

Parallel Evidence Propagation on Multicore Processors^{*}

Yinglong Xia¹, Xiaojun Feng³ and Viktor K. Prasanna^{2,1}

Computer Science Department¹ and Department of Electrical Engineering²
University of Southern California, Los Angeles, CA 90089, U.S.A.
Department of Computer Science and Technology³
Tsinghua University, Beijing 100084, China
{yinglonx, prasanna}@usc.edu, fxj05@mails.tsinghua.edu.cn

Abstract. In this paper, we design and implement an efficient technique for parallel evidence propagation on state-of-the-art multicore processor systems. Evidence propagation is a major step in exact inference, a key problem in exploring probabilistic graphical models. We propose a rerooting algorithm to minimize the critical path in evidence propagation. The rerooted junction tree is used to construct a directed acyclic graph (DAG) where each node represents a computation task for evidence propagation. We develop a collaborative scheduler to dynamically allocate the tasks to the cores of the processors. In addition, we integrate a task partitioning module in the scheduler to partition large tasks so as to achieve load balance across the cores. We implemented the proposed method using Pthreads on both AMD and Intel quadcore processors. For a representative set of junction trees, our method achieved almost linear speedup. The execution time of our method was around twice as fast as the OpenMP based implementation on both the platforms.

Key words: Exact inference, Multicore, Junction tree, Scheduling

1 Introduction

A full joint probability distribution for any real-world system can be used for inference. However, such a distribution increases intractably with the number of variables used to model the system. It is known that independence and conditional independence relationships can greatly reduce the size of the joint probability distributions. This property is utilized by *Bayesian networks* [1]. Bayesian networks have been used in artificial intelligence since the 1960s. They have found applications in a number of domains, including medical diagnosis, consumer help desks, pattern recognition, credit assessment, data mining and genetics [2][3][4].

Inference in a Bayesian network is the computation of the conditional probability of the *query* variables, given a set of *evidence* variables as the knowledge

^{*} 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.

to the network. Inference in a Bayesian network can be *exact* or *approximate*. Exact inference is NP hard [5]. The most popular exact inference algorithm for multiply connected networks was proposed by Lauritzen and Spiegelhalter [1], which converts a Bayesian network into a *junction tree*, then performs exact inference in the junction tree. The complexity of exact inference algorithms increases dramatically with the density of the network, the width of the cliques and the number of states of the random variables in the cliques. In many cases exact inference must be performed in real time.

Almost all recent processors are designed to process simultaneous threads to achieve higher performance than single core processors. Typical examples of multicore processors available today include AMD Opteron and Intel Xeon. While chip multi-processing has been devised to deliver increased performance, an important challenge is to exploit the available parallelism. Prior work has shown that system performance is sensitive to thread scheduling in simultaneous multithreaded (SMT) architectures [6]. To maximize the potential of such multicore processors, users must understand both the algorithmic and architectural aspects to design efficient scheduling solutions.

In this paper, we study parallelization of evidence propagation on state-of-the-art multicore processors. We exploit both structural parallelism and data parallelism to improve the performance of evidence propagation. We achieved speedup of 7.4 using 8 cores on state-of-the-art platforms. This speedup is much higher compared with the baseline methods e.g. OpenMP based implementation. The proposed method can be extended for online scheduling of directed acyclic graph (DAG) structured computations.

The paper is organized as follows: In Section 2, we discuss the background of evidence propagation. Section 3 introduces related work. In Section 4, we present junction tree rerooting. Section 5 defines computation tasks for evidence propagation. Section 6 presents our collaborative scheduler for multicore processors. Experimental results are shown in Section 7. Section 8 concludes the paper.

2 Background

A *Bayesian network* is a probabilistic graphical model that exploits conditional independence to represent compactly a joint distribution. Figure 1 (a) shows a sample Bayesian network, where each node represents a random variable. The edges indicate 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), denoted $\mathbb{G} = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V} = \{A_1, A_2, \dots, A_n\}$ is the node set and \mathcal{E} is the edge set. Each random variable in the Bayesian network has a *conditional probability table* $P(A_j|pa(A_j))$, where $pa(A_j)$ is the parents of A_j . Given the Bayesian network, a joint distribution is given by $P(\mathcal{V}) = \prod_{j=1}^n P(A_j|pa(A_j))$, where $A_j \in \mathcal{V}$ [1].

The *evidence* in a Bayesian network is the variables that have been instantiated, e.g. $E = \{A_{e_1} = a_{e_1}, \dots, A_{e_c} = a_{e_c}\}$, $e_k \in \{1, 2, \dots, n\}$, where A_{e_i} is a variable and a_{e_i} is the instantiated value. Evidence can be propagated to other

variables in the Bayesian network using Bayes' Theorem. Propagating the evidence throughout a Bayesian network is called *inference*, which can be *exact* or *approximate*. Exact inference is proven to be NP hard [5]. The computational complexity of exact inference increases dramatically with the size of the Bayesian network and the number of states of the random variables.

Traditional exact inference using Bayes' theorem fails for networks with undirected cycles [1]. 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 (Figure 1 (a)) in Figure 1 (b), where all undirected cycles in are eliminated. Each vertex in Figure 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 to formulate a junction tree. A junction tree is defined as $J = (\mathbb{T}, \hat{\mathbb{P}})$, 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 .

