EXACT INFERENCE ON MANYCORE PROCESSORS USING POINTER JUMPING

Nam Ma
Computer Science Department
University of Southern California
Los Angeles, CA 90089
email: namma@usc.edu

Yinglong Xia
Computer Science Department
University of Southern California
Los Angeles, CA 90089
email: yinglonx@usc.edu

Viktor Prasanna
Ming Hsieh Department of
Electrical Engineering
University of Southern California
Los Angeles, CA 90089
email: prasanna@usc.edu

ABSTRACT
Exact inference is a key problem in exploring probabilistic graphical models. Most parallel algorithms for exact inference explore data and structural parallelism. These algorithms result in limited performance if the input model offers limited data and structural parallelism. In this paper, we study a pointer jumping based method on manycore systems for exact inference in junction trees. We adapt the technique for both evidence collection and evidence distribution so as to efficiently process junction trees with multiple evidence cliques. We also study the impact of junction tree topology on evidence collection. We implement the proposed method on state-of-the-art manycore systems. Experimental results show that, for junction trees with limited data and structural parallelism, pointer jumping is well suited to accelerate exact inference on manycore systems.

KEY WORDS
Exact Inference, Junction Tree, Pointer Jumping, Manycore System

1 Introduction

Many real-world systems can be modeled using full joint probability distribution of a set of random variables. However, such a distribution grows intractably with the number of variables. It is known that independence and conditional independence relationships can greatly reduce the size of the joint probability distributions. This property is utilized in *Bayesian networks*, which have been used in artificial intelligence since the 1960s. Bayesian networks have found applications in a number of domains, including medical diagnosis, consumer help desk, pattern recognition, credit assessment, data mining and genetics [1] [2] [3].

Given a set of variables with observed values called *evidence*, the computation of the conditional probability of other variables is called *inference*. Inference in a Bayesian network can be *exact* or *approximate*. In general, exact inference is NP hard [4]. By exploiting network structure, many algorithms have been proposed to make exact inference practical for a wide range of applications. The most popular exact inference algorithm proposed by Lauritzen and Spiegelhalter [5] converts a Bayesian network into a *junction tree*, then performs exact inference in the junction tree. Each vertex in the junction tree is a clique consisting of a subset of the variables. The complexity of exact inference in junction tree increases dramatically with the number of cliques in the junction tree, the maximum number of variables in a clique, and the number of states of the random variables [5].

Many parallel techniques have been developed to accelerate exact inference. In [6] [7], the authors exploit *structural parallelism* by parallelizing independent computations across cliques that are not related by an ancestor-descendent relationship in the network. In [6] [8] [9], the authors exploit *data parallelism* by parallelizing independent computations within a clique. In [10], the authors propose a pointer jumping based method for exact inference in Bayesian networks. This method achieves logarithmic execution time regardless of the network topology. To the best of our knowledge, no experimental study of pointer jumping based exact inference on multicore/manycore processors has been performed.

In this paper, we make the following contributions:

- We adapt the pointer jumping technique for both evidence collection and evidence distribution in junction trees. The proposed method is suitable for junction trees that offer limited data and structural parallelism.
- We implement the proposed method using Pthreads on two state-of-the-art manycore systems: Intel Nehalem-EX and AMD Magny-Cours.
- We evaluate our implementation and compare it with our baseline method - a parallel implementation that exploits structural parallelism. Our experimental results show that the technique is suitable for a class of junction trees.

The rest of the paper is organized as follows: In Section 2, we review exact inference in Bayesian networks and the pointer jumping technique. Section 3 discusses related work. Section 4 presents our proposed algorithm with alternative formulas for evidence propagation. We describe our implementation and experiment facilities in Sections 5. Experimental results are presented in Section 6. We provide our conclusion and future research directions in Section 7.

2 Background

2.1 Exact Inference

A *Bayesian network* is a probabilistic graphical model that exploits conditional independence to compactly represent a joint distribution. Figure 1 (a) 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 can *not* form directed cycles. Thus, the structure of a Bayesian network is a *directed acyclic graph* (DAG). The *evidence* in a Bayesian network is the set of variables that have been instantiated.

Traditional exact inference using Bayes' theorem fails for networks with undirected cycles [5]. Most inference methods for networks with undirected cycles convert a network to a cycle-free hypergraph called a *junction tree*. We illustrate a junction tree converted from the Bayesian network in Fig. 1, where all undirected cycles in are eliminated. Each vertex in Fig. 1(b) 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. Assuming $\mathcal{C}_i$ and $\mathcal{C}_j$ are adjacent, 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 $\mathcal{C}_i$. For a clique with w variables, each having r states, the number of entries in $\mathcal{C}_i$ is r^w.

