Data Parallelism for Belief Propagation in Factor Graphs

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*—We investigate data parallelism for belief propagation in acyclic factor graphs on multicore/manycore processors. Belief propagation is a key problem in exploring factor graphs, a probabilistic graphical model that has found applications in many domains. In this paper, we identify basic operations called node level primitives for updating the distribution tables in a factor graph. We develop algorithms for these primitives to explore data parallelism. We also propose a complete belief propagation algorithm to perform exact inference in such graphs. We implement the proposed algorithms on state-of-the-art multicore processors and show that the proposed algorithms exhibit good scalability using a representative set of factor graphs. On a 32-core Intel Nehalem-EX based system, we achieve 30× speedup for the primitives and 29× for the complete algorithm using factor graphs with large distribution tables.**

I. Introduction

Graphical models have been essential tools for probabilistic reasoning. *Factor graph*s [1] have recently emerged as a unified model of directed graphs (e.g. Bayesian networks) and undirected graphs (e.g. Markov networks). A factor graph naturally represents a joint probability distribution that is written as a product of factors, each involving a subset of variables. Factor graphs have found applications in a variety of domains such as image processing, bioinformatics, and especially error-control decoding used in digital communications [2], [3], [4], [5], [6].

Inference is the problem of computing posterior probability distribution of certain variables given some value-observed variables as evidence. In factor graphs, inference is proceeded by the well-known *belief propagation* algorithm [1]. Belief propagation is a process of passing messages along the edges of a graph. Processing each message requires a set of operations with respect to the probability distribution of the random variables in a graph. Such distribution is represented by *potential tables*. The complexity of belief propagation increases dramatically as the number of states of variables and the node degrees of a graph increase. In many applications, such as digital communications, belief propagation must be performed in real time. Therefore, parallel techniques are needed to accelerate the inference.

This research was partially supported by the U.S. National Science Foundation under grant number CNS-1018801.

Data parallelism is amongst the most popular techniques in parallel computing [7]. In [8], the authors proposed scalable data parallel algorithms for image processing with a focus on Gibbs and Markov Random Field model representation for textures. In [9], the authors introduced techniques for data orchestration and tuning on OpenCL devices by which data parallelism is supported. In [10], the authors discussed data parallelism in global address space programming. Data parallelism was also discussed in [11], [12], [13] for various computation intensive applications on multicore processors, accelerators, and grids.

In this paper, we explore data parallelism for belief propagation in factor graphs. Our target platform is general-purpose multicore systems where data is shared among the processors (cores). Various operations with respect to potential tables are parallelized by dividing the potential tables into small chunks each chunk is processed by a core. For factor graphs with large potential tables, data parallelism is an efficient approach to accelerate belief propagation. To the best of our knowledge, no data parallelism techniques have been discussed for belief propagation in factor graphs.

Our contributions in this paper include:

- We explore data parallelism for two primitives for belief propagation in factor graphs: table multiplication and table marginalization.
- We develop algorithms for the above two primitives on shared memory platforms. The primitives are combined for optimizing belief propagation in factor graphs.
- We implement the proposed algorithms on a 32-core Intel Nehalem-EX based system. Our implementation scales almost linearly with the number of processors. It shows 30× speedup compared with a serial implementation on a platform with 32 cores.

The rest of the paper is organized as follows. In Section II, we review factor graphs and belief propagation. Section III discusses related work. Section IV presents our proposed algorithms for node level primitives and complete belief propagation in factor graphs. Experiments are discussed in Section V. Section VI concludes the paper.

II. Background

A. Factor Graphs

Given a set of random variables $X = \{x_1, x_2, \ldots, x_n\}$, a joint probability distribution of X can be written as a factorized function [14]:

$$P(X) \propto \prod_{j=1}^{m} f_j(X_j) \tag{1}$$

where $\propto$ denotes *proportional to*; X_j is a subset of $\{x_1, \ldots, x_n\}$; factor $f_j(X_j)$ is called a local function of X_j, and m is the number of factors. A *local function* defines an unnormalized probability distribution of a subset of variables from X.

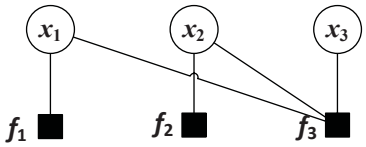

Fig. 1. A factor graph for the factorization $f_1(x_1)f_2(x_2)f_3(x_1, x_2, x_3)$.

A factor graph is a type of probabilistic graphical model that naturally represents such factorized distributions [1]. A factor graph is defined as $\mathbb{F} = (\mathbb{G}, \mathbb{P})$, where $\mathbb{G}$ is the graph structure and $\mathbb{P}$ is the parameter of the factor graph. $\mathbb{G}$ is a *bipartite graph* $\mathbb{G} = (\{X, F\}, E)$, where X and F are nodes representing *variables* and *factors*, respectively. E is a set of edges, each connecting a factor f_j and a variable $x \in X_j$ of f_j. Figure 1 shows a factor graph corresponding to the factorization $g(x_1, x_2, x_3) = f_1(x_1)f_2(x_2)f_3(x_1, x_2, x_3)$. Parameter $\mathbb{P}$ is a set of factor *potentials*, given by the definitions of local functions $f_i(X_j)$. For discrete random variables, a factor potential is represented by a table in which each entry corresponds to a state of the set of variables of the factor. We focus on discrete random variables in this paper.

