Parallel Exact Inference on Multicore Using MapReduce

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

Yinglong Xia
IBM T.J. Watson Research Center
Yorktown Heights, NY 10598
Email: yxia@us.ibm.com

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

***Abstract*—Inference is a key problem in exploring probabilistic graphical models for machine learning algorithms. Recently, many parallel techniques have been developed to accelerate inference. However, these techniques are not widely used due to their implementation complexity. MapReduce provides an appealing programming model that has been increasingly used to develop parallel solutions. MapReduce though has been mainly used for data parallel applications. In this paper, we investigate the use of MapReduce for exact inference in Bayesian networks. MapReduce based algorithms are proposed for evidence propagation in junction trees. We evaluate our methods on general-purpose multi-core machines using Phoenix as the underlying MapReduce runtime. The experimental results show that our methods achieve $20\times$ speedup on an Intel Westmere-EX based system.**

***Index Terms*—exact inference; Bayesian networks; MapReduce; multi-core**

I. Introduction

A full joint probability distribution can be used to model a real-world system. 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 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].

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 to the network. 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 was proposed by Lauritzen and Spiegelhalter [5]. It converts a Bayesian network into a *junction tree*, then performs *evidence propagation* 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 [5].

Several parallel techniques have been developed to accelerate exact inference. In [6] [7], the authors exploit *task parallelism* by parallelizing independent computations across cliques in the network. In [6] [8] [9], the authors exploit *data parallelism* by parallelizing independent computations within a node. Although these techniques exhibit good speedup, the implementation complexity has limited the applicability of these especially in application domains where the experts may not be aware of parallel computing techniques and tools.

In recent years, MapReduce [10] has emerged as a programming model for large-scale data processing. It allows for easy parallelism with well-known supporting runtimes such as Hadoop, Twister for clusters or Phoenix for multicore [11], [12], [13]. Many applications with data parallelism have gained good scalability using MapReduce [14], [15], [16], [17]. However, there are a very limited number of implementations using MapReduce for applications with data dependency constraints such as exact inference.

In this paper, we investigate the use of MapReduce for exact inference in Bayesian networks on multicore platforms. Specifically, our contributions include:

- We propose algorithms for evidence propagation in junction trees using MapReduce. Our methods explore task parallelism and resolve data dependency constraints based on tree traversal methods.
- We implement the algorithms on state-of-the-art multicore machines using Phoenix as the underlying MapReduce runtime. Our methods demonstrate a framework for using MapReduce to parallelize applications with data dependencies using existing runtimes.
- We conduct experiments with various datasets. Our experimental results show $20\times$ speedup on a 40-core Intel Westmere-EX based system.

The rest of the paper is organized as follows. In Section II, we review exact inference and the MapReduce programming model. Section III discusses related work. Section IV presents our proposed algorithms for exact inference in junction trees using MapReduce. Experimental setup and results are presented in Section V. Section VI concludes the paper.

This research was partially supported by the U.S. National Science Foundation under grant number CNS-1018801. Support from the Intel Manycore Testing Lab is gratefully acknowledged.

II. Background

A. Exact Inference in Bayesian Networks

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 $\psi_{\mathcal{C}_i}$ is r^w.

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 an adjacent clique $\mathcal{C}_\mathcal{X}$. 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]:

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

$$\psi^*_\mathcal{X} = \psi_\mathcal{X} \frac{\psi^*_\mathcal{S}}{\psi_\mathcal{S}} \tag{2}$$

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 $\mathcal{C}_\mathcal{X}$. Hence, propagating evidence from $\mathcal{C}_\mathcal{Y}$ to $\mathcal{C}_\mathcal{X}$ includes computing the marginal $\psi_\mathcal{S}$ from $\psi_\mathcal{Y}$ with Eq. 1 and then updating $\psi_\mathcal{X}$ from $\psi_\mathcal{S}$ with Eq. 2.

A two-stage method ensures that evidence at any cliques in a junction tree can be propagated to all the other cliques [5]. The first stage is called *evidence collection*, where evidence is propagated from leaf cliques to the root; the second stage is called *evidence distribution*, where the evidence is propagated from the root to leaf cliques. Figure 2 illustrates the first three steps of evidence collection and distribution in a sample junction tree.

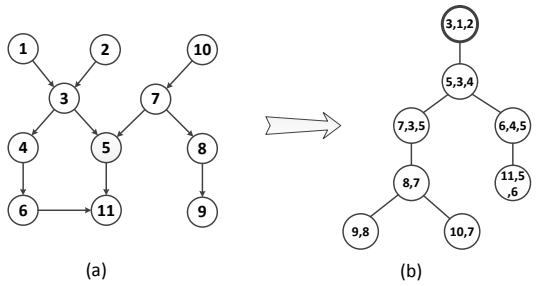


Fig. 1. (a) A sample Bayesian network and (b) corresponding junction tree.

