TECHNICAL UNIVERSITY OF CRETE, GREECE
SCHOOL OF ELECTRONIC AND COMPUTER ENGINEERING

BePadoop
Large-Scale Exact Belief Propagation in Hadoop

Evangelos E. Vazaios

Thesis Committee
Professor Minos Garofalakis (ECE)
Associate Professor Michail Lagoudakis (ECE)
Assistant Professor Vasilios Samoladas (ECE)

Chania, October 2013

Abstract

The critical need for effective processing of inference queries on massive amounts of uncertain/probabilistic data arises naturally in numerous modern application domains. At the same time, the widespread use of large-scale parallel infrastructures (e.g., Hadoop-based clusters) has placed massive processing power at the fingertips of users and applications around the globe, thus enabling fast data analytics over previously unimaginable volumes of real-life data. Still, due to the inherent difficulty and complexity of probabilistic inference, the effective parallelization of such large-scale inference queries continues to pose several difficult research challenges.

In this paper, we present BePadoop the first efficient, Hadoop-based exact inference algorithm (based on Belief Propagation (BP)) for large-scale probabilistic data analysis. BePadoop relies on smart pre-processing of the graphical model and takes advantage of the crucial observation that, during BP over the model's junction tree only a small slice of vertices are ready to send informative messages to their neighbors; furthermore, these computations are independent of each other and can be effectively parallelized. This also allows us to reduce the communication cost between the Hadoop map and reduce phases. To further improve efficiency, we provide an alternate representation for model cliques which has linear space requirements, thus drastically reducing the size of each junction-tree vertex. Extensive experiments with BePadoop over large probabilistic datasets have verified the effectiveness of our approach.

Contents

List of Figures

Chapter 1

Introduction

Graphical models have been established as a powerful tool for a wide range of domains that require processing of uncertain data such as medical diagnosis, natural language processing, data mining and so forth. Inference on graphical models is an integral part for such applications, thus a lot of effort has been put into the development of both exact and approximate inference algorithms that, for example, can compute the marginal distribution of a set of variables in a graphical model given some evidence. It has been proven [3, 22] that both exact and approximate inference are NP-hard problems; furthermore, the storage requirements for exact inference in generic graphs with cycles is exponential. Thus, a lot of effort has been put into parallelizing graph algorithms [6, 9] and easing the development of such algorithms. In recent years, the high availability of processing power has given rise to Google's MapReduce parallel programming paradigm (introduced in [4]), which enables wider audience to develop parallel data processing tools. As a result, machine learning, data mining [1] and approximate inference [12] algorithms for large datasets are now publicly available. Additionally, new vertex-centric paradigms have been proposed such as Pregel[19], GraphLab[17] and PowerGraph[8], in order to simplify the development of parallel graph algorithms, however, these models are not as widely deployed as MapReduce and work efficiently only when the vertex function could either be factorized or is relatively small.

Belief Propagation (BP) is a widely adopted iterative message passing inference algorithm, which is used in a wide variety of applications including computer vision [5], statistical physics [24], information theory [20] and medical diagnosis. In order to apply BP for exact inference on general graphical models, we must induce a new acyclic hyper-graph (Junction Tree), which in most cases has larger storage requirements than the original graphical model. Various BP algorithms have been proposed for exact inference on uncertain data [18, 23], but these algorithms are designed for multicore systems with shared memory across all CPU rather than scalable cluster architectures. In order to tackle an exact inference problem at a larger scale, in this work we introduce BePadoop, a novel MapReduce algorithm for exact inference on massive

graphical models, that takes advantage of the parallel nature of exact inference both structurally and computationally. We present a junction tree representation that tries to circumvent the enormous storage requirements and, in fact, requires only linear space requirements in terms of the original graphical model. The aforementioned representation also enables us to introduce a lazy message computation technique that reduces the space requirements of huge messages. Additionally, through smart pre-processing, BePadoop minimizes the number of iterations and the communication cost between the Map and the Reduce phases.

The rest of the thesis is organized as follows. In Chapter 2, we provide background knowledge for exact inference and the Apache Hadoop MapReduce framework and then, we introduce and analyze our approach in chapter 3,next in chapter 4 we present the related work and briefly discuss other frameworks, which were not chosen for implementing our algorithm. In chapter 5 provides the experimental results and finally, in chapter 6 we discuss future research directions and conclude this thesis.

Chapter 2

Background

2.1 Graphical Models

Probability is a common notion in our day-to-day life, refering to the degree of confidence that an event of uncertain nature will occur, such as the validation of a weather report, and the result of rolling a dice, a football game or a medical test. The fundamental mathematical concept used to handle such events is that of *random variables*. A random variable defines an association of the possible outcomes of each event to a numeric value. Over the years, a large number of real-world applications have been developed to enable storing, processing and inference over a joint probability distribution of random variables.

In general, representing a joint distribution P of a set of random variables $X = \{x_1, .., x_n\}$, where each variable can take k values requires k^n numbers. It becomes apparent that naively generating a large table with k^n values is inefficient and essentially impossible for large k, n. However, for the vast majority of the problems at hand, each random variable is correlated only with a small subset of random variables. Graphical models [11, 14] are a well established tool for succintly representing joint probability distributions over a large set of variables $\mathbf{X} = \{x_1, ..., x_n\}$ with limited correlations. They have been used successfully in a wide range of domains including, computer vision, sensor networks, social networks and data integration systems. There is a large number of different graphical models, but in most cases, a graphical model consists of two components, *a graph* $G(V, E)$ depicting the correlations between the random variables and a set of functions over subsets of random variables called *factors*. In terms of the directionality of the graph component, graphical models are divided into two categories, directed and undirected graphical models.