Evidence in factor graphs is a set of variables with observed values, for example, $E = \{x_{e_1} = a_{e_1}, \ldots, x_{e_k} = a_{e_k}\}$, $e_k \in \{1, 2, \ldots, n\}$. Given evidence, an updated marginal distribution of any other variable can be inquired. *Inference* is the process of computing the posterior marginals of variables, given evidence. *Belief propagation* is a well-known inference algorithm introduced in [15], [16] and later formalized for factor graphs in [1]. After evidence E is *absorbed* at the involved variables, belief propagation is performed. Belief propagation is based on message passing processes where messages are computed locally and sent from a node to its neighbors. Two types of messages are given in Eqs. (2) and (3), one sent from variable node x to factor node f, and the other sent from factor node f to variable node x ([1]):

$$\mu_{x \to f}(x) \propto \prod_{h \in \mathcal{N}_x \setminus \{f\}} \mu_{h \to x}(x) \tag{2}$$

$$\mu_{f \to x}(x) \propto \sum_{\mathcal{N}_f \setminus \{x\}} \left(f(X_f) \prod_{y \in \mathcal{N}_f \setminus \{x\}} \mu_{y \to f}(y) \right) \tag{3}$$

where x and y are variables; f and h are factors; $\mathcal{N}_f$ and $\mathcal{N}_x$ are the sets of neighbors of f and x respectively; $\sum$ denotes marginalization over a potential table; $\prod$ denotes products of the potential tables. Note that a message is a distribution of a random variable. Both message computations require the products of the incoming messages from all the neighbors excluding the one where the message will be sent. Computing a message from f to x requires a marginalization for x.

We assume that each variable has at most r states and each node has at most d neighbors. Thus, the size of the potential table of a factor f is at most r^d, and the size of a message for a variable x is r. The serial complexity of computing $\mu_{x \to f}(x)$ is $O(d \cdot r)$, and that of computing $\mu_{f \to x}(x)$ is $O(d \cdot r^d)$. It can be seen that the complexity of computing $\mu_{f \to x}(x)$ is dominant the complexity of computing $\mu_{x \to f}(x)$.

In *cycle-free* factor graphs, message passing is initiated at leaves. Each node starts computing the message and sends it to a neighbor *after* receiving messages from *all* the other neighbors. The process terminates when two messages have been passed on every edge, one in each direction. Therefore, $|E|$ messages in Eq. (2) and $|E|$ messages in Eq. (3) are computed during the execution of belief propagation in cycle-free factor graphs. The overall serial complexity of belief propagation is $O(|E| \cdot d \cdot r + |E| \cdot d \cdot r^d) = O(|E| \cdot d \cdot r^d) = O(m \cdot d^2 \cdot r^d)$, where m is the number of factor nodes and $|E| = m \cdot d$. In this case, belief propagation leads to *exact inference*, since all the final results are guaranteed to be exact [1].

In *cyclic* factor graphs, belief propagation must be performed in an iterative message passing process. All messages are computed simultaneously using the messages from the previous iteration [1]. The process is terminated when the changes of messages between two consecutive iterations are less than a threshold. The final approximate results imply *approximate inference* for cyclic factor graphs.

The parallel version of belief propagation in cyclic factor graphs is known as an embarrassingly parallel algorithm. In this paper, we focus on belief propagation in cycle-free factor graphs only, where exploring parallelism is a challenge. Our techniques of exploiting data parallelism in computing a message are beneficial for belief propagation in cycle-free factor graphs. However, since computing a message in both types of graphs is identical, our techniques can also be applied for belief propagation in cyclic factor graphs.

Finally, once the belief propagation completes, the posterior marginals of variables are computed by Eqs. (4) and (5). The exact $P(x)$ and $P(X_f)$ is obtained by normalizing the right sides of the equations.

$$P(x) \propto \prod_{f \in \mathcal{N}_x} \mu_{f \to x}(x) \tag{4}$$

$$P(X_f) \propto f(X_f) \prod_{x \in \mathcal{N}_f} \mu_{x \to f}(x) \tag{5}$$

III. Related Work

There are a few earlier papers studying data parallelism in graphical models. In [17], the authors investigated data parallelism in exact inference with respect to Bayesian networks, where a Bayesian network must be converted into a junction tree before inference. The authors proposed node level primitives for evidence propagation in junction trees. The primitives including table marginalization, table extension, and table multiplication/division were optimized for distributed memory platforms. In [18], data parallelism in exact inference in junction trees is explored on Cell BE processors. Although all synergistic processing elements (SPEs) in a Cell BE processor access a shared memory, data must be explicitly moved to the local store of a SPE so as to be processed.

In this paper, we adapt the data layout for potential tables and the node level primitives for belief propagation for exact inference in factor graphs. For the node level computations, we optimize the primitives by taking advantage of the fact that only the relationship between a factor node and a single variable node needs to be considered. This leads to constant-time index mapping between a factor potential and a message. The mapping not only eliminates a space-expensive operation called table extension, but also reduces the execution time of table multiplication and table marginalization. In addition, our algorithms are developed for shared memory platforms, with a focus on general-purpose multicore/manycore systems. Thus, unlike [18], no explicit data copy is needed during the execution.

Several parallel techniques for belief propagation in factor graphs have been discussed in [19], [20]. However, those techniques were proposed for general factor graphs which lead to approximate inference, while we focus on acyclic factor graphs for exact inference. In addition, they focus on structural parallelism given by the factor graph topology. To the best of our knowledge, no data parallel techniques have been discussed for belief propagation in factor graphs.

