Data Parallelism for Belief Propagation in Factor Graphs

Nam Ma · Yinglong Xia · Viktor K. Prasanna

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 computations called node level primitives for propagating the belief in a factor graph. Algorithms for these primitives are developed using data parallel techniques. We propose a complete belief propagation algorithm using the primitives to perform exact inference in factor graphs. We implement the proposed algorithms on state-of-the-art multi-socket multi-core systems with additional NUMA-aware optimizations. Our proposed algorithms exhibit good scalability using a representative set of factor graphs. On a four-socket Intel Westmere-EX based system with 40 cores, we achieve 39.5× speedup for the primitives and 39.2× for the complete algorithm using factor graphs with large potential tables.

Keywords Data Parallelism · Belief Propagation · Multi-core · NUMA Platforms
1 Introduction

Graphical models have been essential tools for probabilistic reasoning. Factor graphs [12] 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 [26,17,22,23,15].

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 [12]. Belief propagation is a process of passing messages along the edges of a graph. Updating a message requires a set of operations with respect to the probability distribution of the random variables. 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 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.

Data parallelism is a key technique in parallel computing [8]. With its simplicity yet efficiency, data parallel technique has been used extensively. In [3], the authors proposed scalable data parallel algorithms for image processing with a focus on Gibbs and Markov Random Fields model representation for textures. In [25], the authors introduced techniques for data orchestration and tuning on OpenCL devices by which data parallelism is supported. In [5], the authors discussed data parallelism in global address space programming. Data parallelism was also discussed in [19,13,4] 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 state-of-the-art multi-socket multi-core systems. In these systems, multi-core processors are interconnected to form a non-uniform memory-access (NUMA) machine with a large number of cores. Various operations with respect to potential tables are parallelized by dividing the potential tables into small chunks, each 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 in two defined 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 multi-socket multi-core system. NUMA-aware optimizations with data placement are also implemented. Our implementation shows almost linear scalability with the number of cores.
The rest of the paper is organized as follows. In Section 2, we review factor graphs and belief propagation. Section 3 discusses related work. Section 4 presents our proposed algorithms for node level primitives and complete belief propagation in factor graphs. Experimental setup is given in Section 5. Section 6 discusses our techniques for NUMA-aware implementation and its results. Experimental results with various table sizes are presented in Section 7. Section 8 concludes the paper.

2 Background

2.1 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 [10]:

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

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 \( X \).

A factor graph is a type of probabilistic graphical model that naturally represents such factorized distributions [12]. A factor graph is defined as \( F = (G, P) \), where \( G \) is the graph structure and \( P \) is the parameter of the factor graph. \( G \) is a bipartite graph \( G = (\{X, F\}, E) \), 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 \( P \) 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, \( E = \{x_{e_1} = a_{e_1}, \ldots, x_{e_k} = a_{e_k}\} \), where \( e_i \in \{1, 2, \ldots, n\}, i = 1, \ldots, k \). 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 [20, 14] and later formalized for factor graphs in [12]. 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 $x$ (12):

$$\mu_{x\rightarrow f}(x) \propto \prod_{h \in N_x \setminus \{f\}} \mu_{h\rightarrow x}(x)$$  \hspace{1cm} (2)$$

$$\mu_{f\rightarrow x}(x) \propto \sum_{N_f \setminus \{x\}} \left( f(X_f) \prod_{y \in N_f \setminus \{x\}} \mu_{y\rightarrow f}(y) \right)$$  \hspace{1cm} (3)$$

where $x$ and $y$ are variables; $f$ and $h$ are factors; $N_f$ and $N_x$ 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 that the message will be sent to. 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 $\mu_{x\rightarrow f}(x)$ is $O(d \cdot r)$, and that of computing $\mu_{f\rightarrow x}(x)$ is $O(d \cdot r^d)$. It can be seen that the complexity of computing $\mu_{f\rightarrow x}(x)$ is dominant the complexity of computing $\mu_{x\rightarrow f}(x)$.

