Scalable Distributed Belief Propagation with Prioritized Block Updates

Jiangtao Yin
University of Massachusetts Amherst,
Amherst, Massachusetts, USA
jyin@ecs.umass.edu

Lixin Gao
University of Massachusetts Amherst,
Amherst, Massachusetts, USA
lgao@ecs.umass.edu

ABSTRACT

Belief propagation (BP) is a popular method for performing approximate inference on probabilistic graphical models. However, its message updates are time-consuming, and the schedule for updating messages is crucial to its running time and even convergence. In this paper, we propose a new scheduling scheme that selects a set of messages to update at a time and leverages a novel priority to determine which messages are selected. Additionally, an incremental update approach is introduced to accelerate the computation of the priority. As the size of the model grows, it is desirable to leverage the parallelism of a cluster of machines to reduce the inference time. Therefore, we design a distributed framework, Prom, to facilitate the implementation of BP algorithms. We evaluate the proposed scheduling scheme (supported by Prom) via extensive experiments on a local cluster as well as the Amazon EC2 cloud. The evaluation results show that our scheduling scheme outperforms the state-of-the-art counterpart.

Categories and Subject Descriptors

I.2.6 [**Artificial Intelligence**]: Learning

Keywords

Belief Propagation; Distributed Framework; Prioritized Block Updates; Incremental Updates

1. INTRODUCTION

Probabilistic graphical models have been used for reasoning in a wide range of application domains [9, 13, 23, 29, 32]. Inference in these models, including marginalization and maximum a posteriori estimation, forms the basis of many statistical methods in knowledge management. Usually, exact inference in a probabilistic graphical model is NP-hard. As a result, there have been many approaches on introducing both variational and sampling approximations to inference. Among them, loopy belief propagation (BP) and its variants [12, 19, 21, 25] are popular message passing methods for performing approximate inference.

It has been shown that the schedule for updating messages can make a huge difference to the running time of BP algorithms. Specifically, dynamic scheduling schemes, which determine the order of updating messages by the changes of message values, can significantly speedup BP algorithms [6–8, 22]. Although dynamic scheduling schemes have potential to speedup BP algorithms, existing ones cannot fully utilize the potential. Most of them typically select one message for updating each time, e.g., the message with the highest priority value. As a result, many operations need to be performed so as to select next message. That is, the cost of realizing such a dynamic scheduling scheme is high.

In this paper, we propose to select a set of messages instead of a single one to update at a time. Hence, the amortized cost of selecting one message is low. Moreover, a novel priority is leveraged to determine which messages are selected. In other words, we present a *prioritized block scheduling* scheme, which selects a block of messages to update via a priority. The priority allows messages that are more useful towards achieving convergence to be selected, and the computation cost of the priority is low. To this end, we introduce an efficient incremental update mechanism, which propagates only the changes of original messages. The change of a message is efficiently computed using the changes of original incoming messages. Also, the change can be directly utilized to calculate the priority. We refer to this mechanism as an *incremental-update* approach.

As the probabilistic graphical models are applied to model large and complex applications, such as image restoration for high-resolution images, it is desirable to leverage the parallelism of a cluster of machines to reduce the inference time. Therefore, we design and implement a distributed framework, *Prom*, which facilitates the implementation of BP and other graph algorithms in a distributed environment. Prom uses the proposed scheduling scheme as its built-in scheduling and supports the incremental-update approach. We evaluate two BP algorithms, the sum-product algorithm and the max-product algorithm on Prom, on a local cluster of machines as well as the Amazon EC2 cloud [1].

More specifically, our main contributions are as follows:

- We propose a novel scheduling scheme for BP algorithms. It selects a set of vertices to update at a time (in turn, a set of messages are selected, since all its outgoing messages are selected when a vertex is selected). As a result, it performs the selection of vertices for

CIKM'14, November 3–7, 2014, Shanghai, China.

http://dx.doi.org/10.1145/2661829.2662081.

many message updates simultaneously instead of for one message update, and thus reduces the overhead of scheduling (since the amortized cost of selecting one message is low).

- We present a novel priority, which is leveraged to determine which messages are selected. The priority is vertex-based and can well capture the gain of updating a vertex (updating its outgoing messages). In other words, updating a vertex with large priority value will send out highly useful outgoing messages towards achieving convergence. To keep the computation of the priority inexpensive, an incremental-update approach is introduced. The message computed by the incremental-update approach can be directly used to derive priority. Furthermore, the message update in the incremental-update approach can be done by accumulating incoming changes rather than by computing from scratch.

- We develop an asynchronous distributed framework, Prom, to support the proposed scheduling scheme and the incremental-update approach. Prom eases the process of programming BP and other graph algorithms in a distributed environment and does not require users to have distributed programming experience. Prom is evaluated via extensive experiments with both synthetic and real-world data. The evaluation results show that the proposed scheduling scheme outperforms the state-of-the-art counterpart and the incremental-update approach can further boost it. Moreover, a scalability test on a 50-node cluster demonstrates nearly linear scaling performance for large graphical models.

2. BELIEF PROPAGATION

Probabilistic graphical models, such as Bayesian networks, factor graphs, and pairwise Markov Random Fields (MRFs), are popular tools to capture uncertainty in real-world applications. Without loss of generality, we consider factor graphs, since any other graphical models can be converted to factor graphs [13]. A factor graph is a bipartite graph with two types of vertices: variable vertices and factor vertices. Each variable vertex represents a single random variable (e.g., x_i). Each factor vertex (e.g., f_j) denotes a function that maps a subset of random variable values (e.g., X_j) to a non-negative real-valued number so as to capture the compatibility of an assignment to those variables. The arguments are graphically represented by edges, which connect a particular function vertex with its variable vertices. Therefore, a factor graph is a factored representation of a joint probability distribution: $P(x_1, x_2, ..., x_n) = \frac{1}{Z}\prod_{j\in J} f_j(X_j)$, where Z is the normalization constant.

We next briefly review two BP algorithms, the sum-product algorithm and the max-product algorithm, and then discuss asynchronous BP algorithms.

2.1 Sum-Product Algorithm

Marginal probabilities of the distribution represented by a factor graph are central to inference. The sum-product algorithm provides an efficient way to compute marginal probabilities on a factor graph. It propagates messages in both directions along edges. Each vertex sends and receives messages till reaching a stable situation, and then the incoming messages are used to estimate the marginal probabilities of the vertex. Let $\mathfrak{m}_{i\to a}(x_i)$ and $\mathfrak{m}_{a\to i}(x_i)$ denote the message sent from variable vertex x_i to factor vertex f_a and the message sent from f_a to x_i, respectively. They can be updated by the following equations:

$$\mathfrak{m}^t_{i\to a}(x_i) = \lambda \prod_{k\in N(i)\backslash a} \mathfrak{m}^{t-1}_{k\to i}(x_i), \tag{1}$$

$$\mathfrak{m}^t_{a\to i}(x_i) = \lambda \sum_{X_j\backslash x_i} f(X_j) \prod_{j\in N(a)\backslash i} \mathfrak{m}^{t-1}_{j\to a}(x_j), \tag{2}$$

where $N(i)\backslash a$ denotes the set of neighbors of a given vertex i (x_i) excluding vertex a (f_a), and λ is a normalization factor to ensure all elements of the messages sum to 1.

