Inference of Beliefs on Billion-Scale Graphs

U Kang, Duen Horng “Polo” Chau, Christos Faloutsos
School of Computer Science
Carnegie Mellon University
5000 Forbes Ave. Pittsburgh, PA, USA
{ ukang, dchau, christos }@cs.cmu.edu

ABSTRACT

How do we scale up the inference of graphical models to billions of nodes and edges? How do we, or can we even, implement an inference algorithm for graphs that do not fit in the main memory? Can we easily implement such an algorithm on top of an existing framework? How would we run it? And how much time will it save us? In this paper, we tackle this collection of problems through an efficient parallel algorithm for Belief Propagation(BP) that we developed for sparse billion-scale graphs using the HADOOP platform.

Inference problems on graphical models arise in many scientific domains; BP is an efficient algorithm that has successfully solved many of those problems. We have discovered and we will demonstrate that this useful algorithm can be implemented on top of an existing framework — the crucial observation in the discovery is that the message update process in BP is essentially a special case of GIM-V(Generalized Iterative Matrix-Vector multiplication) [10], a primitive for large scale graph mining, on a *line graph* induced from the original graph.

We show how we formulate the BP algorithm as a variant of GIM-V, and present an efficient algorithm. We experiment with our parallelized algorithm on the largest publicly available Web Graphs from Yahoo!, with about 6.7 billion edges, on M45, one of the top 50 fastest supercomputers in the world, and compare the running time with that of a single-machine, disk-based BP algorithm.

Categories and Subject Descriptors

H.2.8 [**Database Applications**]: Data Mining

General Terms

Algorithms; Experimentation.

Keywords

Belief Propagation, Hadoop, GIM-V

KDD-LDMTA'10, July 25, 2010, Washington, DC, USA.

1. INTRODUCTION

Belief Propagation(BP) [17, 21] is a popular algorithm for inferring the states of nodes in Markov Random fields. BP has been successfully used for social network analysis, computer vision, error correcting codes, etc. One of the current challenges in BP is the issue of scalability; it is not trivial to run BP on a very large graph with billions of nodes and edges.

To address the problem, we propose an efficient HADOOP algorithm for computing BP on a very large scale. The contributions are the following.

1. We show that the Belief Propagation algorithm is essentially a special case of GIM-V [10], a primitive for parallel graph mining in HADOOP.

2. We provide an efficient algorithm for the Belief Propagation in HADOOP.

3. We run experiments on a HADOOP cluster and analyze the running time.

The rest of the paper is organized as follows. Section 2 explains the related works on the Belief Propagation and HADOOP. Section 3 describes our formulation of the Belief Propagation in terms of GIM-V, and Section 4 provides a fast algorithm in HADOOP. Section 5 shows the experimental results. We conclude in Section 6. Table 1 lists the symbols used in this paper.

Symbol	Definition
V	Set of nodes in a graph
E	Set of edges in a graph
n	Number of nodes in a graph
l	Number of edges in a graph
S	Set of states
$\phi_i(s)$	Prior of node i being in state s
$\psi_{ij}(s', s)$	Edge potential when nodes i and j being in states s' and s, respectively
$m_{ij}(s)$	Message that node i sends to node j expressing node i's belief of node j's being in state s
$b_i(s)$	Belief of node i being in state s

Table 1: Table of symbols

2. BACKGROUND

The related work forms two groups, belief propagation and MapReduce/Hadoop.

2.1 Belief Propagation(BP)

Belief Propagation(BP) [17] is an efficient inference algorithm for probabilistic graphical models. Since its proposal, it has been widely, and successfully, used in a myriad of domains to solve many important problems (some are seemingly unrelated at the first glance). For example, BP is used in some of the best error-correcting codes, such as the *Turbo code* and *low-density parity-check* code, that approach channel capacity. In computer vision, BP is among the top contenders for stereo shape estimation and image restoration (e.g., denoising) [5]. BP has also been used for fraud detection, such as for unearthing fraudsters and their accomplices lurking in online aunctions [3], and pinpointing misstated accounts in general ledger data for the financial domain [13].