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 (12).

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 (12). 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 versions of belief propagation in cyclic factor graphs are known as embarrassingly parallel algorithms. In this paper, we focus on belief propagation in acyclic factor graphs only, where exploring parallelism is a challenge. Our techniques of exploiting data parallelism in computing a message are directly beneficial for belief propagation in acyclic 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.

$$P(x) \propto \prod_{f \in N_x} \mu_{f \rightarrow x}(x) \quad (4)$$

$$P(X_f) \propto f(X_f) \prod_{x \in N_f} \mu_{x \rightarrow f}(x) \quad (5)$$

3 Related Work

Many parallel techniques for inference in general graphical models have been proposed. In [11], the authors explore parallelism for inference in Bayesian network. The authors in [21,24] presented map-reduce techniques for parallelizing inference in Conditional Random Fields and in general undirected graphical models. For factor graphs, several parallel techniques for belief propagation have been discussed in [16,7]. 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.

For factor graph used in LDPC decoding, parallel belief propagation is implemented on FPGAs, GPUs, multi-core processors [9,6]. Those techniques are also for cyclic factor graphs with the employment of embarrassingly parallel message passing algorithms.

There are a few earlier papers studying data parallelism in graphical models. In [27], the authors investigated data parallelism for exact inference with respect to Bayesian networks, where a Bayesian network must be first converted into a junction tree. 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 [28], 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 in acyclic 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 of the 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 [28], no explicit data copy is needed during the execution. To the best of our knowledge, no data parallel techniques have been discussed for belief propagation in factor graphs.
4 Data Parallelism in Belief Propagation

4.1 Representation of Potential 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 2, 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 [27]. 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 $F$. The potential values are stored in an order such that index $i$ of a value in $F$ 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 $F$ of $f$ is $|F| = \prod_{k=1}^{d} r_k$. Similarly, the message at $f$ to/from its variable $x_k$ is stored by potential table $X_k$; $|X_k| = r_k$. Index $i$ of an entry in $F$ is determined by the corresponding state of $X_f$ as given in Eq. 6:

$$i = \sum_{k=1}^{d} x_k \prod_{j=1}^{k-1} r_j$$  \hspace{1cm} (6)

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

$$\tilde{i} = \left\lfloor \frac{i}{os_k} \right\rfloor \mod r_k$$  \hspace{1cm} (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 $F$ to the corresponding index $\tilde{i}$ in $X_k$ takes only $O(1)$ time.

Figure 2 illustrates the organization of a potential table $F$ of a factor $f$ with $X_f = \{x_1, x_2, x_3, x_4, x_5\}$. Each variable has $r = 2$ states. Only the potential values of $X_f$, which are in the shaded column, are actually stored. As given by Eqs. 6 and 7, there is a one-to-one mapping between the indices of entries in $F$ and the states of $X_f$. Thus, the state of the variables at an entry is determined by the index of the entry. For example, the entry at index $i = 13$ corresponds to state $(x_1, x_2, x_3, x_4, x_5) = (1, 0, 1, 1, 0)$ of $X_f$. Since the state of $x_1$ is 1, the corresponding index $\tilde{i}$ in $X_1$ of $i$ is also 1.
4.2 Data parallelism in Node Level Primitives

Section 2 shows that 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 $X_k$ sent to $x_k$, $1 \leq k \leq d$, is driven from Eq. 3 as follows:

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

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 $X_k$. Hence, we have two node level primitives for computing a new message:

- **Table multiplication** $\mathcal{F}^* = \mathcal{F} \times \mathcal{X}_j$, realized by:
  $$\mathcal{F}^*[i] = \mathcal{F}[i] \cdot \mathcal{X}_j[i], \forall i \in \{0, 1, ..., |\mathcal{F}| - 1\} \quad (9)$$

- **Table marginalization** $\mathcal{X}_k = \sum_{\mathcal{X}_k \setminus \{x_k\}} (\mathcal{F}^*)$, realized by:
  $$\mathcal{X}_k[i] = \mathcal{X}_k[i] + \mathcal{F}^*[i], \forall i \in \{0, 1, ..., |\mathcal{F}| - 1\} \quad (10)$$

*Fig. 2* Index mapping between the potential table $\mathcal{F}$ of factor node $f(x_1, x_2, x_3, x_4, x_5)$ and the message $X_1$ of variable $x_1$. Only potential values in the shaded area are actually stored.
where the relationship between \( i \) and \( \overline{7} \) 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.