IV. Data Parallelism in Belief Propagation

A. Representation of Distribution Tables

In a factor graph, each factor node f represents a local function of a set X_f of random variables. As given in Section II, this local function describes a probability distribution of the variables. For discrete random variables, the local function is defined by a potential table where each entry is a non-negative real number corresponding to the probability of a state of X_f. So as to efficiently store the potential tables, we follow the organization of clique potential tables as described in [17]. Accordingly, instead of storing the states of X_f along with their potential values, we only store the potential values in a one-dimensional table $\mathcal{F}$. The potential values are stored in an order such that index i of a value in $\mathcal{F}$ is determined based on the corresponding state of X_f as described below. Note that a message is also a potential table with respect to a variable.

Given a factor $f(X_f)$ with d variables. Let $X_f = x_1, x_2, ..., x_d$, where x_k is the k^{th} variable of X_f. Without loss of generality, we assume that the state of random variable x_k is taken from $D(x_k) = \{0, 1, ..., r_k - 1\}$, where $r_k = |D(x_k)|$. State of X_f is determined by the states of $x_1, x_2, ..., x_d$. The size of the potential table $\mathcal{F}$ of f is $|\mathcal{F}| = \prod_{k=1}^{d} r_k$. Similarly, the message at f to/from its variable x_k is stored by potential table $\mathcal{X}_k$; $|\mathcal{X}_k| = r_k$. Index i of an entry in $\mathcal{F}$ is determined by the corresponding state of X_f as given in Eq. 6:

$$i = \sum_{k=1}^{d} x_k \prod_{j=1}^{k-1} r_j \tag{6}$$

Thus, given any index i in potential table $\mathcal{F}$ of f, we determine the corresponding state $\bar{i}$ of variable x_k of f by:

$$\bar{i} = \left\lfloor \frac{i}{os_k} \right\rfloor \bmod r_k \tag{7}$$

where $os_k = \prod_{j=1}^{k-1} r_j$ with $os_1 = 1$. os_k is precomputed for each variable x_k of f. Thus, mapping an index i in $\mathcal{F}$ to the corresponding index $\bar{i}$ in $\mathcal{X}_k$ takes only $O(1)$ time.

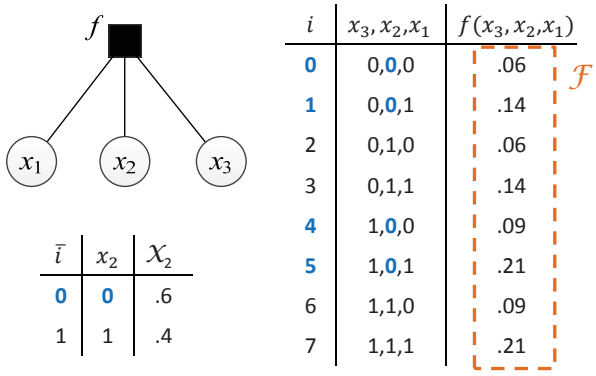

Fig. 2. A sample factor node $f(x_1, x_2, x_3)$, and the potential tables $\mathcal{F}$ of f and $\mathcal{X}_2$ of x_2. Only potential values in the dashed area are actually stored.

Figure 2 illustrates the organization of a potential table $\mathcal{F}$ of a factor f with $X_f = \{x_1, x_2, x_3\}$. Each variable has $r = 2$ states. Only the potential values of X_f are actually stored. There is a one-to-one mapping between the indices of entries in $\mathcal{F}$ and the states of X_f, given by Eq. 6. For example, the entry at index $i = 5$ corresponds to state $(x_1, x_2, x_3) = (1, 0, 1)$ of X_f. Thus, according to Eq. 7, each index of $\mathcal{F}$ determines the state of every variable in X_f. For example, for $i = 0, 1, 4, 5$, the state of x_2 is $\bar{i} = 0$.

B. Data parallelism in Node Level Primitives

Section II shows that the the complexity of computing a message from a factor node to a variable node is dominant the complexity of computing a message from a variable node to a factor node. Thus, we focus on parallelizing the computation at factor nodes. The computation at factor node f to produce a message $\mathcal{X}_k$ sent to $x_k, 1 \leq k \leq d$, is driven from Eq. 3 as follows:

$$\mathcal{X}_k = \sum_{X_f \setminus \{x_k\}} (\mathcal{F} \prod_{j \neq k} \mathcal{X}_j) \quad (8)$$

Eq. 8 is performed in two phases: (1) multiplying factor potential table $\mathcal{F}$ with the $(d-1)$ incoming messages to obtain resulting potential table $\mathcal{F}^*$ and (2) marginalizing the resulting $\mathcal{F}^*$ for x_k to obtain message $\mathcal{X}_k$. Hence, we have two node level primitives for computing a new message:

- Table multiplication $\mathcal{F}^* = \mathcal{F} \times \mathcal{X}_j$, realized by:

$$\mathcal{F}^*(i) = \mathcal{F}(i) \cdot \mathcal{X}_j(\bar{i}), \forall i \in \{0, 1, ..., |\mathcal{F}| - 1\} \quad (9)$$

- Table marginalization $\mathcal{X}_k = \sum_{\setminus \{x_k\}} (\mathcal{F}^*)$, realized by:

$$\mathcal{X}_k(\bar{i}) = \mathcal{X}_k(\bar{i}) + \mathcal{F}^*(i), \forall i \in \{0, 1, ..., |\mathcal{F}| - 1\} \quad (10)$$

where the relationship between i and $\bar{i}$ is determined by Eq. 7.