2.1.1 Directed Graphical Models

The class of directed graphical models, (that includes the popular Bayesian networks [10]) are used to represent causal or asymmetric interactions among a set of variables, using a directed acyclic graph (DAG). Each node in the graph correpsonds to a random variable, and for each directed edge from variable x_i to variable x_j, we refer to x_i as parent of x_j, indicating that the value of x_i directly influences x_j's possible value. In general, variable x_j can have a set of parent variables, denoted $pa(x_j)$. A factor is a conditional probability table, that for each node x_j quantifies the dependency of x_j on its parents. Consider the following example shown in Fig.2.1, where we have a simple bayesian network for modeling whether we are going to be informed when our house's alarm is activated.

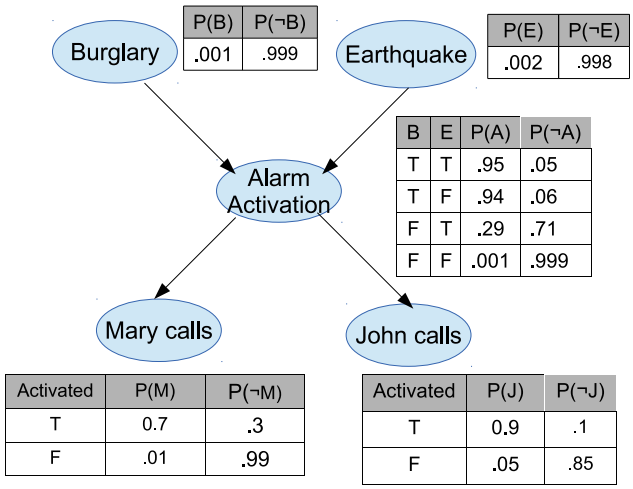


Figure 2.1: A simple Bayesian Network for modeling the event of one of our neighbors calling us.

The probability distribution modelled by a directed graphical model can be factorized as follows:

$$P(x_1, .., x_n) = \prod_{i=1}^{n} P(x_i|pa(x_i)) \tag{2.1}$$

Note that in Equation 2.1, the factors associated with the graphical model are only the conditional probability distributions over a node given its parents in the graph. Since Bayesian networks are easy to design and interpret, they and their variants such as Kalman filters, dynamic bayesian networks are extensively used in practice.

2.1.2 Undirected Graphical Models

Contrary to directed graphical models, undirected graphs or Markov networks are used in problems where there is no natural directionality in the interactions between variables and the interactions are more symmetric. Some examples include the interactions between atoms in a molecular structure, the dependencies between the values of pixels of an image or modeling the effect of society on a an individual. The joint probability is represented by an undirected graph, where each vertex v_i corresponds to a random variable x_i and the edges capture the interactions between those variables, encoded by factor functions[1] $\{f_i : X_i \rightarrow \Re^+ | X_i \subseteq V\}$. Let a set of factors F over a set of variables $X = \{x_1, x_2, ...x_n\}$, then the joint probability is denoted by

$$p(X) \propto \prod_{f_i \in F} f_i(X_i) \tag{2.2}$$

An example of an undirected graphical model is shown in Fig 2.2a, which is a 3x3 grid pairwise Markov random field (MRF). Pairwise MRFs are widely used in the computer vision domain for representing the correlation between adjacent pixels in images. Each vertex corresponds to a random variable representing the value of the respective pixel (black or white) and each edge to a factor depicting the interactions between adjacent pixels. Neighboring pixels are more likely to have the same value. Note, that contrary to the directed graphical models, the factors are not necessarily probability distributions.

2.2 Junction Tree

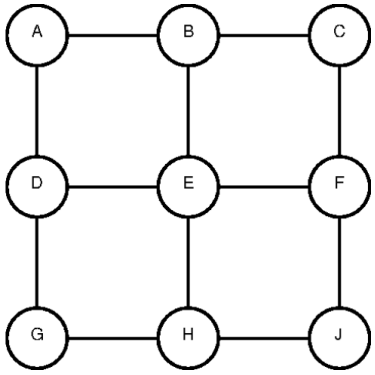
In this thesis, we focus on junction trees. A junction tree is a cycle-free undirected hyper-graph induced by a graphical model. It allows reusing work for several inference queries on the same graph. Alg. 1 shows the Hugin algorithm that is used for creating a junction tree from a graphical model. In cases where we have a directed graphical model we must first moralize the graph, by dropping the directionality of the edges and connecting all parents of each vertex. The next step is to triangulate the graph. A triangulated graph is a graph in which there is no chord-less cycle with more than three vertices. To triangulate a graph, additional edges are added in order to eliminate cycles of four or more vertices. Fig. 2.2b depicts one potential triangulated graph resulting from Fig 2.2a. From the triangulated graph, we induce another graph which is usually called clique graph[2]. The vertices of a clique graph are the maximal cliques of the triangulated graph for instance, in 2.2b, these maximum cliques are: $ABD, GHD, BCF, FHJ, BEHD, BEHF$. Each clique C_i contains a subset

[1] with a lower case x we denote a random variable and with a capital $\mathbf{X}$ a set of random variables

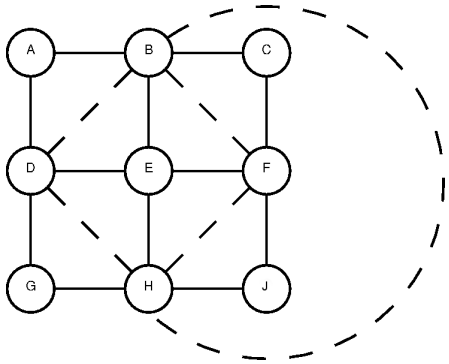
[2] In the bibliography, a clique graph is also referred to as a cluster graph.