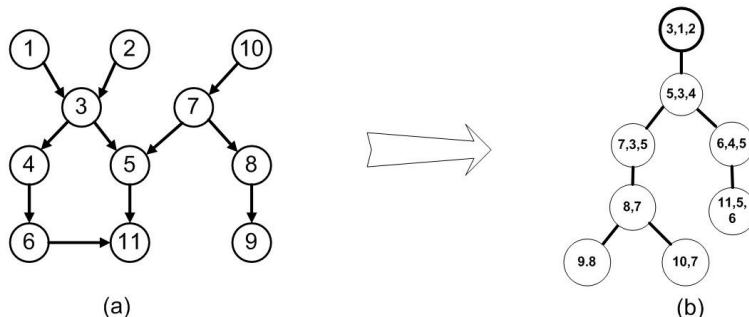


Fig. 1. (a) A sample Bayesian network and (b) corresponding junction tree.

In a junction tree, exact inference proceeds as follows: Assuming evidence is $E = \{A_i = a\}$ and $A_i \in \mathcal{C}_y$, E is *absorbed* at \mathcal{C}_y by instantiating the variable A_i and renormalizing the remaining variables of the clique. The evidence is then propagated from \mathcal{C}_y to any adjacent cliques \mathcal{C}_x . Let ψ_y^* denote the potential table of \mathcal{C}_y after E is absorbed, and ψ_x the potential table of \mathcal{C}_x . Mathematically, evidence propagation is represented as [1]:

$$\psi_S^* = \sum_{\mathcal{Y} \setminus \mathcal{S}} \psi_{\mathcal{Y}}^*, \quad \psi_{\mathcal{X}}^* = \psi_{\mathcal{X}} \frac{\psi_{\mathcal{S}}^*}{\psi_{\mathcal{S}}} \quad (1)$$

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 $\mathcal{C}_{\mathcal{X}}$.

3 Related Work

There are several works on parallel exact inference, such as Pennock [5], Kozlov and Singh [7] and Szolovits. However, some of those methods, such as [7], are dependent upon the structure of the Bayesian network. The performance is adversely affected if the structure of the input Bayesian network is changed. Our method can be used for Bayesian networks and junction trees with various structures. Some other methods, such as [5], exhibit limited performance for multiple evidence inputs. The performance of our method does not depend on the number of evidence cliques. In [8], the authors discuss the structure conversion of Bayesian networks, which is different from evidence propagation addressed in this paper. In [9], the node level primitives are parallelized using message passing on distributed memory platforms. The optimization proposed in [9] is not applicable in this paper, since the multicore platforms have shared memory. However, the idea of parallelization of node level primitives is adapted by our scheduler to partition large tasks. A junction tree decomposition method is provided in [10] to partition junction trees for distributed memory platforms. This method reduces communication between processors by duplicating some cliques. We do not apply junction tree decomposition on our multicore platforms, because the clique duplication consumes memory that is shared by all the cores. A centralized scheduler for exact inference is introduced in [11], which is implemented on Cell BE, a heterogeneous multicore processor with a PowerPC element and 8 computing elements. However, the multicore platforms studied in this paper are homogeneous, and the number of cores is small. Using a separate core for centralized scheduling leads to performance loss. We deviate from the above approaches and explore collaborative task scheduling techniques for exact inference.

4 Junction Tree Rerooting For Minimizing Critical Path

A junction tree can be rerooted at any clique [5]. Consider rerooting a junction tree at clique \mathcal{C} . Let α be a preorder walk of the underlying undirected tree, starting from \mathcal{C} . Then, α encodes the desired new edge directions, i.e. an edge in the rerooted tree points from \mathcal{C}_{α_i} to \mathcal{C}_{α_j} if and only if $\alpha_i < \alpha_j$. In the rerooting procedure, we check the edges in the given junction tree and reverse any edges inconsistent with α . The result is a new junction tree rooted at \mathcal{C} , with the same underlying undirected topology as the original tree.

Rerooting a junction tree can lead to acceleration of evidence propagation on parallel computing systems. Let $P(\mathcal{C}_i, \mathcal{C}_j) = \mathcal{C}_i, \mathcal{C}_{i_1}, \mathcal{C}_{i_2}, \dots, \mathcal{C}_j$ denote a path from \mathcal{C}_i to \mathcal{C}_j in a junction tree, and $L_{(\mathcal{C}_i, \mathcal{C}_j)}$ denote the weight of path $P(\mathcal{C}_i, \mathcal{C}_j)$. Given clique width $w_{\mathcal{C}_t}$ and clique degree k_t for any clique $\mathcal{C}_t \in P(\mathcal{C}_i, \mathcal{C}_j)$, the

weight of the path $L_{(\mathcal{C}_i, \mathcal{C}_j)}$ is given by:

$$L_{(\mathcal{C}_i, \mathcal{C}_j)} = \sum_{\mathcal{C}_t \in P(\mathcal{C}_r, \mathcal{C}_j)} k_t w_{\mathcal{C}_t} \prod_{l=1}^{w_{\mathcal{C}_t}} r_l \quad (2)$$

The *critical path* (CP) of a junction tree is defined as the longest weighted path in the junction tree. Given a junction tree \mathbb{J} , the weight of a critical path, denoted L_{CP} , is given by $L_{CP} = \max_{\mathcal{C}_j \in \mathbb{J}} L_{(\mathcal{C}_r, \mathcal{C}_j)}$, where \mathcal{C}_r is the root. Notice that evidence propagation in a critical path takes at least as much time as that in other paths. Thus, among the rerooted junction trees, the one with the minimum critical path leads to the best performance on parallel computing platforms.

A straightforward approach to find the optimal rerooted tree is as follows: First, reroot the junction tree at each clique. Then, for each rerooted tree, calculate the complexity of the critical path. Finally, select the rerooted tree corresponding to the minimum complexity of the critical path. Given the number of cliques N and maximum clique width $w_{\mathcal{C}}$, the computational complexity of the above procedure is $O(N^2 w_{\mathcal{C}})$.