The belief at a variable vertex (e.g., i) is proportional to the product of all the messages coming to the vertex: $b_i(x_i) \propto \prod_{k\in N(i)} \mathfrak{m}_{k\to i}(x_i)$. Then, the estimate of the marginal probability is $P(x_i) \approx b_i(x_i)$. While the sum-product algorithm converges to the exact marginal probabilities in acyclic graphs, there are no guarantees of convergence or correctness for graphs with loops. Nonetheless, the sum-product algorithm is widely applied on cyclic graphs for approximate inference with great success [4, 17, 26].

2.2 Max-Product Algorithm

In some cases, we are interested in determining which valid configuration has the largest probability, rather than determining the marginal probabilities for the individual variables. The max-product algorithm addresses this problem efficiently. Message updates in the max-product algorithm are similar with those in the sum-product algorithm. In fact we only need to replace $\sum$ with max in computing factor-to-variable messages. The message updates in the max-product algorithm are as follows:

$$\mathfrak{m}^t_{i\to a}(x_i) = \lambda \prod_{k\in N(i)\backslash a} \mathfrak{m}^{t-1}_{k\to i}(x_i), \tag{3}$$

$$\mathfrak{m}^t_{a\to i}(x_i) = \lambda \max_{X_j\backslash x_i} f(X_j) \prod_{j\in N(a)\backslash i} \mathfrak{m}^{t-1}_{j\to a}(x_j). \tag{4}$$

2.3 Asynchronous BP

We can represent each message as a vector in the vector space $\mathfrak{S} \subset R^d$, and represent an entire set $\mathfrak{M}$ of messages as a vector in $\mathfrak{S}^{|\mathfrak{M}|}$. The BP algorithm can be considered as the iterative algorithm with an update function $F : \mathfrak{S}^{|\mathfrak{M}|} \to \mathfrak{S}^{|\mathfrak{M}|}$, i.e. $\mathfrak{m}^t = F(\mathfrak{m}^{t-1})$.

BP aims to find a fixed point $\mathfrak{m}^*$ where $\mathfrak{m}^* = F(\mathfrak{m}^*)$. BP is guaranteed to converge to a unique fixed point $\mathfrak{m}^*$, if the update function F is a contraction under a message norm,

$$\|F(\mathfrak{m}) - \mathfrak{m}^*\| \le \alpha\|\mathfrak{m} - \mathfrak{m}^*\|, 0 \le \alpha < 1,$$

where the message norm $\|\cdot\|$ measures the distance between messages. If F is a max-norm contraction, then we have $\|F(\mathfrak{m}) - \mathfrak{m}^*\|_\infty \le \alpha\|\mathfrak{m} - \mathfrak{m}^*\|_\infty$, where the max-norm $\|\cdot\|_\infty$ is defined as the maximum of the individual message norms, $\|\mathfrak{m}^t - \mathfrak{m}^{t-1}\|_\infty = \max_{i,j} \|\mathfrak{m}^t_{i\to j} - \mathfrak{m}^{t-1}_{i\to j}\|$. In this paper, we use the max-norm to measure the convergence of BP. Mooij and Kappen [18] present sufficient conditions for F to be a contraction under the max-norm.

Function F can also be viewed as a set of individual functions, and each individual function F_i applies to one message. These individual update functions can be used to define synchronous BP and asynchronous BP. In *synchronous BP*, the functions compute the new values of all messages simultaneously at every iteration using their values from last

iteration. In *asynchronous BP*, the functions update messages using the most recent values. The convergence rate of asynchronous BP (with a pre-defined update order) is proven to be at least as good as that of synchronous BP [6].

For asynchronous BP, it has been shown that the dynamic scheduling, which uses a priority to determine the order of updating messages, converges much faster than the static scheduling [6–8,22]. The intuition behind the dynamic scheduling is that sending a message whose current value is very different from its previous value is perhaps more useful, and thus leads to more rapid transfer of information across the graph, while sending a message whose value does not change is useless.

3. INCREMENTAL UPDATES

The general techniques of incremental updates have shown efficiency in many algorithms, such as Nonnegative Matrix Factorization [27] and Expectation-Maximization [28]. In this section, we present an incremental update mechanism for BP algorithms, referred to as an *incremental-update* approach. In contrast, the traditional way of updating messages (described in the previous section) is referred to as a *basic-update* approach. The incremental-update approach propagates only the incremental part (change) of the original message. The message update in the incremental-update approach can be performed by accumulating incoming changes instead of computing from scratch, and thus is much more efficient than that in the basic-update approach. Furthermore, since it usually calculates the priority value using the changes of messages, the dynamic scheduling can benefit from the incremental-update approach.

The basic idea of the incremental update is inspired by the Hugin architecture [5], an approach proposed for the exact inference. It uses an efficient way to update messages, which computes the marginal of a vertex as the product of messages once and then divides a message out from the marginal when one needs to update a message. However, the incremental update we proposed aims to support asynchronous computation. The order of asynchronous computations is based on a priority-based scheduling. The message computed by our incremental update can be directly used to derive priority, while there is no concept of priority in the Hugin architecture. Furthermore, our incremental update performs log-space calculations, so it can use addition/subtraction to update messages, while the Hugin architecture uses more expensive multiplication/division.

To derive an incremental update mechanism for a BP algorithm, we treat messages in log-space. A message in log-space is the logarithmic equivalent of the original message, i.e., $m(x_i) = \ln \mathfrak{m}(x_i)$.

3.1 Incremental Updates for Sum-Product

When the messages are in log-space, the message computation for the sum-product algorithm is as follows:

$$m^t_{i\to a}(x_i) = \sum_{k\in N(i)\setminus a} m^{t-1}_{k\to i}(x_i) + \beta, \tag{5}$$

$$m^t_{a\to i}(x_i) = \ln(\lambda \sum_{X_j\setminus x_i} f(X_j) e^{g^{t-1}_{a\to i}(x_j)}), \tag{6}$$

where $m(x_i) = \ln \mathfrak{m}(x_i)$, $\beta = \ln(\lambda)$, and $g^{t-1}_{a\to i}(x_j) = \sum_{j\in N(a)\setminus i} m^{t-1}_{j\to a}(x_j)$. Then, the belief at a variable vertex (e.g., i) can be computed as: $b_i(x_i) \propto e^{\sum_{k\in N(i)} m_{k\to i}(x_i)}$.

We can make a slight modification to Eq. (5) in which we omit normalization factor β. As Pearl [19] pointed out, normalizing the messages is only for avoiding numerical underflow and makes no differences to the final beliefs. Since we still keep the normalization factor in Eq. (6) and messages are in log-space, there is no numerical underflow problem. Then, the message computation can be performed incrementally. The message $m^t_{i\to a}(x_i)$ can be incrementally computed as follows:

$$\Delta m^t_{i\to a}(x_i) = \sum_{k\in N(i)\setminus a} \Delta m^{t-1}_{k\to i}(x_i), \tag{7}$$

$$m^t_{i\to a}(x_i) = m^{t-1}_{i\to a}(x_i) + \Delta m^t_{i\to a}(x_i), \tag{8}$$

where $m^0_{i\to a}(x_i) = 0$, and $\Delta m^0_{k\to i}(x_i) = m^0_{k\to i}(x_i)$ is the initial message.

In our incremental-update approach, a vertex sends the incremental part of the original message instead of the message itself. For example, vertex x_i sends message $\Delta m^t_{i\to a}(x_i)$ to factor vertex f_a. In order to compute the belief, variable vertex x_i also accumulates the messages received from its neighbors, e.g., $m^t_{k\to i}(x_i) = m^{t-1}_{k\to i}(x_i) + \Delta m^t_{k\to i}(x_i)$.