BP is typically used for computing the *marginal distribution* for the unobserved nodes in a graph, conditional on the observed ones; we will only discuss this version in this paper, though with slight and trivial modifications to our implementation, the *most probable distribution* of node states can also be computed.

BP was first proposed for *trees* [17] and it could compute the exact marginal distributions; it was later applied on general graphs [18] as an approximate algorithm. When the graph contains cycles or loops, the BP algorithm applied on it is called *loopy BP*, which is also the focus of this work.

BP is generally applied on graphs whose nodes have finite number of states (treating each node as a *discrete* random variable). Gaussian BP is a variant of BP where its underlying distributions are Gaussian [20]. Generalized BP [21] allows messages to be passed between subgraphs, which can improve accuracy in the computed beliefs and promote convergence.

BP is computationally-efficient; its running time scales linearly with the number of edges in the graph. However, for graphs with *billions* of nodes and edges — a focus of our work — this cost becomes significant. There are several recental works that investigated parallel BP on multicore shared memory [7] and MPI [6, 14]. However, all of them assume the graphs would fit in the main memory (of a single computer, or a computer cluster). Our work specifically tackles the important, and increasingly prevelant, situation where the graphs would not fit in main memory.

2.2 MapReduce and Hadoop

MapReduce is a parallel programming framework [4] for processing web-scale data. MapReduce has two advantages: (a) The data distribution, replication, fault-tolerance, load balancing is handled automatically; and furthermore (b) it uses the familiar concept of functional programming. The programmer needs to define only two functions, a *map* and a *reduce.* The general framework is as follows [12]: (a) the *map* stage reads the input file and emits (key, value) pairs; (b) the *shuffling* stage sorts the output and distributes them to reducers; (c) the *reduce* stage processes the values with the same key and emits another (key, value) pairs which become the final result.

Hadoop [1] is the open source version of MapReduce. Hadoop uses its own distributed file system HDFS, and provides a high-level language called PIG [15]. Due to its excellent scalability and ease of use, Hadoop is a very promising tool for large scale graph mining(see [16] [10] [9]). Other variants which provide advanced MapReduce-like systems include SCOPE [2], Sphere [8], and Sawzall [19].

3. BP AND GIM-V

In this section, we describe our parallel algorithm for BP on billion-scale graphs on Hadoop.

3.1 Belief Propagation

Now, we provide a quick overview of the Belief Propagation(BP) algorithm, which briefly explains the key steps in the algorithm and their formulation; this information will help our readers better understand how our implementation nontrivially captures and optimizes the algorithm in latter sections. For detailed information regarding BP, we refer our readers to the excellent article by Yedidia et al. [21].

The BP algorithm is an efficient method to solve inference problems for probabilistic graphical models, such as Bayesian networks and pairwise Markov random fields (MRF). In this work, we focus on pairwise MRF, which has seen empirical success in many domains (e.g., Gallager codes, image restoration) and is also simpler to explain; the BP algorithms for other types of graphical models are mathematically equivalent [21].

When we view an undirected simple graph $G = (V, E)$ as a pairwise MRF, each node i in the graph becomes a random variable X_i, which can be in a discrete number of states S. The goal of the inference is to find the marginal distribution $P(x_i)$ for all node i, which is an NP-complete problem.

Fortunately, BP may be used to solve this problem approximately (for MRF; exactly for trees). At a high level, BP infers the "true" (or so-called "hidden") distribution of a node from some prior (or "observed") knowledge about the node, and from the node's neighbors. This is accomplished through iterative message passing between all pairs of nodes v_i and v_j. We use $m_{ij}(x_j)$ to denote the message sent from i to j, which intuitively represents i's opinion about j's likelihood of being in state x_j. The prior knowledge about a node i, or the prior probabilities of the node being in each possible state are expressed through the *node potential function* $\phi(x_i)$. This prior probability may simply be called a *prior.* The message-passing procedure stops if the messages no longer change much from one iteration to the another — or equivalently when the nodes' marginal probabilities are no longer changing much. The estimated marginal probability is called *belief*, or symbolically $b_i(x_i)$ ($\approx P(x_i)$).