We present an efficient rerooting method to minimize the critical path (see Algorithm 1), which is based on the following lemma:

Lemma 1: Suppose that $P(\mathcal{C}_x, \mathcal{C}_y)$ is the longest weighted path from a leaf clique \mathcal{C}_x to another leaf clique \mathcal{C}_y in a given junction tree, and $L_{(\mathcal{C}_r, \mathcal{C}_x)} \geq L_{(\mathcal{C}_r, \mathcal{C}_y)}$, where \mathcal{C}_r is the root. Then, $P(\mathcal{C}_r, \mathcal{C}_x)$ is a critical path in the given junction tree.

Proof sketch: Assume a critical path is $P(\mathcal{C}_r, \mathcal{C}_z)$ where $\mathcal{C}_z \neq \mathcal{C}_x$. Let $P(\mathcal{C}_r, \mathcal{C}_{b1})$ denote the longest common segment between $P(\mathcal{C}_r, \mathcal{C}_x)$ and $P(\mathcal{C}_r, \mathcal{C}_y)$, and $P(\mathcal{C}_r, \mathcal{C}_{b2})$ the longest common segment between $P(\mathcal{C}_r, \mathcal{C}_x)$ and $P(\mathcal{C}_r, \mathcal{C}_z)$. Without loss of generality, assume $\mathcal{C}_{b2} \in P(\mathcal{C}_r, \mathcal{C}_{b1})$. Since $P(\mathcal{C}_r, \mathcal{C}_z)$ is a critical path, we have $L_{(\mathcal{C}_r, \mathcal{C}_z)} \geq L_{(\mathcal{C}_r, \mathcal{C}_x)}$. Because $L_{(\mathcal{C}_r, \mathcal{C}_z)} = L_{(\mathcal{C}_r, \mathcal{C}_{b2})} + L_{(\mathcal{C}_{b2}, \mathcal{C}_z)}$ and $L_{(\mathcal{C}_r, \mathcal{C}_x)} = L_{(\mathcal{C}_r, \mathcal{C}_{b2})} + L_{(\mathcal{C}_{b2}, \mathcal{C}_{b1})} + L_{(\mathcal{C}_{b1}, \mathcal{C}_x)}$, therefore, $L_{(\mathcal{C}_{b2}, \mathcal{C}_z)} \geq L_{(\mathcal{C}_{b2}, \mathcal{C}_{b1})} + L_{(\mathcal{C}_{b1}, \mathcal{C}_x)} > L_{(\mathcal{C}_{b1}, \mathcal{C}_x)}$. Thus, we can find path $P(\mathcal{C}_z, \mathcal{C}_y) = P(\mathcal{C}_z, \mathcal{C}_{b2})P(\mathcal{C}_{b2}, \mathcal{C}_{b1})P(\mathcal{C}_{b1}, \mathcal{C}_y)$ which leads to:

$$\begin{aligned} L_{(\mathcal{C}_z, \mathcal{C}_y)} &= L_{(\mathcal{C}_z, \mathcal{C}_{b2})} + L_{(\mathcal{C}_{b2}, \mathcal{C}_{b1})} + L_{(\mathcal{C}_{b1}, \mathcal{C}_y)} > L_{(\mathcal{C}_{b1}, \mathcal{C}_x)} + L_{(\mathcal{C}_{b2}, \mathcal{C}_{b1})} + L_{(\mathcal{C}_{b1}, \mathcal{C}_y)} \\ &> L_{(\mathcal{C}_x, \mathcal{C}_{b1})} + L_{(\mathcal{C}_{b1}, \mathcal{C}_y)} = L_{(\mathcal{C}_x, \mathcal{C}_y)} \end{aligned} \quad (3)$$

The above inequality contradicts the assumption that $P(\mathcal{C}_x, \mathcal{C}_y)$ is a longest weighted path in the given junction tree. \square

According to Lemma 1, the new root can be found once we identify the longest weighted path between two leaves in the given junction tree. We introduce a tuple $\langle v_i, p_i, q_i \rangle$ for each clique \mathcal{C}_i to find the longest weighted path (Lines 1-6 Algorithm 1), where v_i records the complexity of a critical path of the subtree rooted at \mathcal{C}_i ; p_i and q_i represent \mathcal{C}_{p_i} and \mathcal{C}_{q_i} , respectively, which are two children of \mathcal{C}_i . If \mathcal{C}_i has no child, p_i and q_i are empty.

The path from \mathcal{C}_{p_i} to some leaf clique in the subtree rooted at \mathcal{C}_i is the *longest* weighted path among all paths from a child of \mathcal{C}_i to a leaf clique, while the path from \mathcal{C}_{q_i} to a certain leaf clique in the subtree rooted at \mathcal{C}_i is the *second longest* weighted path. The two paths are concatenated at \mathcal{C}_i and form a leaf-to-leaf path