For factors with large potential tables, those primitives can be parallelized to accelerate execution. A large potential table $\mathcal{F}$ is divided into small chunks that can be processed concurrently. Detailed algorithms for these two primitives are given in the following section.

C. Parallel Algorithms for the Primitives

On a shared memory parallel system with P threads, the potential table $\mathcal{F}$ is divided into P chunks. In table multiplication, each thread updates its own chunk. In table marginalization, the output message needs to be updated using the potential factors from all chunks. Thus, when computing marginal, we let each thread keep a local resulting message computed from its own chunk. Then, these local messages are aggregated to generate the final message. The parallel algorithms for table multiplication and table marginalization are given in Algorithm 1 and Algorithm 2, respectively.

Algorithm 1 Table multiplication

Input: factor f, variable x_k of f, potential table $\mathcal{F}$, message $\mathcal{X}_k$ from x_k to f, number of threads P
Output: resulting potential table $\mathcal{F}^*$

1: **for** $p = 0$ to $P - 1$ **pardo**
   {*local computation*}
2: **for** $i = \frac{|\mathcal{F}|}{P} p$ to $\frac{|\mathcal{F}|}{P}(p+1) - 1$ **do**
3: $\bar{i} =$`index_mapping`(i, f, x_k) using Eq. 7
4: $\mathcal{F}^*(i) = \mathcal{F}(i) * \mathcal{X}_k(\bar{i})$
5: **end for**
6: **end for**

In Algorithm 1, the input includes factor f, the k^{th} variable (x_k) of f, a potential table $\mathcal{F}$ w.r.t f, a message $\mathcal{X}_k$ sent from x_k to f, and the number of threads P. The output is the product of $\mathcal{F}$ and $\mathcal{X}_k$, which is another potential table $\mathcal{F}^*$ of the same size of $\mathcal{F}$. Line 1 launches P-way parallelism and assigns an ID, $p \in \{0, ..., P-1\}$, to each thread. Lines 2-5 perform local computations of a thread on its own chunk of data that consists of $\frac{|\mathcal{F}|}{P}$ contiguous entries. Lines 3-4 update value of the i^{th} entry in $\mathcal{F}$ using the corresponding $\bar{i}^{th}$ entry in $\mathcal{X}_k$. Function `index_mapping`(i, f, x_k) computes $\bar{i}$ using Eq. 7 given that r_k and os_k are available for each variable x_k of factor f.

Algorithm 2 Table marginalization

Input: factor f, variable x_k of f, potential table $\mathcal{F}$, number of threads P
Output: marginal $\mathcal{X}_k$ for x_k from $\mathcal{F}$

1: **for** $p = 0$ to $P - 1$ **pardo**
   {*local computation*}
2: $\mathcal{X}_k^{(p)} = \overrightarrow{0}$
3: **for** $i = \frac{|\mathcal{F}|}{P} p$ to $\frac{|\mathcal{F}|}{P}(p+1) - 1$ **do**
4: $\bar{i} =$`index_mapping`(i, f, x_k) using Eq. 7
5: $\mathcal{X}_k^{(p)}(\bar{i}) = \mathcal{X}_k^{(p)}(\bar{i}) + \mathcal{F}(i)$
6: **end for**
7: **end for**
   {*global update for $\mathcal{X}_k$ from the local results*}
8: $\mathcal{X}_k = \sum_p \mathcal{X}_k^{(p)}$

In Algorithm 2, the input includes factor f, the k^{th} variable (x_k) of f, a potential table $\mathcal{F}$, and the number of threads P. The output is the marginal $\mathcal{X}_k$ computed from $\mathcal{F}$ for x_k. Line 2 initiates a local message for each thread. Lines 3-6 perform local computations of a thread on its own chunk of data to update its local message. Note that those locally updated messages can be read by all the threads. In Lines 4-5, value of the i^{th} entry in $\mathcal{F}$ is used to update the corresponding $\bar{i}^{th}$ entry in the local $\mathcal{X}_k^{(p)}$ of thread p. After Line 7, all the threads complete the local computations. Finally, Line 8 sums all the local messages to form the final output message $\mathcal{X}_k$. Note that all $|\mathcal{X}_k|$ entries of $\mathcal{X}_k$ can be processed in parallel.

We analyze the complexity of the algorithms using the concurrent read exclusive write parallel random access machine (CREW-PRAM) model [7] with P processors. Each thread runs in a separate processor. In Algorithm 1, the mapping function at Line 3 takes $O(1)$ time to complete, as given in Section IV-C. Line 4 also takes $O(1)$ time. Therefore, for $|\mathcal{F}| \geq P$, the complexity of Algorithm 1 is $O(\frac{|\mathcal{F}|}{P})$.

In Algorithm 2, Lines 1-7 take $O(\frac{|\mathcal{F}|}{P})$ time. Line 8 essentially computes the sum of P vectors, each vector has $|\mathcal{X}_k|$ entries ($|\mathcal{X}_k|$-by-P computations). If $|\mathcal{X}_k| \geq P$, we use P processors to compute P rows at a time step, with one processor for each row. In this case, Line 8 takes $O(\lceil \frac{|\mathcal{X}_k|}{P} \rceil P) = O(|\mathcal{X}_k|)$ time. If $\log P \leq |\mathcal{X}_k| < P$, we use P processors to compute $\log P$ rows at a time step, with $P/\log P$ processors for each row. Line 8 also takes $O(|\mathcal{X}_k|)$ time in this case. If $|\mathcal{X}_k| < \log P$, we use P processors to compute one row at a time step. In this case, Line 8 takes $O(|\mathcal{X}_k| \log P)$ time. Thus, for $|\mathcal{X}_k| \geq \log P$, the complexity of Algorithm 2 is $O(\frac{|\mathcal{F}|}{P} + |\mathcal{X}_k|)$.