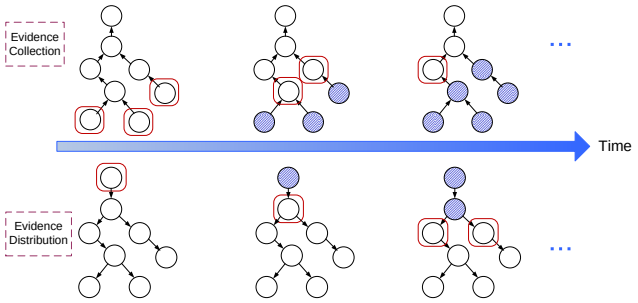


Fig. 2. Illustration of evidence collection and evidence distribution in a junction tree. The cliques in boxes are under processing. The shaded cliques have been processed.

B. The MapReduce Programming Model

MapReduce is a parallel programming model inspired from the functional programming concepts and proposed for data-intensive applications [10]. A MapReduce based computation takes a set of input `<key,value>` pairs, and produces a set of output `<key,value>` pairs. The user of MapReduce expresses the computation using two functions: *Map* and *Reduce*.

- The Map function takes an input pair, performs computation on it, and produces a set of intermediate `<key,value>` pairs. All input pairs can be processed by the Map function simultaneously in the Map phase. At the end of the Map phase, all intermediate values associated with the same intermediate key are grouped together and passed to the Reduce function.
- The Reduce function receives an intermediate key and a set of values associated with that key. These values are merged with some aggregation operations to form a possibly smaller set of values. All intermediate keys and the associated sets of values can also be processed by the Reduce function simultaneously.

Simplicity is the main benefit of the MapReduce programming model. The user only needs to provide a simple description of the algorithm using Map and Reduce functions and leave all parallelization and concurrency control to the runtime system. The MapReduce runtime system is responsible of distributing the work to the available processing units.

In many applications, there must be a sequence of MapReduce invocations during the entire execution. The term Iterative

MapReduce [12] is used to refer to the repetitive invocations of MapReduce.

III. Related Work

There are several works on parallel exact inference in graphical models. In [6], *structural parallelism* in exact inference is explored by assigning independent cliques of the junction tree to separate processors for concurrent processing. In [7], structural parallelism is also exploited by a dynamic task scheduling method, where the evidence propagation in each clique is viewed as a task. In [8], [9], *data parallelism* is explored in node level computations by updating parts of a potential table in parallel. These techniques require complex implementation and optimization using low-level programming models such as Pthreads or MPI.

MapReduce was originally proposed for large-scale data processing applications on clusters [10]. It has then evolved to a computational model for a broader class of problems on a range of computing platforms [18], [13], [14], [19], [20]. In [14], MapReduce is used for a class of machine learning algorithms on multicore. These algorithms are written in a summation form which is easily fit into MapReduce model. MapReduce is also used for loopy belief propagation in graphical models [21], [22]. Howerver, these problems employ embarrassingly parallel algorithms that have no data dependency; that is each MapReduce iteration scan on the same entire dataset.

The use of MapReduce has been introduced for some graph algorithms such as minimum spanning tree, shortest path, or tree-based prefix sum [17], [18]. Design patterns for graph algorithms in MapReduce are discussed in [23] with the illustration of the PageRank algorithm. It is also assumed that the computations occur at every node of the graph. For exact inference, there exist data dependencies given by the graph, only potential tables at some nodes in the graph are ready in a MapReduce invocation. Graph traversal methods are used in our paper to handle data dependency constraints.

Several runtime systems have been developed to support the MapReduce model. Hadoop [11] is a well-known MapReduce infrastructure used intensively in industry. Twister [12] was introduced to support iterative MapReduce. It aims to minimize the overhead between iterations of MapReduce. These runtime systems focus on handling data-intensive applications with very large data files on distributed platforms. On multicore, the Phoenix runtime [13] allows easy implementation of MapReduce based applications by handling the low-level parallelism, load balancing and locality. In this paper, we use Phoenix because it maintains the basic MapReduce model and is developed for multicore platforms.

IV. MapReduce for Exact Inference

A. Task Definition

As given in II, exact inference in junction tree proceeds in two stages: evidence collection and evidence distribution. In evidence collection, evidence is propagated from the leaves to the root. When a clique is fully updated from all of its children, it is ready to compute the marginal of the separator connecting to its parent. The update of a clique from the separators connecting to its children is considered as an aggregation operation. Thus, it is natural to define the computation of marginal of a separator as a map task, and the update of a clique from all of its children as a reduce task. Details of the map task and reduce task for evidence collection are given in Algorithm 1.