Algorithm 1 Root selection for minimizing critical path**Input:** Junction tree J **Output:** New root \mathcal{C}_r

```

1: initialize a tuple  $\langle v_i, p_i, q_i \rangle = \langle k_i w_{\mathcal{C}_i} \prod_{j=1}^{w_{\mathcal{C}_i}} r_j, 0, 0 \rangle$  for each  $\mathcal{C}_i$  in  $J$ 
2: for  $i = N$  downto 1 do
3:    $p_i = \arg_j \max(v_j), \forall pa(\mathcal{C}_j) = \mathcal{C}_i$ 
4:    $q_i = \arg_j \max(v_j), \forall pa(\mathcal{C}_j) = \mathcal{C}_i$  and  $j \neq p_i$ 
5:    $v_i = v_i + v_{p_i}$ 
6: end for
7: select  $\mathcal{C}_m$  where  $m = \arg_i \max(v_i + v_{q_i}), \forall i$ 
8: initialize path  $P = \{\mathcal{C}_m\}; i = m$ 
9: while  $\mathcal{C}_i$  is not a leaf clique do
10:   $i = p_i; P = \{\mathcal{C}_i\} \cup P$ 
11: end while
12:  $P = P \cup \mathcal{C}_{q_m}; i = m$ 
13: while  $\mathcal{C}_i$  is not a leaf node do
14:   $i = p_i; P = P \cup \{\mathcal{C}_i\}$ 
15: end while
16: denote  $\mathcal{C}_x$  and  $\mathcal{C}_y$  the two end cliques of path  $P$ 
17: select new root  $\mathcal{C}_r = \arg_i \min |L_{(\mathcal{C}_x, \mathcal{C}_i)} - L_{(\mathcal{C}_i, \mathcal{C}_y)}| \forall \mathcal{C}_i \in P(\mathcal{C}_x, \mathcal{C}_y)$ 

```

in the original junction tree. In Lines 3 and 4, $\arg_j \max(v_j)$ stands for the value of the given argument (parameter) j for which the value of the given expression v_j attains its maximum value. In Line 7, we detect a clique \mathcal{C}_m on the longest weighted path and identify the path in Lines 8-15 accordingly. The new root is then selected in Line 17.

We briefly analyze the serial complexity of Algorithm 1. Line 1 take $w_{\mathcal{C}}N$ computation time for initialization, where $w_{\mathcal{C}}$ is clique width and N is the number of cliques. The loop in Line 2 has N iterations and both Lines 3 and 4 take $O(k)$ time, where k is the maximum number of children of a clique. Line 7 takes $O(N)$ time, as do Lines 8-15, since a path consists of at most N cliques. Lines 16-17 can be completed in $O(N)$ time. Since $k < w_{\mathcal{C}}$, the complexity of Algorithm 1 is $O(w_{\mathcal{C}}N)$, compared to $O(w_{\mathcal{C}}N^2)$, the complexity of the straightforward approach.

5 Task Definition and Dependency Graph Construction

5.1 Task Definition

Evidence propagation consists of a series of computations called node level primitives. There are four types of node level primitives: *marginalization*, *extension*, *multiplication* and *division* [9]. In this paper, we define a *task* as the computation of a node level primitive. The input to each task is one or two potential tables, depending on the specific primitive type. The output is an updated potential table. The details of the primitives are discussed in [9]. We illustrate the tasks related to clique \mathcal{C} in Figure 2 (b). Each number in brackets corresponds to a task

of which the primitive type is given in Figure 2 (c). The dashed arrows in Figure 2 (b) illustrate whether the task works on the same potential table or between two potential tables. The edge in Figure 2 (c) represent precedence order of the execution of the tasks.

A property of the primitives is that the potential table of a clique can be partitioned into independent activities and processed in parallel. The results from each activity are combined (for extension, multiplication and division) or added (for marginalization) to obtain the final output. This property is utilized in Section 6.

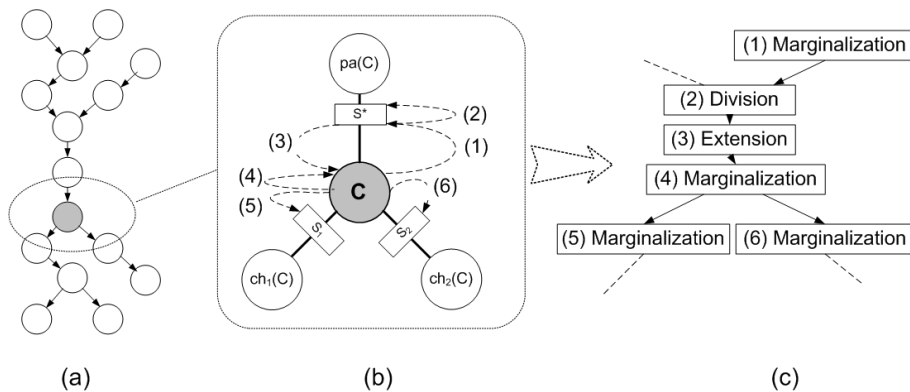


Fig. 2. (a) Clique updating graph; (b) Primitives used to update a clique; (c) Local task dependency graph with respect to the clique in (b).

5.2 Dependency Graph Construction

Given an arbitrary junction tree, we reroot it according to Section 4. The resulting tree is denoted \mathbb{J} . We construct a *task dependency graph* G from \mathbb{J} to describe the precedence constraints among the tasks. The task dependency graph is created in the following two steps:

First, we construct a *clique updating graph* to describe the coarse grained dependency relationship between cliques in \mathbb{J} . In exact inference, \mathbb{J} is updated twice [1]: (1) evidence is propagated from leaf cliques to the root; (2) evidence is then propagated from the root to the leaf cliques. Thus, the clique updating graph has two symmetric parts. In the first part, each clique depends on all its children in \mathbb{J} . In the second part, each clique depends on its parent in \mathbb{J} . Figure 2 (a) shows a sample clique updating graph from the junction tree given in Figure 1 (b).

Second, based on the clique updating graph, we construct *task dependency graph* G to describe the fine grained dependency relationship between the tasks defined in Section 5.1. The tasks related to a clique C are shown in Figure 2 (b).

Considering the precedence order of the tasks, we obtain a small DAG called a *local task dependency graph* (see Figure 2 (c)). Replacing each clique in Figure 2 (a) with its corresponding local task dependency graph, we obtain the task dependency graph G for junction tree \mathbb{J} .

