8 in Algorithm 3) takes a very small amount of time compared with the overall execution time.

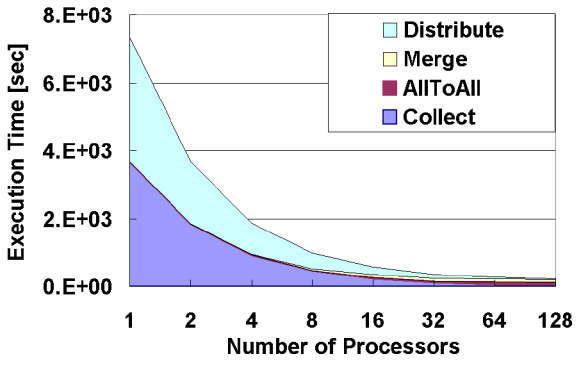
We illustrate the scalability for each stage, i.e., evidence collection (`collect`), all-to-all communication (`AllToAll`), junction tree merging (`Merge`), and evidence distribution (`Distribute`), of the proposed parallel evidence propagation algorithm (Lines 9-20 in Algorithm 3) in Fig. 7. Fig. 7a shows the results where we used a single bitmap. Thus, there was only one all-to-all communication. In Fig. 7b, we divided the bitmap into multiple bitmaps. Thus, we had multiple all-to-all communication steps, but the bandwidth utilization efficiency was improved. We controlled the number of bitmaps by altering the tradeoff factor η in (8).

Figs. 7a and 7b show that the impact of η was very limited when a small number of processors were used. As the number of processors increases, we observed the impact of η on the execution time for each stage. In Fig. 8, we can observe that using a single all-to-all communication step results in less overhead due to startup latency. In Fig. 8, we show the normalized execution time for the results in Figs. 7a and 7b when 128 processors were used. When a single bitmap was used, the all-to-all communication took a small percentage of execution time, since we sent only one message. So, the overhead due to startup latency was minimized in this case. However, since unused data were transferred in all-to-all communication, junction tree merging took additional time to identify useful separators from all received data, as shown in Fig. 8. When multiple bitmaps were used, we improved the efficiency of junction tree merging, but the communication time was longer because of the increased overhead due to startup latency.

Finally, we evaluated our proposed method using various junction trees. In Fig. 9, the label *Original JTree* represents the junction tree having the following parameters: $N = 65{,}536$, $W_C = 15, r = 2, d = 4, W_S = 4$. We varied a parameter each time as shown in the labels to illustrate the impact of various parameters on the execution time of our method. When we



(a) Evidence propagation with a single bitmap



(b) Evidence propagation with a set of sub-bitmaps

Fig. 7. Execution time of various steps.

reduced the number of cliques in the junction tree from 65,536 to 32,768, the execution time of the new junction tree exhibited similar scalability as the original tree. The execution time was almost half of that for the original junction tree. When we used the junction tree with $W_C = 10$, the execution time was much smaller than that for the original junction tree. For the original junction tree, we must update 2^{15} entries for each clique potential table. However, for the junction tree with $W_C = 10$, we only update 2^{10} entries, 1/32 of that for the original junction tree. Thus, it is reasonable that the execution time was much faster. When W_S was reduced from 4 to 2, the impact was negligible. When we increased d from 2 to 4, we observed in Fig. 9 that the scalability was adversely affected. The execution time increased slightly as the number of processors increases. The reason is that large d results in more separators being transferred.

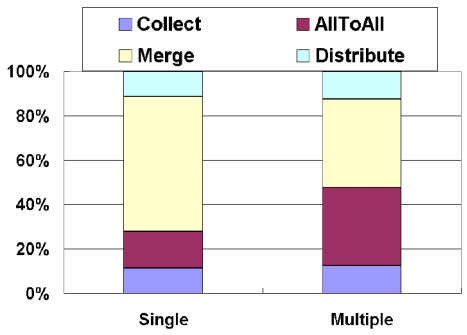


Fig. 8. Normalized execution time of various steps.

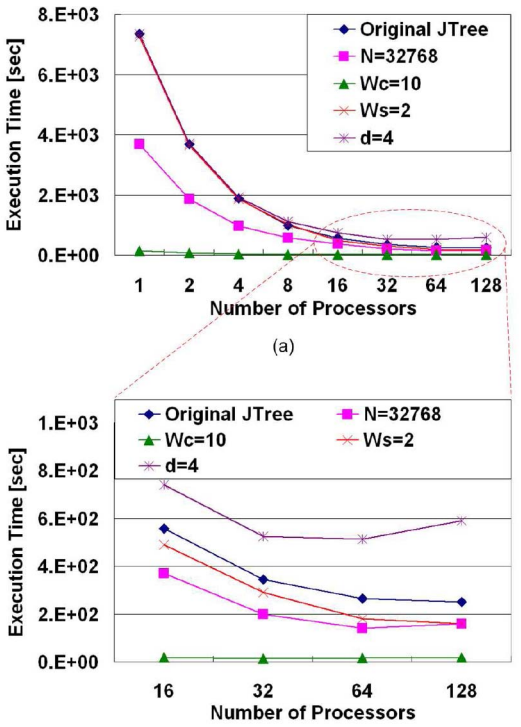


Fig. 9. (a) Impact of various parameters of junction tree on overall execution time with respect various number of processors. (b) Close-up view for the selected segment in the dashed circle.

