CS239 Report: Distributed Belief Propagation
With Circular Message Passing

Xilai Zhang, Haiying Huang

1. Introduction

In this project, we design and implement a distributed version of the ***Iterative Belief Propagation*** **(IBP)** algorithm for doing inference on Bayesian Networks (BNs) using *Pthread* library and evaluate its performance. Belief propagation is a message-passing algorithm: in each iteration, a node X collects messages from its neighbors, do computation on collected message, and push the updated message to its neighbor Y, and the belief states of each node will converge to its marginal probability. We develop a distributed version of the IBP algorithm, and our contributions are as follows.

- Develop a parallel Iterative Belief Propagation (IBP) algorithm over 1D-partitioning of the network, using *pthread* in *C++*.
- Propose a ***circular message passing*** scheme in IBP, which is inspired by the Token Ring protocol in computer Network, to reduce the communication cost between adjacent nodes. To our knowledge, our distributed IBP system is the first one that adopts this message-passing scheme.
- Evaluate the performance/scalability of our IBP algorithm on networks of different sizes, showing that our algorithm provides upto **4.0x** Speedup, as compared to the un-distributed version of IBP (the ***baseline***).

2. Prior Arts

2.1 Bayesian Networks

A Bayesian Network (BN) over variables $\mathcal{X} = \{X_1, \ldots, X_n\}$ is a directed acyclic Graph G=(V,E) that encodes a joint probability over $\mathcal{X}$ (See Figure 1). Each node $X \in \mathcal{X}$ corresponds to a random variable and is attached with one Conditional Probability Table (CPT) that represents the conditional $p(X|\mathbf{U})$ where $\mathbf{U}$ denotes the parents of X in the DAG. The goal of probabilistic inference is to compute the prior/posterior marginals for each variable X in the network

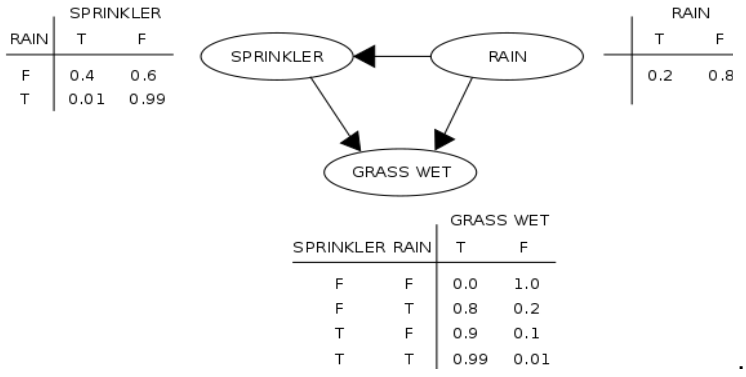

Figure 1. An example Bayesian network with CPT tables

2.2 Belief Propagation

Iterative Belief Propagation (IBP) is a message-passing algorithm that is used to do approximate probabilistic inference on BNs of general structures. Consider each pair adjacent nodes $X \rightarrow Y$. During each iteration, node X will collect message from all its neighbors other than Y, multiply them with its own CPT, eliminate according to Y, and push the updated message to Y. A message from parent X to child Y is called *causal* message, denoted $\lambda_Y(X)$; and a message from child Y to parent X is called *diagnostic* message, denoted $\pi_Y(X)$. At the beginning, all messages are uniformly initialized, and during each iteration, the messages are propagated according to the above rules, and after many iterations, the algorithm can provide high-quality approximation of the marginals of every node in the network. The algorithm is illustrated in the figure shown below (Darwiche, 2009).

Algorithm 40 IBP($\mathcal{N}$, **e**)

input:

$\mathcal{N}$: a Bayesian network inducing distribution Pr

e: an instantiation of some variables in network $\mathcal{N}$

output: approximate marginals, $BEL(X\mathbf{U})$, of $\Pr(X\mathbf{U}|\mathbf{e})$ for each family $X\mathbf{U}$ in $\mathcal{N}$

main:

1: $t \leftarrow 0$
2: initialize all messages π^0, λ^0 (uniformly)
3: **while** messages have not converged **do**
4: $t \leftarrow t + 1$
5: **for** each node X with parents $\mathbf{U}$ **do**
6: **for** each parent U_i **do**
7: $\lambda_X^t(U_i) = \eta \sum_{X\mathbf{U}\setminus\{U_i\}} \lambda_{\mathbf{e}}(X)\Theta_{X|\mathbf{U}} \prod_{k\neq i} \pi_X^{t-1}(U_k) \prod_j \lambda_{Y_j}^{t-1}(X)$
8: **end for**
9: **for** each child Y_j **do**
10: $\pi_{Y_j}^t(X) = \eta \sum_{\mathbf{U}} \lambda_{\mathbf{e}}(X)\Theta_{X|\mathbf{U}} \prod_i \pi_X^{t-1}(U_i) \prod_{k\neq j} \lambda_{Y_k}^{t-1}(X)$
11: **end for**
12: **end for**
13: **end while**
14: **return** $BEL(X\mathbf{U}) = \eta\, \lambda_{\mathbf{e}}(X)\Theta_{X|\mathbf{U}} \prod_i \pi_X^t(U_i) \prod_j \lambda_{Y_j}^t(X)$ for families $X\mathbf{U}$

Figure 2. The iterative Belief Propagation algorithm

3. Algorithm Design and Architecture

3.1 Distributed Belief Propagation

We provide a distributed version of the IBP algorithm. Take a second look at the pseudo-code in Figure 2. Notice that during each iteration, the computation of one node is independent of all the other nodes. Therefore, we can partition the network into different lists of nodes, and use one thread for handling one partition.

Also, we propose a ***circular message passing*** scheme for our IBP, which is inspired by the Token Ring protocol (Eigen, 2004) in computer networking to reduce the communication cost between adjacent nodes. We will discuss this in detail in section 3.3. To our knowledge, our distributed IBP implementation is the first one that adopts such a scheme.

The multi-threading incurs two new challenges. First, we need to synchronize threads among iteration: a thread needs to wait for all the other threads to finish its current iteration before it can proceed to the next iteration, i.e. a barrier at the end of every iteration. Second, we need to pass around data across threads for each pair of adjacent nodes, and therefore need to prevent data-racing. We will discuss how our implementation overcomes these challenges in the next section.

3.2 Thread Communication

For now, let us assume one thread for each node. For each two adjacent nodes/threads $X \rightarrow Y$, they own two pipes (which are implemented as shared pointers), one for the causal message from X to Y and the other for the diagnostic message from Y to X. A mutex lock is used for each pipe to prevent two threads from reading/writing to the shared pointer at the same time; moreover, we keep track of the iteration number enclosed in the message to prevent a thread from writing an updated message before the current message has been collected by the other thread. We design our *Node* class as below to implement the above idea.

```
class Node{
public:
    int id;
    int card; // cardinality
    vector<int> parents; // parents id
    vector<int> children; // children id
    vector<int> parentsCard; //parents cardinality
    vector<int> childrenCard; // children cardinality
    int num_parents;
    int num_children;
    int num_iter=0;
    NdArray CPT;

    vector<pthread_mutex_t*> lock_from_neighbors; // locks for msg pipes from neighbors
    vector<pthread_mutex_t*> lock_to_neighbors; // locks for msg pipes to neighbors
    vector<Message*> msg_from_neighbors; // msg pipes from neighbors
    vector<Message*> msg_to_neighbors; // msg pipes to neighbors
    vector<Message> msg_from_neighbors_cache;
    vector<Message> msg_to_neighbors_cache;

    Node(){
```

Figure 3. Shared message pointers and mutex locks between adjacent nodes

3.3 Circular Message Passing

Consider each pair adjacent nodes $X \rightarrow Y$. In each iteration, node X need to collect message from all its neighbors $adj(X)$ before it can compute the message to be sent to Y. As discussed above, for each neighbor, node X need to acquire a lock before it can read the message from that neighbor; however, the common *pthread_lock()* is a blocking function call, which could generate a long waiting time if we naively read the message from each neighbor one by one.

Instead, we collect messages from neighbors in a circular order using the non-blocking *pthread_mutex_trylock()*. The neighbors of X are arranged into a ring topology Y_1,Y_2,...,Y_n in Figure 4. Node X will first try to acquire the lock from node Y_i. If successful, it will continue to read the message from Y_i; otherwise, it will proceed to acquire the lock for the next node Y_i+1 without waiting for Y_i. This procedure is repeated until messages from all neighbors are collected. This circular topology is used for both collecting/pushing messages from every neighbor.