6 Collaborative Scheduling

We propose a *collaborative scheduler* to allocate the tasks in the task dependency graph G to the cores. We assume that there are P cores in a system. The framework of the scheduler is shown in Figure 3. The *global task list (GL)* in Figure 3 stores the tasks from the task dependency graph. Each entry of the list stores a task and the related data, such as the task size, the task dependency degree, and the links to its succeeding tasks. Initially, the *dependency degree* of a task is the number of incoming edges of the task in G . Only the tasks with dependency degree equal to 0 can be processed. The global task list is shared by all the threads, so any thread can fetch a task, append new tasks, or decrease the dependency degree of tasks. Before an entry of the list is accessed by a thread, all the data in the entry must be protected by a lock to avoid concurrent write.

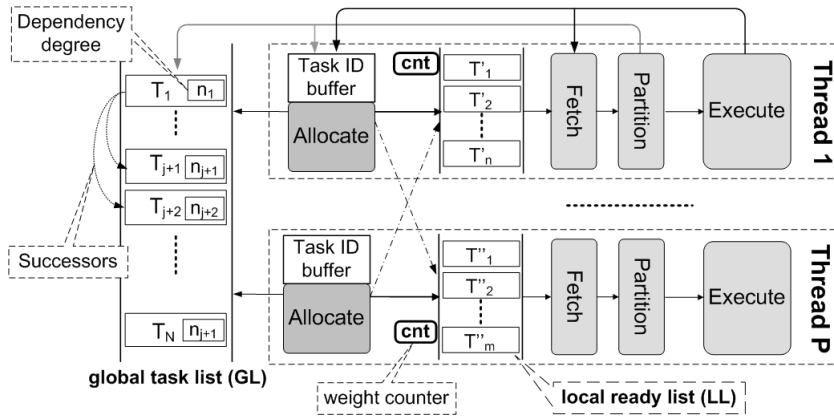


Fig. 3. Components of the collaborative scheduler.

Every thread has an *Allocate module* which is in charge of decreasing task dependency degrees and allocating tasks to the threads. The module only decreases the dependency degree of the tasks if their predecessors appear in the *Task ID buffer* (see Figure 3). The task ID corresponds to the offset of the task in the GL, so the module can find the task given the ID in $O(1)$ time. If the dependency degree of a task becomes 0 after the decrease operation, the module allocates it to a thread with the aim of load balancing across threads. Various heuristics can be used to balance the workload. In this paper, we allocate a task to the thread with the smallest workload.

Each thread has a *local ready list* (LL) to store the tasks allocated to the thread. All the tasks in a LL are processed by the same thread. However, since the tasks in the LL can be allocated by all the Allocate modules, the LLs are actually global. Thus, locks are used to prevent concurrent write to LL. Each LL has a *weight counter* to record the workload of the tasks in the LL. Once a new task is inserted to (fetched from) the list, the workload of the task is added to (subtracted from) the weight counter.

The *Fetch module* takes tasks from the LL in the same thread. Heuristics can be used to select tasks from the LL. In this paper, we use a straightforward method where the task at the head of the LL is fetched.

The *Partition module* checks the workload of the fetched task. The tasks with heavy workload are partitioned for load balancing. As we discussed in Section 5.1, a property of the primitives is that the potential table of a clique can be partitioned easily. Thus, a task T can be partitioned to subtasks $\hat{T}_1, \hat{T}_2, \dots, \hat{T}_n$, each processing a part of the potential table related to task T . Each subtask inherits the parents of T in the task dependency graph G . However, we let \hat{T}_n be the successor of $\hat{T}_1, \dots, \hat{T}_{n-1}$, and only \hat{T}_n inherits the successor of T . Therefore, the results from the subtasks can be concatenated or added by \hat{T}_n . The Partition module preserves the structure of G , except replacing T by the subtasks. The module replaces T in the GL with \hat{T}_n , and appends other subtasks to the GL. \hat{T}_1 is sent to the Execute module and $\hat{T}_2, \dots, \hat{T}_{n-1}$ are evenly distributed to local lists, so that these subtasks can be executed by several threads.

Each thread also has a local *Execute module* where the primitive related to a task is performed. Once the primitive is completed, the Execute module sends the ID of the task to the Task ID buffer, so that the Allocate module can accordingly decrease the dependency degree of the successors of the task. The Execute module also signals the Fetch module to take the next task, if any, from LL.

The collaborative scheduling algorithm is shown in Algorithm 2. We use the following notations in the algorithm: GL is the global list. LL_i is the local ready list in Thread i . d_T and w_T denote the dependency degree and the weight of task T , respectively. W_i is the total weight of the tasks in LL_i . δ is the threshold of the size of potential table. Any potential table larger than δ is partitioned. Line 1 in Algorithm 2 initializes the Task ID buffers. As shown in Line 3, the scheduler keeps on working until all tasks are processed. Lines 4-10 correspond to the Allocate module. Line 11 is the Fetch module. Lines 12-18 correspond to the Partition module and Execute Module.

Algorithm 2 achieves load balancing by two means: First, the Allocate module ensures that the new tasks are allocated to the threads where the total workload of the tasks in its LL is the lowest. Second, the Partition module guarantees that each single large task can be processed in parallel.