The function $g_{a\to i}(x_j)$ in Eq. (6) can be also incrementally computed. We have

$$\Delta g^t_{a\to i}(x_j) = \sum_{j\in N(a)\setminus i} \Delta m^t_{j\to a}(x_j), \tag{9}$$

$$g^t_{a\to i}(x_j) = g^{t-1}_{a\to i}(x_j) + \Delta g^t_{a\to i}(x_j), \tag{10}$$

where $g^0_{a\to i}(x_j) = 0$.

Then, the incremental message sent from factor vertex f_a to variable vertex x_i can be computed as follows:

$$\begin{aligned}\Delta m^t_{a\to i}(x_i) &= m^t_{a\to i}(x_i) - m^{t-1}_{a\to i}(x_i) \\ &= \ln(\lambda \sum_{X_j\setminus x_i} f(X_j) e^{g^{t-1}_{a\to i}(x_j)}) - m^{t-1}_{a\to i}(x_i),\end{aligned} \tag{11}$$

where $m^0_{a\to i}(x_i)$ is the initial message. Factor vertex f_a also keeps records of $g^{t-1}_{a\to i}(x_j)$ and $m^{t-1}_{a\to i}(x_i)$.

Since the incremental-update approach uses only new incoming incremental messages to compute outgoing incremental messages, the complexity of computing an outgoing message for a vertex depends on the number of new incoming messages the vertex has received (since last update) rather than the vertex's degree. This is highly useful especially in the asynchronous communication model (e.g., under the dynamic scheduling), in which only part of a vertex's incoming messages may be updated when the algorithm computes its outgoing messages. In contrast, the basic-update approach always computes messages from scratch no matter how many incoming messages are updated. Its computation complexity is determined by the vertex's degree.

3.2 Incremental Updates for Max-Product

When the messages are in log-space, the message computation for the max-product algorithm is as follows:

$$m^t_{i\to a}(x_i) = \sum_{k\in N(i)\setminus a} m^{t-1}_{k\to i}(x_i) + \beta, \tag{12}$$

$$m^t_{a\to i}(x_i) = \ln(\lambda \max_{X_j\setminus x_i} f(X_j) e^{g^{t-1}_{a\to i}(x_j)}), \tag{13}$$

where $m^t_{i\to a}(x_i) = \ln \mathfrak{m}^t_{i\to a}(x_i)$, $m^{t-1}_{k\to i}(x_i) = \ln \mathfrak{m}^{t-1}_{k\to i}(x_i)$, $m^t_{a\to i}(x_i) = \ln \mathfrak{m}^t_{a\to i}(x_i)$, $\beta = \ln(\lambda)$, and $g^{t-1}_{a\to i}(x_j) = \sum_{j\in N(a)\setminus i} m^{t-1}_{j\to a}(x_j)$.

The only difference in computing messages between the max-product algorithm and the sum-product algorithm is that the former one replaces $\sum$ with max in computing factor-to-variable messages. As a result, the message update for the max-product algorithm can be performed incrementally as well. Computing the incremental variable-to-factor message is the same with that in the sum-product algorithm (so is $g_{a\to i}(x_j)$). Here, we only show how to incrementally compute the factor-to-variable message. The incremental message sent from factor vertex f_a to variable vertex x_i can be computed as follows:

$$\begin{aligned}\Delta m^t_{a\to i}(x_i) &= m^t_{a\to i}(x_i) - m^{t-1}_{a\to i}(x_i) \qquad (14)\\ &= \ln(\lambda \max_{X_j \setminus x_i} f(X_j) e^{g^{t-1}_{a\to i}(x_j)}) - m^{t-1}_{a\to i}(x_i),\end{aligned}$$

where $m^0_{a\to i}(x_i)$ is the initial message. Factor vertex f_a also keeps records of $g^{t-1}_{a\to i}(x_j)$ and $m^{t-1}_{a\to i}(x_i)$.

Using mathematical induction, it is straightforward to verify that performing message updates traditionally and performing message updates incrementally are equivalent.

4. OUR SCHEDULING SCHEME

In this section, we present our scheduling scheme, which is inspired by the *residual scheduling* [6]. The residual scheduling leverages the difference in values of the message before and after the update as the residual of the message. By giving the message with high residual a high execution priority, the BP algorithm can potentially converge fast. The residual scheduling uses a priority queue to order all outgoing messages' residuals. Every time it sends out the outgoing message with the largest residual in the priority queue and then updates the queue.

The issue of the residual scheduling is that it has high overhead. It always selects one message to update at a time. Once the message is updated, it needs to recompute the priorities of the messages that have been affected and maintain the priority queue so as to select next message. Moreover, the residual scheduling determines a message's priority by actually computing the message. Many messages are computed only for the purpose of obtaining their priority values, and are never sent out. As a result, in order to select one message, many operations have to be performed.

Our scheduling scheme selects a set of messages instead of a single one to update each time so as to reduce the cost. It utilizes a priority to determine which messages are selected. In addition, we also present a novel priority, which allows messages that are more useful towards achieving convergence to be selected (without actually computing the messages in advance).

4.1 Prioritized Block Scheduling

Our scheduling scheme is over vertices. That is, when a selected vertex is updated, all its outgoing messages will be computed and sent out. Scheduling over vertices rather than messages can reduce the cost of selecting messages, since a vertex usually has at least several messages. Updating a vertex always uses the most recently available data (i.e., incoming messages). Our scheduling scheme selects a block of k vertices to update each time. Once the block of selected vertices are updated, it selects another block of vertices to update. A priority is used to determine which vertices are selected. Every time our scheduling scheme selects the top-k vertices in terms of the priority value. Since our scheduling scheme selects a block of vertices to update via a priority, we refer to it as the *prioritized block scheduling*.

The size of the block (i.e., k) balances the tradeoff between the gain from the prioritized block scheduling and the cost of selecting the k vertices. Setting k too small may incur considerable cost, e.g., when $k = 1$, the prioritized block scheduling can be in principle seen as a vertex-based version of the residual scheduling (since it selects one vertex to update at a time). Setting k too large may degrade the effect of the prioritized block scheduling, e.g., if setting k as the number of vertices, it degrades to the round-robin scheduling. We will show in experiments (Section 6.3) that a quite large range of k can allow the prioritized block scheduling to have better performance than the round-robin scheduling.

The prioritized block scheduling uses an efficient way to select the top-k vertices. The naive way is to first sort all the vertices by their priority values and then pick the top ones. However, sorting all the vertices can be expensive and time consuming (at least $O(n \log n)$ time). Instead, the prioritized block scheduling first finds the vertex with the k-th largest priority value. Then, it utilizes the k-th largest priority value as a threshold to filter the vertices. That is, it scans all the vertices once and picks only the vertices with larger or equivalent priority values. Randomized-Select [3] is utilized to find the k-th largest value. It has an expected running time of $O(n)$. In this way, the prioritized block scheduling takes $O(n)$ time (including the time in scanning all the vertices) in extracting the top-k vertices.