Algorithm 1 Map and Reduce tasks for evidence collection

1: **procedure** MAP(id i, POT $\psi_{\mathcal{Y}}$)
2: Let $j = pa(i)$ be the identifier of the parent of $\mathcal{C}_i$
3: Let $\mathcal{S} = sep(i, j)$ be the separator of $\mathcal{C}_i$ and $\mathcal{C}_j$
4: Compute marginal $\psi_{\mathcal{S}}$ of $\mathcal{S}$ from $\psi_{\mathcal{Y}}$ according to Eq. 1
5: EMIT(j, $\psi_{\mathcal{S}}$)
6: **end procedure**

7: **procedure** REDUCE(id j, list $[\psi_{\mathcal{S}_1}, \psi_{\mathcal{S}_2}, \dots])$
8: Denote $\psi_{\mathcal{X}}$ as the POT of clique $\mathcal{C}_j$
9: **for all** $\psi_{\mathcal{S}} \in [\psi_{\mathcal{S}_1}, \psi_{\mathcal{S}_2}, \dots, \psi_{\mathcal{S}_d}]$ **do**
10: Update $\psi_{\mathcal{X}}$ from $\psi_{\mathcal{S}}$ according to Eq. 2
11: **end for**
12: EMIT(j, $\psi_{\mathcal{X}}$)
13: **end procedure**

Algorithm 2 Map and Reduce tasks for evidence distribution

1: **procedure** MAP(id i, POT $\psi_{\mathcal{Y}}$)
2: Let $\mathcal{L} = ch(i)$ be identifiers of all children of $\mathcal{C}_i$
3: **for all** $j \in \mathcal{L}$ **do**
4: Let $\mathcal{S} = sep(i, j)$ be the separator of $\mathcal{C}_i$ and $\mathcal{C}_j$
5: Compute marginal $\psi_{\mathcal{S}}$ of $\mathcal{S}$ from $\psi_{\mathcal{Y}}$ according to Eq. 1
6: EMIT(j, $\psi_{\mathcal{S}}$)
7: **end for**
8: **end procedure**

9: **procedure** REDUCE(id j, list $[\psi_{\mathcal{S}_1}, \psi_{\mathcal{S}_2}, \dots])$
10: Denote $\psi_{\mathcal{X}}$ as the POT of clique $\mathcal{C}_j$
11: **for all** $\psi_{\mathcal{S}} \in [\psi_{\mathcal{S}_1}, \psi_{\mathcal{S}_2}, \dots]$ **do**
12: Update $\psi_{\mathcal{X}}$ from $\psi_{\mathcal{S}}$ according to Eq. 2
13: **end for**
14: EMIT(j, $\psi_{\mathcal{X}}$)
15: **end procedure**

In the map task, defined by procedure MAP, the key i is the identifier of the clique, and the value $\psi_{\mathcal{Y}}$ is the potential table (POT) of clique $\mathcal{C}_i$. Line 2 assigns the identifier of the parent clique of $\mathcal{C}_i$ to j. The marginal $\psi_{\mathcal{S}}$ of the separator connecting $\mathcal{C}_i$ with its parent is computed in Line 3. In Line 4, this map task produces j as the intermediate key and $\psi_{\mathcal{S}}$ as the intermediate value.

In the reduce task, defined by procedure REDUCE, the inputs include identifier j as the intermediate key and a list of the associated intermediate values $[\psi_{\mathcal{S}_1}, \psi_{\mathcal{S}_2}, \dots]$ produced by the map tasks. Clique $\mathcal{C}_j$ has potential table $\psi_{\mathcal{X}}$ which will be

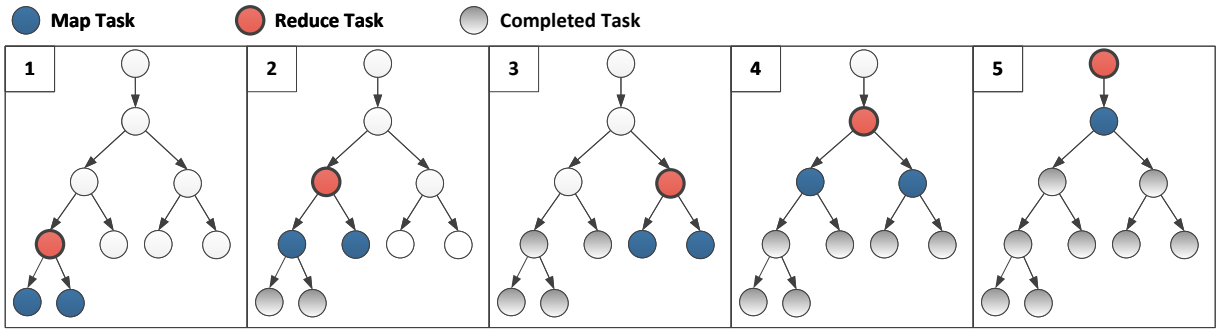


Fig. 3. Map tasks and Reduce tasks in the post-order sequence of MapReduce invocations for evidence collection.