A	D	value
B	B	30
B	W	5
W	B	5
W	W	30

D	G	value
B	B	30
B	W	5
W	B	5
W	W	30



(a) A 3x3 grid like pairwise Markov Random Field of a 3x3 black and white image. For demonstration purposes only, we have only included two factors out of twelve



(b) The triangulated graph of Fig. 2.2a. The edges, which are added due to triangulation are marked with dotted lines.

Figure 2.2

of variables $X_{C_i} \subseteq \mathbf{X}$ and stores a factor function over these variables, called clique potential and denoted by $\phi_{C_i}(X_{C_i})$. Each edge connecting two cliques C_i, C_j is associated with a separator set $S_{i,j} = X_{C_i} \cap X_{C_j}$, which also represents the joint potential of the separator's variables $\phi(S_{i,j})$[1]. Furthermore, we assign to each edge connecting cliques C_i, C_j a weight which is equal to the number of the common variables between the cliques, $|C_i \cap C_j|$. The final step to create a junction tree is to run a maximum-weight spanning tree[2] algorithm on the clique graph; the end product of all these five steps is a junction tree of the original graph. After the construction of the junction

[1] we use ϕ to denote factors of the junction tree and f for factors of the original graphical model.

[2] A maximum-weight spanning tree can be found by running a minimum spanning tree algorithm with negated weights.

tree, each factor of the original graphical model is assigned to one clique. Then, these factors are multiplied to compute the value of each clique potential. Fig. 2.3 depicts the junction tree of the 3x3 grid of Fig. 2.2a, cliques are denoted with ellipses and separator sets are marked with squares. The full joint probability distribution represented by the original graphical model is computed by

$$p(X) = \frac{\prod_{C_i \in C} \phi_{C_i}(X_{C_i})}{\prod_{S_{i,j} \in S} \phi(S_{i,j})} \tag{2.3}$$

.

By construction, a junction tree satisfies two important properties:

1. **Family Preserving**: each factor f_i of the original graphical model must be associated with one clique C_i, such that all of the factor's variables are present in the clique. For example the factor of the edge AD in Fig. 2.2a can be associated only with the ABD clique, since there is no other clique that contains the factor variables. On the other hand, the factor associated with the edge BE can be assigned either to BEHD or BEHF. Note that, the assignment of factors affects only the initial value of clique potentials and not the result of the message passing algorithm, we present in the next section.

2. **Running intersection property**: If two cliques C_i, C_j contain the same variable x_i, then this variable x_i must be in all cliques across the path from C_i to C_j. Consider our exampmle, Fig. 2.3, ABD and BCF cliques have B variable in common. So, B is present in all the cliques along the path that connects them.

These properties are very important in order to ensure that the induced junction tree represents the original distribution, and that the marginal probabilities for each variable present in more than one cliques agree with each other.

Algorithm 1: Hugin Algorithm for constructing a Junction Tree

Input: Graphical model g
Output: Junction Tree J

```
if g.directed then
    g ← Moralize (g)
g ← Triangulate (g)
C ← FindMaximalCliques (g)
J ← MaximumSpanningTree (C)
return J
```

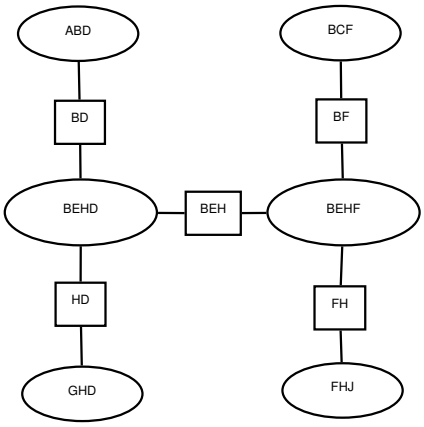


Figure 2.3: The junction tree of the 3x3 MRF of Fig. 2.2a

Each clique/separator contains a subset of variables , whose joint potential is exponential in the number of states of a variable, thus in the worst case where all variables are present in one vertex,[1] the space complexity of a junction tree is $O(max(dom(x_i)^N)$, where N is the total number of variables and $dom(x)$ the domain of a variable x.

2.3 Belief Propagation

Given a graphical model, the most fundamental task is computing the marginal distribution of a set of graphical model's variables. This task is usually referred to as inference. Belief Propagation (BP) is a widely used message passing algorithm for inference on graphical models. Vertices send messages to their neighboring vertices. These messages encode the "belief" of the vertex about the marginal distribution of their common variables. A vertex v_i sends a message $m_{i\to j}$ to a vertex v_j by multiplying its own potential $\phi_i(X_{C_i})$ with all the incoming messages from its neighbors except for the one coming from v_j and then calculates the marginal distribution of their common variables $S_{i,j}$ as shown in the equation below.

$$m_{i\to j}(S_{i,j}) = \sum_{X_{C_i}\setminus S_{i,j}} \phi_i(X_{C_i}) \prod_{k\in N(i)\setminus j} m_{k\to i}(S_{k,i}) \tag{2.4}$$

where $N(i)$ denotes the neighbors of vertex i and $S_{k,i}$ denotes the common variables (i.e. the separator set) between two vertices v_i, v_k. As an example, Fig. 2.4 denotes how the message from the vertex v_{BEHD} to v_{BEHF} is computed. Note that in Eq. 2.4, messages are multi-dimensional arrays, and, as a result, the product is a point-wise operation between tables of different sizes. A point-wise product (division) is obtained

[1] From this point the terms clique and vertex will be used interchangeably

by multiplying (dividing) the outputs of two functions at each combination of common domain values. A full example is shown in Fig 3.5, where three factors are multiplied (A,AD,AB) to create ABD. For general graphs, BP is an approximate inference algorithm that terminates when a convergence condition is satisfied. For example, the difference between all the messages of two consecutive iterations is less than a user defined value. An exact BP algorithm that follows the message passing protocol terminates when all the vertices have produced all their messages. After BP terminates, we can compute the marginal distribution of a variable x_i by choosing a clique C_j which contains x_i and multiplying all the incoming messages with the initial distribution of the clique and then summing out all the variables but x_i as shown below:

$$belief(x_i) = \sum_{X_{C_j} \setminus x_i} \phi_j(X_{C_j}) \prod_{k \in N(j)} m_{k \to j}(S_{k,j}) \tag{2.5}$$

Pearl [21] showed that if the graph has cycles then there are few convergence guarantees and in some practical cases BP may not converge at all. On the other hand, when BP is employed on acyclic structures such as junction trees, is guaranteed to compute exact marginals. Unfortunately, the computational complexity of exact inference is proven to be exponential in the size of the largest clique, which grows quickly in loopy models with many variables. Thus, a lot of work has been put into devising good approximate schemes [2, 7].