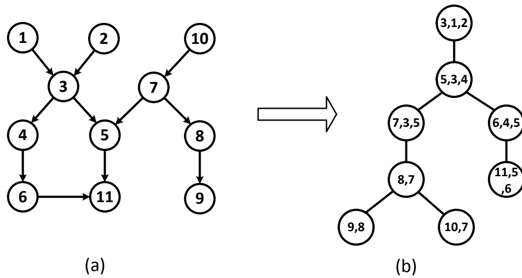


Figure 1. (a) A sample Bayesian network and (b) corresponding junction tree.

In a junction tree, exact inference is performed as follows: Assuming 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 any 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 [5]:

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

where $\mathcal{S}$ is a 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 $\mathcal{C}_\mathcal{X}$.

A junction tree is updated using the above evidence propagation in two passes, i.e., *evidence collection* and *evidence distribution*. In evidence collection, evidence is propagated from the leaves to the root. A clique $\mathcal{C}$ is ready to propagate evidence to its parent when $\mathcal{C}$ has been already updated, i.e. when all of the children of $\mathcal{C}$ finished propagating evidence to $\mathcal{C}$. Evidence distribution is the same as collection, except that the evidence propagation direction is from the root to the leaves. This two-pass procedure ensures that evidence at any cliques in a junction tree can be propagated to all the other cliques [5].

2.2 Pointer Jumping

Pointer jumping technique is used in many parallel algorithms operating on lists [11][12]. Given a linked list of N elements, each element i has a successor $s(i)$ which is initialized as $s(i) = i+1, 0 \leq i \leq N-2$. Each element sends data to its successor. Pointer jumping technique conceptually performs data transfer from all the elements to the last node of the list (node $(N-1)$) in $\log\ N$ iterations. All logarithms are to base 2. Each iteration of pointer jumping performs $s(i) = s(s(i)), 0 \leq i \leq N-2$, unless $s(i) = N-1$. In other words, the successor of each element is updated by that successor's successor. In each iteration, the data of element i is transferred to the updated successor of i independently of the operations performed by other elements. Figure 2 illustrates the iterations of pointer jumping on a list of N elements. After $\log\ N$ iterations, all the elements complete transferring their data to node $(N-1)$.

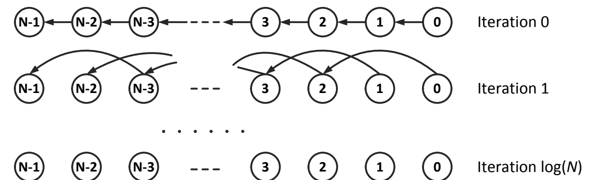


Figure 2. Pointer jumping on a list.

On the concurrent read exclusive write parallel random access machine (CREW-PRAM) [12] with P processors, $1 \leq P \leq N$, each element of the list is assigned to a processor. Note that the update operations are implicitly synchronized at the end of each pointer jumping iteration. Each iteration takes $O(N/P)$ time to complete $O(N)$ update operations. Thus, the pointer jumping based algorithm runs in $O((N \cdot \log\ N)/P)$ time using a total number of $O(N \log\ N)$ update operations.

3 Related Work

There are several works on parallel exact inference. In [6], the authors explore *structural parallelism* in exact inference by assigning independent cliques to separate proces-

sors for concurrent processing. In [7], the structural parallelism is explored by a dynamic task scheduling method, where the evidence propagation in each clique is viewed as a task. These methods are suitable for "bushy" junction trees which offer parallel activities at structural level. Unlike the above techniques, *data parallelism* in exact inference is explored in [8][6] [9]. Since the solutions for exploring data parallelism partition the potential tables in a clique and update each part in parallel, they are suitable for junction trees with large potential tables. If the input junction tree offers limited structural and data parallelism, the performance of the above solutions are adversely affected.

Pointer jumping has been used in many algorithms, such as list ranking, parallel prefix [12]. In [13], the authors use pointer jumping to develop a new spanning tree algorithm for symmetric multiprocessors that achieves parallel speedups on arbitrary graphs. In [10], the authors propose a pointer jumping based method for exact inference in Bayesian networks. The method achieves logarithmic execution time regardless of structural or data parallelism in the networks. However, the method in [10] exhibits limited performance for multiple evidence inputs. In addition, no experimental study of the method has been provided. Our proposed algorithm combines pointer jumping technique with Lauritzen and Spiegelhalter's two-pass algorithm, which is efficient for multiple evidence inputs. In this paper, the proposed algorithm is empirically evaluated using a class of junction trees on state-of-the-art manycore processors.

4 Pointer Jumping Based Algorithms for Exact Inference in Junction Tree

4.1 Evidence Propagation in a Chain of Cliques Using Pointer Jumping

Given a chain consisting of N cliques $\mathcal{C}_0 \rightarrow \mathcal{C}_1 \rightarrow \ldots \rightarrow \mathcal{C}_{N-1}$, in which evidence comes at $\mathcal{C}_0$, we use pointer jumping technique to propagate the evidence from $\mathcal{C}_0$ to the remaining cliques as follows: Let $pa(\mathcal{C}_i)$ denote the parent of $\mathcal{C}_i$, i.e., $\mathcal{C}_{i-1}$, $0 < i < N$. In the first iteration, we propagate the evidence from $pa(\mathcal{C}_i)$ to $\mathcal{C}_i$. Then, we propagate the evidence from $pa^2(\mathcal{C}_i) = pa(pa(\mathcal{C}_i))$ to $\mathcal{C}_i$. We repeat such operations for $\log N$ iterations so that the evidence is propagated to all the cliques.