Algorithm 3 Evidence collection using MapReduce with post-order traversal

```
1:  procedure MR_Postorder_Propagation(JTree 𝒥, id i)
2:      Let ℒ = ch(i) be identifiers of all children of 𝒞_i in 𝒥
3:      if ℒ ≠ ∅ then
4:          for all j ∈ ℒ do
5:              MR_Postorder_Propagation(𝒥, j)
6:          end for
7:          Set ℒ as input data for the map phase
8:          Perform MapReduce invocation on ℒ
9:      end if
10: end procedure
```

Algorithm 4 Evidence distribution using MapReduce with pre-order traversal

```
1:  procedure MR_Preorder_Propagation(JTree 𝒥, id i)
2:      Let ℒ = ch(i) be identifiers of all children of 𝒞_i in 𝒥
3:      if ℒ ≠ ∅ then
4:          Set {i} as input data for the map phase
5:          Perform MapReduce invocation on {i}
6:          for all j ∈ ℒ do
7:              MR_Preorder_Propagation(𝒥,j)
8:          end for
9:      end if
10: end procedure
```

updated from all child cliques via the separators. Lines 9-11 perform the update of $\psi_{\mathcal{X}}$ from each new $\psi_{\mathcal{S}}$ of its children.

Computing marginal of a separator and updating a clique can be performed by node level primitives given in [9]. At a clique with w variables, each having r states, and d children, the complexity of a map task is $O(w \cdot r^w)$ and the complexity of a reduce task is $O(d \cdot w \cdot r^w)$.

In evidence distribution, evidence is propagated from the root to the leaves. A clique is updated from its single parent. The clique then computes the marginal of each separator for all of its child cliques. We define the map task as the computation of the marginal and the reduce task as the update of the clique. Details of the map task and reduce task for evidence distribution are given in Algorithm 2. In the algorithm, the map task computes the marginal for all of its child cliques. The map task now produces a list of intermediate pairs of key and value, each for a separator connecting to a child clique. The reduce task is exactly the same as that in evidence collection. However, the list of intermediate values $[\psi_{\mathcal{S}_1}, \psi_{\mathcal{S}_2}, \dots]$ should now contain only one element corresponding to the single parent. For evidence distribution, the complexity of a map task is $O(d \cdot w \cdot r^w)$ and the complexity of a reduce task is $O(w \cdot r^w)$.

B. Depth-First Search based Iterative MapReduce

Iterative MapReduce is used to complete evidence propagation in junction trees. In each iteration, there is one MapReduce invocation in which the Map tasks are executed in parallel and so are the Reduce tasks. It is essential to determine the input data for each MapReduce invocation so that data dependency constraints are preserved. In evidence collection, a clique is ready to compute marginal of the separator connecting to its parent *only after* it is updated from *all* the separators connecting to its children. In evidence distribution, when a clique is updated from its parent, it is ready to compute separator marginal for all of its children.

Our first approach to preserving the data dependency constraints is using *depth-first search* (DFS) traversal for the sequence of MapReduce invocations. We call this approach DFS-based approach. The DFS-based approach employs *post-order* traversal in evidence collection and *pre-order* traversal in evidence distribution. Data input for each MapReduce invocation is set up accordingly. This approach is illustrated in Algorithms 3 and 4.

In Algorithm 3, the post-order traversal for the sequence of MapReduce invocations is produced by recursive procedure calls. The initial value of input i is the identifier of the root. Lines 7-8 perform the MapReduce invocation with the defined data input for its map phase. By the definition of the map task and reduce task, there is one reduce task and multiple map tasks corresponding to the child nodes of the reduce node in each MapReduce invocation. The map tasks at the child nodes are executed in parallel in the map phase. Figure 3 illustrates the map tasks and reduce tasks in the post-order sequence of MapReduce invocations for evidence collection in a sample junction tree. Only non-leaf cliques become reduce tasks. It

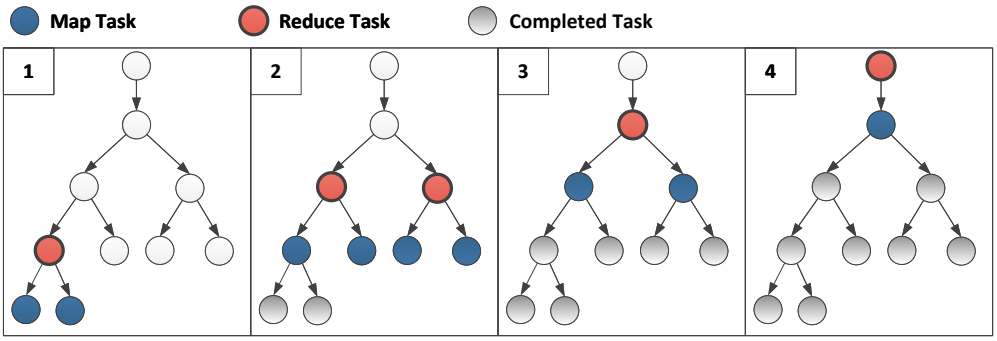


Fig. 4. Map tasks and Reduce tasks in the level-based sequence of MapReduce invocations for evidence collection.