In detail, messages are obtained as follows. Each edge e_{ij} is associated with messages $m_{ij}(x_j)$ and $m_{ji}(x_i)$ for each possible state. Provided that all messages are passed in every iteration, the order of passing can be arbitrary. Each message vector m_{ij} is normalized, so that it sums to one. Normalization also prevents numerical underflow (or zeroing-out values). Each outgoing message from a node i to a neighbor j is generated based on the incoming messages from the node's other neighbors. Mathematically, the message-update equation is:

$$m_{ij}(x_j) = \sum_{x_i} \phi_i(x_i)\psi_{ij}(x_i, x_j) \prod_{k \in N(i) \backslash j} m_{ki}(x_i) \qquad (1)$$

where $N(i)$ is the set of nodes neighboring node i, and

$\psi_{ij}(x_i, x_j)$ is called the *edge potential*; intuitively, it is a function that *transforms* a node's incoming messages collected into the node's outgoing ones. Formally, $\psi_{ij}(x_i, x_j)$ equals the probability of a node i being in state x_i and that its neighbor j is in state x_j.

The algorithm stops when the beliefs converge (within some threshold, e.g., 10^{-5}), or a maximum number of iterations has finished. Although convergence is not guaranteed theoretically for general graphs, except for those that are trees, the algorithm often converges in practice, where convergence is quick and the beliefs are reasonably accurate. When the algorithm ends, the node beliefs are determined as follows:

$$b_i(x_i) = k\phi_i(x_i) \prod_{j \in N(i)} m_{ji}(x_i) \tag{2}$$

where k is a normalizing constant.

3.2 GIM-V

GIM-V(Generalized Iterative Matrix-Vector multiplication) [10] is a primitive for parallel graph mining algorithms in HADOOP. It is based on the observation that many graph mining algorithms, including PageRank, RWR, radius estimation, and connected components, are essentially repeated matrix-vector multiplication. GIM-V generalizes the matrix-vector multiplication by customizing the sub-operations in the matrix-multiplication, thereby resulting in multiple algorithms. The main idea of GIM-V is as follows. Given a n by n matrix M and a n-vector v, the matrix-vector multiplication is represented by

$M \times v = v'$ where $v'_i = \sum_{j=1}^{n} m_{i,j} v_j$.

By careful observation, we notice three implicit operations in the above formula:

1. `combine2`: multiply $m_{i,j}$ and v_j.
2. `combineAll`: sum n multiplication results.
3. `assign`: overwrite previous value of v_i with new result to make v'_i.

GIM-V customizes the three operations by defining a generalized multiplication operator $\times_G$ so that the multiplication becomes

$v' = M \times_G v$
where $v'_i = \texttt{assign}(v_i, \texttt{combineAll}_i(\{x_j \mid j = 1..n,$
$x_j = \texttt{combine2}(m_{i,j}, v_j)\}))$.

Different customizations lead to various algorithms which seem unrelated at first sight. For example, computing the PageRank and the connected components are special cases of GIM-V, as we show below.

PageRank.

The Pagerank score can be computed by a power iteration which is represented in GIM-V by
$p^{next} = M \times_G p^{cur}$
where M is an adjacency matrix, p is a vector of length n which is updated by
$p_i^{next} = \texttt{assign}(p_i^{cur}, \texttt{combineAll}_i(\{x_j \mid j = 1..n,$
$x_j = \texttt{combine2}(m_{i,j}, p_j^h)\}))$,
and the three operations are defined as follows:

1. $\texttt{combine2}(m_{i,j}, v_j) = c \times m_{i,j} \times v_j$
2. $\texttt{combineAll}_i(x_1, ..., x_n) = \frac{(1-c)}{n} + \sum_{j=1}^{n} x_j$
3. $\texttt{assign}(v_i, v_{new}) = v_{new}$

Connected Components.