We formulate evidence propagation in a chain of cliques as follows. Let $\mathcal{C}_i = \{S_{i+1,i}, V_i, S_{i,i-1}\}$, where $S_{i+1,i} = \mathcal{C}_{i+1} \cap \mathcal{C}_i$, $S_{i,i-1} = \mathcal{C}_i \cap \mathcal{C}_{i-1}$ and $V_i = \mathcal{C}_i \backslash \{S_{i+1,i} \cup S_{i,i-1}\}$, $0 < i < N$. Indeed $S_{i+1,i}$ is the separator between $\mathcal{C}_{i+1}$ and $\mathcal{C}_i$. For the sake of simplicity, we let $S_{0,-1} = S_{N,N-1} = \emptyset$ and $P(\mathcal{C}_i) = P(S_{i+1,i}, V_i, S_{i,i-1}) = \psi_{\mathcal{C}}$. We rewrite $P(\mathcal{C}_i)$ as a conditional distribution:

$$\begin{aligned} P(S_{i+1,i}, V_i | S_{i,i-1}) &= \frac{P(S_{i+1,i}, V_i, S_{i,i-1})}{P(S_{i,i-1})} \\ &= \frac{P(S_{i+1,i}, V_i, S_{i,i-1})}{\sum_{S_{i+1,i} \cup V_i} P(S_{i+1,i}, V_i, S_{i,i-1})} \\ &= \frac{P(\mathcal{C}_i)}{\sum_{S_{i+1,i} \cup V_i} P(\mathcal{C}_i)} \end{aligned} \quad (2)$$

$$\begin{aligned} P(S_{i+1,i} | S_{i,i-1}) &= \frac{\sum_{V_i} P(S_{i+1,i}, V_i, S_{i,i-1})}{P(S_{i,i-1})} \\ &= \sum_{V_i} \frac{P(\mathcal{C}_i)}{\sum_{S_{i+1,i} \cup V_i} P(\mathcal{C}_i)} \end{aligned} \quad (3)$$

Using Eqs. 2 and 3, in iteration 0, we propagate the evidence from $pa(\mathcal{C}_i)$ to $\mathcal{C}_i$ by:

$$\begin{aligned} &P(S_{i+1,i}, V_i | S_{i-1,i-2}) \\ &= \sum_{S_{i,i-1}} P(S_{i+1,i}, V_i | S_{i,i-1}) P(S_{i,i-1} | S_{i-1,i-2}) \end{aligned} \quad (4)$$

$$P(S_{i+1,i} | S_{i-1,i-2}) = \sum_{V_i} P(S_{i+1,i}, V_i | S_{i-1,i-2}) \quad (5)$$

In iteration k, we propagate evidence from $pa^{2^k}(\mathcal{C}_i)$ to $\mathcal{C}_i$ by:

$$\begin{aligned} &P(S_{i+1,i}, V_i | S_{i-2^k, i-2^k-1}) \\ &= \sum_{S_{i-2^{k-1}, i-2^{k-1}-1}} P(S_{i+1,i}, V_i | S_{i-2^{k-1}, i-2^{k-1}-1}) \\ &\qquad P(S_{i-2^{k-1}, i-2^{k-1}-1} | S_{i-2^k, i-2^k-1}) \end{aligned} \quad (6)$$

$$P(S_{i+1,i} | S_{i-2^k, i-2^k-1}) = \sum_{V_i} P(S_{i+1,i}, V_i | S_{i-2^k, i-2^k-1}) \quad (7)$$

Performing Eqs. 6 and 7 for each $k, 0 \leq k \leq \log N - 1$, we propagate the evidence from $\mathcal{C}_0$ to all the cliques in a chain of N cliques. For the rest of the paper, we define *evidence propagation* as the computation using Eqs. 6 and 7 to propagate evidence from one clique from another clique.

4.2 Pointer Jumping Based Algorithm for Exact Inference in a Junction Tree

Algorithm 1 shows exact inference in a junction tree using pointer jumping. The algorithm is based on Lauritzen and Spiegelhalter's two-pass algorithm described in Section 2.1. The two passes are called evidence collection and evidence distribution. Evidence collection propagates evidence from leaf cliques to the root. Evidence distribution propagates evidence from the root to the leaf cliques.