D. Complete Algorithm for Belief Propagation

According to Eq. 8, computing a message at a factor node requires a series of node level primitives: $(d-1)$ table

multiplications and one table marginalization, where d is the node-degree of the factor node. The node level primitives discussed above can be utilized here. The incoming messages $\mathcal{X}_j$ from neighbor nodes are the inputs to the primitives. The computation process to produce a message at a factor node is described in Algorithm 3.

Algorithm 3 Message computation kernel

Input: factor f, variable x_k of f, potential table $\mathcal{F}$ of f, number of threads P
Output: message $\mathcal{X}_k$ sent from f to x_k

1: $\mathcal{F}^* = \mathcal{F}$
2: **for** $p = 0$ to $P-1$ **pardo**
{*local computation for the product of $\mathcal{F}$ with $\mathcal{X}_j$*}
3: **for** $j = 1$ to d, $j \neq k$ **do**
4: Compute $\mathcal{X}_j$
5: **for** $i = \frac{|\mathcal{F}|}{P}p$ to $\frac{|\mathcal{F}|}{P}(p+1)-1$ **do**
6: $\bar{i} =$`index_mapping`(i, f, x_j) using Eq. 7
7: $\mathcal{F}^*(i) = \mathcal{F}^*(i) * \mathcal{X}_j(\bar{i})$
8: **end for**
9: **end for**
{*local computation for the marginal $\mathcal{X}_k$ from $\mathcal{F}^*$*}
10: $\mathcal{X}_k^{(p)} = \overrightarrow{0}$
11: **for** $i = \frac{|\mathcal{F}|}{P}p$ to $\frac{|\mathcal{F}|}{P}(p+1)-1$ **do**
12: $\bar{i} =$`index_mapping`(i, f, x_k) using Eq. 7
13: $\mathcal{X}_k^{(p)}(\bar{i}) = \mathcal{X}_k^{(p)}(\bar{i}) + \mathcal{F}^*(i)$
14: **end for**
15: **end for**
{*global update for $\mathcal{X}_k$ from the local results*}
16: $\mathcal{X}_k = \sum_p \mathcal{X}_k^{(p)}$

In Algorithm 3, the input includes factor f, the k^{th} variable (x_k) of f, the original potential table $\mathcal{F}$ of f, and the number of threads P. The output is the message $\mathcal{X}_k$ to send from factor node f to variable node x_k. Line 1 initiates a copy $\mathcal{F}^*$ of $\mathcal{F}$ that will be used as an intermediate result. Lines 3-9 perform $(d-1)$ table multiplications between $\mathcal{F}^*$ with $(d-1)$ incoming messages. Line 4 computes the incoming message $\mathcal{X}_j$ using Eq. 2. Lines 10-15 perform table marginalization on the final $\mathcal{F}^*$ to generate the output message $\mathcal{X}_k$ for x_k. Line 16 combines all the local results to form the final message $\mathcal{X}_k$. In the CREW-PRAM model, the complexity of Algorithm 3 is $O(\frac{d|\mathcal{F}|}{P} + d\sum_{j\neq k}|\mathcal{X}_j| + |\mathcal{X}_k|)$, $1 \leq P \leq |\mathcal{F}|$.

The process of belief propagation consists of a sequence of local computations and message passing. The order of passing the messages guarantees that each message at a node is computed after all of the other messages have arrived at the node. A common strategy is to order the nodes by Breadth first search (BFS) with respect to an arbitrarily selected root [1]. Based on this order, a queue of directed edges $\mathbb{Q} = \{(f, x)\}$ from factors to variables is formed for the sequence of messages to be computed.

Given a queue of edges, belief propagation in factor graph

Algorithm 4 Belief propagation using message computation kernels

Input: factor graph $\mathbb{F} = (\mathbb{G}, \mathbb{P})$, edge queue $\mathbb{Q}$, number of threads P
Output: updated messages on all edges in both directions

1: **for** $p = 0$ to $P-1$ **pardo**
2: **for** $i = 1$ to $|\mathbb{Q}|$ **do**
3: Let $(f, x) = \mathbb{Q}(i)$
4: Let $\mathcal{F} = \mathbb{P}(f)$
5: `ComputeMessage`$(f, x, \mathcal{F})$
6: **end for**
7: **end for**

$\mathbb{F}$ is performed as shown in Algorithm 4. For each directed edge from factor f to variable x, it retrieves the potential table $\mathcal{F}$ of f from parameter $\mathbb{P}$. Then, it applies the message computation kernel given in Algorithm 3. A synchronization barrier is implied at the end of each iteration before moving to another edge. At the end of the algorithm, the messages in both directions on all the edges are updated.

V. Experiments

A. Facilities

We conducted experiments on a 32-core Intel Nehalem-EX based system and a 8 core AMD system. The former system consists of four Intel Xeon X7560 processors fully connected through 6.4 GT/s QPI links. Each processor has 24 MB L3 Cache and 8 cores running at 2.26 GHz. All the cores share a 256 GB DDR3 memory.