Another example is the computation of connected components of graph. The component membership vector c is updated in GIM-V by

$c^{h+1} = A \times_G c^h$

where A is an adjacency matrix, c^{h+1} is the component membership vector at $(h+1)$th iteration. The c vector is updated by
$c_i^{h+1} = \texttt{assign}(c_i^h, \texttt{combineAll}_i(\{x_j \mid j = 1..n,$
$x_j = \texttt{combine2}(A_{i,j}, c_j^h)\}))$,
and the three operations are defined as follows:

1. $\texttt{combine2}(A_{i,j}, v_j) = A_{i,j} \times v_j$.
2. $\texttt{combineAll}_i(x_1, ..., x_n) = \text{MIN}\{x_j \mid j = 1..n\}$
3. $\texttt{assign}(v_i, v_{new}) = \text{MIN}(v_i, v_{new})$.

For details and fast algorithms for GIM-V, see [10].

3.3 BP using GIM-V

We first present the high level idea of the BP algorithm in Algorithm 1.

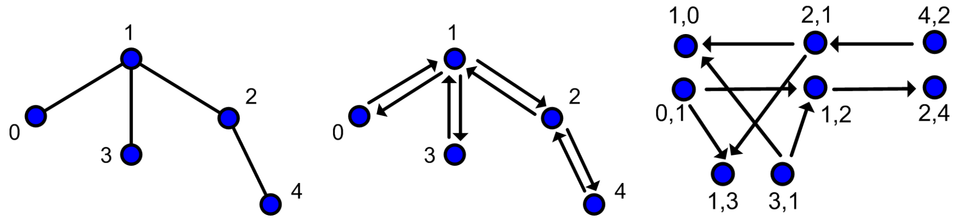
Algorithm 1: Belief Propagation

```
Input : Edge $E$,
        node prior $\phi^{n \times 1}$, and
        propagation matrix $\psi^{S \times (S-1)}$
Output: Belief matrix $b^{n \times S}$
1 begin
2   while $m$ does not converge do
3     for $(i,j) \in E$ do
4       for $s \in S$ do
5         $m_{ij}(s) \leftarrow \sum_{s'} \phi_i(s')\psi_{ij}(s', s) \prod_{k \in N(i) \backslash j} m_{ki}(s')$;
6   for $i \in V$ do
7     for $s \in S$ do
8       $b_i(s) \leftarrow k\phi_i(s) \prod_{j \in N(i)} m_{ji}(s)$;
9 end
```

The key observation is that the message update process (line 5 of Algorithm 1) can be represented by GIM-V on an induced graph from the original graph. To formalize this, we first define the 'directed line graph'.

DEFINITION 1 (DIRECTED LINE GRAPH). *Given a directed graph G, its directed line graph L(G) is a graph such that each node of L(G) represents an edge of G, and there is an edge from v_i to v_j of L(G) if the corresponding edges e_i and e_j form a length-two directed path from e_i to e_j in G.*

To convert a undirected line graph G to a directed line graph $L(G)$, we first convert G to a directed graph by converting each undirected edge to two directed edges. Then, we connect two nodes in $L(G)$ if their corresponding edges



(a) Original graph (b) Directed graph (c) Directed line graph

Figure 1: Converting a undirected graph to a directed line graph. (a to b): replace a undirected edge with two directed edges. (b to c): for an edge (i, j) in (b), make a node (i, j) in (c). Make a directed edge from (i, j) to (k, l) in (c) if $j = k$ and $i \neq l$.

form a directed path in G . For example, see Figure 1 for a graph and its directed line graph.