Given a junction tree J, let $ch(\mathcal{C}_i)$ denote the *set* of children of $\mathcal{C}_i \in$ J, i.e., $ch(\mathcal{C}_i) = \{\mathcal{C}_j | pa(\mathcal{C}_j) = \mathcal{C}_i\}$. We define $ch^d(\mathcal{C}_i) = \{\mathcal{C}_j | pa^d(\mathcal{C}_j) = \mathcal{C}_i\}$, and $pa^d(\mathcal{C}_j) = pa(pa^{d-1}(\mathcal{C}_j)) = pa(\cdots(pa(\mathcal{C}_j)))$, where $d \geq 1$. We denote H as the height of J. Updating a clique means that we perform all evidence propagations from its predecessors that are either its children in evidence collection or its parent in evidence distribution.

Algorithm 1 Pointer Jumping Based Exact Inference in a Junction Tree

Input: Junction tree J with a height of H
Output: J with updated potential tables

```
    {evidence collection}
 1: for k = 0, 1, ..., (log H) - 1 do
 2:   for C_i ∈ J pardo
 3:     for all [C_j ∈ ch^{2^k}(C_i)] do
 4:       Propagate evidence from C_j to C_i according to Eqs. 6
          and 7
 5:     end for
 6:   end for
 7: end for
    {evidence distribution}
 8: for k = 0, 1, ..., (log H) - 1 do
 9:   for C_i ∈ J pardo
10:     [C_j = pa^{2^k}(C_i)]
11:     Propagate evidence from C_j to C_i according to Eqs. 6
        and 7
12:   end for
13: end for
```

In Algorithm 1, Lines 1-7 and 8-13 correspond to evidence collection and distribution, respectively. Lines 3 and 10 show how pointer jumping technique is used in the algorithm. In each iteration, all the cliques can be updated independently. In evidence collection, in iteration k, a clique $\mathcal{C}_i$ in J is updated by performing all the evidence propagations from the set $ch^{2^k}(\mathcal{C}_i)$. In evidence distribution, in iteration k, a clique $\mathcal{C}_i$ in J is updated by performing the evidence propagation from its only parent $pa^{2^k}(\mathcal{C}_i)$.

Assume that P processors, $1 \leq P \leq N$, are employed. Let $W = \max_i\{W_i\}$, where W_i is the number of cliques at level i of the junction tree, $0 \leq i < H$. Assuming each evidence propagation takes unit time, updating a clique takes $O(W)$ time during evidence collection, since the clique must be updated by sequentially performing $O(W)$ evidence propagations from its children. In contrast, updating a clique takes $O(1)$ time during evidence distribution since only one evidence propagation from its parent is needed. In each iteration, there are at most $O(N)$ clique updates performed by P processors. The algorithm performs $\log\ H$ iterations for each evidence collection and distribution. Thus, the complexity of the algorithm is $O((W \cdot N \cdot \log\ H)/P)$ for evidence collection and $O((N \cdot \log\ H)/P)$ for evidence distribution. The overall complexity is $O((W \cdot N \cdot \log\ H)/P)$.

The algorithm is attractive if the input junction tree offers limited structural parallelism. For example, for a chain of N cliques, we have $W = 1$, $H = N$, the complexity of the algorithm is $O((N \cdot \log N)/P)$.

5 Experimental Setup

5.1 Platforms

We conducted experiments on two state-of-the-art many-core platforms:

- 32-core Intel Nehalem-EX based system which consisted of four Intel Xeon X7560 processors fully connected through 6.4 GT/s QPI links. Each processor had 24 MB L3 Cache and 8 cores running at 2.26 GHz. All the cores shared a 512 GB DDR3 memory. The operating system on this system was Red Hat Enterprise Linux Server release 5.4.

- 24-core AMD Magny-Cours based system which consisted of two AMD Opteron 6164 HE processors. Each processor had 12 MB L3 Cache and 12 cores running at 1.70 GHz. All the cores shared a 64 GB DDR3 memory. The operating system on the system was CentOS release 5.3.

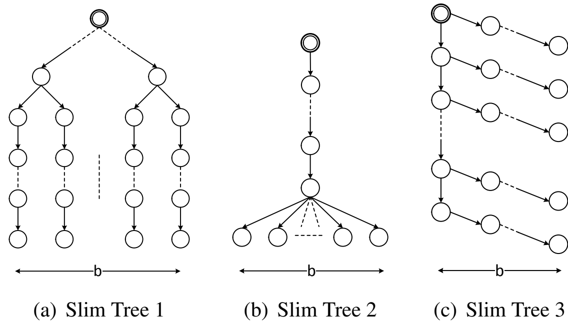
Due to the similarity of the results from the two platforms, we only present the results from the 32-core Nehalem-EX based system.

5.2 Junction Tree Topologies

We evaluated the performance of our proposed algorithm using three families of junction trees: chains, balanced binary trees, and slim trees. Given $W = \max_i\{W_i\}$ for a junction tree, where W_i is the number of cliques at level i of the junction tree, we define a *slim tree* as a junction tree satisfying $W < H$, where H is the height of the tree. Note that W is an upper bound on the number of sequential evidence propagations required for updating a clique. H determines the number of pointer jumping iterations. Using slim trees, we can observe the performance of the proposed algorithm, even though there exists some amount of structural parallelism in such junction trees. The clique size, i.e. the number of variables in the clique, was $W_d = 12$ in our experiments. This offered limited amount of data parallelism. The random variables were set as binary, i.e. $r = 2$, in our experiments.