7 Experiments

Algorithm 2 Collaborative Task Scheduling

```

1:  $\forall T$  s.t.  $d_T = 0$ , evenly distribute the ID of  $T$  to Task ID buffers
2: for Thread  $i$  ( $i = 1 \dots P$ ) in parallel do
3:   while  $GLULL_i \neq \emptyset$  do
4:     for  $T \in \{ \text{successors of tasks in the } i\text{-th Task ID buffer} \}$  do
5:        $d_T = d_T - 1$ 
6:       if  $d_T = 0$  then
7:         allocate  $T$  to  $LL_j$  where  $j = \arg_t \min(W_t), t = 1 \dots P$ 
8:          $W_k = W_k + w_T$ 
9:       end if
10:    end for
11:    fetch task  $T'$  from  $LL_i$ 
12:    if the size of potential table  $\psi_{T'} > \delta$  then
13:      partition  $T'$  into subtasks  $\hat{T}'_1, \hat{T}'_2, \dots, \hat{T}'_n$  s.t.  $\psi_{\hat{T}'_j} \leq \delta, j = 1, \dots, n$ 
14:      replace  $T'$  in GL with  $\hat{T}'_n$ , and allocate  $\hat{T}'_1, \dots, \hat{T}'_{n-1}$  to local lists
15:      execute  $\hat{T}'_1$  and place the task ID of  $\hat{T}'_1$  into the  $i$ -th Task ID buffer
16:    else
17:      execute  $T'$  and place the task ID of  $T'$  into the  $i$ -th Task ID buffer
18:    end if
19:  end while
20: end for

```

We conducted experiments on two state-of-the-art homogeneous multi-core processor systems: Intel Xeon quadcore and AMD Opteron quadcore system. The former contained two Intel Xeon x86.64 E5335 processors, each having four cores. The processors ran at 2.00 GHz with 4 MB cache and 16 GB memory. The operating system was Red Hat Enterprise Linux WS release 4 (Nahant Update 7). We installed GCC version 4.1.2 compiler and Intel C/C++ Compiler (ICC) version 10.0 with streaming SIMD extensions 3 (SSE 3), also known as Prescott New Instructions (PNI). The latter platform had two AMD Opteron 2347 quadcore processors, running at 1.9 GHz. The system had 16 GB DDR2 memory and the operating system was Red Hat Linux CentOS version 5. We also used GCC 4.1.2 compiler on the AMD platform.

To evaluate the performance of the junction tree rerooting method shown in Algorithm 1, we generated four junction trees using the template shown in Figure 4. The template in Figure 4 is a tree with $b + 1$ branches. R is the root. Using $b = 1, 2, 4, 8$, we obtained four junction trees. Every junction tree had 512

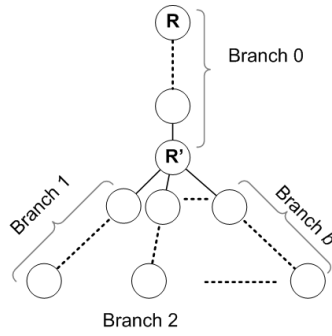


Fig. 4. Junction tree template for evaluating rerooting algorithm.

cliques, each consisting of 15 binary variables. Thus, the serial complexity of each Branch is approximately equal. Using Algorithm 1, clique R' became the new root after rerooting. For each junction tree, we performed evidence propagation on both the original tree and the rerooted tree, using various number of cores. We disabled task partitioning, which provided parallelism at fine grained level.

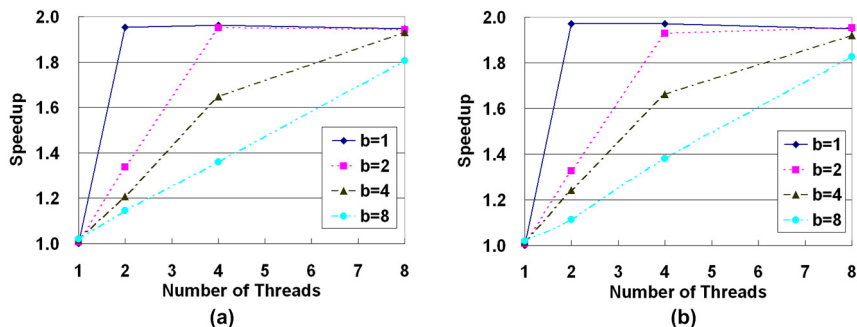


Fig. 5. Speedup caused by rerooting on (a) Intel Xeon and (b) AMD Opteron. $(b + 1)$ is the total number of branches in the junction tree.

The results are shown in Figure 5. The speedup in Figure 5 was defined as $Sp = t_R/t_{R'}$, where t_R ($t_{R'}$) is the execution time of evidence propagation in the original (rerooted) junction tree. According to Section 4, we know that when clique R is the root, Branch 0 plus Branch 1 is a critical path. When R' is the root, only Branch 0 is the critical path. Thus, the maximum speedup is 2 for the four junction trees, if the number of concurrent threads P is larger than b . When $P < b$, Sp was less than 2, since some cliques without precedence constraint can not be processed in parallel. From the results in Figure 5, we can see that the rerooted tree led to speedup around 1.9, when 8 cores were used. In addition, the maximum speedup was achieved using more threads as b increases. These observations matched the analysis above. Notice that some speedup curves fell slightly when 8 concurrent threads were used. This was caused by the overheads such as the lock contention.

We also observed that, compared with the time for evidence propagation, the percentage of overall execution time spent on junction tree rerooting was very small. Rerooting the junction tree with 512 cliques took 24 microseconds on the AMD Opteron quadcore system, compared to 2.8×10^7 microseconds for the overall execution time. Thus, although Algorithm 1 was not parallelized, it causes negligible overhead in parallel evidence propagation.

We generated junction trees of various sizes to analyze and evaluate the performance of the proposed evidence propagation method. The junction trees were generated using Bayes Net Toolbox [12]. The first junction tree (Junction tree 1) had 512 cliques and the average clique width was 20. The average degree for each clique was 4. All random variables were binary. The second junction

tree (Junction tree 2) had 256 cliques and the average clique width was 15. The number of states of random variables was 3 and the average clique degree was 4. The third junction tree (Junction tree 3) had 128 cliques. The clique width was 10 and the number of states of random variables was 3. The average clique degree was 2. All the three junction trees were rerooted using Algorithm 1. In our experiments, we used double precision floating point numbers to represent the probabilities and potentials.