Our prioritized block scheduling has much lower cost of selecting one message than the residual scheduling. Updating one message in the residual scheduling needs to reset the message's residual and adjust the dependent messages' residuals (the messages sent from the message's destination vertex). Assuming the degree of the message's destination vertex is d, there are $(d-1)$ dependent messages. We know that adjusting an element's priority value in a priority queue with n elements typically needs $O(\log n)$ time. Given a factor graph with $|V|$ vertices and $|E|$ edges, there are $O(|E|)$ messages in the priority queue. Hence, selecting a message to update in the residual scheduling needs $d * O(\log |E|)$ time, $O(\log |E|)$ for the selected message itself and $(d-1) * O(\log |E|)$ for the $(d-1)$ dependent messages. In our prioritized block scheduling, selecting k vertices to update only needs $O(|V|)$ time. Suppose the averaged degree of these k vertices is d'. Then, $(k * d')$ messages will be updated once the k vertices are selected. As a result, the amortized cost of selecting one message to update in our prioritized block scheduling is $\frac{O(|V|)}{k*d'}$. For a reasonably large k (e.g., k is one tenth of $|V|$), the cost is low and much lower than that in the residual scheduling.

4.2 Priority

We define the residual of an incremental message $\Delta m(x_i)$ as its L^1-norm (in log-space),

$$r(\Delta m) = \sum_{x_i} |\Delta m(x_i)|.$$

Next, we derive the priority utilized in our prioritized block scheduling for the sum-product algorithm and for the max-product algorithm, respectively. The priority is vertex-based, and the priority of a vertex is directly computed from the residuals of its incoming messages.

4.2.1 Priority in Sum-Product

For any outgoing message sending from a variable vertex (e.g., i), its residual can be computed as follows:

$$r(\Delta m_{i\to a}) = \sum_{x_i} |\Delta m^t_{i\to a}(x_i)| = \sum_{x_i} |\sum_{k\in N(i)\setminus a} \Delta m^{t-1}_{k\to i}(x_i)|.$$

Therefore, we use the summation over all assignments of incoming messages in log-space,

$$pr_i = \sum_{x_i} |\sum_{k\in N(i)} \Delta m^{t-1}_{k\to i}(x_i)|,$$

as the priority of a variable vertex (i), which well approximates the residual of each individual outgoing message of the variable vertex.

For any outgoing message sending from a factor vertex (e.g., a), its residual can be computed as follows:

$$\begin{aligned} r(\Delta m_{a\to i}) &= \sum_{x_i} |\Delta m^t_{a\to i}(x_i)| \\ &= \sum_{x_i} |\ln \frac{\sum_{X_j\setminus x_i} f(X_j) e^{g^{t-1}_{a\to i}(x_j)}}{\sum_{X_j\setminus x_i} f(X_j) e^{g^{t-2}_{a\to i}(x_j)}}|. \end{aligned}$$

Applying the fact for any $y_1 > 0, y_2 > 0, z_1 > 0, z_2 > 0$, $\frac{y_1+y_2}{z_1+z_2} \le \max\{\frac{y_1}{z_1}, \frac{y_2}{z_2}\}$, we have

$$\begin{aligned} r(\Delta m_{a\to i}) &\le \sum_{x_i} |\ln \max_{X_j\setminus x_i} \frac{f(X_j) e^{g^{t-1}_{a\to i}(x_j)}}{f(X_j) e^{g^{t-2}_{a\to i}(x_j)}}| \\ &\le \sum_{x_i} |\max_{X_j\setminus x_i} \ln \frac{f(X_j) e^{g^{t-1}_{a\to i}(x_j)}}{f(X_j) e^{g^{t-2}_{a\to i}(x_j)}}| \\ &\le \sum_{x_i} \max_{X_j\setminus x_i} |\ln \frac{e^{g^{t-1}_{a\to i}(x_j)}}{e^{g^{t-2}_{a\to i}(x_j)}}| \\ &= \sum_{x_i} \max_{X_j\setminus x_i} |\Delta g^{t-1}_{a\to i}(x_j)| \\ &= \sum_{x_i} \max_{X_j\setminus x_i} |\sum_{j\in N(a)\setminus i} \Delta m^{t-1}_{j\to a}(x_j)|. \end{aligned}$$

Applying the fact for any $y_1 > 0, y_2 > 0, z_1 > 0, z_2 > 0$, $\frac{y_1+y_2}{z_1+z_2} \ge \min\{\frac{y_1}{z_1}, \frac{y_2}{z_2}\}$, we have

$$\begin{aligned} r(\Delta m_{a\to i}) &\ge \sum_{x_i} |\ln \min_{X_j\setminus x_i} \frac{f(X_j) e^{g^{t-1}_{a\to i}(x_j)}}{f(X_j) e^{g^{t-2}_{a\to i}(x_j)}}| \\ &\ge \sum_{x_i} |\min_{X_j\setminus x_i} \ln \frac{f(X_j) e^{g^{t-1}_{a\to i}(x_j)}}{f(X_j) e^{g^{t-2}_{a\to i}(x_j)}}| \\ &\ge \sum_{x_i} \min_{X_j\setminus x_i} |\ln \frac{e^{g^{t-1}_{a\to i}(x_j)}}{e^{g^{t-2}_{a\to i}(x_j)}}| \\ &= \sum_{x_i} \min_{X_j\setminus x_i} |\Delta g^{t-1}_{a\to i}(x_j)| \\ &= \sum_{x_i} \min_{X_j\setminus x_i} |\sum_{j\in N(a)\setminus i} \Delta m^{t-1}_{j\to a}(x_j)|. \end{aligned}$$

We have derived the lower bound and the upper bound for $r(\Delta m_{a\to i})$. Then, we use a value between these two bounds to approximate $r(\Delta m_{a\to i})$. Let $v_{a\to i} = \sum_{x_i} \frac{1}{s} \sum_{X_j\setminus x_i} |\sum_{j\in N(a)\setminus i} \Delta m^{t-1}_{j\to a}(x_j)|$, where s is the number of possible states of $X_j\setminus x_i$. We can see that (since $v_{a\to i}$ is the average) $v_{a\to i}$ is between those bounds. Therefore, we use $v_{a\to i}$ to approximate $r(\Delta m_{a\to i})$, and use the summation of averaged values over all assignments of incoming messages in log-space,

$$pr_a = \sum_{x_i} \frac{1}{s} \sum_{X_j\setminus x_i} |\sum_{j\in N(a)} \Delta m^{t-1}_{j\to a}(x_j)|,$$

as the priority of a factor vertex (a). Intuitively, this priority well captures the importance of new incoming messages available to the factor vertex.

4.2.2 Priority in Max-Product

The message update for a variable vertex in the max-product algorithm is the same with that in the sum-product algorithm. Accordingly, the priority for a variable vertex defined in the sum-product algorithm also applies to the max-product algorithm. Next, we derive the priority for a factor vertex in the max-product algorithm.