Notice that an edge $(i, j) \in E$ in the original graph G correspond to a node in $L(G)$. Thus, line 5 of Algorithm 1 is essentially updating nodes in $L(G)$. Let (V, E) be the nodes and the edges of the original graph G. Also, let A' be the adjacency matrix of $L(G)$. Then the belief propagation is formulated in GIM-V as follows:

$$m(s)^{next} = A' \times_G m^{cur} \tag{3}$$

where $m(s)$ is the message vector of length $|E|$, and $m_i(s)$, the ith element of $m(s)$, contains the message regarding the state s. $m(s)$ is updated by
$m(s)_i^{next}$ =$\texttt{assign}(m_i^{cur}$,$\texttt{combineAll}_i(\{x_j \mid j = 1..r$,
x_j =$\texttt{combine2}(A'_{i,j}, m_j^{cur})\}))$,
and the three operations are defined as follows:

1. $\texttt{combine2}(A'_{i,j}, m_j) = A'_{i,j} \times m_j$.
2. $\texttt{combineAll}_i(x_1, ..., x_r) = \sum_{s'} \phi_i(s')\psi_{ij}(s', s) \prod_{j=1}^{r} x_j(s')$
3. $\texttt{assign}(v_i, v_{new}) = v_{new}$.

Thus, GIM-V on the directed line graph leads to the Hadoop algorithm for BP which is summarized in Algorithm 2.

Algorithm 2: Hadoop-BP using GIM-V

```
Input : Edge $E$ of a undirected graph,
        node prior $\phi^{n \times 1}$, and
        propagation matrix $\psi^{S \times (S-1)}$
Output: Belief matrix $b^{n \times S}$
begin
    $A' \leftarrow$ directed line graph from $E$ ;
    while $m$ does not converge do
        for $s \in S$ do
            $m(s)^{next} = A' \times_G m^{cur}$;
    for $i \in V$ do
        for $s \in S$ do
            $b_i(s) \leftarrow k\phi_i(s) \prod_{j \in N(i)} m_{ji}(s)$;
end
```

4. FAST ALGORITHM FOR HADOOP

In this section, we first describe the naive algorithm for Hadoop-BP and propose an efficient algorithm.

4.1 Naive Algorithm

The formulation of BP in terms of GIM-V provides an intuitive way to understand the computation. However, a naive algorithm without careful design is not efficient for the following reason. In a naive algorithm, we first build the matrix for the line graph $L(G)$ and the message vector, and apply the GIM-V on them. The problem is that a node in G with degree d will generate $d(d - 1)$ edges in $L(G)$. Since there exists many nodes with a very large degree in real-world graphs due to the well-known power-law degree distribution, the number of nonzero elements will grow too large. For example, the YahooWeb graph in Section 5 has several nodes with the several-million degree. Then, the number of nonzero elements in the corresponding line graph will be more than 1 trillion. Thus, we need an efficient algorithm for dealing with the problem.

4.2 Lazy Multiplication

The main idea to solve the problem in the previous section is not to build the line graph explicitly: instead, we do the same computation on the original graph, or perform a 'lazy' multiplication. The crucial observation is that the edges in the original graph G contains all the edge information in $L(G)$: each edge $e \in E$ of G is a node in $L(G)$, and $e_1, e_2 \in G$ are adjacent in $L(G)$ if and only if they share the node in G. Thus, grouping all the edges by nodes in G allows us to work on adjacent nodes in $L(G)$. Since we encodes the message value by adding an additional field to each edge, the message update operation(line 5 of Algorithm 1) can be computed after the grouping.

A computational issue in computing $\prod_{k \in N(i) \backslash j} m_{ki}(s')$ is that a straightforward implementation requires $N(i)(N(i) - 1)$ multiplication which is prohibitively large. However, we decrease the number of multiplication to $2N(i)$ by first computing $t = \prod_{k \in N(i)} m_{ki}(s')$, and for each $j \in N(i)$ computing $t/m_{ji}(s')$.