2.3.1 Message Passing Protocol

Most of the exact inference algorithms on junction trees respect the message passing protocol, which decides when a given clique is allowed to pass a message to one of its neighbors.

BP message passing protocol A clique can send a message to a neighboring clique only when it has received messages from all of its other neighbors

A belief propagation algorithm that respects the above message passing protocol is an iterative task, which sends the minimum number of messages for the convergence of BP. When used on junction trees, BP results in a procedure that starts from the leaves of the tree and propagates information up to the root vertex, which is an arbitrary vertex chosen at the beginning. When all the information reaches the root, the reverse procedure is initiated, so messages flow from the root down to the leaf vertices. The upward procedure is usually referred to as forward pass and the latter as backward pass. During the forward pass, each vertex v_i sends only one message after receiving $deg(v_i) - 1$ messages, where $deg(v_i) = |N(v_i)|$. On the other hand, during the backward pass each vertex receives the reverse message that it produced during the

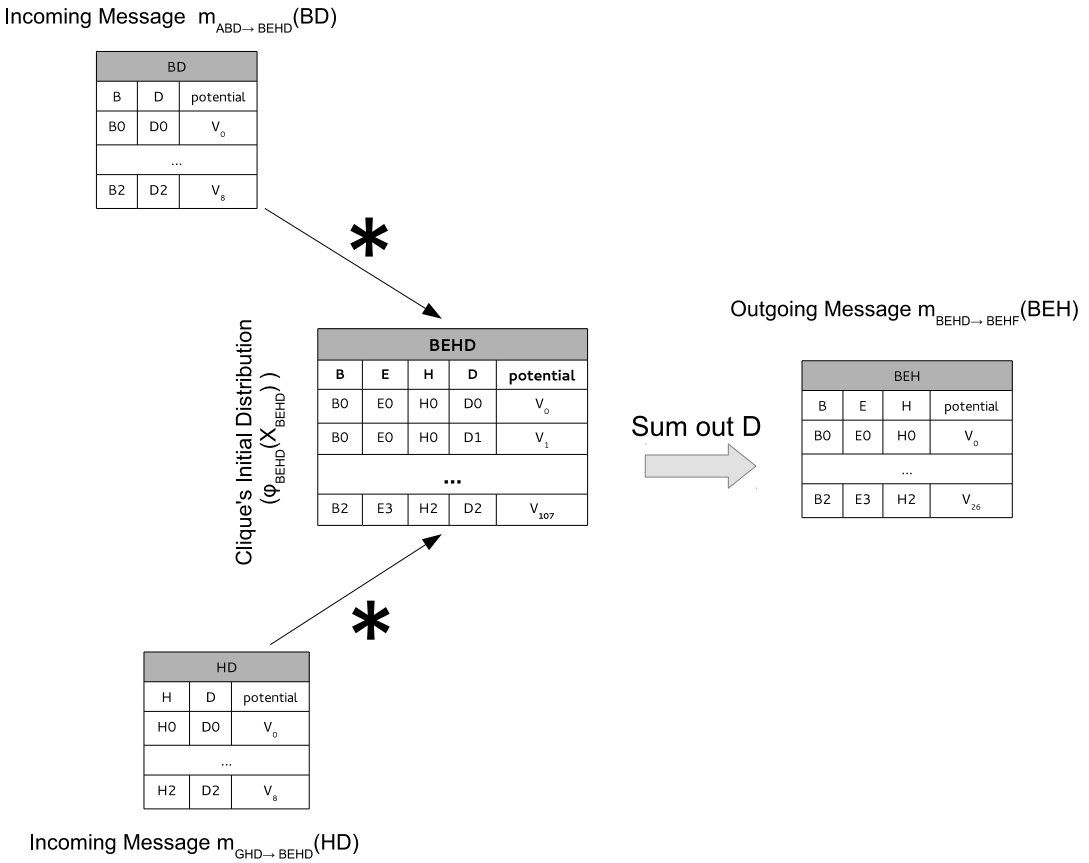


Figure 2.4: Message $m_{BEHD \to BEHF}$ produced by multiplying clique's distribution table $\phi_{BEHD}(X_{C_{BEHD}})$ with incoming messages tables $m_{ABD \to BEHD}(BD)$, $m_{GHD \to BEHD}(HD)$.

forward pass, and, as a result, each vertex has all the required messages to broadcast its remaining messages to its neighbors. The message passing protocol has great impact on parallel BP algorithms because it implies a structural parallelization which can be exploited when running BP on junction trees. Additionally, by following the protocol, we can guarantee that there are no redundant message computations in our algorithm, since each vertex produces only one message for each of its edges.

2.4 Hadoop

Hadoop is a software framework that supports data-intensive distributed applications. Hadoop was inspired by Google's MapReduce framework [4] designed for batch processing large amounts of data. Over the last couple of years a lot of applications have

been developed and a wide variety of algorithms have been parallelized and implemented in Hadoop's setting.

