Accelerating Bayesian Network Parameter Learning Using Hadoop and MapReduce

Aniruddha Basak
Carnegie Mellon University
Silicon Valley Campus
NASA Research Park,
Moffett Field, CA 94035-0001
abasak@cmu.edu

Irina Brinster
Carnegie Mellon University
Silicon Valley Campus
NASA Research Park,
Moffett Field, CA 94035-0001
irina.brinster@sv.cmu.edu

Xianheng Ma
Carnegie Mellon University
Silicon Valley Campus
NASA Research Park,
Moffett Field, CA 94035-0001
xianhengma@cmu.edu

Ole J. Mengshoel
Carnegie Mellon University
Silicon Valley Campus
NASA Research Park,
Moffett Field, CA 94035-0001
ole.mengshoel@sv.cmu.edu

ABSTRACT

Learning conditional probability tables of large Bayesian Networks (BNs) with hidden nodes using the Expectation Maximization algorithm is heavily computationally intensive. There are at least two bottlenecks, namely the potentially huge data set size and the requirement for computation and memory resources. This work applies the distributed computing framework MapReduce to Bayesian parameter learning from complete and incomplete data. We formulate both traditional parameter learning (complete data) and the classical Expectation Maximization algorithm (incomplete data) within the MapReduce framework. Analytically and experimentally we analyze the speed-up that can be obtained by means of MapReduce. We present the details of our Hadoop implementation, report speed-ups versus the sequential case, and compare various Hadoop configurations for experiments with Bayesian networks of different sizes and structures. For Bayesian networks with large junction trees, we surprisingly find that MapReduce can give a speed-up compared to the sequential Expectation Maximization algorithm for learning from 20 cases or fewer. The benefit of MapReduce for learning various Bayesian networks is investigated on data sets with up to 1,000,000 records.

Categories and Subject Descriptors

I.2.6 [**Learning**]: Parameter Learning; D.1.3 [**Programming Techniques**]: Distributed Programming

General Terms

Algorithms, Performance, Experiments

BigMine'12 August 12, 2012 Beijing, China

1. INTRODUCTION

The estimation of parameters in Bayesian Networks (BNs) can be very compute intensive. The size of the conditional probability table grows exponentially in the number of parents in the network; as the input data size increases, sequential learning for large and complex BNs becomes challenging even in the case of complete data. For learning from incomplete data, the limiting factor is often the inference step that must be performed on each instance in the data set. Running inference requires calculation of the posterior distribution over all families, i.e. over all instantiations of variables and their parents.

Expectation Maximization (EM) is an iterative algorithm that enables learning statistical models from data with missing values or/and latent variables [4]. EM is a powerful technique as it guarantees convergence to a local maximum of the log-likelihood function. Due to its numerical stability and ease of implementation, EM has become the algorithm of choice in many areas. It is widely used for Bayesian clustering [9] in machine learning and computer vision, for gene clustering, motif finding, and protein identification in computational biology, for medical imaging, and for word alignment in machine translation [5]. Most of the listed applications benefit from representation of data in the form of graphical probabilistic models, such as hidden Markov chains or BNs.

This work addresses the application of a MapReduce based distributed computing framework, Hadoop, to Bayesian parameter learning from complete and incomplete data. We implement the traditional Bayesian update and the classical EM algorithms in Hadoop to speed up parameter learning in BNs for a wide range of input data sizes. In the MapReduce EM (MREM) formulation, the inference step is performed independently for each data record. By running inference in parallel, we accelerate each iteration of EM, alleviating the computational cost. We present an analytical framework for understanding the scalability and achievable speed up of MREM versus the sequential algorithm. Finally, we test the performance of map-reduced algorithms on a variety of BNs for a wide range of data sizes. We find that for complex networks, our map-reduced implementation outperforms se-

quential learning from data sizes as small as 20 data records.

2. BACKGROUND AND RELATED WORK

2.1 Bayesian Network Parameter Learning

Parameter learning is one of the key issues in BNs, which are widely used in artificial intelligence, machine learning, statistics, and bioinformatics [12]. Learning parameters from complete data is discussed in [13] and [1]. Unfortunately, as the data size becomes hugh, learning time of traditional sequential learning becomes unacceptable. In this paper, we decompose the Bayesian Update algorithm for learning parameters from complete data on MapReduce to get speed up.

One of the ways to speed up parameter estimation in graphical models focuses on parallelizing the inference step, which introduces the main computational bottleneck in many machine learning algorithms. Theoretical analysis of the parallel implementation of belief propagation and its scaling limitations appears in [7]. Near linear parallel scaling has been demonstrated by parallelization of the junction tree (JT) inference using OpenMP and MPI [11]. The belief propagation algorithm has been accelerated on GPUs [14] and CPU-GPGPU heterogenous system [8].

The EM algorithm has been implemented in MapReduce for a variety of purposes. It has been shown that a family of machine learning (ML) algorithms that fit the Statistical Query model (including EM) are embarrassingly parallel and hence, lend themselves well to the MapReduce parallelization on multicore computers [2] . The authors demonstrate nearly linear speed ups with increasing number of cores for a variety of ML algorithms. Performance and implementation issues associated with adapting various types of ML algorithms to MapReduce have been investigated [6].

Though EM has been implemented on MapReduce for a variety of tasks, to the best of our knowledge, in previous work there has been no formulation of the MapReduce algorithm for learning from complete and incomplete data in BNs.