For any outgoing message sending from a factor vertex (e.g., a), its residual can be computed as follows:

$$\begin{aligned} r(\Delta m_{a\to i}) &= \sum_{x_i} |\Delta m^t_{a\to i}(x_i)| \\ &= \sum_{x_i} |\ln \frac{\max_{X_j\setminus x_i} f(X_j) e^{g^{t-1}_{a\to i}(x_j)}}{\max_{X_j\setminus x_i} f(X_j) e^{g^{t-2}_{a\to i}(x_j)}}|. \end{aligned}$$

Applying the fact for any $y_1 > 0, y_2 > 0, z_1 > 0, z_2 > 0$, $\frac{\max\{y_1,y_2\}}{\max\{z_1,z_2\}} \le \max\{\frac{y_1}{z_1}, \frac{y_2}{z_2}\}$, we can derive the following inequations:

$$\begin{aligned} r(\Delta m_{a\to i}) &= \sum_{x_i} |\ln \frac{\max_{X_j\setminus x_i} f(X_j) e^{g^{t-1}_{a\to i}(x_j)}}{\max_{X_j\setminus x_i} f(X_j) e^{g^{t-2}_{a\to i}(x_j)}}| \\ &\le \sum_{x_i} \max_{X_j\setminus x_i} |\ln \frac{f(X_j) e^{g^{t-1}_{a\to i}(x_j)}}{f(X_j) e^{g^{t-2}_{a\to i}(x_j)}}| \\ &= \sum_{x_i} \max_{X_j\setminus x_i} |\sum_{j\in N(a)\setminus i} \Delta m^{t-1}_{j\to a}(x_j)|. \end{aligned}$$

Applying the fact for any $y_1 > 0, y_2 > 0, z_1 > 0, z_2 > 0$, $\frac{\max\{y_1,y_2\}}{\max\{z_1,z_2\}} \ge \min\{\frac{y_1}{z_1}, \frac{y_2}{z_2}\}$, we can derive the following inequations:

$$\begin{aligned} r(\Delta m_{a\to i}) &= \sum_{x_i} |\ln \frac{\max_{X_j\setminus x_i} f(X_j) e^{g^{t-1}_{a\to i}(x_j)}}{\max_{X_j\setminus x_i} f(X_j) e^{g^{t-2}_{a\to i}(x_j)}}| \\ &\ge \sum_{x_i} \min_{X_j\setminus x_i} |\ln \frac{f(X_j) e^{g^{t-1}_{a\to i}(x_j)}}{f(X_j) e^{g^{t-2}_{a\to i}(x_j)}}| \\ &= \sum_{x_i} \min_{X_j\setminus x_i} |\sum_{j\in N(a)\setminus i} \Delta m^{t-1}_{j\to a}(x_j)|. \end{aligned}$$

From the above inequations, we can see that the max-product algorithm has the same bounds for the residual of an outgoing message sending from a factor vertex as the sum-product algorithm. Accordingly, the priority for a factor vertex defined in the sum-product algorithm applies to the max-product algorithm as well.

The defined priority uses summation to aggregate incoming messages, and thereby we call it the *sum priority*. From the above derivation, we can see that the sum priority has strong connections with the residuals of its outgoing messages and thus well captures the gain of updating the vertex. That is, updating a vertex with large sum priority will send out highly useful outgoing messages. In contrast, updating

a vertex with zero sum priority will waste a update, since the outgoing messages will not change.

4.3 Convergence

The prioritized block scheduling guarantees that BP algorithms converge if update function F is a max-norm contraction. It has been shown that when F is a max-norm contraction, if a scheduling scheme can guarantee that every message is updated infinitely often (until convergence), the BP algorithm will converge [6]. We first show that our prioritized block scheduling can fulfill this requirement.

LEMMA 4.1. *If update function F is a max-norm contraction, the prioritized block scheduling guarantees that every message is updated infinitely often.*

PROOF. We prove this lemma by contradiction. Assume there are a set of messages that belong to (sent from) a set of vertices, C, which are updated only before a time point t. We use pr_i to denote the priority value of vertex i. Since update function F is a contraction, the messages that are updated will move towards their fixed points. Consequently, at some time point after t, for any vertex that does not belong to C (i.e., $i \in (V - C)$, where V is the whole set of vertices), its outgoing messages can reach the fixed points (since they are always being updated). At that time, for any $i \in (V - C)$, we have $pr_i = 0$; if we also have $pr_i = 0$ for any $i \in C$, the BP algorithm has converged; otherwise, a vertex in C (e.g., j, $pr_j > 0$) must be selected to update, which contradicts with the assumption that any vertex in C is updated only before time point t. □

Therefore, we have the following theorem.

THEOREM 4.2. *If update function F is a max-norm contraction, BP algorithms with the prioritized block scheduling converge.*

5. DISTRIBUTED FRAMEWORK

BP algorithms and its variants are commonly used to perform inference on large real-world probabilistic graphical models. It is desirable to leverage the parallelism of a cluster of machines to reduce the completion time, and to have a general framework to facilitate the implementation in a distributed environment. BP algorithms (and its many extensions) are graph algorithms. Actually, graph algorithms have become an essential component in knowledge discovery, since graphs can capture complex dependencies and interactions. Therefore, we propose *Prom*, an asynchronous distributed framework for graph algorithms.

Prom provides several high-level APIs to users for implementing BP or other graph algorithms without worrying about the complexity of parallel computation. Prom supports asynchronous executions on graphs, in which vertices are updated using the latest available values, and leverages the proposed prioritized block scheduling as its default scheduling in order to efficiently order vertex updates.

Prom is built upon *Maiter* [31], an open-source graph processing framework. Maiter has shown good performance for several graph algorithms. In Maiter, users specify the application logic simply through a vertex update function. However, Maiter assumes that each vertex (or message) has only one scalar value (e.g. a floating-point number), and thus cannot support algorithms with vector values, such as BP and Personalized PageRank [10]. Additionally, Maiter assumes that the update function has only one operation (e.g., addition) with commutative and associative properties, but there are many graph algorithms with more than one operations in the update function (e.g., sum-product has addition and multiplication). These limitations need to be removed so as to accommodate more graph algorithms. To this end, Prom extends Maiter to support a broader class of graph algorithms efficiently. Prom makes two basic assumptions: (1) the graph structure is static and will not change during execution; (2) asynchronous execution with dynamically ordering vertex updates does not affect the correctness of the algorithm. Graph algorithms satisfying these two assumptions can be implemented on Prom and can benefit from the efficient prioritized block scheduling.

A vertex-centric programming model (which has been shown to be efficient for many graph algorithms) is adopted by Prom. That is, each vertex is considered as an independent computing unit, and the operations are performed over vertices until termination. Vertex updates are performed on workers, and there is a master controlling the flow of computation. All workers (and the master) run in parallel and communicate through MPI.

5.1 Data Partition and Storage

The input graph is split into partitions and each worker is responsible for one partition. Each partition consists of a set of vertices and all their (outgoing) edges. Each worker leverages an in-memory table, *info table*, to store the vertices in its partition. For graph algorithms under the vertex-centric programming model, storing the following information is typically sufficient for a vertex: ID, incoming messages, outgoing messages, priority, state, and edges (with edge data associated with each edge). Hence, as shown in Figure 1, Prom represents a vertex by a tuple with six fields, $\{v, im, om, pr, st, sd\}$, where field v for the vertex ID, im for the incoming messages, om for the outgoing messages, pr for the priority value, st for the state, and sd for the static data (e.g., edges and their associated data).

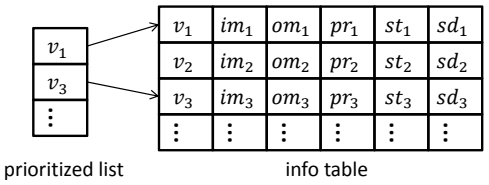


Figure 1: Data storage in a worker.

Prom allows users to define each field of the info table. For example, to implement the incremental-update approach for BP, we can define the incoming message field (im) of a vertex with $[\Delta m_{v_a}, \Delta m_{v_b}, \ldots, \Delta m_{v_l}, m_{v_a}, m_{v_b}, \ldots, m_{v_l}]$ (each item can be a vector), where Δm_{v_a} stores the new incoming incremental message from neighbor v_a, and m_{v_a} accumulates the incoming messages already received from v_a. The static data (sd) is usually defined to contain edges and the data associated with edges (e.g., factor functions of the factor graph). Each tuple is stored in one entry of the info table, which is indexed by the vertex ID (v).