### 4.3 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 \( F \) w.r.t \( f \), message \( X_k \) 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 \( i = \lfloor \frac{|F|}{P} \rfloor p \) to \( \lfloor \frac{|F|}{P} \rfloor (p + 1) - 1 \) do

3. \( \overline{7} = \text{index\_mapping}(i, f, x_k) \) using Eq. 7

4. \( \mathcal{F}^*[i] = \mathcal{F}[i] \ast X_k[\overline{7}] \)

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 \( X_k \) sent from \( x_k \) to \( f \), and the number of threads \( P \). The output is the product of \( \mathcal{F} \) and \( X_k \), 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 \( \lfloor \frac{|F|}{P} \rfloor \) contiguous entries. Lines 3-4 update value of the \( i^{th} \) entry in \( \mathcal{F} \) using the corresponding \( \overline{7}^{th} \) entry in \( X_k \). Function \( \text{index\_mapping}(i, f, x_k) \) computes \( \overline{7} \) using Eq. 7 given that \( r_k \) and \( os_k \) are available for each variable \( x_k \) of factor \( f \).

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 \( X_k \) 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
Algorithm 2 Table marginalization

**Input:** factor \( f \), variable \( x_k \) of \( f \), potential table \( F \), number of threads \( P \)

**Output:** marginal \( \mathcal{X}_k \) for \( x_k \) from \( F \)

1. for \( p = 0 \) to \( P - 1 \) pardo

   \{local computation\}

2. \( \mathcal{X}^{(p)}_k = 0 \)

3. for \( i = \lceil \frac{|F|}{P} \rceil p \) to \( \lceil \frac{|F|}{P} \rceil (p + 1) - 1 \) do

4. \( i = \text{index mapping}(i, f, x_k) \) using Eq. 7

5. \( \mathcal{X}^{(p)}_k[i] = \mathcal{X}^{(p)}_k[i] + F[i] \)

6. end for

7. end for

\{global update for \( \mathcal{X}_k \) from the local results\}

8. \( \mathcal{X}_k = \sum_p \mathcal{X}^{(p)}_k \)

of the \( i^{th} \) entry in \( F \) is used to update the corresponding \( i^{th} \) entry in the local \( \mathcal{X}^{(p)}_k \) 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 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 4.3. Line 4 also takes \( O(1) \) time. Therefore, for \( |F| \geq P \), the complexity of Algorithm 1 is \( O\left(\frac{|F|}{P}\right) \).

In Algorithm 2, Lines 1-7 take \( O\left(\frac{|F|}{P}\right) \) 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 \( O\left(\frac{|\mathcal{X}_k|}{P}\right) \). 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 \( O\left(|\mathcal{X}_k|\right) \) 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 \( O\left(|\mathcal{X}_k| \log P \right) \) time. Thus, for \( |\mathcal{X}_k| \geq \log P \), the complexity of Algorithm 2 is \( O\left(\frac{|F|}{P} + |\mathcal{X}_k|\right) \).