2.2 MapReduce and Hadoop

MapReduce is a programming framework for distributed computing on large data sets which was introduced by Google in 2004. It is an abstraction that allows users to easily create parallel applications while hiding the details of data distribution, load balancing, and fault tolerance [3].

MapReduce requires decomposition of an algorithm into map and reduce steps. In the map phase, the input data split into blocks is processed as a set of input key-value pairs in parallel by multiple mappers. Each mapper applies to each assigned datum a user specified map function and produces as its output a set of intermediate key-value pairs. Then the values with the same key are grouped together (the sort and shuffle phase) and passed on to a reducer, which merges the values belonging to the same key according to a user-defined reduce function.

Hadoop, an implementation of MapReduce, provides a framework for distributing the data and user-specified mapreduce jobs across a large number of cluster nodes or machines. It is based on the master/slave architecture. The single master server, referred to as jobtracker, receives a job assignment from the user, distributes the map and reduce tasks to slave nodes (tasktrackers) and monitors their progress.[1] Storage and distribution of data to slave nodes is handled by the Hadoop Distributed File System(HDFS). In the following text, a Hadoop node might denote a tasktracker or jobtracker machine. A map task describes the work executed by a mapper on one input split. A reduce task process records with the same intermediate key. A mapper/reducer might be assigned multiple map/reduce tasks. In this work, Hadoop is run on the Amazon Elastic Compute Cloud (EC2) – a web service that provides reconfigurable compute resources.

3. ALGORITHMS

3.1 Map-Reduced Bayesian Update (MRBU)

In this section, we introduce the BN Learning MapReduce algorithm for large data sets. Let tuple $\beta = (\boldsymbol{X}, \boldsymbol{W}, \boldsymbol{\Theta})$ be a BN, where $(\boldsymbol{X}, \boldsymbol{W})$ is a directed acyclic graph, with $n = |\boldsymbol{X}|$ nodes, $m = |\boldsymbol{W}|$ edges, and associated set of conditional probability distributions $\boldsymbol{\Theta} = \{\Pr(X_1|\Pi_{X_1}), \cdots, \Pr(X_n|\Pi_{X_n})\}$. Here, $Pr(X_i|\Pi_{X_i})$ is the conditional probability distribution for $X_i \in \boldsymbol{X}$, also called conditional probability table (CPT). If X_i is a root node, we define $\Pi_{X_i} = \{\}$ and thus $\boldsymbol{\Theta}$ contains the probabilities of the root nodes. In the pseudocode for MRBU Algorithm 1, we use $\mathcal{S} = \{\mathcal{S}_1, \mathcal{S}_2, ..., \mathcal{S}_n\}$ to denote the observation value for each node assignment in a training sample, ϕ for its corresponding position in CPT, π_{x_i} for the parent assignments of node X_i, $\overline{M}[x_i, \pi_{x_i}]$ for counts of all the child-parents' combinations, and c_i for pseudo counts.

We specify different mappers to count the observed cases of the positions in the CPT of each node. The *Map(key, value)* method emits "1" for each case that occurs in the training sample. The reducer adds up the counts for each value in the CPT of each node. The *Reduce(key, values)* method emits the sum. The *MergeCPT()* method uses the immediate results emitted by the reducer to calculate the probabilities for all the CPTs. The pseudocount c_i is introduced to solve the zero-probability problem.

3.2 Sequential EM (SEM)

In this work, we implement the basic EM algorithm for parameter learning of BNs with table conditional probability distributions (CPD). In SEM, given the BN structure, its junction tree decomposition, and a number of incomplete data records, we want to determine the probability distribution (BN parameters) that is most likely to produce the observed data.

SEM is an iterative algorithm that alternates between two steps. The computation starts with the initial guess of parameters. In the expectation step, we use the current parameters to compute the expected sufficient statistics. Given each data record and each family, we calculate the joint probability distribution using junction tree propagation as our inference engine [10]. Estimated probabilities of all data records are summed up to give the expected sufficient statistics [9]

$$\overline{M}_{\theta^t}[x_i, \pi_{x_i}] = \sum_m P(x_i, \pi_{x_i}|D[m], \theta^t), \qquad (1)$$

where θ^t represents the parameters at step t, x_i is the variable assignment, π_{x_i} denotes its parents' assignment, and

[1] `http://wiki.apache.org/hadoop/ProjectDescription`

Algorithm 1 MR Learning from Complete Data

Inputs: $\beta, \mathcal{D}$//BN and dataset respectively
Outputs: $\theta_{x_i|\pi_{x_i}}$//BN parameters
Driver:
Execute $Mapper$
Execute $Reducer$
$MergeCPT()$

Mapper:
//$value$: one observation of all the nodes.
//ϕ : corresponding position in CPT
Method $Map(key, value)$
$\mathcal{S} \leftarrow value$
for each $X_i \in \boldsymbol{X}$ **do**
$\Pi(X_i) \leftarrow ReadParents(X_i)$
for each $\pi_{X_i} \in \Pi(X_i)$ **do**
$\phi \leftarrow$ getPositionCPT$(X_i, \pi_{X_i}, \mathcal{S})$
end for
EMIT$(string(\text{“}X_i\ \phi\text{”}), 1)$
end for

Reducer:
//key : the key emitted from mapper
//$values$: the value emitted from mapper
Method $Reduce(key, values)$
$sum \leftarrow 0$
for each val in $values$ **do**
$sum \leftarrow sum + val$
end for
EMIT(key, sum)