MapReduce is a simple programming model composed by two stages, Map and Reduce. A high level overview of the workflow is shown in Fig 2.5. During the map phase, the input is split and each split is assigned to one Mapper. Each split is read as series of key/value pairs and each pair is given to the Mappers as input. Mappers process those pairs and output other key/value pairs themselves. Those pairs are then shuffled and distributed to the reducers. Before the reduce phase takes place, all key/value pairs are sorted by key and all the values with the same key are grouped together in a list. Reducers take as input and process a key/list of values pair and output key/value pairs. Note that each mapper/reducer is independent of all other mappers/reducers, so they all can be run in parallel. On the other hand, there is a strict synchronization barrier between the Map and Reduce stages.

Hadoop MapReduce provides programmers with high throughput, distributed filesystem (HDFS) and fault tolerance services. When a mapper or reducer fails, Hadoop automatically restarts it to another machine.

Despite the simple programming model and the benefits provided by Hadoop MapReduce there are a lot of challenges when developing a MapReduce algorithm. There are algorithms, for which an efficient port to the MapReduce paradigm is a non-trivial task. An indicative example is some graph algorithms, which require a large number of iterations accessing only a small portion of the graph. In such cases, Hadoop is not the ideal framework to work with. Furthermore, the data shuffling cost (network communication) between mappers and reducers is often the bottleneck in Hadoop MapReduce. Generally, large network data transfers are slow even between machines that are on the same high speed local network; thus, an iterative algorithm that unnecessarily processes the full input in each iteration can be adversely affected by communication cost.

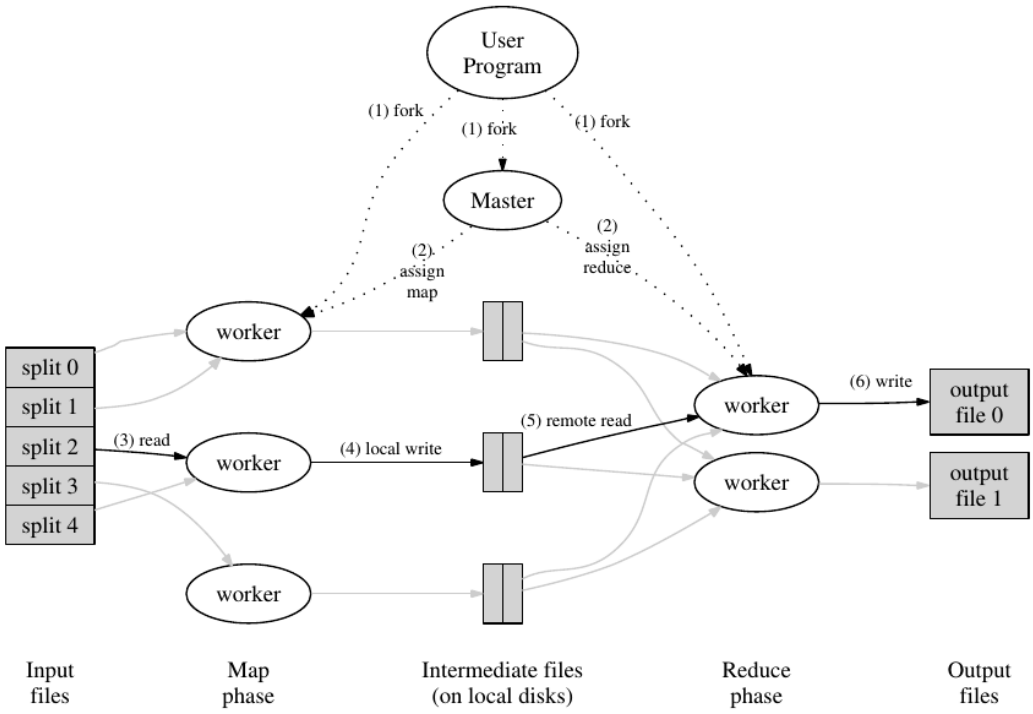


Figure 2.5: Workflow of Hadoop MapReduce

Chapter 3

BePadoop

In this chapter we present our algorithm, BePadoop, an efficient MapReduce algorithm for exact inference on large junction trees. In order to reduce the intermediate data between the Map and Reduce phases, BePadoop takes advantage of a direct corollary of the Message Passing Protocol, namely that each vertex and each message is used only twice; once during the forward pass and once during the backward pass. BePadoop employs a ranking algorithm that ranks vertices according to the iteration vertices are ready to send at least one message. As a result, given an iteration k BePadoop reads only the vertex information required for that specific iteration. Moreover, to address the exponential space requirements we introduce a new clique representation that has linear space requirements in the factors assigned to each clique, in addition to a lazy way for computing messages.

Apart from the space requirements of our problem, computational complexity is another difficult problem that we must address. During an iteration, if there is a clique that requires processing time significantly greater than the rest, then this clique will dominate the time of that iteration. In order to process such huge cliques efficiently, BePadoop effectively parallelizes their computation.

In the rest of this chapter, we first discuss the pre-processing steps we employ in order to reduce communication cost. Then, we present the alternative representation for the clique potential tables, which does not suffer from the aforementioned exponential space requirements, followed by the introduction of a simple lazy computation approach for message computation. Finally, we introduce our initialization procedure and the two MapReduce jobs used by the BePadoop algorithm.

3.1 Ranking

BePadoop follows the message passing protocol, as a consequence, each vertex is used exactly twice, once during the forward pass and another during the backward pass.