The only remaining pieces of the computation is to incorporate the prior ϕ and the propagation matrix ψ. The propagation matrix ψ is a tiny bit of information, so it can be sent to every reducer by a variable passing functionality of Hadoop. The prior vector ϕ can be large, since the length of the vector can be the number of nodes in the graph. In the Hadoop algorithm, we also group the ϕ by the node id: each node prior is grouped together with the edges(messages) whose source id is the node id. Algorithm 3 and 4 shows the Hadoop algorithm for the BP message ini-

```
Algorithm 3: HADOOP-BP Initialization
Input  : Edge $E = \{(id_{src}, id_{dst})\}$,
         Set of states $S = \{s_1, ..., s_p\}$
Output: Message Matrix $M =$
         $\{(id_{src}, id_{dst}, m_{dst,src}(s_1), ..., m_{dst,src}(s_p))\}$
Initialize-Map(Key k, Value v);
begin
    Output($(k, v)$, ($\frac{1}{|S|}$, ..., $\frac{1}{|S|}$));   // (k: $id_{src}$, v: $id_{dst}$)
end
```

```
Algorithm 4: HADOOP-BP Message Update
Input  : Set of states $S = \{s_1, ..., s_p\}$,
         Current Message Matrix $M^{cur} =$
         $\{(sid, did, m_{did,sid}(s_1), ..., m_{did,sid}(s_p))\}$,
         Prior Matrix $\Phi = \{(id, \phi_{id}(s_1), ..., \phi_{id}(s_p))\}$,
         Propagation Matrix $\psi$
Output: Updated Message Matrix $M^{next} =$
         $\{(id_{src}, id_{dst}, m_{dst,src}(s_1), ..., m_{dst,src}(s_p))\}$
Stage1-Map(Key k, Value v);
begin
    if $(k, v)$ is of type $M$ then
        Output$(k, v)$;                    // (k: $sid$, v: $did, m_{did,sid}(s_1), ..., m_{did,sid}(s_p)$)

    else if $(k, v)$ is of type $\Phi$ then
        Output$(k, v)$;     // (k: $id$, v: $\phi_{id}(s_1), ..., \phi_{id}(s_p)$)

end
Stage1-Reduce(Key k, Value v[1..r]);
begin
    $temps[1..p] \leftarrow [1..1]$;
    $saved\_prior \leftarrow [\,]$;
    HashTable<int, double[1..p]> $h$;
    foreach $v \in v[1..r]$ do
        if $(k, v)$ is of type $\Phi$ then
            $saved\_prior[1..p] \leftarrow v$;

        else if $(k, v)$ is of type $M$ then
            $(did, m_{did,sid}(s_1), ..., m_{did,sid}(s_p)) \leftarrow v$;
            $h.add(did, (m_{did,sid}(s_1), ..., m_{did,sid}(s_p)))$;
            foreach $i \in 1..p$ do
                $temps[i] = temps[i] \times m_{did,sid}(s_i)$;

    foreach $(did, (m_{did,sid}(s_1), ..., m_{did,sid}(s_p))) \in h$ do
        $outm[1..p] \leftarrow 0$;
        foreach $u \in 1..p$ do
            foreach $v \in 1..p$ do
                $outm[u] = outm[u] + saved\_prior[v]\psi(v, u)temps[v]/m_{did,sid}(s_v)$;
        Output($did$, ($sid, outm[1], ..., outm[p]$));
end
```

tialization and the message update which implements the algorithm described above.

5. EXPERIMENTS

In this section, we present experimental results to answer the following questions:

Q1 How fast is our algorithm, compared to a single-machine disk-based Belief Propagation algorithm?

Q2 How does our HADOOP Belief Propagation algorithm scales up?

We performed experiments in the M45 HADOOP cluster by Yahoo!. The cluster has total 480 machines 1.5 Petabyte total storage and 3.5 Terabyte memory. The single-machine experiment was done in a machine with 3 Terabyte of disk and 48 GB memory. The single-machine BP algorithm is a scaled-up version of a memory-based BP which reads all the nodes and the edges into a memory. That is, the single-machine BP loads only the node information into a memory, but it reads the edges sequentially from the disk for every message update, instead of loading all the edges into a memory once for all.

For the data, we used the YahooWeb graph, a snapshot of the Web at the year 2002 with 1.4 billion nodes and 6.7 billion edges saved in a 120-GB file.

5.1 Results

Between HADOOP-BP and the single-machine BP, which one runs faster? At which point does the HADOOP-BP outperform the single-machine BP? Figure 2 (a) shows the comparison of running time of the HADOOP-BP and the single-machine BP. Notice that HADOOP-BP outperforms the single-machine BP when the number of machines exceeds 40. The HADOOP-BP requires more machines to beat the single-machine BP due to the fixed costs for writing and reading the intermediate results to and from the disk. However, for larger graphs which do not fit into a memory, HADOOP-BP is the only solution.