Method $MergeCPT()$ //Reads all the reducer's results
for each $i \leftarrow 1, \ldots, n$
for each x_i,$\pi_{x_i} \in Val(X_i, \Pi_{X_i})$ **do**
$\theta_{x_i|\pi_{x_i}} \leftarrow (\bar{M}[x_i, \pi_{x_i}] + c_i)/(\sum_{\Pi_{X_i}} \bar{M}[x_i, \pi_{x_i}] + \sum c_i)$
//c_i: pseudo counts
end for
end for

$D[m]$ is the observation m. In the M-step, we treat the estimated sufficient statistics as complete data and perform maximum likelihood estimation to obtain a new estimate of parameters:

$$\theta^{t+1}_{x|\pi_x} = \frac{\bar{M}_{\theta^t}[x, \pi_x]}{\bar{M}_{\theta^t}[\pi_x]}. \quad (2)$$

The next E-step is performed with the updated set of parameters. The steps are repeated until convergence.

3.3 Map-Reduced EM (MREM)

In this work, we decompose the basic EM algorithm using MapReduce. Since all records in the input data are independent of each other for calculation of the expected sufficient statistics, they can be processed in parallel. The input records can be split between multiple mappers, each running the E-step. The M-step is performed on the reducers.

E-Step: Each mapper takes as input the BN structure β, the current estimate of parameters $\boldsymbol{\theta}^t$, the JT decomposition T of β, and incomplete data $\mathcal{D}$. A mapper runs the E-step on its input records and accumulates pseudo-counts $\bar{M}[x_i, \pi_{x_i}]$ for all child-parents combinations in a hash map. Once the mapper processes all records assigned to it, it emits an intermediate key-value pair for each hash map entry. The emitted key contains state assignments to parents π_{x_i} of the node X_i, whereas the value represents the child variable assignment x_i appended with the soft counts $\bar{M}[x_i, \pi_{x_i}]$ for this entry. This intermediate key makes sure that all variables with the same parents are grouped and processed in the same reduce task.

Algorithm 2 MR Learning from Incomplete Data

Inputs:
β, T // BN and JT Structure
$\boldsymbol{\theta}^0$ // Initial guess of BN parameters
$\mathcal{D}$ // Incomplete data
Outputs: $\boldsymbol{\theta}^{final}$
Driver:
repeat
Execute $Mapper$
Execute $Reducer$
$t \leftarrow t + 1$
until convergence

Mapper:
Method $Map(key, value)$
$s \leftarrow (s_0, s_1, \ldots s_m)$ //read in one observation
Run JT inference on $(\beta, \boldsymbol{\theta})$ to compute $\Pr(X, \Pi_X | s)$
for each $i \leftarrow 1, \ldots, n$
for each x_i,$\pi_{x_i} \in Val(X_i, \Pi_{X_i})$ **do**
$\bar{M}[x_i, \pi_{x_i}] \leftarrow \bar{M}[x_i, \pi_{x_i}] + \Pr(x_i, \pi_{x_i} | S)$
end for
end for

Method $Run()$
for each $(key, value)$
$Map(key, value)$
end for
for each $i \leftarrow 1, \ldots, n$
for each x_i,$\pi_{x_i} \in Val(X_i, \Pi_{X_i})$ **do**
EMIT$(string(\text{“}\pi_{x_i}\text{”}), string(\text{“}x_i; \bar{M}[x_i, \pi_{x_i}]\text{”})$
end for
end for

Reducer:
Method $Reduce(key, values)$
$sum = 0$
$HashMap \leftarrow newHashMap()$
for each val in $values$ **do**
$sum \leftarrow sum + \bar{M}[x_i, \pi_{x_i}]$
$HashMap.put(string(\text{“}x_i; \pi_{x_i}\text{”}), \bar{M}[x_i, \pi_{x_i}])$
end for
for each $Entry$ in $HashMap$ **do**
key$\leftarrow string(\text{“}x_i; \pi_{x_i}\text{”})$
$\theta^{t+1}_{x_i|\pi_{x_i}} \leftarrow \bar{M}[x_i, \pi_{x_i}]/sum$
EMIT$(key, \theta^{t+1}_{x_i|\pi_{x_i}})$
end for

M-Step: Each reduce method performs the M-step for families with the same parent assignment: it iterates through all the values with the same key, parses the value, and fills a hash map, in which keys correspond to child-parent combinations and their states, and values correspond to the soft counts. Values are summed up to obtain the parent count. Finally, each reduce function emits an output key-value pair

for each hash map entry. The output key is of the form x_i, π_{x_i}; the output value represents a newly estimated parameter $\theta^{t+1}_{x_i|\pi_{x_i}}$, i.e. the quotient of the soft and parent counts.

3.4 Design Choices for MREM

We use *hard-coded combiner* and *optimal resource utilization* to improve the performance of MREM.
In a naive map-reduced implementation, each mapper emits a key-value pair per family assignment per processed datum, while in MREM, a mapper would combine the values associated with the same key from all the data records assigned to it before passing intermediate key-values to a reducer. This *hard-coded combiner* represents a major optimization as it dramatically decreases the number of records transmitted to the reducer.
The MREM combine operation will be effective if the number of items to be combined is large. Therefore, instead of relying on Hadoop's input splitting policy, we split the data set into chunks whose size depends on the number of map tasks that can be run in parallel. To ensure that no mapper remains idle while others are processing large amount of data, we assign each mapper only one map task which corresponds to processing one input split.
Figure 1 shows the speed-ups on the top of the bars for naive implementation and our MREM with respect to the sequential one.The time axis (vertical) refers to the average time to complete one EM iteration. The map-reduced algorithms are run on large EC2 instances with 4 tasktracker nodes; the sequential version is run on the small EC2 instance. For a 100K input dataset, the naive implementation gives $\sim$ 1.4x speed-up over the sequential version; the optimized one shows over 14x speed-up.