In addition, we conducted experiments for a baseline using OpenMP on an 8-core AMD Opteron based system. The system consists of the Dual AMD Opteron 2350 (Barcelona) Quad-Core processors running at 2.0 GHz, and 16 GB DDR2 memory shared by all the cores.

B. Datasets

We generated cycle-free factor graphs using *libDAI*, an open source library for probabilistic inference in factor graphs[21]. The factor graphs are generated with modifiable parameters: number of factors m, degree of each factor d, and number of states of each variable r. We examined the complexity and the scalability of our algorithms with regards to r and d by changing each of them at a time. In the experiments, we keep $m = 100$. With binary variables ($r = 2$), we conducted the experiments using $d = 16$, $d = 18$, and $d = 20$. With the fixed node degree $d = 12$, we conducted experiments using various numbers of states of variables $r = 2$, $r = 3$, and $r = 4$.

C. Performance Metrics

Metrics for evaluating the performance of our method are execution time and speedup. Speedup is defined as the ratio of the execution time of belief propagation on a single processor (core) to the execution time of belief propagation on P processors (cores). The serial code for belief propagation in factor graphs was obtained from libDAI. Our parallel code

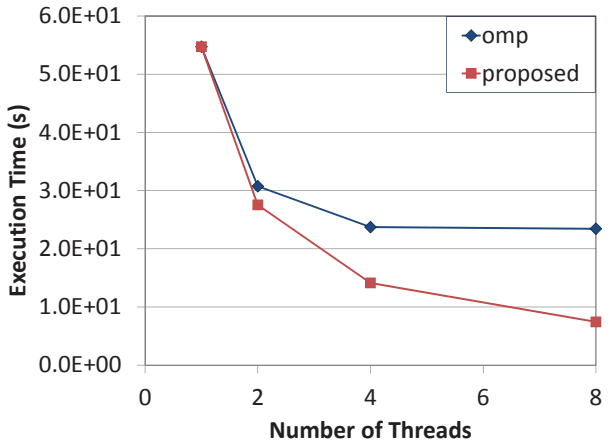

Fig. 3. Execution time of the OpenMP based implementation for belief propagation in factor graphs with $r = 2, d = 16, m = 100$ on an 8-core AMD Opteron based system.

was developed based on this libDAI serial code. Experimental results show that the execution time of our parallel code running on one core is almost the same as that of the serial code.

D. An OpenMP Implementation

An OpenMP implementation for belief propagation was developed based on libDAI serial code. In this OpenMP implementation, we used the directives `#pragma omp parallel for` in front of the loop for table multiplication in Eq. 9. For table marginalization in Eq. 10, we used `#pragma omp for reduction` for each entry in $\mathcal{X}_k$ and enabled nested loop with `omp_set_nested`. These directives guide the runtime system to automatically parallelize the computations within the loops using the given number of threads. We also used additional supporting options like static scheduling or dynamic scheduling. Figure 3 shows the experimental results for the OpenMP baseline and our proposed method on the 8-core AMD Opteron system. The dataset is a factor graph with $m = 100, r = 2, d = 16$. It can be seen that the OpenMP baseline does not scale as the number of threads is increased beyond 4. Our method scales well on this system. The experimental results of our method on the 32-core system are presented below.

E. Experimental Results

The algorithms were implemented on the 32-core system using Pthreads. We initiated as many threads as the number of hardware cores, and bound each thread to a separate core. These threads persisted over the entire program execution and communicated with each other using the shared memory. We conducted the experiments with various numbers of threads chosen from $1, 2, 4, 8, 16,$ and 32. We evaluated the performance of each individual primitive and of the entire belief propagation process.

From Section IV-C, the number of entries of a factor potential table is determined by $|\mathcal{F}| = \prod_{k=1}^{d} r_k = r^d$, assuming that $r_i = r, i = 1, ...d$. The number of entries of a message to/from x_k is $|\mathcal{X}_k| = r$. Double-precision floating-point numbers were used for potential values. Thus, with $r = 2$ and $d = 16, d = 18, d = 20$, the size of the potential tables are 512KB, 2GB, 8GB respectively. With $d = 12$ and $r = 2, r = 3, r = 4$, the size of the potential tables are 32KB, 4GB, 128GB respectively.

Figure 4 illustrates the execution time of table multiplication. In Figure 4(a), with $d = 18$ and $d = 20$, table multiplication scales almost linearly with the number of threads ranging to 32. With $d = 16$ however, the primitive does not show scalability when the number of threads increases from 16 to 32. Similarly in Figure 4(b), with $r = 3$ and $r = 4$, we achieved almost linear scalability for the primitive. The speedup achieved using 32 threads is 30. However with $r = 2$, the performance declines when the number of threads is greater than 8. This is because With $r = 2$ and $d = 12, d = 16$, the potential tables are too small to exploit data parallelism.

The execution time of table marginalization is illustrated in Figure 5. It has almost the same results with table multiplication. Although table marginalization requires the accumulation of the local results at the end, this primitive still scales very well with the number of threads for large potential tables. This is because in the experiments, $|\mathcal{F}| \gg |\mathcal{X}_k|$, making the computation time almost decrease almost linearly with the number of threads.

The overall execution time of belief propagation on the input factor graphs are shown in Figure 6. The results are consistent with the performance of the primitives. It is predictable since the belief propagation performs a series of message computations that are composed of a set of the primitives. With the scalable primitives, the overall program also scales well. However, due to the synchronization required between the primitives, the overall speedup is slightly lower than the speedup of the primitives. Using 32 threads, we observed a $29\times$ speedup for the factor graphs with $d = 12, r = 4$ or $d = 20, r = 2$.