Let N denote the number of cliques in a slim tree. Three different types of slim trees were generated for the experiments. As illustrated in Figure 3, these are:

- *Slim Tree 1* was formed by first creating a balanced binary tree with b leaf nodes. Then each of the leaf nodes was connected to a chain consisting of $\lceil \frac{N-(2b-1)}{b} \rceil$ nodes. The height of the tree is $(\lceil \log b \rceil + \lceil \frac{N-(2b-1)}{b} \rceil)$. Increasing b increases the amount of structural parallelism.



(a) Slim Tree 1 (b) Slim Tree 2 (c) Slim Tree 3

Figure 3. Three types of slim junction trees

- *Slim Tree 2* was formed by connecting b cliques to the head of a chain consisting of $(N - b)$ nodes. The height of the tree is $(N - b + 1)$. Increasing b increases the amount of structural parallelism. Larger b also results in more sequential operations to update a clique during evidence collection.
- *Slim Tree 3* was formed by first creating a chain of length $\lceil \frac{N}{b} \rceil$. Then, each clique in the chain was connected to another chain consisting of $(b - 1)$ cliques, except that the last chain connected to the root may have less than $(b - 1)$ cliques. The height of the tree is $(\lceil \frac{N}{b} \rceil + b - 1)$. Increasing b reduces the amount of structural parallelism.

In the experiments, for each given number of cliques N, we varied b to change slim tree topology.

5.3 Baseline Implementation

We use a parallel implementation as the baseline to evaluate our propose method. The parallel implementation explores structural parallelism offered by junction trees. A collaborative scheduler proposed in [14] was employed to schedule tasks to threads, where each thread was bound to a separate core. We call this baseline implementation as the scheduling based exact inference (SEI). In this implementation, we represented a junction tree as an adjacency array, each element representing a clique and the links to the children of the clique. Each task had an attribute called the dependency degree, which was initially set as the in-degree of the corresponding clique. The scheduling activities were distributed across the threads. The adjacency array was shared by all the threads. In addition to the shared adjacency array, each clique had a ready task list to store the assigned tasks that are ready to be executed. Each thread fetched a task from its ready task list and executed it locally. After the task completed, the thread reduced the dependency degree of the children of the clique. If the dependency degree of a child became 0, the child clique was assigned to the ready task list of the thread with the lightest workload.

5.4 Implementation Details

We implemented our proposed algorithm using Pthreads on manycore systems. We initiated as many threads as the number of hardware cores in the target platform. Each thread was bound to a separate core. These threads persisted over the entire program execution and communicated with each other using the shared memory.

The input junction tree was stored as an adjacency array in the memory. Similar to the SEI, the adjacency array consists of a list of nodes, each corresponding to a clique in the given junction tree. Each node has links to its children and a link to its parent. Note that the links were dynamically updated due to the nature of pointer jumping.

We define a *task* as the computation for updating a clique in an iteration of pointer jumping. The input to a task is the clique to be updated and the set of cliques propagating evidence to it. The output of a task is the clique after updated. In iteration k of pointer jumping in evidence collection, a task corresponds to Lines 3-5 in Algorithm 1, which updates clique $\mathcal{C}_i$ by sequentially performing multiple evidence propagations using cliques in $ch^{2^k}(\mathcal{C}_i)$. In iteration k of pointer jumping in evidence distribution, a task corresponds to Lines 10-11 in Algorithm 1, which updates clique $\mathcal{C}_i$ by performing a single evidence propagation using clique $pa^{2^k}(\mathcal{C}_i)$.

In each iteration of pointer jumping (in both evidence collection and evidence distribution), all the tasks can be executed in parallel. These tasks were statically distributed to the threads by a straightforward scheme: the task corresponding to clique $\mathcal{C}_i$, $O \leq i < N$, is distributed to thread $(i \mod P)$, where P is the number of threads. Hence, a clique is always updated by the same thread, although the cliques propagating evidence to it vary from iteration to iteration. Note that, in an iteration, the execution time of the tasks can vary due to different numbers of evidence propagations performed in the tasks.

We explicitly synchronized the update operations across the threads at the end of each iteration of pointer jumping using a barrier.

6 Results and Analysis

Figures 4, 5, and 6 illustrate the performance of the proposed algorithm and the scheduling-based exact inference (SEI) for various junction trees. We consider the execution time and the scalability with respect to the number of processors for each algorithm.

For chains, as shown in Figure 4, SEI showed limited scalability; in contrast, the proposed algorithm scaled very well with the number of cores. This is because for chains, SEI had no structural parallelism to exploit. For a chain of 4096 cliques, on a single core, the proposed algorithm ran much slower than SEI; however, when the number of cores exceeded 16, the proposed algorithm ran faster than SEI.