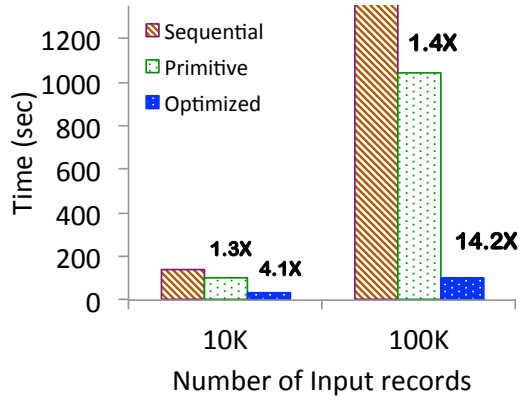


Figure 1: Total execution times of the sequential, naive and optimized implementations on ADAPT_T1. Sequential code is run on a small EC2 instance; the map-reduced code is run on a large EC2 instance with 4 tasktracker nodes.

4. ANALYSIS OF EXECUTION TIME

In this section, we estimate the performance benefit of the MREM algorithm over its sequential counterpart. We derive analytical expressions for the runtime of one iteration of the sequential EM and MREM. Due to space limitations, we omit some detailed derivations.

4.1 Sequential EM

The **E-phase** consists of two steps: *computing marginals using belief propagation* and *calculation of pseudocounts* for all input data records. We will denote the time taken by these steps for each data record as t_{bp} and t_{pc} respectively. Hence, the total time to complete this phase for n input records is $T_{E_s} = [t_{bp} + t_{pc}]n$.
In the **M-phase** all parameters in the CPT are recalculated and this requires calculation of parent counts. As the total time required for this phase is directly proportional to the number of parameters $|\theta|$ of the Bayesian network, we get $T_{M_s} = t_{c_1}|\theta|$.
Since the implementation is sequential, we can add the time of the two phases to find the total time (T_{seq}) taken by one EM iteration,

$$T_{seq} = [t_{bp} + t_{pc}]n + t_{c_1}|\theta|. \quad (3)$$

4.2 MREM

In MREM, the E-phase and the M-phase are done by M mappers and R reducers present in the compute cluster. Unlike the sequential EM, at the end of each MREM iteration the newly computed BN parameters need to be updated in the HDFS so that every mapper gets these values before the beginning of the E-phase in the next iteration. Thus, along with E- and M-phases there is one more step *Update BN*.
Map phase: According to the implementation of MREM, each mapper processes at most $\lfloor \frac{n}{M} \rfloor$ records from the input file. As Mappers execute concurrently, the time required to complete the E-step in MREM is

$$T_{Emr} = (t_{bp} + t_{pc}) \left\lfloor \frac{n}{M} \right\rfloor + t_{trns}|\theta|, \quad (4)$$

where t_{trns} reflects the time to transmit each key-value pair over the network. We considered this time to be a part of the E-step. This time is proportional to the size of transmitted data $|\theta|$ for each mapper.
Reduce phase: Mappers emit key-value pairs with keys as *parent assignments* "π_{x_i}". Let us define a set Ξ to represent all possible parent assignments for the network β, i.e. $\Xi = \{\pi_{x_i} | \pi_{x_i} \in Val(\Pi_{X_i}) \forall X_i \in \mathbf{X}\}$. We will denote the members of the set Ξ as ξ_j and its cardinality as $|\Xi|$. Hence, each mapper can emit at most $|\Xi|$ intermediate keys. All values associated with every intermediate key ξ_j for $j \in [1, |\Xi|]$ will generate one reduce task which results in $|\Xi|$ reduce tasks. So each of the R Reducers in the MapReduce framework will be assigned at most $\lceil \frac{|\Xi|}{R} \rceil$ reduce tasks.
Each reduce task obtains the parent counts and re-estimates the parameters $\theta_{x_i|\pi_{x_i}}$ as mentioned in Section 3.4. Among all key-value pairs emitted by each mapper, those pairs will have the same key ξ_j which correspond to node assignments associated with the same parent assignment i.e $\{(x_i, \bar{M}[x_i, \pi_{x_i}]) | \pi_{x_i} = \xi_j\}$. We will denote this set as ν_{ξ_j} and note that

$$|\nu_{\xi_1}| + |\nu_{\xi_2}| + |\nu_{\xi_3}| + ... + |\nu_{\xi_{|\Xi|}}| = |\theta|. \quad (5)$$

As all the intermediate key-value pairs emitted by all mappers are accumulated by the MapReduce library, the maximum possible value with the key ξ_j is $M\nu_{\xi_j}$. Hence, a reducer with r ($r \leq \lceil \frac{|\Xi|}{R} \rceil$) reduce tasks will take maximum time to finish if ($|\nu_{\xi_1}| + |\nu_{\xi_2}| + ... + |\nu_{\xi_r}|$) is maximum for

it:

$$T_{Mmr} = \sum_{k=1}^{r}(M|\nu_{\xi_k}|t_h + |\nu_{\xi_k}|t_{div}), \qquad r \leq \lceil \frac{|\Xi|}{R} \rceil$$
$$= (Mt_h + t_{div}) \sum_{k=1}^{r} |\nu_{\xi_k}|. \quad (6)$$