VI. Conclusions

In this paper, we explored data parallelism for belief propagation in factor graphs. We developed algorithms for node level primitives to exploit data parallelism in the table-based operations. We then developed an algorithm for belief propagation using these primitives. We implemented the algorithms and conducted experiments on state-of-the-art general-purpose multicore systems. For factor graphs with large potential tables, the experimental results exhibited almost linear scalability. As part of our future work, we plan to explore other forms of parallelism in belief propagation with various granularity. For factor graph with small potential tables, it can be more efficient to explore structural parallelism. Hence, our future work will also include task scheduling that is essential to efficiently exploit structural parallelism.

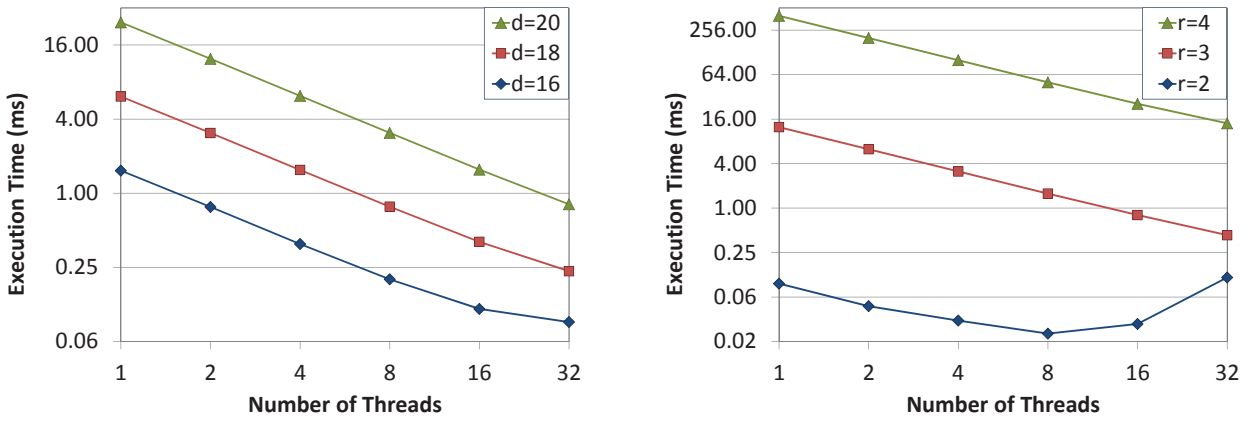

Fig. 4. Performance of table multiplication on the 32-core Intel Nehalem-EX based system with (a) $r = 2$ and various node degrees, (b) $d = 12$ and various numbers of states of the variables.

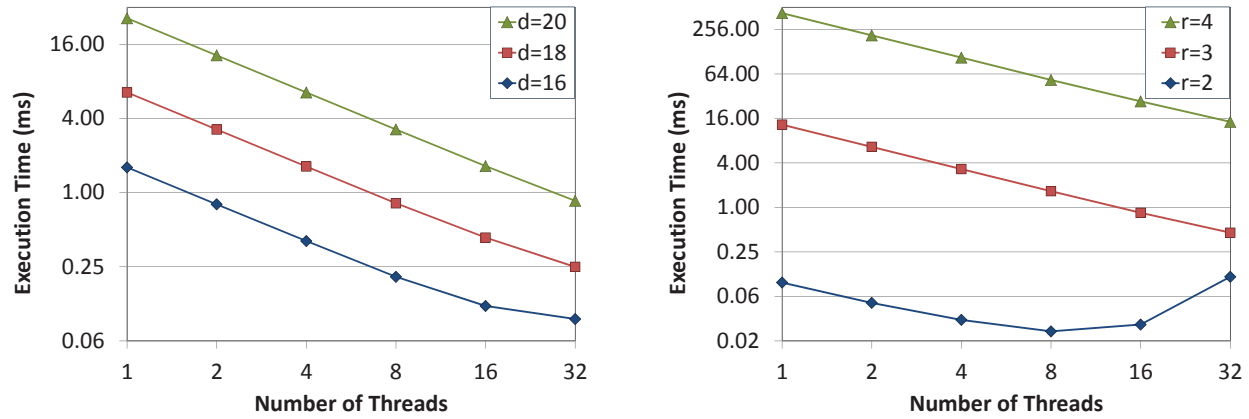

Fig. 5. Performance of table marginalization on the 32-core Intel Nehalem-EX based system with (a) $r = 2$ and various node degrees, (b) $d = 12$ and various numbers of states of the variables.