We performed exact inference with respect to the above three junction trees using Intel Open Source Probabilistic Network Library (PNL) [13]. The scalability of the results is shown in Figure 6. PNL is a full function, free, open source, graphical model library released under Berkeley Software Distribution (BSD) style license. PNL provides an implementation for junction tree inference with discrete parameters. The parallel version of PNL is also provided by Intel [13]. The results shown in Figure 6 were obtained on a IBM P655 multiprocessor system, where each processor runs at 1.5 GHz with 2 GB of memory. We can see from Figure 6 that, for all the three junction trees, the execution time increased when the number of processors was greater than 4.

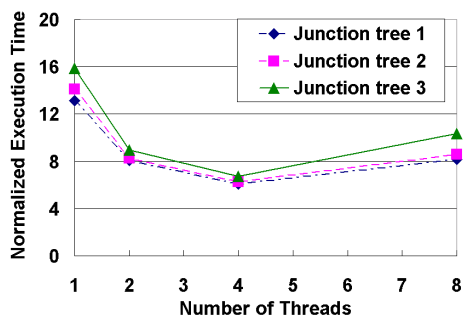


Fig. 6. Scalability of exact inference using PNL library for various junction trees.

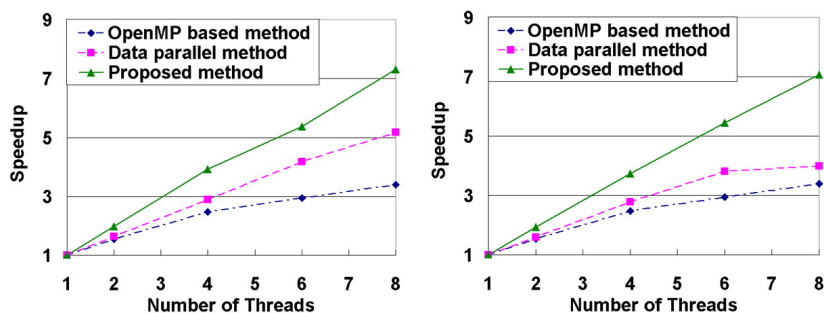


Fig. 7. Scalability of exact inference using various methods on (a) Intel Xeon and (b) AMD Opteron.

We compared three parallel methods for evidence propagation on both Intel Xeon and AMD Opteron in Figure 7. The first two methods were the baselines. The first parallel method was the *OpenMP based method*, where the OpenMP intrinsic functions were used to parallelize the sequential code. We used ICC to compile the code on Xeon, while GCC was used on Opteron. The second method is called *data parallel method*, where we created multiple threads for each node level primitive. That is, the node level primitives were parallelized every time they were performed. The data parallel method is similar to the task partitioning mechanism in our collaborative scheduler, but the overheads were large. The third parallel method was the proposed method. Using Junction trees 1-3 introduced above, we conducted experiments on both the platforms. For all the three methods, we used level 3 optimization (`-O3`). The OpenMP based method also benefited from the SSE3 optimization (`-msse3`). We show the speedups in Figure 7. The results show that the proposed method exhibited linear speedup and was superior compared with the baseline methods on both the platforms. Performing the proposed method on 8 cores, we observed speedup of 7.4 on Intel Xeon and 7.1 on AMD Opteron. Compared to the OpenMP based method, our approach achieved speedup of 2.1 when 8 cores were used. Compared to the data parallel method, we achieved speedup of 1.8 on AMD Opteron.

To show the load balance we achieved and the overhead of the collaborative scheduler, we measured the computation time for each thread. In our context, the computation time for a thread is the total time taken by the thread to perform node level primitives. Thus, the time taken to fetch tasks, allocate tasks and maintain the local ready list were not considered. The computation time reflects the workload for each thread. We show the results in Figure 8 (a), which were obtained on Opteron using Junction tree 1 defined above. We observed very similar results for Junction tree 2 and 3. Due to space constraints, we show results on Junction tree 1 only. We also calculated the ratio of the computation time over the total parallel execution time. This ratio illustrates the quality of the scheduler. From Figure 8 (b), we can see that, although the scheduling time increased a little as the number of threads increases, it was not exceeding 0.9% of the execution time for all the threads.

Finally, we modified parameters of Junction tree 1 to obtain a dozen junction trees. We applied the proposed method on these junction trees to observe the performance of our method in various situations. We varied the number of cliques N , clique width w_C , number of states r and average number of children k . We obtained almost linear speedup for all cases. From the results in Figure 9 (a), we observe that the speedups achieved in the experiments with various values for N were all above 7. All of them exhibited linear speedups. In Figure 9 (b) and (c), all results showed linear speedup except the ones with $w_C = 10$ and $r = 2$. The reason was that the size of the potential table was small. For $w_C = 10$ and $r = 2$, the potential table had 1024 entries, about 1/1000 of the number of entries in a potential table with $w_C = 20$. However, since N and the junction tree structure were the same, the scheduling requires approximately the same time for junction tree with small potential tables. Thus, the overheads became relatively large. In

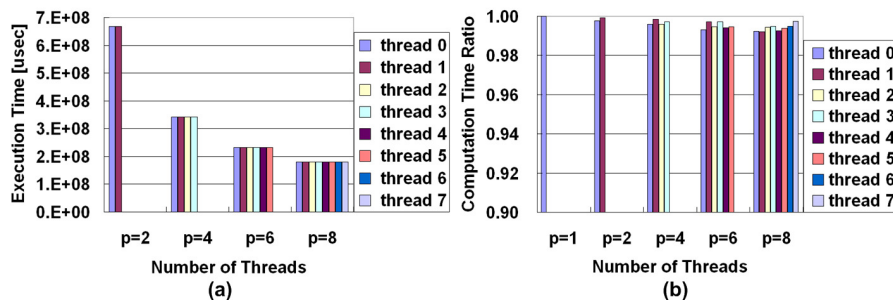


Fig. 8. (a) Load balance across the threads; (b) Computation time ratio for each thread.