The BePadoop Ranking procedure computes iterations of our algorithm at which each vertex is used to produce messages. It is composed of two procedures - the forward and backward ranking. Forward Ranking is a bottom up procedure starting from the leaf vertices, which have forward rank equal to one. We rank vertex v_i after $deg(v_i) - 1$ of its neighbors have been forward ranked. So, the forward rank of v_i, $fw(v_i)$, is calculated by:

$$fw(v_i) = \max_{v_j \in N(v_i) \setminus unranked\, neighbor} fw(v_j) + 1$$

The procedure stops when all vertices have been forward ranked. Fig. 3.1a shows the junction tree of Fig. 2.3 fully ranked. During the first iteration of the forward rank, we assign forward rank equal to 1 to the leaf vertices. Then, we can forward rank the two remaining junction tree vertices (BDEH,BEHF) with rank 2. These two vertices are the root vertices of our junction tree; note that, in general a tree can have either one or two roots. Now, we can start the backward rank of the junction tree. Contrary to the forward rank, the backward rank is defined by a breadth first procedure starting from the root vertices and going downwards to the leaf vertices. Let us give a more formal definition of the forward and backward rank. We define the following quantities:

Definition 1 *The **eccentricity** $e(v_i)$ of a vertex v_i is the maximum distance from v_i to any other vertex. If the graph is a tree, then the eccentricity of a vertex v_i is the maximum distance of v_i from any leaf node.*

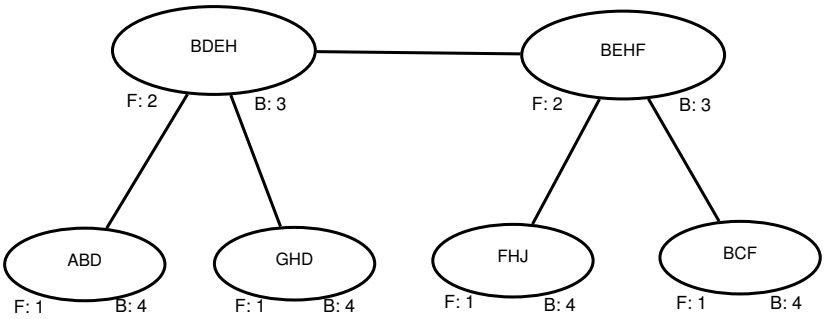
Definition 2 *The **ancestor** of v_i $ancs(v_i)$ is the neighbor of v_i with maximum eccentricity. $ansc(v_i) = \arg\max_{v_j \in N(v_i)} e(v_j)$*

Definition 3 *The **forward rank** $fw(v_i)$ of a vertex v_i is equal to $fw(v_i) = \max_{v_j \in N(v_i) \setminus ancs(v_i)} e(v_j) + 1$. All leaf nodes v_{leaf} have $fw(v_{leaf}) = 1$.*

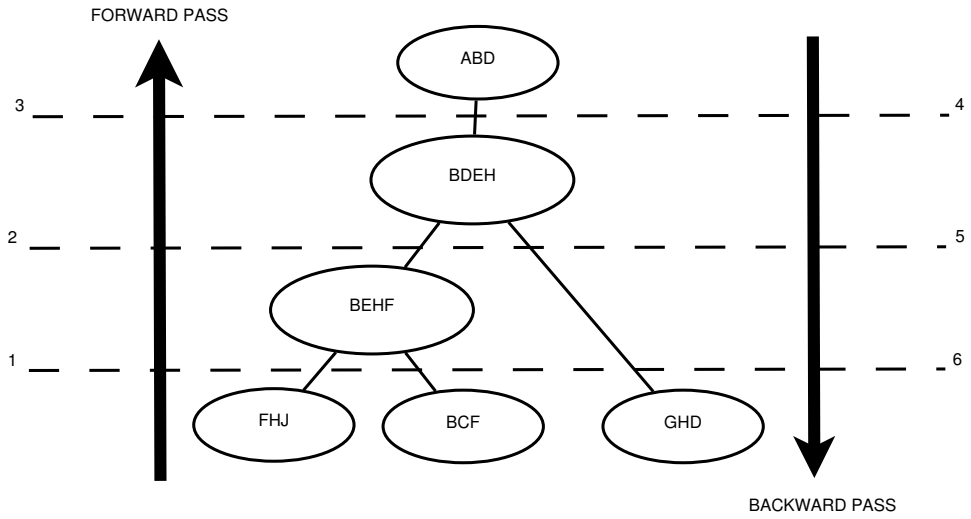
Definition 4 *The **backward rank** $bw(v_i)$ of a vertex v_i is equal to $bw(v_i) = bw(ancs(v_i)) + 1$. When v_i is the only root then the $bw(v_i) = fw(v_i)$. In case the junction tree has two roots, they are the ancestor of each other, as a consequence, the backward rank of each root is equal to $bw(v_{roots}) = fw(v_{roots}) + 1$.*

In practice, the forward rank of a vertex v_i is the minimum iteration, that v_i is ready to send its first message. Similarly, backward rank is the minimum iteration, that v_i can send its remaining $deg(v_i) - 1$ messages. One the greatest advantages of Ranking is that it provides us with sets of vertices that can be processed in parallel. This is shown by the following theorem:

Theorem 1 *All vertices with the same forward (backward) rank f_r (b_r) can be processed in parallel as the computations for their outgoing message(s) produced during rank f_r (b_r) are independent of each other.*



(a)



(b)

Figure 3.1: On top, the fully ranked junction tree of 2.3 and below the forward and backward pass iterations, when we choose as root a leaf node

Proof of theorem 1 We will prove the theorem for vertices with equal forward ranks, but the same holds for the vertices with equal backward ranks. Suppose we are at iteration k and there are two vertices v_i, v_j which have the same forward rank, $fw(v_i) = fw(v_j) = k$,so they are ready to send one message. We are going to prove that the only message each vertex v_i, v_j can produce is along the path that connects them. Additionally, the computations for producing these messages are independent of each other.