Algorithm 5 Level-based MapReduce for evidence collection.

```
1: procedure MR_Level-based_EviCol(JTree 𝒥)
2:     Let H = height(J)
3:     for l = H, H − 1, ..., 1 do
4:         Let ℒ = nodes_at_level(𝒥, l)
5:         Set ℒ as input data for the map phase
6:         Perform MapReduce invocation on ℒ
7:     end for
8: end procedure
```

Algorithm 6 Level-based MapReduce for evidence distribution.

```
1: procedure MR_Level-based_EviDist(JTree 𝒥)
2:     Let H = height(J)
3:     for l = 0, 1, ..., H − 1 do
4:         Let ℒ = nodes_at_level(𝒥, l)
5:         Set ℒ as input data for the map phase
6:         Perform MapReduce invocation on ℒ
7:     end for
8: end procedure
```

can be seen that the number of MapReduce invocations is equal to the number of non-leaf cliques in the junction tree.

In Algorithm 4, the pre-order traversal for the sequence of MapReduce invocations is produced. The initial value of input i is also the identifier of the root. The MapReduce invocation is performed before the procedure is called recursively on the current clique's children. This case, the data input for the map phase includes only the current clique. Thus, in each MapReduce invocation, there is one map task and multiple reduce tasks that corresponds to the children of the map clique.

C. Level based Iterative MapReduce

The DFS-based approach provides a straightforward way to preserve the data dependencies. However, parallelism is available only in the map phase for evidence collection or in the reduce phase for evidence distribution. In addition, parallelism is limited by the number of child nodes of a clique. We propose a level-based approach that can still preserve data dependencies but exploits more parallelism. Level of a clique is defined as the number of edges on the path from it to the root in the junction tree. In this approach, each MapReduce invocation receives all the nodes at a certain level as the data input for the map phase. The sequence of MapReduce invocations proceeds with a descending-level order in evidence collection, and proceeds with a ascending-level order in evidence distribution. Hence, the data dependencies are preserved by the order of map tasks and reduce tasks in the sequence of MapReduce invocations. This approach is illustrated in Algorithm 5 and 6.

In Algorithm 5, data input for the map phase of each MapReduce invocation includes all the cliques at a corresponding level. Map task and reduce tasks for evidence distribution are defined in Algorithm 1. The map tasks at level l produce a set of intermediate keys that are identifiers of some parent cliques at level $(l-1)$. The intermediate values for an intermediate key are grouped by the MapReduce invocation, giving separator input to clique update to be performed by a reduce task. Note that only non-leaf cliques at level $(l-1)$ correspond to the reduce tasks. The sequence of MapReduce invocations are performed following the descending order of level l from H to 1. The execution of Agorithm 5 in a sample junction tree is illustrated in Figure 4.

We analyze the complexity of the algorithm using the concurrent read exclusive write parallel random access machine (CREW-PRAM) model [24]. We assume that each non-leaf node has d children and w variables with each variable having r states. Let N denote the number of nodes, H denote the tree height, and P denote the number of processors. Assume that n_l is the total number of nodes and m_l is the number of non-leaf nodes at level l. Note that $n_l = d \cdot m_{l-1}$. In the MapReduce invocation at tree level l, $H \geq l \geq 1$, there are n_l map tasks executed in parallel in the map phase, and m_{l-1} reduce tasks executed in parallel in the reduce phase. Thus, the execution time in the MapReduce invocation is $t_M^{(l)} = w \cdot r^w \cdot \lceil n_l/P \rceil \leq w \cdot r^w \cdot (\lfloor n_l/P \rfloor + 1)$ for the map phase, and $t_R^{(l)} = d \cdot w \cdot r^w \cdot \lceil m_{l-1}/P \rceil \leq d \cdot w \cdot r^w \cdot (\lfloor m_{l-1}/P \rfloor + 1)$ for the reduce phase. The number of MapReduce invocations is H. Thus, the time complexity of Algorithm 5 is $t = \sum_{l=H}^{1} (t_M^{(l)} + t_R^{(l)}) = O(d \cdot w \cdot r^w \cdot (N/P + H))$.

Algorithm 6 shows the level-based approach to using

MapReduce for evidence distribution. In this algorithm, the sequence of MapReduce invocations follows the ascending order of the level l, from 0 to $(H-1)$. Also note that the map task and reduce task in evidence distribution, defined in Algorithm 2, are different from those in evidence collection. Similar analysis shows that the time complexity of Algorithm 6 is $t = O(d \cdot w \cdot r^w \cdot (N/P + H))$, which is the same as that of Algorithm 5.