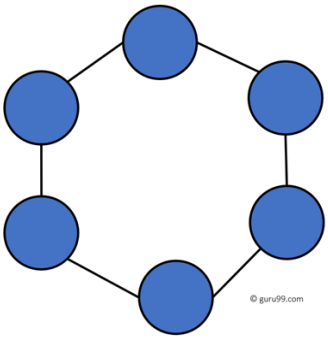

Figure 4. The neighbors of each node are arranged into a Ring topology

3.4 Conditional Probability Table

As discussed above, each node in the network is attached with one Conditional Probability Table (CPT) and the message propagated through the network is also a Probability Table. A Probability Table can be viewed as a N-dimensional array (*N-d array*) similar to Numpy. In our C++ implementation, we implement N-d array as a flattened 1-d array in Figure 5. We provide the *get*/*set* method to access the probability table using N-d index and the *dotProduct*/*project* function to support basic operations on the probability tables.

```
class NdArray {
public:
    int ndims; // n-dimensions
    vector<int> cards; // size of each dimension
    vector<double> array; // 1D representation of ND array
    int array_size;

    NdArray() {};
    NdArray(int ndims, vector<int> cards);
    int computeIndex(vector<int> indices); //indices---S array,
    double get(vector<int> indices);
    void set(vector<int> indices, double value);
    void setData(vector<double> array);
    string to_string();

};
```

Figure 5. N-D array as flattened 1-D array

3.5 Network Generation

To test our distributed IBP algorithm, we randomly sample networks of different sizes. We randomly generate a Directed Acyclic Graph (DAG) by the following procedure. First we generate a chain with NUM_NODES nodes. This is to ensure that all nodes are connected. Then, for each node and a subsequent node, we randomly create an edge between them according to certain probability. After the DAG is sampled, we randomly sample CPT for each node.

4. Experiments

4.1 Experiment methodologies

We design experiments to evaluate the accuracy/speedup/scalability of our distributed IBP algorithm and report the results as below. The experiments are conducted on AWS ec2 instance. We have also tried testing on UCLA's *huffman2* supercomputer. However, our C++ implementation requires *g++* compiler, which seems not installed on the huffman2 server.

▼ Instance Type Edit instance type

Instance Type	ECUs	vCPUs	Memory (GiB)	Instance Storage (GB)	EBS-Optimized Available	Network Performance
c5a.8xlarge	-	32	64	EBS only	Yes	10 Gigabit

Figure 5. Configuration of our c5a.8xlarge ec2 instance on AWS, which has 32 vCPUs

4.2. Parallel v.s Sequential Speedup

We measure the execution time of our parallel IBP against the sequential version over networks of different sizes. We randomly generate networks of 10,20,...,100 nodes, run parallel IBP using N=10 threads and sequential IBP respectively, and report the execution time and speedup in the below table.

Num of Nodes	Seq IBP time (s)	Parallel IBP time (s)	Speedup
10	2979.82	1771.8	1.68
20	6262.25	1767.49	3.54
30	27439.3	6718.58	4.08
40	37072.4	8018.18	4.62
50	51673.8	12228.9	4.23
60	52117.4	10022	5.2
70	86865.2	21669.1	4.01
80	65155.2	16105.4	4.05
90	99397.2	15731.7	6.32
100	95919.9	18230	5.26

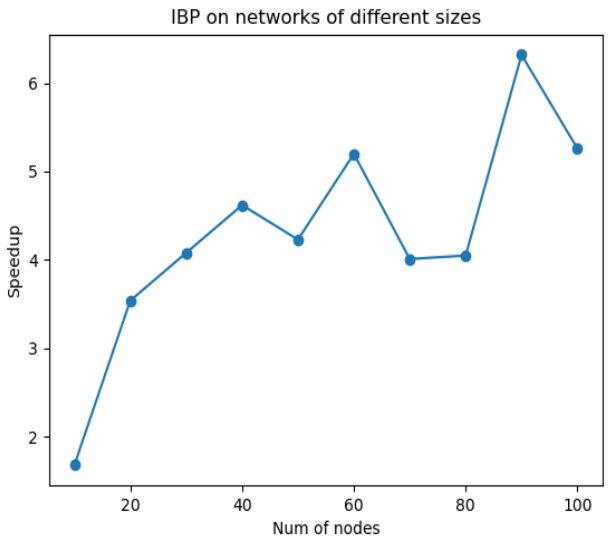

Table 1. Speedup of distributed IBP using N=10 threads across network of different sizes

As we can see, our distributed IBP provides **4.0+** speedup compared to the sequential IBP when the network size is fairly large. Notice that the execution time does **NOT** necessarily increase with the number of nodes. The reason is that the complexity of belief propagation not only depends on the number of nodes but also the density of the network, and is not taken into account when we randomly generate the networks.

4.3 Scalability

Next, we fix the input network and run our distributed IBP with increasing number of threads and measure the speedup. As shown in Table below, as the number of threads N increases, the speedup first increases, peaks at certain N, and then decreases. The reason is that starting from a certain point, the communication cost for safe message passing starts to outweigh the benefits by dividing node computation.