For balanced trees, as shown in Figure 5, both methods scaled well due to available structural parallelism. The scalability of SEI was even better than that of the proposed algorithm. This is because for balanced trees, during evidence collection in the proposed algorithm, some clique are

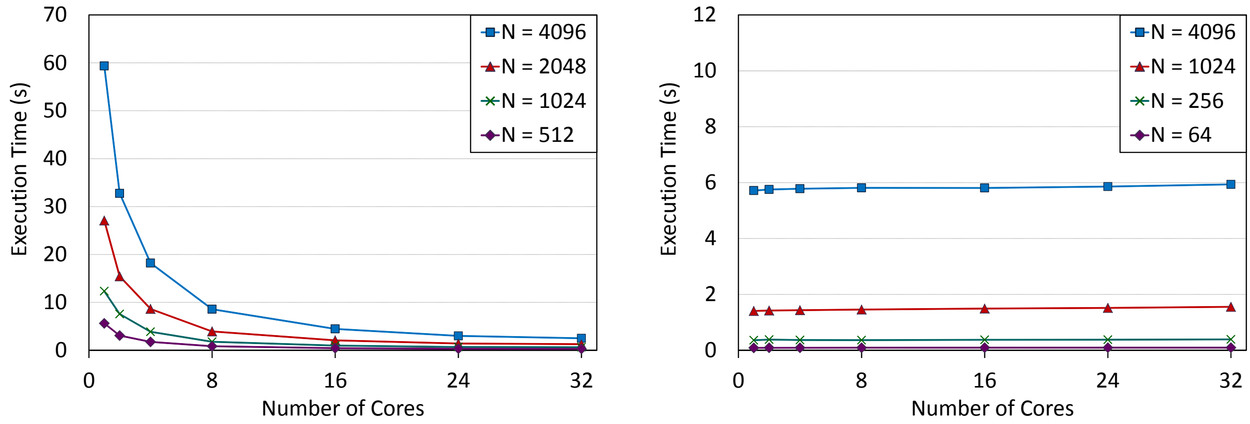


Figure 4. Comparison between the proposed algorithm (left) and SEI (right) for chains on the 32-core Intel Nehalem-EX based system.

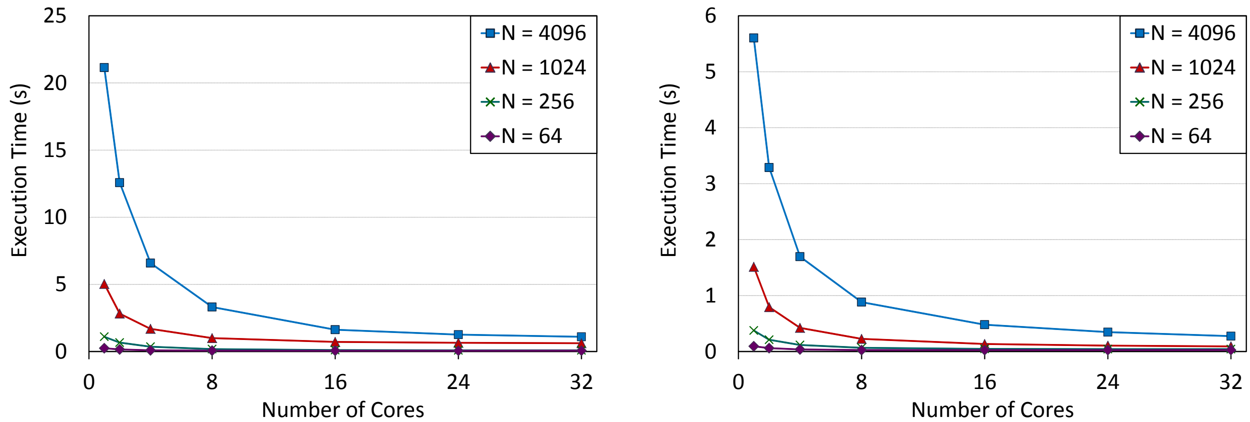


Figure 5. Comparison between the proposed algorithm (left) and SEI (right) for balanced trees on the 32-core Intel Nehalem-EX based system.

updated by sequentially performing a large number of evidence propagations from its children. Load imbalance can occur due to the straightforward scheduling scheme employed by us. By employing the work-sharing scheduler, SEI had better load balance. Note that SEI ran much faster than the proposed algorithm.