Since our graph is a tree, then there is a unique path between every pair of vertices. Let $p = \langle v_i, v_x, ..., v_y, v_j \rangle$ be the unique path between v_i and v_j. We must first prove that at k^{th} iteration there has not been any messages produced along p, thus the only message that v_i and v_j can produce is along the path that connects them. Since, we

are at iteration k, v_i, v_j received $deg(v_i) - 1, deg(v_j) - 1$ messages respectively and they are ready to send one message each. Any vertex v can send its first message after it has received $deg(v) - 1$ messages. Additionally, any intermediate vertex along p has degree at least two, thus either v_i or v_j must first send a message for intermediate vertices to be ready for producing messages. As a result, the forward rank for intermediate vertices along p is greater than the forward rank of both ends as shown in Fig 3.2. Moreover, being at iteration k means that v_j,v_i have received all their messages but one, specifically the one that will pass through p. Consequently,both vertices can send one outgoing message each along p ($m_{i\to x}$,$m_{j\to y}$), but the computations for those messages are independent from Eq. 2.4. Now, we must prove that v_i, v_j cannot produce any other message during the given iteration to ensure that the computations that will take place in v_i and v_j are independent. We follow the message passing protocol, thus only the messages along the edges $\langle v_i, v_x\rangle$,$\langle v_j, v_y\rangle$ can be computed, since all the other messages are dependent on messages that have not yet been produced. As a consequence, we have that v_i, v_j can produce only the messages along p. Moreover, the computations for those messages are independent of each other, because they are not dependent on any message along path p. ■

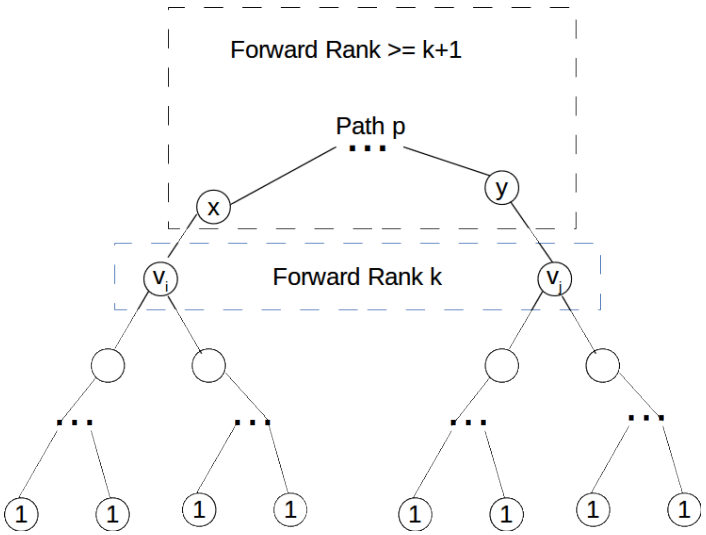


Figure 3.2: Two vertices v_i, v_j with equal rank k

Ranking identifies independent sets of vertices, thus, we can create for each rank files that contain those independent sets in order to read in only the active portion of the tree at a given iteration. Apart from vertices, we can extend a similar idea to messages, as they are also used only twice in BP. As a result, by knowing both the forward and backward ranks for each vertex, we can easily determine at which iterations each message is used. Thus, we can further reduce the input of each iteration by storing messages to files according to the ranks that are going to be used in a similar way

as vertices. In simple words, ranking pre-computes all the required information for each iteration. One side note: vertices with equal forward ranks do not necessarily mean they will have equal backward ranks, so we duplicate vertex information into their respective rank files during BePadoop's initialization. Thus, we only read in the necessary vertices at a given iteration in order to produce new messages, but we never update them. Messages, in the same manner as vertices, are also duplicated into files when they are produced, however they too do not need to be updated after their generation.

Forward/Backward ranking a junction tree not only helps minimize the input of each iteration, i.e., the amount of exchanged data, but also *minimizes the number of iterations* required by our algorithm. The number of iterations in belief propagation on junction trees is dependent on the choice of root, thus we must carefully choose the vertex that will act as root. Since the forward and backward pass have equal numbers of iterations, if we minimize the iterations for the former we also minimize iterations needed by the latter. Intuitively, vertices that have the minimum eccentricity in the tree are the best candidates for roots, and, there can be either one vertex or two vertices with that property. Fortunately, it is not difficult to prove that forward ranking identifies the vertices with the minimum eccentricity, thereby explicitly finding the optimal choice of vertices to be used as roots. As a result, by choosing those vertices we minimize the number of BePadoop's iterations, in order for information to flow from leaf vertices to the root and backwards. Fig 3.1b shows the number of iterations when a random root is chosen, where the total number of iterations will be 6 compared to the optimal 3 iterations required by BePadoop.

3.2 Outlier Cliques

When processing vertices of a given rank there might be some cliques which require significantly more time to be processed than the rest. This is usually the case either because those cliques are bigger in size or have larger degree. Having a way to systematically identify these cliques is important, because we can then parallelize their processing in order to handle them efficiently. Using Eq. 2.4,we can calculate the required number of multiplications and summations for computing an outgoing message. As a result, we define the computation cost for processing clique C_j at rank r, denoted by κ_j^r, as the total number of operations required by the computation of all outgoing messages of C_j at r. Assuming the cost distribution for each rank is Gaussian, we introduce the notion of outlier cliques, which are all cliques that satisfy the following conditions:

1. $\kappa_j^r \geq \mu_r + 2\sigma_r$, where μ_r denotes the mean clique computation cost for rank r and σ_r the standard deviation of the cost distribution at r and,

2. $\frac{\kappa_j^r - \mu_r}{\mu_r} \geq p\%$, where p is a user defined threshold.