4.4 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 \( F \) of \( f \), number of threads \( P \)

**Output:** message \( X_k \) sent from \( f \) to \( x_k \)

1. \( F^* = F \)
2. \( \text{for } p = 0 \text{ to } P - 1 \) pardo
   
   \{local computation for the product of \( F \) with \( X_j \)\}
   
   3. \( \text{for } j = 1 \text{ to } d, j \neq k \) do
   4. \( \text{Compute } X_j \)
   5. \( \text{for } i = \lfloor \frac{|F|}{P} \rfloor p \text{ to } \lfloor \frac{|F|}{P} \rfloor (p + 1) - 1 \) do
   6. \( i = \text{index mapping}(i, f, x_j) \) using Eq. 7
   7. \( F^*[i] = F^*[i] \times X_j[i] \)
   8. end for
   9. end for

\{local computation for the marginal \( X_k \) from \( F^* \)\}

10. \( X_k^{(p)} = 0 \)
11. \( \text{for } i = \lfloor \frac{|F|}{P} \rfloor p \text{ to } \lfloor \frac{|F|}{P} \rfloor (p + 1) - 1 \) do
12. \( i = \text{index mapping}(i, f, x_k) \) using Eq. 7
13. \( X_k^{(p)}[i] = X_k^{(p)}[i] + F^*[i] \)
14. end for
15. end for

\{global update for \( X_k \) from the local results\}

16. \( X_k = \sum_p X_k^{(p)} \)

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

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 [12]. Based on this order, a queue of directed edges \( Q = \{(f, x)\} \) from factors to variables is formed for the sequence of messages to be computed.

Given a queue of edges, belief propagation in factor graph \( F \) is performed as shown in Algorithm 4. For each directed edge from factor \( f \) to variable \( x \), it retrieves the potential table \( F \) of \( f \) from parameter \( P \). Then, it applies the message computation
Algorithm 4 Belief propagation using message computation kernels

Input: factor graph $F = (G, P)$, edge queue $Q$, number of threads $P$

Output: updated messages on all edges in both directions

1: for $p = 0$ to $P - 1$ par do
2: for $i = 1$ to $|Q|$ do
3: Let $(f, x) = Q(i)$
4: Let $F = P(f)$
5: ComputeMessage($f, x, F$)
6: end for
7: end for

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.

5 Experimental Setup

5.1 Platforms and Implementation Setup

We conducted experiments on a 4-socket Westmere-EX system as representation of the modern multisocket multicore systems. The system consists of four Xeon E7-4860 processors fully connected through 6.4 GT/s QPI links. Each processor has 10 cores, each running at 2.26 GHz, and 16 GB DDR3 memory. Thus, the system has total 64 GB shared memory. Figure 3 shows the detailed architectures of the E7-4860 and the 4-socket system.

Fig. 3 Architectures of (a) a single Intel Westmere-EX processor and (b) the 4-socket Westmere-EX system with 4 processors fully connected via QPIs.
The algorithms were implemented using Pthreads. When the experiments were conducted, the system disabled hyperthreading, supporting one hardware thread per core. In our initial experiments, running multiple threads on each core showed no improvement, if not a decline in the performance. Thus, 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. So as to evaluate the scalability of the algorithms, various numbers of threads were chosen from $1, 2, 4, 6, 8, \ldots, 36, 38, 40$.

5.2 Datasets

We generated cycle-free factor graphs using libDAI, an open source library for probabilistic inference in factor graphs\cite{18}. The factor graphs are generated with modifiable parameters: number of factor nodes $m$, degree of each node $d$, and number of states of each variable $r$. Since computing a message requires $O(d \cdot r^d)$ operations on a potential table of size $r^d$, the computation is more memory-bound with small $d$ and more compute-bound with large $d$. 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 $d = 18$, $d = 20$ and $d = 22$. With node degree $d = 3$, we conducted experiments using various numbers of states of variables $r = 60$, $r = 100$, and $r = 140$. Factor graphs with binary variables are the most popular datasets, while factor graphs with large number of states of variable can be found in bioinformatics or computer vision domain.

5.3 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 on a single core to the execution time on $P$ cores. The serial code for belief propagation in factor graphs was obtained from libDAI. Our parallel code 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.