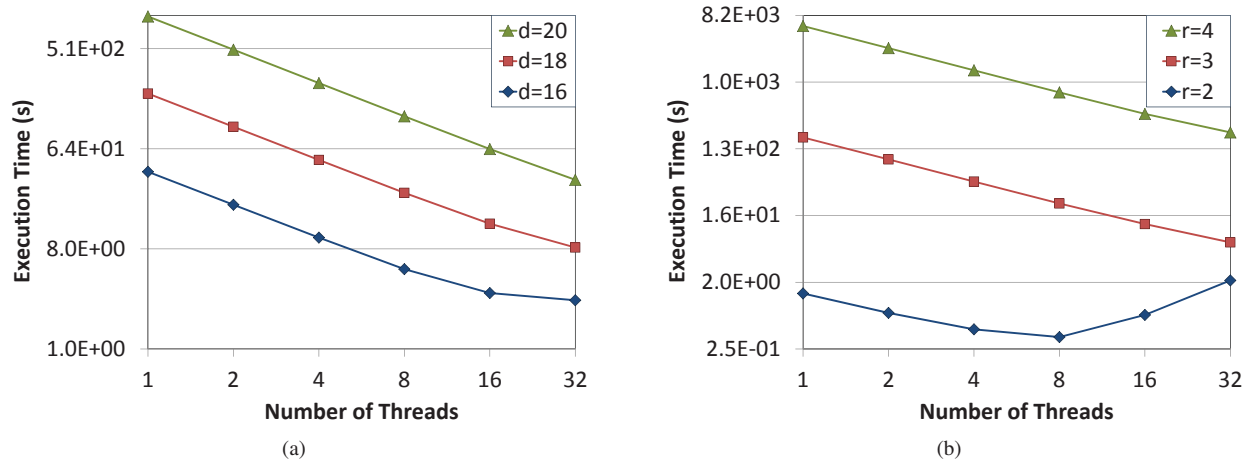

Fig. 6. Overall performance of belief propagation on the 32-core Intel Nehalem-EX based system with (a) $r = 2$ and various node degrees, (b) $d = 12$ and various numbers of states of the variables.

VII. Acknowledgements

Thanks to the management, staff, and facilities of the Intel® Manycore Testing Lab ([22], [23]).

References

[1] F. R. Kschischang, B. J. Frey, and H.-A. Loeliger, "Factor graphs and the sum-product algorithm," *IEEE Transactions on Information Theory*, vol. 47, no. 2, pp. 498–519, 2001.

[2] E. Sudderth and W. T. Freeman, "Signal and Image Processing with Belief Propagation," *IEEE Signal Processing Magazine*, vol. 25, Mar. 2008.

[3] A. Mitrofanova, V. Pavlovic, and B. Mishra, "Integrative protein function transfer using factor graphs and heterogeneous data sources," in *Proceedings of the 2008 IEEE International Conference on Bioinformatics and Biomedicine*. IEEE Computer Society, 2008, pp. 314–318.

[4] T. Richardson and R. Urbanke, "The Capacity of Low-Density Parity Check Codes under Message-Passing Decoding," *IEEE Transactions on Information Theory*, vol. 47, no. 2, pp. 599–618, 2001.

[5] C. Schlegel and L. Perez, *Trellis and Turbo Coding*. John Wiley & Sons, 2003.

[6] H. Loeliger, "An Introduction to factor graphs," *IEEE Signal Processing Magazine*, vol. 21, no. 1, pp. 28–41, 2004.

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

[8] D. A. Bader, J. JáJá, and R. Chellappa, "Scalable data parallel algorithms for texture synthesis and compression using gibbs random fields," *IEEE Transactions on Image Processing*, vol. 4, no. 10, pp. 1456–1460, 1995.

[9] K. Spafford, J. Meredith, and J. Vetter, "Maestro: data orchestration and tuning for opencl devices," in *Proceedings of the 16th international Euro-Par conference on Parallel processing: Part II*, ser. Euro-Par'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 275–286.

[10] C. Coarfa, Y. Dotsenko, J. Mellor-Crummey, F. Cantonnet, T. El-Ghazawi, A. Mohanti, Y. Yao, and D. Chavarría-Miranda, "An evaluation of global address space languages: co-array fortran and unified parallel c," in *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, ser. PPoPP '05. New York, NY, USA: ACM, 2005, pp. 36–47.

[11] M. Parashar and X. Li, *Advanced Computational Infrastructures for Parallel and Distributed Applications*, 2010, ch. Enabling Large-Scale Computational Science: Motivations, Requirements and Challenges, pp. 1–7.

[12] J. Kurzak, D. A. Bader, and J. Dongarra, *Scientific Computing with Multicore and Accelerators*. Chapman & Hall / CRC Press, 2010.

[13] R. Buyya, *High Performance Cluster Computing: Architectures and Systems, Vol. 1*. Prentice Hall, 1999.

[14] D. Koller and N. Friedman, *Probabilistic Graphical Models: Principles and Techniques*. The MIT Press, 2009.

[15] J. Pearl, *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1988.

[16] 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.

[17] Y. Xia and V. K. Prasanna, "Scalable Node Level Computation Kernels for Parallel Exact Inference," *IEEE Transactions on Computers*, vol. 59, no. 1, pp. 103–115, 2009.

[18] ——, "Parallel Evidence Propagation on Multicore Processors," *The Journal of Supercomputing*, 2010.

[19] A. Mendiburu, R. Santana, J. A. Lozano, and E. Bengoetxea, "A parallel framework for loopy belief propagation," in *GECCO (Companion)*, 2007, pp. 2843–2850.

[20] J. Gonzalez, Y. Low, C. Guestrin, and D. O'Hallaron, "Distributed parallel inference on large factor graphs," in *Conference on Uncertainty in Artificial Intelligence (UAI)*, Montreal, Canada, July 2009.

[21] J. M. Mooij, "libDAI: A free and open source C++ library for discrete approximate inference in graphical models," *Journal of Machine Learning Research*, vol. 11, pp. 2169–2173, Aug. 2010. [Online]. Available: http://www.jmlr.org/papers/volume11/mooij10a/mooij10a.pdf

[22] Intel® Manycore Testing Lab. [Online]. Available: http://www.intel.com/software/manycoretestinglab

[23] Intel® Software Network. [Online]. Available: http://www.intel.com/software