The experimental results in this section show that the proposed method scales well with respect to almost all of the junction trees. In addition, compared with several baseline algorithms for evidence propagation, our method shows improved performance in terms of the total execution time. Like many other MPI-based implementations, clusters with higher bandwidth and lower latency interconnections, e.g., InfiniBand or Myrenet, lead to better performance for our proposed method, due to less communication overhead.

6 Conclusion

In this paper, we developed a novel distributed exact inference algorithm based on junction tree decomposition. We proposed an efficient heuristic to decompose a junction tree into a set of subtrees with approximately equal weight. Since the cost of communication is high for distributed exact inference, we discussed the tradeoff between startup latency and bandwidth utilization efficiency. Additional details available in the online Supplemental Document associated with this paper. We used a bitmap to represent the subtrees. We proposed a bitmap partitioning method to group the cliques so as to reduce communication overhead. In the future, we plan to take advantage of both junction tree decomposition and node level primitives so as to simultaneously explore structural parallelism and data parallelism. In addition, we plan to consider overlapping communication and computation in distributed exact inference.

ACKNOWLEDGMENTS

This research was partially supported by the US National Science Foundation (NSF) under grant number 1018801.

REFERENCES

[1] D.A. Bader, "High-Performance Algorithm Engineering for Large-Scale Graph Problems and Computational Biology," *Proc. Fourth Int'l Workshop Efficient and Experimental Algorithms*, pp. 16-21, 2005.

[2] K.W. Cameron, R. Ge, and X.-H. Sun, "$\log_n P$ and $\log_3 P$: Accurate Analytical Models of Point-to-Point Communication in Distributed Systems," *IEEE Trans. Computers*, vol. 56, no. 3, pp. 314-327, Mar. 2007.

[3] D. Heckerman, "Bayesian Networks for Data Mining," *Data Mining and Knowledge Discovery*, vol. 1, pp. 79-119, 1997.

[4] T. Ito, T. Uno, X. Zhou, and T. Nishizeki, "Partitioning a Weighted Tree to Subtrees of Almost Uniform Size," *Proc. 19th Int'l Symp. Algorithms and Computation*, pp. 196-207, 2008.

[5] A.V. Kozlov and J.P. Singh, "A Parallel Lauritzen-Spiegelhalter Algorithm for Probabilistic Inference," *Proc. Supercomputing*, pp. 320-329, 1994.

[6] S.L. Lauritzen and D.J. Spiegelhalter, "Local Computation with Probabilities and Graphical Structures and Their Application to Expert Systems," *J. Royal Statistical Soc. B*, vol. 50, pp. 157-224, 1988.

[7] M. Lin, I. Lebedev, and J. Wawrzynek, "High-Throughput Bayesian Computing Machine with Reconfigurable Hardware," *Proc. 18th Ann. ACM/SIGDA Int'l Symp. Field Programmable Gate Arrays*, pp. 73-82, 2010.

[8] B. Middleton, M. Shwe, D. Heckerman, H. Lehmann, and G. Cooper, "Probabilistic Diagnosis Using a Reformulation of the INTERNIST-1/QMR Knowledge Base," *Methods of Information Medicine*, vol. 30, pp. 241-255, 1991.

[9] D. Pennock, "Logarithmic Time Parallel Bayesian Inference," *Proc. 14th Ann. Conf. Uncertainty in Artificial Intelligence*, pp. 431-438, 1998.

[10] S.J. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, second ed. Prentice Hall, 2002.

[11] R.D. Shachter, S.K. Andersen, and P. Szolovits, "Global Conditioning for Probabilistic Inference in Belief Networks," *Proc. 10th Conf. Uncertainty in Artificial Intelligence*, pp. 514-522, 1994.

[12] M. Taufer, M. Crowley, D.J. Price, A.A. Chien, and C.L. Brooks III, "Study of a Highly Accurate and Fast Protein-Ligand Docking Method Based on Molecular Dynamics: Research Articles," *IEEE Third Int'l Workshop High Performance Computational Biology*, vol. 17, no. 14, pp. 1627-1641, Dec. 2005.

[13] USC Center for High-Performance Computing and Communications, http://www.usc.edu/hpcc/, 2012.