The next question is, how does our HADOOP-BP scale up? Figure 2 (b) shows the scalability of our algorithm with the increasing number of machines. We see that our HADOOP-BP scales up linearly close to the ideal scale-up.

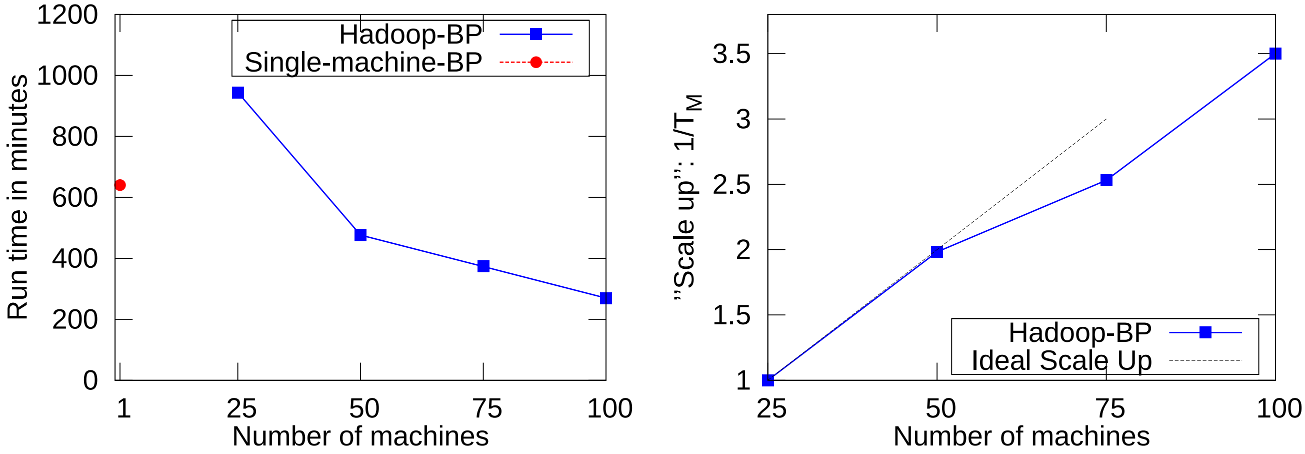
5.2 Discussion

Based on the experimental results, what are the advantages of HADOOP-BP? In what situations should it be used? For a small graph whose nodes and edges fit in the memory, the single-machine BP is recommended since it runs faster. For a medium-to-large graph whose nodes fit in the memory but the edges do not fit in the memory, HADOOP-BP gives the reasonable solution since it runs faster than the single-machine BP. For a very large graph whose nodes do not fit in the memory, HADOOP-BP is the only solution. We summarize the advantages of the HADOOP-BP here:

- **Scalability**: HADOOP-BP is *the only* solution when the nodes information can not fit in memory. Moreover, HADOOP-BP scales up near-linearly.

- **Running Time**: Even for a graph whose node information fits into a memory, HADOOP-BP ran 2.4 times faster.

- **Fault Tolerance**: HADOOP-BP enjoys the fault tolerance that HADOOP provides: data are replicated, and the failed programs due to machine errors are restarted in working machines.



(a) Running Time

(b) Scale-Up

Figure 2: Running time of BP with 10 iterations on the YahooWeb graph with 1.4 billion nodes and 6.7 billion edges. (a) Comparison of the running times of the parallel BP and the single-machine BP. Notice that Hadoop-BP outperforms the single-machine BP when the number of machines exceed $\tilde{4}$0. (b) "Scale-up" (throughput $1/T_M$) versus number of machines M, for the YahooWeb graph. Notice the near-linear scale-up close to the ideal(dotted line).