V. Experiments

A. Facilities

We conducted experiments on a 40-core Intel Westmere-EX based system as a representative state-of-the-art multi-core system. The system consists of four Xeon E7-4860 processors fully connected through 6.4 GT/s QPI links. Each processor has 10 cores running at 2.26 GHz and sharing 24 MB L3 cache. The system has total 64 GB DDR3 shared memory. The operating system is Red Hat Linux 4.1.2-45. We used gcc-4.1.2 with optimization flag -O3 to compile our programs.

Phoenix-2 [25] was used as the MapReduce runtime for our MapReduce invocations. The runtime allows us to define the Map task and Reduce task, and to assign data input for the map phase by defining function `splitter` for each MapReduce invocation. The runtime is responsible for scheduling all the Map tasks and all the Reduce tasks to the available cores. We evaluated the scalability of our proposed method using various numbers of cores up to 40.

B. Datasets

To evaluate the performance of our proposed approaches, we used a set of balanced junction trees generated synthetically and a set of random junction trees converted from real Bayesian networks.

- *Balanced junction trees* were generated as d-ary trees, in which each node except the leaf nodes and the last non-leaf node has d children. Impact of the tree structures and the task size on the performance were evaluated using various parameters such as number of children of each node (d), number of cliques (N), and clique width (w), i.e. number of variables in a clique.
- *Random junction trees* were obtained by extracting parts of the junction trees converted from real Bayesian networks in the Quick Medical Reference knowledge base [26]. The junction trees were extracted with parameters including number of cliques (N) and tree height (H). For each pair of N and H, we obtained 20 random trees and reported the average execution time.

C. Performance Metrics and Baselines

Execution time and speedup are used to evaluate the performance of our method. The speedup on P cores is calculated as the ratio of the execution time on a single core to the execution time on P cores.

We compare our MapReduce based approaches with serial, data parallel, and OpenMP implementations for evidence propagation. *Serial* is a sequential implementation of evidence propagation in which cliques are updated one by one by the BFS-based order of cliques. *Data parallel* is a parallel technique that explores data parallelism in node level computation [9]. The cliques are updated one by one by the BFS-based order, but in each node level primitive, the potential table is divided and distributed to the cores so that different parts of the potential table can be updated in parallel. The *OpenMP* baseline uses the available OpenMP *directives* to parallelize the node level computation. The ease of implementation of the two parallel baselines is comparable with that of the MapReduce based approaches.

D. Experimental Results

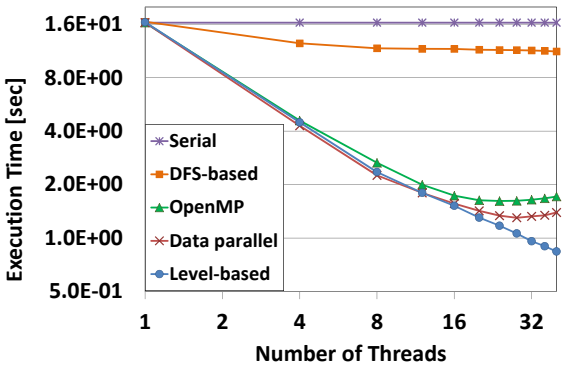
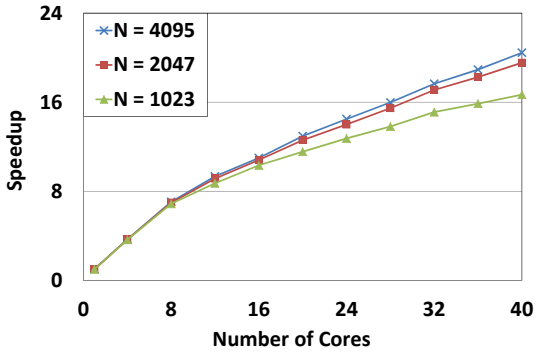


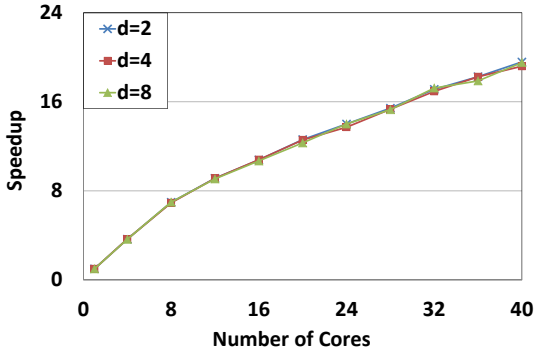
Fig. 5. Scalability of the MapReduce based approaches compared with the baselines.