Figure 9 (d), all the results had similar performance when k was varied. All of them achieved speedups of more than 7 using 8 cores.

8 Conclusions

We presented an efficient rerooting algorithm and a collaborative scheduling algorithm for parallel evidence propagation. The proposed method exploited both task and data parallelism in evidence propagation. Thus, even though one of the levels can not provide enough parallelism, the proposed method still achieves speedup on parallel platforms. Our implementation achieved $7.4\times$ speedup using 8 cores. This speedup is much higher compared with the baseline methods, such as the OpenMP based implementation. In the future, we plan to investigate the overheads in the collaborative scheduler and further improve its performance. As more cores are integrated into a single chip, some overheads such as lock contention will increase dramatically. We intend to improve the design of the collaborative scheduler to reduce such overheads, so that the scheduler can be used for a class of DAG structured computations in the many-core era.

References

1. Lauritzen, S.L., Spiegelhalter, D.J.: Local computation with probabilities and graphical structures and their application to expert systems. *J. Royal Statistical Society B* **50** 157–224 (1988)
2. Heckerman, D.: Bayesian networks for data mining. In: *Data Mining and Knowledge Discovery*. (1997)
3. Russell, S.J., Norvig, P.: *Artificial Intelligence: A Modern Approach* (2nd Edition). Prentice Hall (2002)
4. Segal, E., Taskar, B., Gasch, A., Friedman, N., Koller, D.: Rich probabilistic models for gene expression. In: *9th International Conference on Intelligent Systems for Molecular Biology*. 243–252 (2001)
5. Pennock, D.: Logarithmic time parallel Bayesian inference. In: *Proceedings of the 14th Annual Conference on Uncertainty in Artificial Intelligence*. 431–438 (1998)

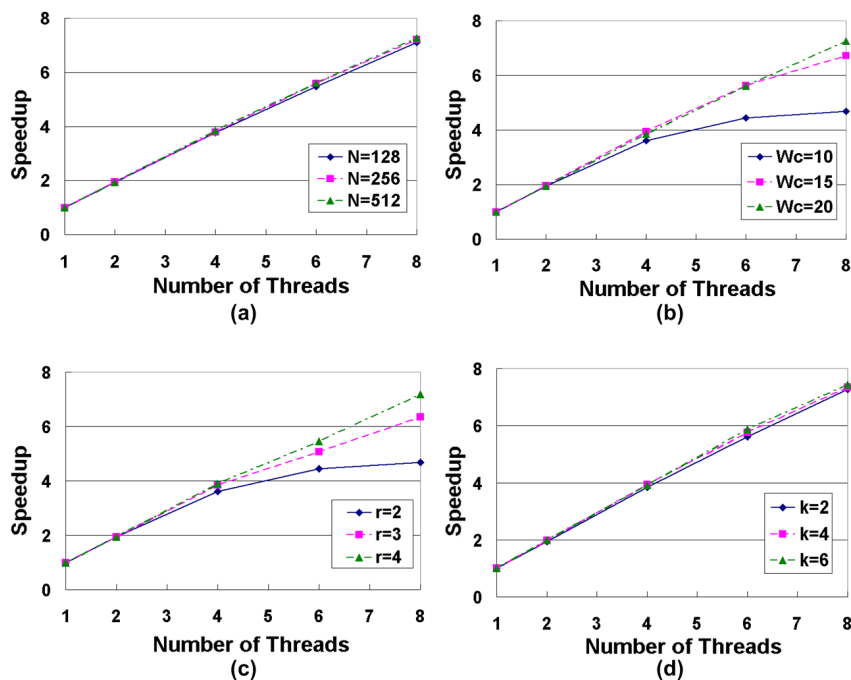


Fig. 9. Speedups of exact inference on multicore systems with respect to various junction tree parameters.

6. De Vuyst, M., Kumar, R., Tullsen, D.: Exploiting unbalanced thread scheduling for energy and performance on a cmp of smt processors. In: IEEE International Symposium on Parallel and Distributed Processing (IPDPS). 1–6 (2006)
7. Kozlov, A.V., Singh, J.P.: A parallel Lauritzen-Spiegelhalter algorithm for probabilistic inference. In: Supercomputing. 320–329 (1994)
8. Xia, Y., Prasanna, V.K.: Parallel exact inference. In: Proceedings of the Parallel Computing. 185–192 (2007)
9. Xia, Y., Prasanna, V.K.: Node level primitives for parallel exact inference. In: Proceedings of the 19th International Symposium on Computer Architecture and High Performance Computing. 221–228 (2007)
10. Xia, Y., Prasanna, V.K.: Junction tree decomposition for parallel exact inference. In: IEEE International Symposium on Parallel and Distributed Processing (IPDPS). 1–12 (2008)
11. Xia, Y., Prasanna, V.K.: Parallel exact inference on the cell broadband engine processor. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC). 1–12 (2008)
12. Murphy, K.: (<http://www.cs.ubc.ca/~murphyk/software/bnt/> bnt.html)
13. Intel Open Source Probabilistic Networks Library: (<http://www.intel.com/technology/computing/pnl/>)