5.4 An OpenMP Baseline

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 item in table $X_k$ 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. In our implementations, each thread was bound to a separate
core. The dataset was a factor graph with $m = 100$, $r = 2$, $d = 20$. Figure 4 shows the experimental results for the OpenMP baseline and our proposed method on a 10-core Xeon E7-4860 processor. It can be seen that the OpenMP baseline has limited scalability as the number of threads is increased beyond 4. In contrast, our proposed method scales linearly with the number of threads in this experiment.

6 NUMA-Aware Implementations

On a single processor with 10 cores, our proposed technique shows linear speedup. However, since each primitive is a memory-bound computation, a NUMA platform may affect the performance of the primitives significantly. In our non NUMA-aware implementation, we let the master thread allocate all the data including the entire potential tables. When executing the primitives, a thread needs to access to its chunk of data that may locate in remote memory. We address this problem by using the first-touch data allocation policy of the operating system and modifying the code for the primitives correspondingly. We evaluate the effect of NUMA-aware implementation using two different datasets: (1) $r = 100$, $d = 3$ and (2) $r = 2$, $d = 20$, for the same number of factors $m = 100$. These two datasets have similar factor potential table sizes and also similar complexities of the primitives ($100^3$ vs. $2^{20}$). However, computing a message for the second dataset is more computationally expensive ($20 \times 2^{20}$ vs. $3 \times 100^3$).

6.1 Node Level Primitives

In Algorithm 1 for table multiplication, thread $p$ updates its own chunk of the potential table $F^*$, from $|F^*_p|p$ to $|F^*_p|(p + 1) − 1$. In the NUMA-aware implementation, each chunk of a potential table is separately allocated to the memory of the processor of the corresponding thread. The original index $i$ of an entry is maintained for the mapping to the index $\bar{i}$ message $\chi_k$. Figure 5 shows the effect of data placement on table multiplication on the 4-socket Westmere-EX system. For both datasets, the
NUMA-aware implementation shows much better speedup compared with the non
NUMA-aware implementation. Using 40 cores, the speedups of the NUMA-aware
and non NUMA-aware are about 37 and 31, respectively.

![Graph](image)

**Fig. 5** NUMA effects on table multiplication with two different datasets on the 4-socket Intel Westmere-EX based system.

In Algorithm 1 for table marginalization, thread \( p \) reads its own chunk of the
potential table \( F \), from \( \frac{|F|}{|F|} p \) to \( \frac{|F|}{|F|} (p+1) - 1 \), and updates its own message copy \( \Lambda_k^{(p)} \).

In the NUMA-aware implementation, in addition to the data placement of potential
table’s chunks, the message copies are also allocated separately for the threads. Thus,
when executing table marginalization, even though the potential tables are already
accessed by the prior table multiplications, accessing the message copy still affects
the performance of non NUMA-aware implementation. As shown in Figure 6, the
speedup of the non NUMA-aware code for dataset \( r = 100, d = 3 \) is significantly
tapered off as the number of threads exceeds 10, 20, 30. For dataset \( r = 2, d = 20, \)
it is slightly affected because the message size is small \((r = 2)\). The speedup of non NUMA-aware code for these two datasets are 30 and 33.4, respectively. The NUMA-aware implementation achieves good scalability, with a speedup of 36.8 using 40 cores.

6.2 Message Computation Kernel

Data placement has shown significant effects on the performance of the individual primitives. As given in Algorithm 6, computing a message at a factor node \(f\) consists of a series of primitives, including \((d - 1)\) table multiplications and 1 table marginalization. However, with non NUMA-aware implementation, only the first table multiplication needs to access the potential table from the remote memory. The remaining primitives in the series will work on the already loaded chunks of the potential table. Table marginalization only needs to access its message copy from remote memory. Thus, with non NUMA-aware implementation, the performance of computing a message is not significantly affected for factor graphs with large node degree \(d\). Figure 7 shows the comparison of the speedups of message computation between NUMA-aware and non NUMA-aware implementations. The improvement of NUMA-aware implementation is very clear for dataset \(r = 100, d = 3\) but not noticeable for dataset \(r = 2, d = 20\).