Figure 5 illustrates the execution time of the proposed approaches compared with the two parallel baselines. The experiments were conducted using a balanced junction tree with $N = 2047$, $r = 2$, $w = 15$, and $d = 2$ except that $d = 40$ was used for the DFS-based MapReduce approach. With $r = 2$ and $w = 15$, data parallelism is sufficient for the two baselines to scale well when less than 20 cores are used. With low synchronization overhead, the *data parallel* technique exhibits even better scalability compared with the level-based MapReduce approach when the number of cores is less than 10. However, as the number of cores increases beyond 20, the two baselines do not scale with the number of cores. The scalability of the *OpenMP* baseline is worse because the OpenMP runtime does not offer efficient support for the irregular data accesses required by the node level computation. In addition, false sharing is another issue in the OpenMP implementation when a potential table, stored as a continuous array, is updated frequently by multiple threads. The level-based MapReduce approach shows good scalability with the number of cores. When 40 cores are used, the improvement in execution time of the level-based MapReduce approach is 50.3% compared with the *OpenMP* baseline and is 39.6% compared with the *data parallel* baseline.

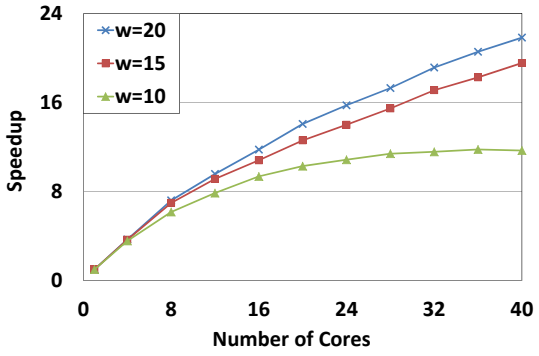
For the DFS-based MapReduce approach, because there are only d map tasks in one MapReduce invocation, we set $d = 40$ to provide sufficient parallelism. This approach shows very limited scalability in Figure 5. However, this result is



(a)



(b)



(c)

Fig. 6. Speedup of the level-based MapReduce approach on the 40-core Intel Westmere-EX based system for balanced junction trees with respect to various parameters.

expected because as analyzed in Section IV-B, the execution time cannot be reduced by more than 50% compared with the sequential execution time regardless of the number of cores used. Thus, the DFS-based approach does not seem to be suitable for exact inference in junction trees. For other computational problems, the DFS-based MapReduce approach may appeal as an effective method due to its simplicity. In the following paragraphs, we present the performance of the level-based MapReduce approach.

Impact of tree structures and task size on the performance of the level-based MapReduce approach are illustrated in Figure 6. The default values of the parameters are: $N = 2047$, $r = 2$, $w = 15$, $d = 2$. Figure 6(a) shows that for larger values

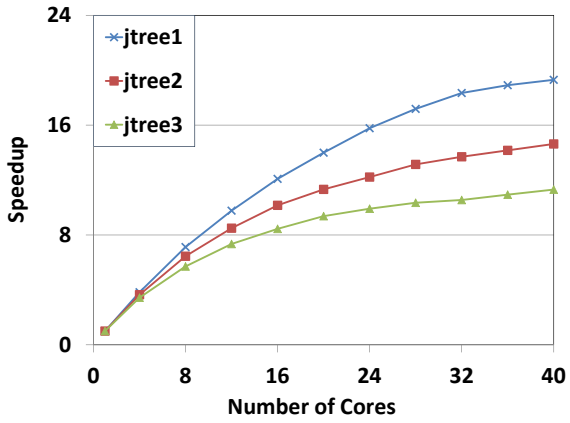


Fig. 7. Speedup of the level-based MapReduce based approach on the 40-core Intel Westmere-EX based system for randome junction trees.

of N, i.e. for larger trees, the proposed approach achieves better speedups. This result is expected because larger trees offer more parallelism and also reduce the impact of the limited parallelism available at the root clique on the overall execution time. In Figure 6(b), changing value of d does not have significant impact on the speedup of the approach. The speedups achieved are almost identical for $d = 2$, $d = 4$, and $d = 8$. In Figure 6(c), clique width w, which reflects the task size defined by $w \cdot r^w$, shows significant impact on the speedup of the approach. The larger the value of w, i.e. the larger the task size, the better speedup the level-based MapReduce approach achieves. Larger task size helps reduce the impact of overhead of the Phoenix system when managing the parallelization.

Finally, we evaluated the level-based MapReduce approach using the random junction trees based on real Bayesian networks in the Quick Medical Reference knowledge base. We obtained junction trees with $N = 2000$, $r = 2$, and $w = 12$ as the average. We varied the density of the junction trees by setting various values for tree height H. The smaller the tree height, the more "bushy" the tree is. Figure 7 shows the speedup of the proposed approach for three junction trees, named *jtree1*, *jtree2*, *jtree3*, corresponding to $H = 5$, $H = 15$, $H = 25$ respectively. As expected, the best speedup is achieved with *jtree1*, the next one is with *jtree2*, and the last one is with *jtree3*. Although the average clique width w is quite small ($w = 12$), the level-based MapReduce method still achieves good scalability with *jtree1*. Compared with balanced trees, these random trees have more even distribution of the nodes to the levels. Thus, when the number of cores is average (from 10 to 30), the speedup tends to be better. However, when the number of cores is more than 30, the speedup tapers off, because the number of nodes at each level becomes relatively small compared with available parallelism.