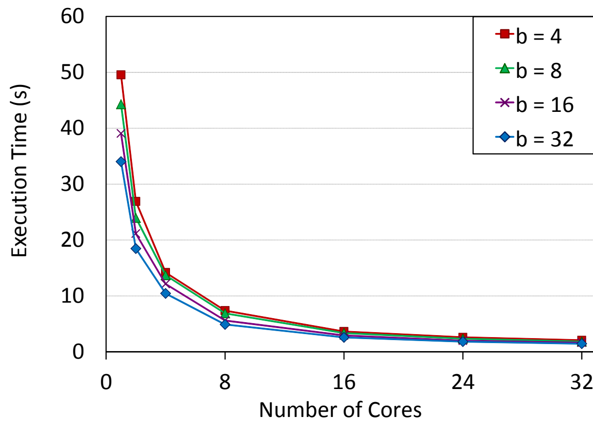
To understand better the impact of junction tree topology on the scalability, we used slim trees for the experiments. The results are shown in Figure 6. The number of cliques (N) was 4096. We varied the amount of structural parallelism offered by the junction trees by changing the parameter b (see Section 5.2) for each type of slim trees.

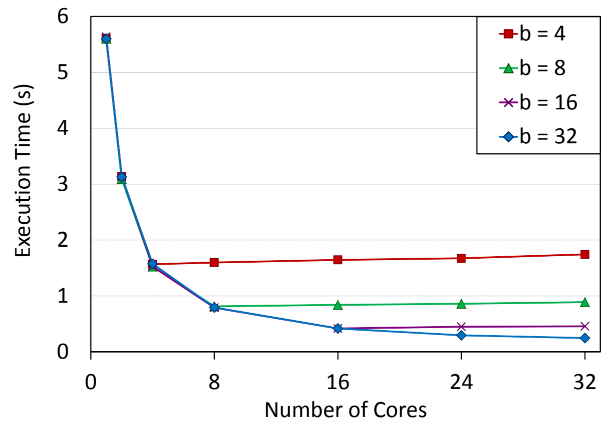
For Slim Tree 1, we observed that SEI did not scale when the number of cores P exceeded b. The proposed algorithm still scaled well. The scalability of the proposed algorithm is not significantly affected by b. Thus, when b was small and P was large, e.g. $b = 4$ and $P = 32$, the proposed algorithm ran faster than SEI.

For Slim Tree 2, increasing b made SEI scale better. In contrast, the proposed algorithm did not scale well when b increased because of the load imbalance. Load imbalance occurs because in each iteration of pointer jumping in evidence collection, some cliques are updated by sequentially performing b evidence propagations, while the other cliques perform only one evidence propagation. Thus, when b was large, e.g. 256 or 1024, for the number of cores used in the experiments, SEI ran faster than the proposed algorithm. When b was small and P was large, e.g. $b = 16$ and $P = 32$, the proposed algorithm ran faster than SEI.

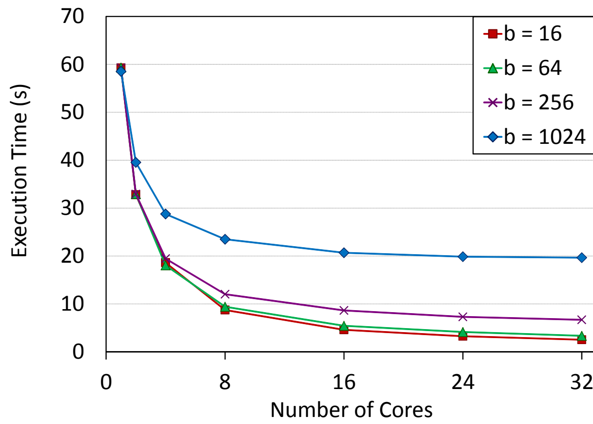
For Slim Tree 3, although increasing b reduces the amount of structural parallelism, it also reduces the length of the sequential path (chains consisting of $\lceil \frac{N}{b} \rceil$ cliques). Hence, as long as N/b is greater than the number of cores P, increasing b makes SEI scale better. If N/b is less than P, then all the processors are not fully utilized by SEI. We observed that for $N = 4096$ and $b = 256$, SEI no longer scaled when the number of cores P exceeded 16. The proposed algorithm scaled best when $b = 4$. When $b = 4$ and $P = 32$, the proposed algorithm ran faster than SEI.

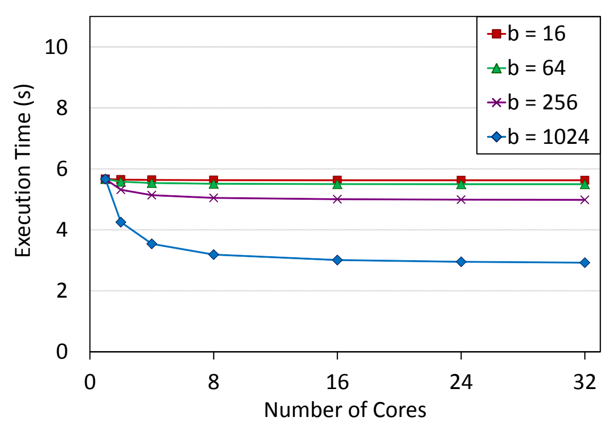
Examining the execution time of the proposed algo-