6. CONCLUSION

In this paper we proposed a HADOOP-BP for the inferences of graphical models in a billion-scale graphs. The main contributions are the followings:

- We show that the inference problem in graphical models is a special case of GIM-V which is a tractable primitive for large scale graph mining in HADOOP.
- We carefully design an efficient algorithms for HADOOP-BP.
- We do the experiments to compare the running time of the HADOOP-BP and the single-machine BP. We also gives the scalability results and show that HADOOP-BP has a near-linear scale up.

One major research direction is to apply our HADOOP-BP for inferences on large, real-world data. Another directions is the tensor analysis on HADOOP ([11]), since tensor is a multi-dimensional extension of matrix.

7. ACKNOWLEDGEMENTS

This work was partially funded by the National Science Foundation under Grants No. IIS-0705359, IIS-0808661, and under the auspices of the U.S. Dept. of Energy by Lawrence Livermore National Laboratory under contract DE-AC52-07NA27344. We would like to thank YAHOO! for the web graph and access to the M45. The opinions expressed are those of the authors and do not necessarily reflect the views of the funding agencies.

8. REFERENCES

[1] Hadoop information. http://hadoop.apache.org/.

[2] R. Chaiken, B. Jenkins, P.-A. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou. Scope: easy and efficient parallel processing of massive data sets. *VLDB*, 2008.

[3] D. H. Chau, S. Pandit, and C. Faloutsos. Detecting fraudulent personalities in networks of online auctioneers. *PKDD*, 2006.

[4] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. *OSDI*, 2004.

[5] P. Felzenszwalb and D. Huttenlocher. Efficient belief propagation for early vision. *International journal of computer vision*, 70(1):41–54, 2006.

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

[7] J. E. Gonzalez, Y. Low, and C. Guestrin. Residual splash for optimally parallelizing belief propagation. *AISTAT*, 2009.

[8] R. L. Grossman and Y. Gu. Data mining using high performance data clouds: experimental studies using sector and sphere. *KDD*, 2008.

[9] U. Kang, C. Tsourakakis, A. P. Appel, C. Faloutsos, and J. Leskovec. Radius plots for mining tera-byte scale graphs: Algorithms, patterns, and observations. *SIAM International Conference on Data Mining*, 2010.

[10] U. Kang, C. Tsourakakis, and C. Faloutsos. Pegasus: A peta-scale graph mining system - implementation and observations. *IEEE International Conference on Data Mining*, 2009.

[11] T. G. Kolda and J. Sun. Scalable tensor decompsitions for multi-aspect data mining. *ICDM*, 2008.

[12] R. Lämmel. Google's mapreduce programming model – revisited. *Science of Computer Programming*, 70:1–30, 2008.

[13] M. McGlohon, S. Bay, M. Anderle, D. Steier, and C. Faloutsos. Snare: a link analytic system for graph labeling and risk detection. In *Proceedings of the 15th ACM SIGKDD international conference on Knowledge*

discovery and data mining, pages 1265–1274. ACM, 2009.

[14] A. Mendiburu, R. Santana, J. Lozano, and E. Bengoetxea. A parallel framework for loopy belief propagation. *GECCO*, 2007.

[15] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a not-so-foreign language for data processing. In *SIGMOD '08*, pages 1099–1110, 2008.

[16] S. Papadimitriou and J. Sun. Disco: Distributed co-clustering with map-reduce. *ICDM*, 2008.

[17] J. Pearl. Reverend Bayes on inference engines: A distributed hierarchical approach. In *Proceedings of the AAAI National Conference on AI*, pages 133–136, 1982.

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

[19] R. Pike, S. Dorward, R. Griesemer, and S. Quinlan. Interpreting the data: Parallel analysis with sawzall. *Scientific Programming Journal*, 2005.

[20] Y. Weiss and W. Freeman. Correctness of belief propagation in Gaussian graphical models of arbitrary topology. *Neural Computation*, 13(10):2173–2200, 2001.

[21] J. S. Yedidia, W. T. Freeman, and Y. Weiss. Understanding belief propagation and its generalizations. *Exploring Artificial Intelligence in the New Millenium*, 2003.