VI. Conclusion

In this paper, we studied the use of MapReduce programming model to exploit task parallelism for exact inference in Bayesian networks. We defined map tasks, reduce tasks,

and methods to utilize the MapReduce framework for the evidence propagation process with respect to data dependency constraints. Our approaches employed depth-first search (DFS) traversal methods and a level-based method to order the sequence of MapReduce invocations for handling the data dependencies. The DFS-based approach provided a natural and basic way of using MapReduce for evidence propagation in junction trees. Although having limited scalability for evidence propagation, the DFS-based approach can be efficient for other computational problems. The level-based approach also provided a simple method and was shown to be efficient for evidence propagation in junction trees. Compared with data parallel baselines that have similar level of implementation complexity, the level-based MapReduce approach achieved better scalability with the number of cores, especially when the data parallelism is limited. Using an existing MapReduce runtime called Phoenix, the level-based approach achieved $20\times$ speedup for exact inference on a 40-core Intel Westmere-EX based system. Our future work includes further study of optimization techniques for MapReduce, such as embedding map tasks to reduce tasks or overlapping map tasks with reduce tasks to increase task parallelism. In addition, we plan to explore the use of MapReduce for data parallelism in the computation within a clique node.

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 *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, "Scalable node-level computation kernels for parallel exact inference," *IEEE Trans. Comput.*, vol. 59, no. 1, pp. 103–115, 2010.

[10] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," in *OSDI*, 2004, pp. 137–150.

[11] Hadoop wiki - powered by. [Online]. Available: http://wiki.apache.org/hadoop/PoweredBy

[12] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S.-H. Bae, J. Qiu, and G. Fox, "Twister: a runtime for iterative mapreduce," in *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, 2010, pp. 810–818.

[13] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis, "Evaluating mapreduce for multi-core and multiprocessor systems," in *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, ser. HPCA '07, 2007, pp. 13–24.

[14] C. T. Chu, S. K. Kim, Y. A. Lin, Y. Yu, G. R. Bradski, A. Y. Ng, and K. Olukotun, "Map-reduce for machine learning on multicore," in *NIPS*. MIT Press, 2006, pp. 281–288.

[15] B. Panda, J. S. Herbach, S. Basu, and R. J. Bayardo, "Planet: massively parallel learning of tree ensembles with mapreduce," *Proc. VLDB Endow.*, vol. 2, no. 2, pp. 1426–1437, 2009.

[16] J. Tang, J. Sun, C. Wang, and Z. Yang, "Social influence analysis in large-scale networks," in *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*, ser. KDD '09, 2009, pp. 807–816.

[17] M. T. Goodrich, N. Sitchinava, and Q. Zhang, "Sorting, searching, and simulation in the mapreduce framework," in *Proceedings of the 22nd international conference on Algorithms and Computation*, 2011, pp. 374–383.

[18] H. J. Karloff, S. Suri, and S. Vassilvitskii, "A model of computation for mapreduce," in *SODA*, 2010, pp. 938–948.

[19] B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang, "Mars: a mapreduce framework on graphics processors," in *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, ser. PACT '08, 2008, pp. 260–269.

[20] J. H. C. Yeung, C. C. Tsang, K. H. Tsoi, B. S. H. Kwan, C. C. C. Cheung, A. P. C. Chan, and P. H. W. Leong, "Map-reduce as a programming model for custom computing machines," in *Proceedings of the 2008 16th International Symposium on Field-Programmable Custom Computing Machines*, ser. FCCM '08, 2008, pp. 149–159.

[21] J. Gonzalez, Y. Low, and C. Guestrin, "Residual splash for optimally parallelizing belief propagation," *Journal of Machine Learning Research - Proceedings Track*, vol. 5, pp. 177–184, 2009.

[22] C. F. U Kang, Duen Horng (Polo) Chau, "Inference of beliefs on billion-scale graphs," in *The 2nd Workshop on Large-scale Data Mining: Theory and Applications (LDMTA 2010)*, July 2010.

[23] J. Lin and M. Schatz, "Design patterns for efficient graph algorithms in mapreduce," in *Proceedings of the Eighth Workshop on Mining and Learning with Graphs*, ser. MLG '10, 2010, pp. 78–85.

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

[25] R. M. Yoo, A. Romano, and C. Kozyrakis, "Phoenix rebirth: Scalable mapreduce on a large-scale shared-memory system," in *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*, ser. IISWC '09, 2009, pp. 198–207.

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