Figure 8 summarizes the effect of data placement on the primitives and message computation kernel. With dataset \(r = 100, d = 3\), using 40 cores, the NUMA-aware code is significantly faster than the non NUMA-aware code for the primitives (20% faster) and the message computation (12% faster). With dataset \(r = 2, d = 20\), the improvement of NUMA-aware code is significant for the first table multiplication (20%), noticeable for the table marginalization (10%), and not very clear for the message computation (1.2%).

Fig. 7 NUMA effects on message computation with two different datasets on the 4-socket Intel Westmere-EX based system.
Using NUMA-aware implementations, we evaluated the impact of table size on the performance of each primitive and of the complete belief propagation process. Table size directly reflects the amount of parallelism provided for belief propagation. From Section 4.3, 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 $|X_k| = r$. Double-precision floating-point numbers were used for potential values. Thus, with $r = 2$ and $d = 18, d = 20, d = 22$, the sizes of each potential table are 2 MB, 8 MB, 32 MB respectively. With $d = 3$ and $r = 60, r = 100, r = 140$, the sizes of each potential table are 1.6 MB, 7.6 MB, 20.9 MB respectively. With the number of factors $m = 100$, the total sizes of the datasets are multiplied by 100.
Figure 9 illustrates the execution time of table multiplication. In Figure 9(a) with \( d = 22 \), table multiplication scales almost linearly with the number of threads. It achieves 39.5\( \times \) speedup as 40 threads are used. With \( d = 20 \), table multiplication also shows very good scalability, achieving 37.4\( \times \) speedup as 40 threads are used. With \( d = 16 \) however, the primitive is tapered off especially when the number of threads increases beyond 20. Its speedup is about 30 when 40 threads are used. Similarly in Figure 9(b), table multiplication achieves very good scalability with \( r = 140 \) and \( r = 100 \), but just fair scalability with \( r = 60 \). As 40 threads are used, the speedups for the primitive with \( r = 160, r = 100, r = 60 \) are 39, 37.2, and 29.5 respectively. The scalability of table multiplication are not good with \( r = 2, d = 18 \) and with \( r = 60, d = 3 \) because the potential tables are too small to offer sufficient parallelism.

![Figure 9](image)

**Fig. 9** Performance of table multiplication on the 40-core Intel Westmere-EX based system with (a) \( r = 2 \) and various node degrees, (b) \( d = 3 \) and various numbers of states of the variables.

The execution time of table marginalization is illustrated in Figure 10. Although table marginalization requires the accumulation of the resulting message copies at the end, this primitive still scales very well with the number of threads for large potential tables. This is because in the experiments, \(|F| \gg |X_k|\), making the time for accumulating the message copies very small compared with the entire time for table marginalization. When 40 threads are used, with \( r = 2 \) and \( d = 22, f = 20, f = 18 \), the speedups for the primitive are 39.3, 37.3, 29.8 respectively. With \( d = 3 \) and \( r = 160, r = 100, r = 60 \), the speedups are 38.6, 36.1, and 28 respectively.

The overall execution time of belief propagation on the input factor graphs are shown in Figure 11. 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 40 threads, we observed a 39.2\( \times \) speedup for the factor graphs with \( r = 2, d = 22 \) or \( r = 140, d = 3 \).
Fig. 11 Overall performance of belief propagation on the 40-core Intel Westmere-EX based system with (a) \( r = 2 \) and various node degrees, (b) \( d = 3 \) and various numbers of states of the variables.

8 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 multi-socket multi-core systems. Data placement was used to address NUMA effects from the system. 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 graphs 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.

Acknowledgements Thanks to the management, staff, and facilities of the Intel® Manycore Testing Lab (1,2).

References

1. Intel® Manycore Testing Lab. URL http://www.intel.com/software/manycoretestinglab
2. Intel® Software Network. URL http://www.intel.com/software