Update phase: At the end of each iteration, the file in HDFS containing the CPT is updated with the recently calculated values. If writing one entry to the file takes t_{write} time, the total time required to update the entire CPT is $T_{Umr} = t_{write}|\theta|$.
Hence, the total time taken by one iteration of MREM,

$$T_{mr} = T_{Emr} + T_{Mmr} + T_{Umr}$$
$$= (t_{bp} + t_{pc}) \left\lfloor \frac{n}{M} \right\rfloor + t_{prop}|\theta|$$
$$+ (Mt_h + t_{div}) \sum_{k=1}^{r} |\nu_{\xi_k}| + t_{write}|\theta|. \quad (7)$$

As equation (5) implies $\sum_{k=1}^{r} |\nu_{\xi_k}| \leq |\theta|$, we can approximate T_{mr} as follows,

$$T_{mr} \approx (t_{bp} + t_{pc}) \left\lfloor \frac{n}{M} \right\rfloor + t_{c_2}|\theta|, \quad (8)$$

where t_{c_2} is a constant for a compute-cluster and captures the aggregate effect of the last three terms in (7).
From (3) and (8) we compute the speed-up (Ψ) for the MREM algorithm compared to the sequential EM:

$$\Psi = \frac{T_{seq}}{T_{mr}} \approx \frac{(t_{bp} + t_{pc})n + t_{c_1}|\theta|}{(t_{bp} + t_{pc}) \left\lfloor \frac{n}{M} \right\rfloor + t_{c_2}|\theta|} \quad (9)$$

From this equation we observe that as n increases, the numerator (T_{seq}) increases at a higher rate compared to the denominator (T_{mr}). At some point T_{seq} exceeds T_{mr}, making $\Psi > 1$. For sufficiently large values of n (depends on network parameters), the following expression holds

$$(t_{bp} + t_{pc}) \left\lfloor \frac{n}{M} \right\rfloor \gg t_{c_2}|\theta|. \quad (10)$$

In this situation, the MREM algorithm reaches its peak performance with the speedup $\Psi_{max} = M$. However, from the expressions of t_{c_1} and t_{c_2} we see the latter is much greater due to the factor $t_{prop} + t_{write}$. Hence, for very small values of n ($n \approx M$), the denominator in (9) will be greater than the numerator. In this case MREM will be slower than sequential EM.

5. EXPERIMENTS ON HADOOP AND DISCUSSION

5.1 Experimental Set Up

We experiment with three types of EC2 compute nodes: *small*, *medium*, and *large* instances.[2] We test both implementations on a number of complex BNs[3](see Table 1) that originate from different problem domains and vary in size and structure. Comparison between different cluster configurations is performed on several variants of ADAPT - a BN representation of an electrical power system, based on the ADAPT testbed provided by NASA for benchmarking of system health management applications.[4]

All algorithms are implemented in Java. In the MREM analysis, we calculate speed-ups based on per-iteration execution time, which is measured as the average of the runtime across 10 iterations of the EM algorithm.

5.2 Correctness Check

First we test the correctness of MREM, by comparing the results generated by MREM and sequential EM. Both algorithms are executed on the EC2 computers in linux environment to estimate the parameters of 3 different ADAPT networks. We find that after every iteration, the parameter estimates of both algorithms are exactly same. This happens due to the inherent similarity of the two implementations and thus we conclude that MREM is a correct implementation of EM.

5.3 MRBU Experimental Results

5.3.1 Varying Bayes Nets and Data Set Sizes

We vary the data set size, using 10K, 50K, 100K, 500K and 1,000K training samples to train each BN. Figure 1 illustrates the training time of different BNs in a single small instance. It is obvious that increasing the size of the training samples leads to increased training time. The training time of Munin2, Munin3 and Munin4 is growing much faster, while the training time of Mildew, Barley and Powerplant is growing much slower. This is because they have different structures and different number of nodes.

Table 1: Summary of BNs

BayesNet	Nodes N	Edges E	$\|\theta\|$
Water	32	66	13,484
Mildew	35	46	547,158
Powerplant	46	42	448
Barley	48	84	130,180
Diabetes	413	604	461,069
Link	724	1,125	20,502
Munin2	1,003	1,244	83,920
Munin3	1,044	1,315	85,855
Munin4	1,041	1,397	98,423
ADAPT_T1	120	136	1,504
ADAPT_P2	493	602	10,913
ADAPT_P1	172	224	4,182
ADAPT_T2	671	789	13,281

5.3.2 Varying the Number of Hadoop Nodes

For each BN of the same training size, we vary the number of Hadoop nodes to parallelize the algorithm. Figure 2 shows the changes in the training time when using different number of Hadoop nodes. This experiment is run in Amazon Elastic MapReduce, and the training data are stored in Amazon S3. Thus, the performance of the MapReduce job is affected by some external factors such as the network bandwidth between the Amazon S3 and Amazon EC2. In 2 (a), the number of training sample is 10K. When increasing the number of Hadoop nodes, the training time does not always decrease. This is because the training data is small (no more than 51MB), and the external factors and

[2] `http://aws.amazon.com/ec2/`
[3] Other than ADAPT: `http://bndg.cs.aau.dk/html/bayesian_networks.html`
[4] ADAPT: `http://works.bepress.com/ole_mengshoel/29/`