5.2 Vertex Operation

Each worker has two main operations for its stored vertices: the catch operation and the update operation. The

catch operation uses a user-defined function ($c\_fun()$) to aggregate a new incoming message for a vertex (say v_j) to its stored incoming messages. That is, function $c\_fun()$ needs to update the incoming message field (im_j) of vertex v_j, upon receiving a new incoming message. Also, it needs to update the priority field (pr_j) to aggregate the importance of the new incoming message. By defining function $c\_fun()$ in different ways, users can realize different update approaches (e.g., incremental-update or basic-update) and priorities.

The update operation uses another user-defined function ($u\_fun()$) to compute outgoing messages (and the state) for scheduled vertices. When it is performed on a vertex, function $u\_fun()$ computes outgoing messages and updates the state (e.g., the belief distribution of the vertex) by incorporating the latest incoming messages, and modifies the incoming message field if necessary as well as resets the priority value to zero.

Prom uses MPI to transmit messages between workers. All messages during transmission are in the format (dst, src, cnt), where dst denotes the message's destination vertex, src indicates the source vertex, and cnt denotes the message's content. The catch operation and the update operation are realized in two threads for asynchronous execution.

5.3 Distributed Prioritized Execution

Prom leverages the prioritized block scheduling (described in Section 4.1) as its default scheduling scheme. Since a centralized ordering is inefficient in a distributed environment, Prom allows each worker to build its own prioritized block scheduling. Round by round, each worker selects its local top-k vertices in terms of the priority value as a block to update. All workers selects vertices independently.

A worker puts the block of selected vertices into a list, *prioritized list*. To minimize the copy cost, only vertex IDs are put in the prioritized list, as shown in Figure 1. Vertex IDs are used to locate corresponding vertices in the info table. All the vertices in the prioritized list will be updated by the update operation during the round. In the first round, all vertices are put into the prioritized list to guarantee that each vertex is updated at least once before convergence.

5.4 Distributed Termination Check

Prom adopts a passively monitoring model to perform termination check. Each worker utilizes a user-defined function ($m\_fun()$) to periodically measure its local progress by scanning the info table (typically looking at the incoming message field), and reports the progress to the master. The master aggregates the local progress reports from workers (in the way that a user specifies) so as to obtain the global progress, and in turn determines whether the termination condition is satisfied. If yes, the master sends termination signals to all workers. Upon receiving the terminate signal, a worker stops updating its info table and dumps the table to a distributed file system (i.e., HDFS) so as to reliably store the converged results.

We use the following convergence criterion (max-norm) for BP algorithms (where $\varepsilon \geq 0$ is a small constant):

$$\max_{i,j} \|\Delta m_{i \to j}\|_1 \leq \varepsilon.$$

6. EVALUATION

In this section, we evaluate the proposed prioritized block scheduling and the priority. Both the sum-product algorithm and the max-product algorithm are implemented on Prom. For the comparison purpose, both the incremental-update approach and the basic-update approach are used. To show the performance of the prioritized block scheduling, we compare it with the round-robin scheduling (static scheduling). We also compare the prioritized block scheduling with the state-of-the-art dynamic scheduling.

6.1 Experiment Setup

The experiments are performed on a local cluster and a large-scale cluster on Amazon EC2 [1]. The local cluster consists of 4 machines, and each of them has Intel E8200 dual-core 2.66GHz CPU, 4GB of RAM, and 1TB of hard disk. These 4 machines are connected through a Gbit switch. The large-scale cluster consists of 50 medium instances.

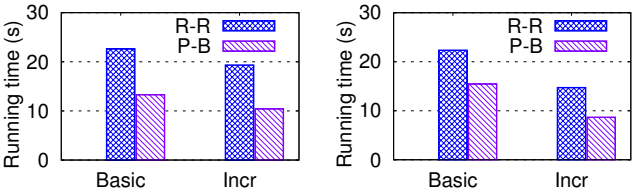
Table 1: Factor Graph Summary

Dataset	# of Vertices	Description
gird-n	$4 * n^2 - 2 * n$	$n \times n$ grid MRF
uw-theory	133,999	uw-theory MLN
uw-systems	414,340	uw-systems MLN

Both synthetic and real-world factor graphs are used. We generate one type of pairwise MRFs, random grids with binary variables (parameterized by the Ising model) [6], and convert them into factor graphs. Random grids are chosen because they are standard benchmarks for evaluating BP algorithms. For real-world graphs, we consider Markov Logic Networks (MLNs) [20]. *Alchemy* is leveraged to compile the MLNs from the UW-CSE data collection [2] into factor graphs. After compiling, the factor functions will be adjusted if BP algorithms on the compiled graphs do not converge. The factor graphs are summarized in Table 1. In order to load the strongly connected vertices to the same worker and thus reduce across-worker communication, we utilizes METIS [11] to split a graph into partitions.

Each worker by default sets k as 10% of the number of its local vertices. The convergence criterion is set to $\varepsilon = 10^{-4}$. Running times are averaged over 10 runs.

6.2 Efficiency of Prioritized Block Scheduling



(a) Sum-Product on grid-200 (b) Max-Product on grid-100

Figure 2: BP algorithms with different scheduling schemes and update approaches.

We first show the running time of BP algorithms with the prioritized block scheduling on the local cluster. The running time is measured as the wall-clock time that BP uses to reach the convergence criterion. The round-robin scheduling is also evaluated as a reference point. For the sum-product algorithm as well as the max-product algorithm, the prioritized block scheduling is faster than the round-robin scheduling with either the incremental-update approach or the basic-update approach, as presented in Figure 2. For example, the prioritized block scheduling is 1.9x faster for the sum-product algorithm on *grid-200* when the incremental-update approach is utilized. In addition, the

incremental-update approach is always superior to the basic-update approach. Note that, in all figures, "P-B" indicates the prioritized block scheduling; "R-R" represents the round-robin scheduling; "Incr" and "Basic" denote the incremental-update approach and the basic-update approach, respectively.

Table 2: Vertex degree comparison

Graph	overall avg. deg.	variable avg. deg.
gird-200	2.5	5.0
uw-theory	3.8	55.7
uw-systems	3.8	78.8

To further show the advantage of the prioritized block scheduling, we evaluate both scheduling schemes for the sum-product algorithm on real-world factor graphs. The performance comparison for the max-product algorithm is similar and therefore omitted here due to space limitations. As plotted in Figure 3, the speedup of the prioritized block scheduling over the round-robin scheduling is up to 2.1x on real-world factor graphs (when the incremental-update approach is used). Moreover, compared with the basic-update approach, the incremental-update approach allows the prioritized block scheduling to achieve up to 4x speedup, much higher than that on the synthetic factor graphs (Figure 2a). The different speedups can be attributed to different structures of the factor graphs. For instance, the real-world factor graphs have much higher degrees for variable vertices, as shown in Table 2.

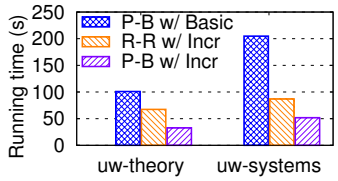


Figure 3: Prioritized block scheduling on real-world graphs.

We also measure the convergence speed of the different scheduling schemes (when the incremental-update approach is used). The test is performed on the real-world factor graph, *uw-theory*, and the max-norm ($\max_{i,j} |\Delta m_{i\to j}(x_i)|$) is used to measure the convergence progress. As shown in Figure 4, the prioritized block scheduling converges much more rapidly than the round-robin scheduling.