[14] Y. Xia, X. Feng, and V.K. Prasanna, "Parallel Evidence Propagation on Multicore Processors," *Proc. PaCT '09: 10th Int'l Conf. Parallel Computing Technologies*, pp. 377-391, 2009.

[15] Y. Xia and V.K. Prasanna, "Node Level Primitives for Parallel Exact Inference," *Proc. 19th Int'l Symp. Computer Architecture and High Performance Computing*, pp. 221-228, Oct. 2007.

[16] Y. Xia and V.K. Prasanna, "Junction Tree Decomposition for Parallel Exact Inference," *Proc. IEEE Int'l Parallel and Distributed Processing Symp. (IPDPS)*, 2008.

[17] Y. Xia and V.K. Prasanna, "Distributed Evidence Propagation in Junction Trees," *Proc. 22nd Int'l Symp. Computer Architecture and High Performance Computing*, Oct. 2010.

[18] Y. Xia and V.K. Prasanna, "Scalable Node-Level Computation Kernels for Parallel Exact Inference," *IEEE Trans. Computer*, vol. 59, no. 1, pp. 103-115, Jan. 2010.

[19] H. Zha, X. He, C. Ding, H. Simon, and M. Gu, "Bipartite Graph Partitioning and Data Clustering," *Proc. 10th Int'l Conf. Information and Knowledge Management*, pp. 25-32, 2001.

[20] A.Y. Zomaya, *Parallel Computing for Bioinformatics and Computational Biology: Models, Enabling Technologies, and Case Studies*. Wiley, 2006.

Yinglong Xia received the BS degree from the University of Electronic Science and Technology of China, the MEng degree from Tsinghua University, and the PhD degree from the University of Southern California (USC), in 2003, 2006, and 2010, respectively. Currently, he is working as a postdoctoral researcher in the IBM T.J. Watson Research Center, working on parallel algorithms for large-scale sparse graphs. His research interests include high-performance computing and parallel algorithm design for various machine learning applications. He has published technical papers in various international journals and conferences. He is a cochair or a committee member of the Workshop on Parallel and Distributed Computing for Machine Learning and Inference Problems (ParLearning). He also serves as a publicity cochair or a committee member for International Parallel and Distributed Processing Symposium (IPDPS), International Conference on Parallel and Distributed Computing and Systems (PDCS), IEEE Cluster, etc. He is a computing innovation fellow (CIFellow) and an ACM member. He is a member of the IEEE.

Viktor K. Prasanna received the BS degree in electronics engineering from the Bangalore University, the MS degree from the School of Automation, Indian Institute of Science, and the PhD degree in computer science from the Pennsylvania State University. He is Charles Lee Powell Chair in engineering and is professor of electrical engineering and professor of computer science at the University of Southern California (USC) and serves as the director of the Center for Energy Informatics (CEI). He is the executive director of the USC-Infosys Center for Advanced Software Technologies (CAST). He is an associate member of the Center for Applied Mathematical Sciences (CAMS). He also serves as an associate director of the USC-Chevron Center of Excellence for Research and Academic Training on Interactive Smart Oilfield Technologies (CiSoft) at USC. Currently, he is serving as the editor-in-chief of the *Journal of Parallel and Distributed Computing (JPDC)*. His research interests include high-performance computing, parallel and distributed systems, reconfigurable computing, cloud computing, and embedded systems. He has published extensively and consulted for industries in the above areas. He is the Steering Committee cochair of the International Parallel and Distributed Processing Symposium (IPDPS) [merged IEEE International Parallel Processing Symposium (IPPS) and Symposium on Parallel and Distributed Processing (SPDP)]. He is the Steering Committee Chair of the International Conference on High Performance Computing (HiPC). In the past, he has served on the editorial boards of the *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*,*IEEE Transactions on Parallel and Distributed Systems (TPDS)*, *Journal of Pervasive and Mobile Computing*, and the Proceedings of the IEEE. He serves on the editorial boards of the *Journal of Parallel and Distributed Computing* and the *ACM Transactions on Reconfigurable Technology and Systems*. During 2003-2006, he was the editor-in-chief of the *IEEE Transactions on Computers*. He was the founding chair of the *IEEE Computer Society Technical Committee on Parallel Processing*. He received an Outstanding Achievement Award at the 2009 World Congress in computer science, computer engineering, and applied computing (WORLDCOMP '09) for his contributions to reconfigurable computing and an Outstanding Engineering Alumnus Award from the Pennsylvania State University, in 2009. He has received best paper awards at several international forums including ACM Computing Frontiers (CF), IEEE International Parallel and Distributed Processing Symposium (IPDPS), International Conference on Parallel and Distributed Systems (ICPADS), International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD), International Conference on Parallel and Distributed Computing and Systems (PDCS), among others. He is a recipient of the 2005 Okawa Foundation Grant. He is a fellow of the IEEE, the Association for Computing Machinery (ACM) and the American Association for Advancement of Science (AAAS).