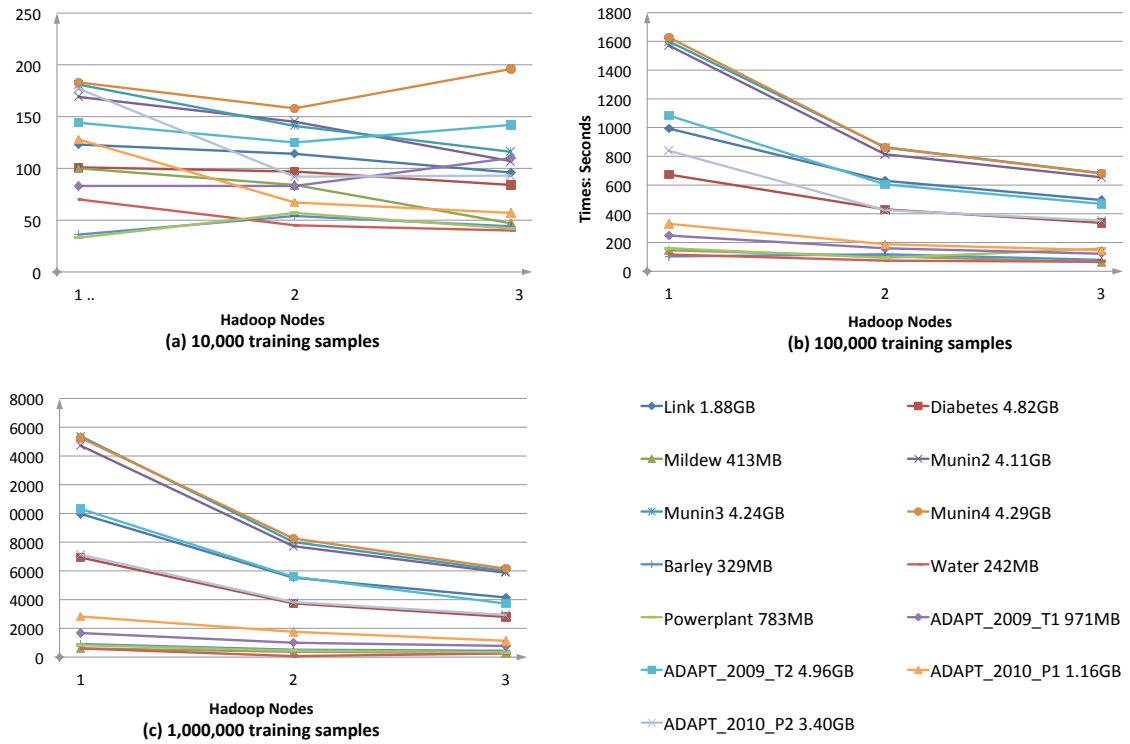


Figure 2: Varying the Bayesian network(Link, Mildew,...) and data set size as well as the number of Small Hadoop nodes.

overheads of Hadoop are more obvious. Figure 2(b) and Figure 2(c) shows that for large data sets (100K and 1,000K training samples), increasing the number of Hadoop nodes significantly reduces the training time. Complexity of the BN also affects the parallelization speed up. For example, Munin2, Munin3 and Munin4 have similar speed up in Figure 2 (b) and Figure 2 (c) because they have very similar number of nodes and edges. They also have many more nodes and edges than Barley, Water and Powerplant; thus they show more significant speed ups with increasing number of Hadoop nodes.

5.3.3 *Varying Hadoop Parameters*

The performance of Hadoop is related to its job configurations, e.g. the number of mappers and reducers. We train the Link BN with 100K training samples using 1, 2, 10, 20, 50 and 100 mappers using a different number of Hadoop nodes. The result is shown in Figure 3.

When increasing the number of map tasks from 1 to 10, the average map time and the run time decrease. However, when the number of map tasks is increased from 20 to 100, even though the average time taken by map tasks decreases, the total running time does not decrease, or even increases a little. This happens because, first, too many mappers lead to over-splitting of the data, introducing too much over-head and, second, the average shuffle time also increases. Figure 3 shows that the minimum running time occurs for 10 map tasks. On a Small instance, two map tasks can be run in parallel. So the right level of parallelism for maps is around 10 map tasks for the 5 nodes case.

5.4 MREM Experimental Results

5.4.1 *Varying Data Set Sizes for MREM*

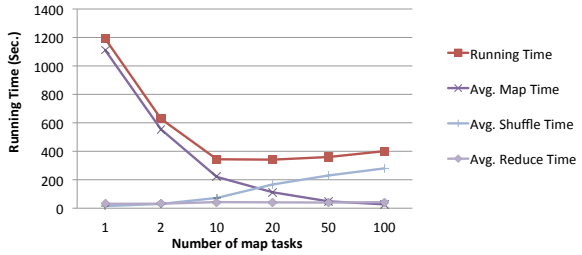
From equations (3) and (8), the runtimes of both the sequential and MREM algorithms increase linearly with increasing number of data records but at different rates. In this section, we compare the sequential EM and MREM for input records varying from 1 to 1,000K. Both algorithms are executed on Small Amazon EC2 instances. For MREM, 4 mapper nodes of the same instance have been used so that the value of $M = 4$.