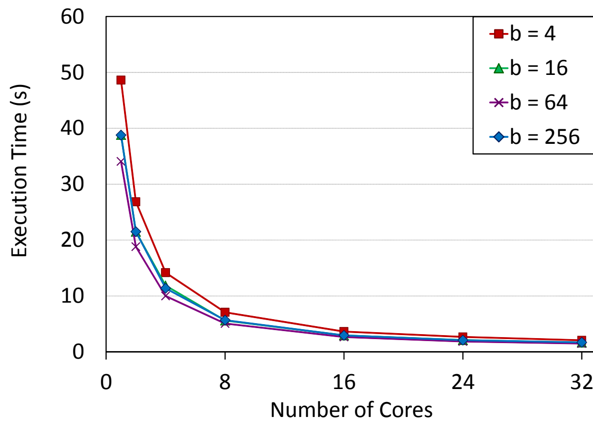


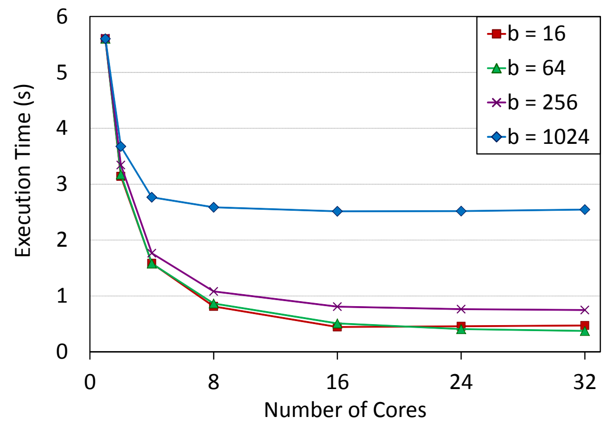
(a) The proposed algorithm (left) and SEI (right) for Slim Tree 1.





(b) The proposed algorithm (left) and SEI (right) for Slim Tree 2.





(c) The proposed algorithm (left) and SEI (right) for Slim Tree 3.

Figure 6. Comparison between the proposed algorithm and the scheduling based exact inference (SEI) for slim trees with $N = 4096$ cliques on the 32-core Intel Nehalem-EX based system.

rithm and SEI in all the cases, we can conclude that when SEI does not scale with the number of cores due to lack of structural parallelism, the proposed algorithm can offer superior performance compared with SEI by using a large number of cores.

7 Conclusions

In this paper, we adapted a pointer jumping based method to explore exact inference in junction trees with limited data and structural parallelism on manycore systems. The proposed method parallelizes both evidence collection and distribution. Due to the sequential update for each clique with respect to its children, the performance of the pointer jumping based evidence collection depends on the topology of the input junction tree. Our experimental results show that for junction trees with limited structural parallelism, the proposed algorithm is well suited for manycore platforms. In the future, we plan to improve the load balance across threads within each iteration of pointer jumping. In addition, we are considering extending the algorithm by integrating the methods exploring structural parallelism and data parallelism so that the algorithm can automatically choose the best method for a given junction tree. We also plan to investigate the parallelization of clique update with respect to its children during evidence collection.

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

References

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

[2] E. Segal, B. Taskar, A. Gasch, N. Friedman, and D. Koller, "Rich probabilistic models for gene expression," in *Proceedings of the 9th International Conference on Intelligent Systems for Molecular Biology*, 2001, pp. 243–252.

[3] D. Heckerman, "Bayesian networks for data mining," *Data Mining and Knowledge Discovery*, vol. 1, no. 1, pp. 79–119, 1997.

[4] G. F. Cooper, "The computational complexity of probabilistic inference using bayesian belief networks," *Artificial Intelligence*, vol. 42, no. 2-3, pp. 393–405, 1990.

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

[6] A. V. Kozlov and J. P. Singh, "A parallel Lauritzen-Spiegelhalter algorithm for probabilistic inference," in *Proceedings of Supercomputing*, 1994, pp. 320–329.

[7] Y. Xia, X. Feng, and V. K. Prasanna, "Parallel evidence propagation on multicore processors," in *PaCT*, 2009, pp. 377–391.

[8] B. D'Ambrosio, T. Fountain, and Z. Li, "Parallelizing probabilistic inference: Some early explorations," in *UAI*, 1992, pp. 59–66.

[9] 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*, October 2007, pp. 221–228.

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

[11] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*. The MIT Press and McGraw-Hill Book Company, 1990.

[12] J. JáJá, *An Introduction to Parallel Algorithms*. Reading, MA: USA: Addison-Wesley, 1992.

[13] D. A. Bader and G. Cong, "A fast, parallel spanning tree algorithm for symmetric multiprocessors (smps)," *Parallel and Distributed Computing*, vol. 65, no. 9, pp. 994–1006, 2005.

[14] Y. Xia and V. K. Prasanna, "Collaborative scheduling of dag structured computations on multicore processors," in *Conf. Computing Frontiers*, 2010, pp. 63–72.