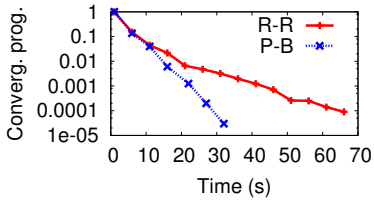
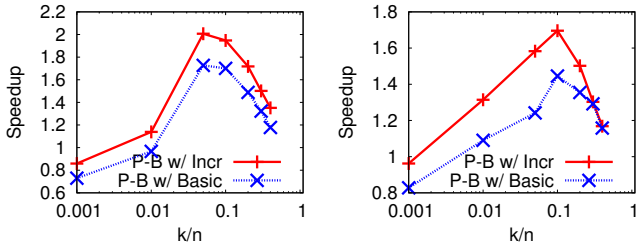


Figure 4: Convergence progress vs. time.

6.3 Impact of k

The block size (i.e., k) balances the tradeoff between the gain from the prioritized block scheduling and the cost of preparing the prioritized list. Figure 5 shows the convergence speedup results with different k. The speedup is measured over the running time when k is the number (n) of a worker's local vertices (i.e., the round-robin scheduling). From the figures, we can see that a quite large range of k can allow the prioritized block scheduling to have better performance than the round-robin scheduling (when either the incremental-update approach or the basic-update approach is used), and that the optimal speedup happens at around $k/n = 0.1$. This is also why we set $k/n = 0.1$ by default.

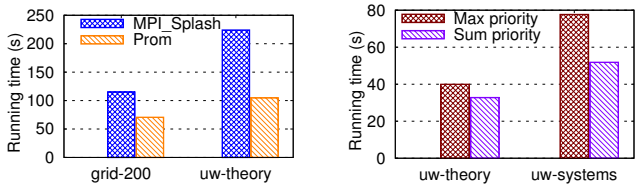


(a) Sum-Product on grid-200 (b) Max-Product on grid-100

Figure 5: The impact of k (varying k/n).

6.4 Comparison with Other Schedules

To further demonstrate the efficiency of its built-in prioritized block scheduling, Prom is also compared with another distributed implementation of the sum-product algorithm, MPI_Splash [8], on the local cluster. MPI_Splash utilizes the *DBRSplash scheduling*, a distributed version of the *ResidualSplash scheduling* [7]. The ResidualSplash scheduling applies a variation of the residual scheduling in a single machine (multiple-core) environment, and it has been shown that ResidualSplash is more efficiently than the original residual scheduling. By recognizing the high overhead of the residual scheduling, ResidualSplash also defines the residual over vertices instead of messages and selects a set of vertices to update at a time via a Splash operation. The Splash operation uses the vertex with the largest residual as a root and updates vertices around the root. However, not all vertices covered by the Splash operation have large residuals, and thus some updates might not be useful. ResidualSplash defines a vertex's priority as the maximum of the residuals of its incoming messages. To differentiate this priority with our sum priority, we refer to it as the *max priority*. The DBRSplash scheduling is the state-of-the-art dynamic scheduling for BP in a distributed environment.



(a) Different schedules (b) Different priorities

Figure 6: Performance comparison with the state-of-the-art dynamic scheduling.

For fairness, Prom uses the same priority and termination condition as MPI_Splash. To compare scheduling schemes only, we leverage the basic-update approach to implement the sum-product algorithm on Prom. As presented in Figure 6a, Prom can be up to 2x faster than MPI_Splash, indicating that the prioritized block scheduling outperforms DBRSplash. In order to verify that the superiority of Prom over MPI_Splash stems from its scheduling scheme, we implement both the prioritized block scheduling and the ResidualSplash scheduling (single machine version of DBRSplash) in a single machine environment and evaluate them with the same

settings. The prioritized block scheduling is 1.8x faster on *grid-200* and 2.3x faster on *uw-theory* than the Residual-Splash scheduling.

In order to show the performance of our sum priority, we compare it with the max priority. We evaluate these two priorities (when both are utilized by the prioritized block scheduling) for the sum-product algorithm on real-world graphs. As presented in Figure 6b, the prioritized block scheduling with our sum priority is 1.2x faster on *uw-theory* and 1.5x faster on *uw-systems* than that with the max priority.

6.5 Accuracy

We also assess accuracy of the beliefs computed by Prom (using the prioritized block scheduling with the incremental-update approach) for the sum-product algorithm. We first compare with the exact result. Since exact inference is intractable on large graphical models, we here use a small factor graph, *grid-10*. The beliefs (of all variable vertices) computed by Prom are compared against the exact beliefs computed by the junction tree algorithm [14]. We use MPI_Splash as a reference point. Kullback-Leibler (KL) divergence is leveraged to measure the difference. From Figure 7, we can see that both Prom and MPI_Splash achieve high accuracy. For example, for more than 90% variable vertices, the KL divergence of the beliefs computed by Prom from the exact beliefs is less than 0.01.

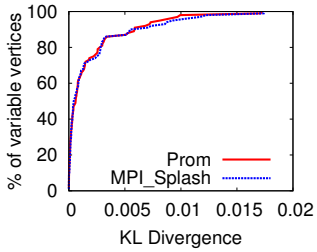
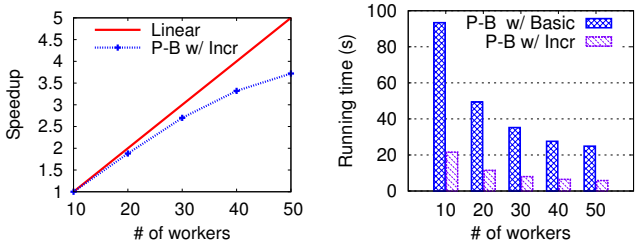


Figure 7: Cumulative percentage of variable vertices as a function of the KL divergence.

For large graphs, since exact inference is intractable, we only compare Prom with MPI_Splash. We evaluate both Prom and MPI_Splash on *grid-200*. Beliefs from both systems are compared by calculating the L^1 difference averaged over all variable vertices. The difference in beliefs computed by the two systems is less than 0.02 in terms of averaged L^1 per variable vertex.

6.6 Scaling Performance



(a) Speedup (b) Performance comparison

Figure 8: Scalability test on us-systems.

Figure 8 presents the scaling performance of the prioritized block scheduling (for the sum-product algorithm) on Prom as the number of workers increases from 10 to 50 on the Amazon EC2 cloud. The real-world factor graph, *us-systems*, is used. The speedup is calculated over the running time of 10 workers. We can see that the prioritized block scheduling exhibits nearly linear speedup, and that it always converges faster when the incremental-update approach is utilized than when the basic-update approach is utilized.

7. RELATED WORK

Several works [6–8, 22] have shown that BP algorithms with the dynamic scheduling converge faster than those with the static scheduling. The earliest work [6] proposes the *residual scheduling*, which selects the outgoing message with largest residual to update each time. It uses a priority queue to order messages. Besides the large priority queue maintenance overhead, the problem of the residual scheduling is that it determines an outgoing message's residual by actually computing it. Later, Sutton and McCallum [22] propose to approximate the residual of an outgoing message rather than compute it in order to reduce the computation overhead. However, the cost of ordering messages so as to select the one with the largest residual is still high. Our prioritized block scheduling scheme selects a set of messages to update each time in order to reduce the cost.