As described in Section 4.2, the speed-up (Ψ) achieved by the MREM in case of small data sets is typically smaller due to the overhead of MapReduce. For small datasets, we can estimate the minimum size of input data records, which exhibits beneficial results (i.e $\Psi > 1$) when using MapReduce.

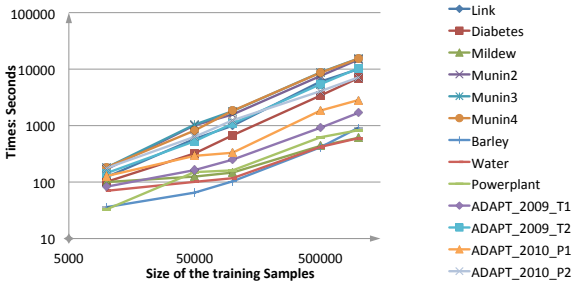
Figure 4 shows the plots of achieved speed-ups of MREM relative to the sequential version in semi-logarithmic scale. Markers denote the experimental data, while continuous lines represent the hyperbolic fit to the data. For all BNs, the curves are well described by a rational function in n, i.e. can be written in the form of (9):

$$\Psi \approx \frac{(t_{bp} + t_{pc})n + t_{c_1}|\theta|}{(t_{bp} + t_{pc})\left\lfloor \frac{n}{M} \right\rfloor + t_{c_2}|\theta|} = \frac{An + B}{Cn + D}$$

The best fit is achieved for small ADAPT BNs that also get up to 4x speed-ups (linear with the number of compute resources). This behavior is consistent with our mathematical analysis of Ψ and confirms that the input data-size required to gain sufficient performance improvement



(a)Run Time, Average Map, Shue and Reduce time in 5 Hadoop nodes



(b)Varying the size of training samples of different Bayesian Networks

Figure 3: Run time, average Map, Shuffle and Reduce time in 5 Hadoop Small nodes.

($\Psi \approx \Psi_{max}$) depends on the BN to a great extent.

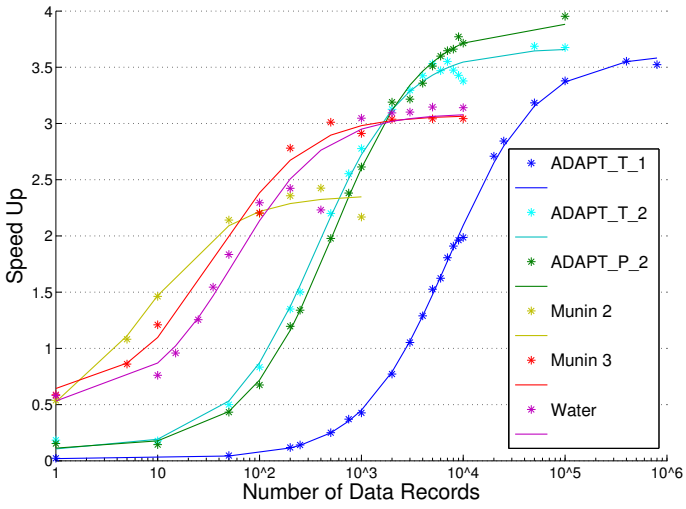


Figure 4: Speed up of MREM relative to sequential EM for data set sizes ranging from $n = 1$ to $n = 1,000K$. MREM is run on Small EC2 instances with 4 mapper nodes.

The cross-over point for which the sequential runtime exceeds the runtime on Hadoop also depends on the type and size of network. We observe that for networks with the largest CPTs (ADAPT_P2 and ADAPT_T2), running Hadoop starts giving improvements for data sizes with less than 200 records. This result is expected since for complex networks, the cluster start-up overhead quickly becomes negligible compared to the runtime of JT inference. In this case, distributing workload across multiple nodes pays off even for small training sets. Yet, for ADAPT_T1 the cross-over point is shifted to 2.8K data records - a point at which inference runtime in sequential code becomes comparable to the cluster set-up time. The *cross-over points* for the three ADAPT networks run on the Amazon EC2 Small instance with four mapper nodes are shown in Table 2.

Table 2: Data size at which MREM (run on Small instance with 4 nodes) completes in the same time as sequential EM

BayesNet	$\|\theta\|$	Total CPT Size	Records
ADAPT_T1	1,504	1,690	2.8K
ADAPT_P2	10,913	32,805	160
ADAPT_T2	13,281	36,396	130
Water	13,484	3,465,948	20
Munin3	85,855	3,113,174	10
Munin2	83,920	4,861,824	5

Figure 4 also shows that for Munin2, having the largest total JT size, Ψ never reaches $\Psi_{max} = 4$. This reminds us of the limitations of the distributed computing instance we are using. For a big JT, the *heap memory* allocated to the Java Virtual Machines is almost exhausted which requires garbage collection to process more records. Consequently, much longer time is required to complete iterations of MREM for very large JTs with sufficiently high data sizes. Using *Medium* or *Large* instances would help to counteract this effect as they have more memory available to be allocated as heap space.

5.4.2 *Effect of Instance Types*

We investigate which instance type suits best our application among the *Small*, *Medium* and *Large* instances. We used three cluster configurations with the same cost per hour as *Small*: 16 small compute nodes, *Medium*: 8 medium compute nodes and *Large*: 4 large compute nodes. On these Hadoop clusters we perform the experiments for three ADAPT networks with two different dataset sizes (10K and 100K records). Figure 5 shows that for almost all cases the performance is better for *Medium* instances compared to the corresponding *Small* and *Large* instances.