Scalability of IBP when size=20		
Threads	Exec Time (s)	Speedup
1	10102.5	1
2	5156.9	1.96
3	4732.75	2.13
4	3733.8	2.71
5	3302.19	3.06
6	2797.86	3.61
7	2386.08	4.23
8	2492.26	4.05
9	3606.93	2.8
10	3125.87	3.23

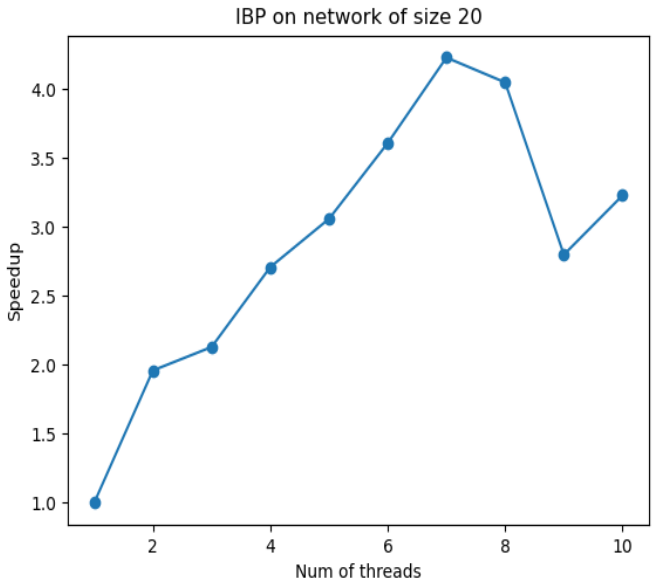

Table 2. Scalability of distributed IBP on a network of size 20. The maximum speedup is achieved with N = 7 threads.

Scalability of IBP when size=30		
Threads	Exec time (s)	Speedup
1	16912.3	1
2	12211	1.39
3	11011	1.54
4	8788.3	1.92
5	7281.66	2.32
6	8031.27	2.11
7	8415.73	2.01
8	8152.56	2.07
9	8111.8	2.08
10	7028.63	2.41

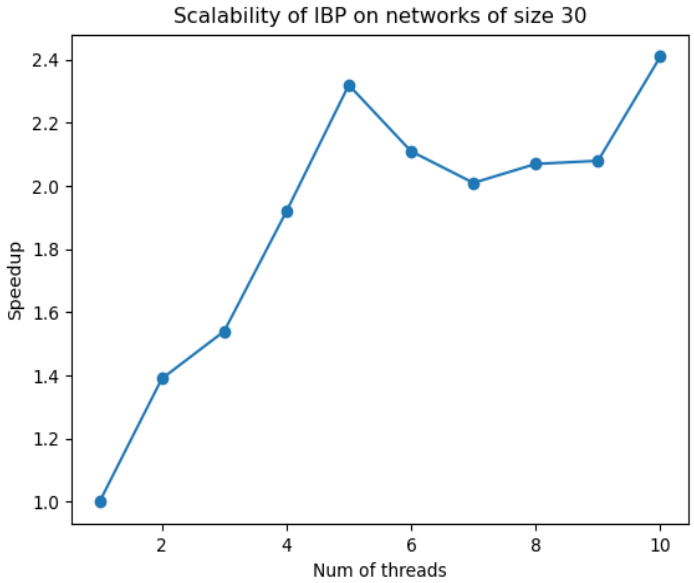

Table 3. Scalability of distributed IBP on network of size 30. The maximum speedup is achieved with N=10 threads.

5. Conclusion

In this project, we develop a distributed Iterative Belief propagation (IBP) algorithm using 1D-partition of the network. We propose a circular message-passing scheme, which is analogous to the Token Ring protocol, for passing messages between adjacent nodes, and integrate this scheme into our algorithm. We implement our distributed IBP algorithm using pthread in C++ and test it on AWS. Our experiments show that our distributed IBP provides more than **4.0x** speedup compared to the sequential IBP when the network size is fairly large. Therefore our implementation provides **faster convergence time** when IBP is applied in fields such as artificial intelligence and information theory. We also measure the scalability of our algorithm, showing that with increasing number of threads, the execution time of our algorithm first increases and then peaks. We believe the reason is that at certain stages, the communication cost between threads starts to outweigh the benefits of dividing computation between nodes.

Bibliography:

1. Darwiche, A. (2009). *Modeling and reasoning with Bayesian networks*. Cambridge university press.
2. Ergen, Mustafa, et al. "WTRP-wireless token ring protocol." *IEEE transactions on Vehicular Technology* 53.6 (2004): 1863-1881.