The *ResidualSplash scheduling* [7] applies a variation of the residual scheduling in the multiple-core environment. It defines the residual over vertices instead of messages. The residual of a vertex is used to determine the Splash ordering, and a Splash operation uses the vertex with the largest residual as a root and propagates messages around the root (i.e., among the neighbors within fixed number hops). That is, it selects a set of messages to update at a time. The Residual-Splash scheduling outperforms the residual scheduling, since it reduces the cost of selecting one single message. However, not all vertices covered by the Splash operation have large residuals, and thus some updates might not be useful. The *DBRSplash scheduling* [8] extends the idea of the Residual-Splash scheduling to a distributed environment. In contrast, our prioritized block scheduling selects vertices with high residuals uniformly, and therefore all scheduled updates are potentially useful.

Since massive graphs become increasingly popular, a series of parallel frameworks have emerged to scale graph processing. Among them, *Priter* [30], *Maiter* [31], *GRACE* [24], and *GraphLab* [15, 16] support prioritized execution. Priter is a MapReduce-based framework, which requires synchronous iterations. Maiter presents asynchronous execution but assumes that each vertex (or message) has only one scalar value. As a result, none of them supports BP with dynamic scheduling. GRACE and GraphLab can support BP. GRACE relies on users to implement their own scheduling schemes and its prototype is built on a shared-memory architecture. GraphLab is the first framework to use a general asynchronous model for graph algorithms and provides the Splash scheduling (based on ResidualSplash) for BP. In comparison, Prom provides a more efficient scheduling scheme, the prioritized block scheduling.

8. CONCLUSIONS

In this paper, we propose an efficient dynamic scheduling scheme, the prioritized block scheduling, with a novel prior-

ity for BP algorithms. In order to efficiently compute the priority and update messages, we introduce an incremental-update approach, which is much more efficient than the traditional basic-update approach. In addition, to facilitate the implementation of BP algorithms and other graph algorithms in a distributed environment, we design and implement an asynchronous distributed framework, Prom. Prom uses the prioritized block scheduling as its default scheduling scheme. We implement two BP algorithms, the sum-product algorithm and the max-product algorithm, on Prom. With both synthetic and real-world data, the evaluation results show that the prioritized block scheduling outperforms the state-of-the-art dynamic scheduling scheme, and that the incremental-update approach can further accelerate the prioritized block scheduling.

Acknowledgments

We would like to thank anonymous reviewers for their insightful comments and suggestions. This work is partially supported by NSF grants CNS-1217284 and CCF-1018114. Any opinions, findings, conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the sponsor.

9. REFERENCES

[1] Amazon Elastic Compute Cloud (Amazon EC2). `http://aws.amazon.com/ec2/`.

[2] UW-CSE MLN. `http://alchemy.cs.washington.edu/mlns/uw-cse/`.

[3] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, 3rd edition, 2009.

[4] C. Crick and A. Pfeffer. Loopy belief propagation as a basis for communication in sensor networks. In *UAI '03*, pages 159–166, 2003.

[5] A. Darwiche. *Modeling and Reasoning with Bayesian Networks*. Cambridge University Press, 2009.

[6] G. Elidan, I. McGraw, and D. Koller. Residual belief propagation: Informed scheduling for asynchronous message passing. In *UAI '06*, pages 165–173, 2006.

[7] J. E. Gonzalez, Y. Low, and C. Guestrin. Residual splash for optimally parallelizing belief propagation. In *AISTATS '09*, pages 177–184, 2009.

[8] J. E. Gonzalez, Y. Low, C. Guestrin, and D. O'Hallaron. Distributed parallel inference on large factor graphs. In *UAI '09*, pages 203–212, 2009.

[9] J. Ha, S.-H. Kwon, S.-W. Kim, C. Faloutsos, and S. Park. Top-N recommendation through belief propagation. In *CIKM '12*, pages 2343–2346, 2012.

[10] G. Jeh and J. Widom. Scaling personalized web search. In *WWW '03*, pages 271–279, 2003.

[11] G. Karypis and V. Kumar. Multilevel k-way partitioning scheme for irregular graphs. *Journal of Parallel and Distributed Computing*, 48:96–129, 1998.

[12] K. Kersting, B. Ahmadi, and S. Natarajan. Counting belief propagation. In *UAI '09*, pages 277–284, 2009.

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

[14] S. L. Lauritzen and D. J. Spiegelhalter. Local computations with probabilities on graphical structures and their application to expert systems. *Journal of the Royal Statistical Society. Series B (Methodological)*, 50(2):pp. 157–224, 1988.

[15] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein. Distributed GraphLab: A framework for machine learning and data mining in the cloud. *Proc. VLDB Endow.*, 5(8):716–727, Apr. 2012.

[16] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. GraphLab: A new parallel framework for machine learning. In *UAI '10*, 2010.

[17] R. McEliece, D. J. C. MacKay, and J.-F. Cheng. Turbo decoding as an instance of pearl's "belief propagation" algorithm. *Selected Areas in Communications, IEEE Journal on*, 16(2):140–152, Sept. 2006.

[18] J. Mooij and H. Kappen. Sufficient conditions for convergence of loopy belief propagation. In *UAI '05*, pages 396–403, 2005.

[19] J. Pearl. *Probabilistic reasoning in intelligent systems: networks of plausible inference*. Morgan Kaufmann Publishers Inc., 1988.

[20] M. Richardson and P. Domingos. Markov logic networks. *Mach. Learn.*, 62(1-2):107–136, Feb. 2006.

[21] L. Song, A. Gretton, D. Bickson, Y. Low, and C. Guestrin. Kernel belief propagation. In *AISTATS '11*, pages 333–341, 2011.

[22] C. Sutton and A. McCallum. Improved dynamic schedules for belief propagation. In *UAI '07*, pages 376–383, 2007.

[23] D. Z. Wang, E. Michelakis, M. N. Garofalakis, and J. M. Hellerstein. BAYESSTORE: Managing large, uncertain data repositories with probabilistic graphical models. *PVLDB*, 1(1):340–351, 2008.

[24] G. Wang, W. Xie, A. Demers, and J. Gehrke. Asynchronous large-scale graph processing made easy. In *CIDR '13*, 2013.

[25] J. S. Yedidia, W. T. Freeman, and Y. Weiss. Exploring artificial intelligence in the new millennium. chapter Understanding Belief Propagation and Its Generalizations, pages 239–269. Morgan Kaufmann Publishers Inc., 2003.

[26] J. S. Yedidia, W. T. Freeman, and Y. Weiss. *Understanding belief propagation and its generalizations*. Morgan Kaufmann Publishers Inc., 2003.

[27] J. Yin, L. Gao, and Z. M. Zhang. Scalable nonnegative matrix factorization with block-wise updates. In *ECML/PKDD '14*, pages 337–352, 2014.

[28] J. Yin, Y. Zhang, and L. Gao. Accelerating expectation-maximization algorithms with frequent updates. In *CLUSTER '12*, pages 275–283, 2012.

[29] X. Yu, W. Lam, and B. Chen. An integrated discriminative probabilistic approach to information extraction. In *CIKM '09*, pages 325–334, 2009.

[30] Y. Zhang, Q. Gao, L. Gao, and C. Wang. PrIter: A distributed framework for prioritized iterative computations. In *SOCC '11*, pages 13:1–13:14, 2011.

[31] Y. Zhang, Q. Gao, L. Gao, and C. Wang. Accelerate large-scale iterative computation through asynchronous accumulative updates. In *ScienceCloud '12*, pages 13–22, 2012.

[32] J. Zou and F. Fekri. A belief propagation approach for detecting shilling attacks in collaborative filtering. In *CIKM '13*, pages 1837–1840, 2013.