5.4.3 *Performance on Large Data Sets*

To test the performance of MREM on very large input data, we chose the ADAPT_T1 network with an incomplete data set having 1,000,000 records. In order to deal with big data, we used bigger Hadoop clusters up to 10 compute nodes of large instance type. Moreover, to compare the performance across different Hadoop clusters, we executed the same experiment on five more cluster configurations as follows: 5 Large instances, 10 Medium instances, 5 Medium instances, 10 Small Instances and 5 Small instances.

From Figure 6 we see that one iteration of MREM, which involved processing of 1,000K input records, was finished in only 8 minutes when 10 Large and Medium compute nodes were used. Given cost considerations, our recommendation here is to use medium instances.

6. CONCLUSION

MapReduce is commonly used to distribute computation for vast amounts of data. In this paper, we have applied the framework to BN parameter learning from complete and incomplete data. Running sequential EM to learn the parameters of the ADAPT_T2 network from 100,000 data records

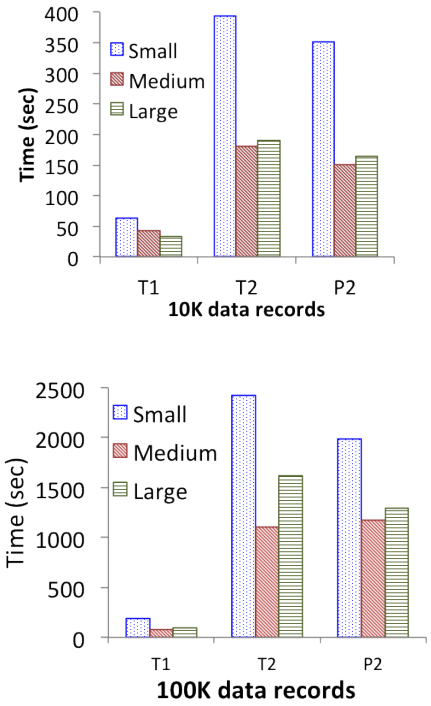


Figure 5: Performance on Small, Medium and Large instances with the same cost for three different Bayesian Networks and two training data-set sizes.

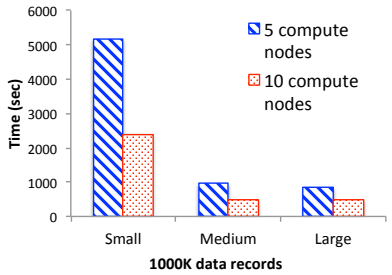


Figure 6: Performance on Small, Medium and Large instances for ADAPT_T1 network with n=1,000K.

takes around 2 hours 30 minutes per iteration. But using our MREM implementation for the same task, on an Amazon EC2 cluster with 5 Large compute nodes, takes only 15 minutes for each iteration.

Moreover, we found that the benefit of using MapReduce depends not only on the size of the input data (as is well known) but also on the size and structure of the network. We have shown that for BNs with large junction trees, MapReduce can give speed-up compared to the sequential EM for learning from 20 cases or fewer. More generally, this work improves the understanding of how to optimize the use of MapReduce and Hadoop when applied to the important task of BN parameter learning.

7. ACKNOWLEDGEMENTS

This material is based, in part, upon work supported by NSF awards CCF0937044 and ECCS0931978.

References

[1] W. L. Buntine. Operations for Learning with Graphical Models. *Journal of Artificial Intelligence Research*, 2:159–225, 1994.

[2] C. Chu, S. Kim, Y. Lin, Y. Yu, G. Bradski, A. Ng, and K. Olukotun. Map-reduce for machine learning on multicore. *Advances in neural information processing systems, NIPS-07*, 19:281, 2007.

[3] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[4] A. Dempster, N. Laird, and D. Rubin. Maximum likelihood from incomplete data via the EM algorithm. *Journal of the Royal Statistical Society. Series B (Methodological)*, pages 1–38, 1977.

[5] C. Do and S. Batzoglou. What is the expectation maximization algorithm? *Nature biotechnology*, 26(8):897–899, 2008.

[6] D. Gillick, A. Faria, and J. DeNero. Mapreduce: Distributed computing for machine learning. *Technical Report, Berkeley*, 2006.

[7] J. Gonzalez, Y. Low, and C. Guestrin. Residual splash for optimally parallelizing belief propagation. *AISTATS-09*, 2009.

[8] H. Jeon, Y. Xia, and V. Prasanna. Parallel exact inference on a CPU-GPGPU heterogenous system. In *Parallel Processing (ICPP), 2010 39th International Conference on*, pages 61–70. IEEE, 2010.

[9] D. Koller and N. Friedman. *Probabilistic graphical models: principles and techniques*. The MIT Press, 2009.

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

[11] V. Namasivayam and V. Prasanna. Scalable parallel implementation of exact inference in Bayesian networks. In *12th International Conference on Parallel and Distributed Systems, 2006. ICPADS 2006.*, volume 1. IEEE, 2006.

[12] J. Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann, San Mateo, CA, 1988.

[13] D. J. Spiegelhalter and S. L. Lauritzen. Sequential updating of conditional probabilities on directed graphical structures. *Networks*, 20:579–605, 1990.

[14] L. Zheng, O. Mengshoel, and J. Chong. Belief propagation by message passing in junction trees: Computing each message faster using gpu parallelization. In *Proc. of the 27th Conference on Uncertainty in Artificial Intelligence (UAI-11)*, 2011.