The assumption that cost distribution is a Gaussian coupled with our first condition limit the percent of outlier cliques to approximately 2% of all the rank's vertices. Moreover, we use the latter condition to filter cases where the relative difference between a possible outlier clique and the mean cost is not large enough to require special handling. The percent value of p can be defined by the user with 100% as default value. Our approach for characterizing outlier cliques is much better than using an arbitrary static threshold because this requires from the user to have in-depth knowledge of the dataset or the problem at hand, which in most cases is neither desirable nor feasible. It is clear that identifying outlier cliques requires basic statistics of the cost distribution for each rank. This data is extracted during the initialization of BePadoop.

3.3 Clique Representation

A factor $f_j(X_j)$ is a function that depicts the correlations between a set of variables X_j. We represent a factor $f_j(X_j)$ with two components: a variable vector V_{f_j}, which contains the variables that are present in the factor and a table T_{f_j} with the function's output values. Each clique C_i in a junction tree must capture the joint potential over a subset $X_{C_i} \subset \mathbf{X}$ from the variables of the original graphical model. Naturally, in order to represent $\phi_i(X_{C_i})$[1], we use the same representation as factors, namely a a variable vector $V_{X_{C_i}}$ and a table $T_{X_{C_i}}$, which captures the correlations between the variables of X_{C_i}. It is easy to see that the required space to represent any factor f_j is equal to the product of the variables' domain sizes $|f_j(X_j)| = \prod_{x_i \in X_j} dom(x_i)$. As a result, both factors and cliques have exponential space complexity in the domain size of their variables but, in most cases, factors are orders of magnitude smaller, because they represent correlations on much smaller variable sets than cliques. Here, we propose an alternative representation, which has linear space complexity in the number of factors assigned to each clique.

During the construction of the junction tree, each factor f_j from the original graphical model is assigned to only one clique C_i. Let $f_j^{(i)}$ be the a factor f_j that has been assigned to C_i and F_i the set of all the factors assigned to C_i. The Clique's potential $\phi_i(X_{C_i})$ is computed by the point-wise product of the factors in F_i, i.e., $\phi_i(X_{C_i}) = \prod_{F_i} f_j^{(i)}$. Instead of generating $\phi_i(X_{C_i})$ at initialization, we propose to calculate on the fly the required values from the factor set F_i assigned to C_i. This approach enables us to represent junction trees that would be otherwise impossible, due to the size of the clique's potential. Naturally, this comes with a price; there is an increase in the computational cost since we calculate the point-wise product of F_i for

[1]Note that we use ϕ_i to denote the joint potential in a clique C_i and f_j to denote a factor from the original graphical model

each value in $\phi_i(X_{C_i})$ on the fly. On the other hand, this is a price we gladly pay in order to be able to process large graphical models, which induce junction trees with huge cliques. Later, when we discuss the complexity of our approach, we will show that this increase in computational cost is not an issue for our problem.

Let us go back and discuss a case of the point-wise multiplication where the two tables are of different size; when the set of variables present in the smaller table T_s is subset of the set of variables in the larger table T_l, then each value of T_l maps to exactly one value of T_s. We denote as $M_{T_s}^{T_l}$ a mapping function for each potential value of T_l to one value in T_s. When computing $\phi_i(X_{C_i})$ from F_i, we need such a mapping function $M_{f_j^{(i)}}^{\phi_i(X_{C_i})}$, as the clique potential is a superset of all the assigned factors of the original model. This is a consequence of the family preserving property of junction trees, because if a factor $f_j(X_{f_j})$ is assigned to a clique C_i, then X_{f_j} must be a subset of X_{C_i}. Thus, in order to represent $\phi_i(X_{C_i})$ we must create one mapping $M_{f_j^{(i)}}^{\phi_i(X_{C_i})}$ for each factor $f_j(X_{f_j})$ assigned to C_i that maps each value of $\phi_i(X_{C_i})$ to one value of f_j.

Fig. 3.3 denotes how the potential of C_{ABD} from Fig. 3.5 is represented by the three factors assigned to that clique. Fig. 3.3 shows also an example of computing the value of a single index. The index is given as input to the mapping components which map that index to an index in the factors (cells colored in light gray), then, we take the product of those values and calculate the requested value.

We are now ready to introduce the mapping component called index-mapper,which takes as input an index for a value of the larger table ($\phi_i(X_{C_i})$) and returns the mapped index of the smaller table ($f_j^{(i)}$). Before we give a detailed description of how an index-mapper works, we describe the general idea behind it. In our example Fig. 3.3, we observe that D changes its value in every row of the table *ABD* in ascending order (D_0, D_1, D_2). Then B changes its value every $dom(D)$ values and A every $dom(B) * dom(D)$ values. We refer to this quantity as the stride of a variable x_i in the table T_{ϕ_i} of potential $\phi_i(X_{C_i})$, $strd(T_{\phi_i}^{x_i})$. In our example we have that $strd(T_{C_{ABD}}^{D}) = 1$, $strd(T_{C_{ABD}}^{B}) = 3$ and $strd(T_{C_{ABD}}^{A}) = 9$. Moreover, each variable has a certain number of rows needed for a full iteration over its values, which is equal to the product of the domain size of the variable and its stride. In the example shown in Fig. 3.3, we have that the variable D needs 3 rows, B needs 9 and finally A requires 18 rows.

We claim that, given an $index$ and a potential $\phi_i(X_{C_i})$, the following equation computes the index (value) of a variable in a table.

$$val(\phi_i(X_{C_i}), x_i, index) = \left\lfloor \frac{(index \mod (dom(x_i) * strd(T_{\phi_i}^{x_i})))}{strd(T_{\phi_i}^{x_i})} \right\rfloor \tag{3.1}$$

In Eq. 3.1 the nominator limits the possible values to the number of rows needed by x_i for a full iteration and by dividing with the stride of the variable we get the index of x_i's value at that specific index of table $T_{\phi_i(X_{C_i})}$. To continue, in our example of Fig. 3.3,

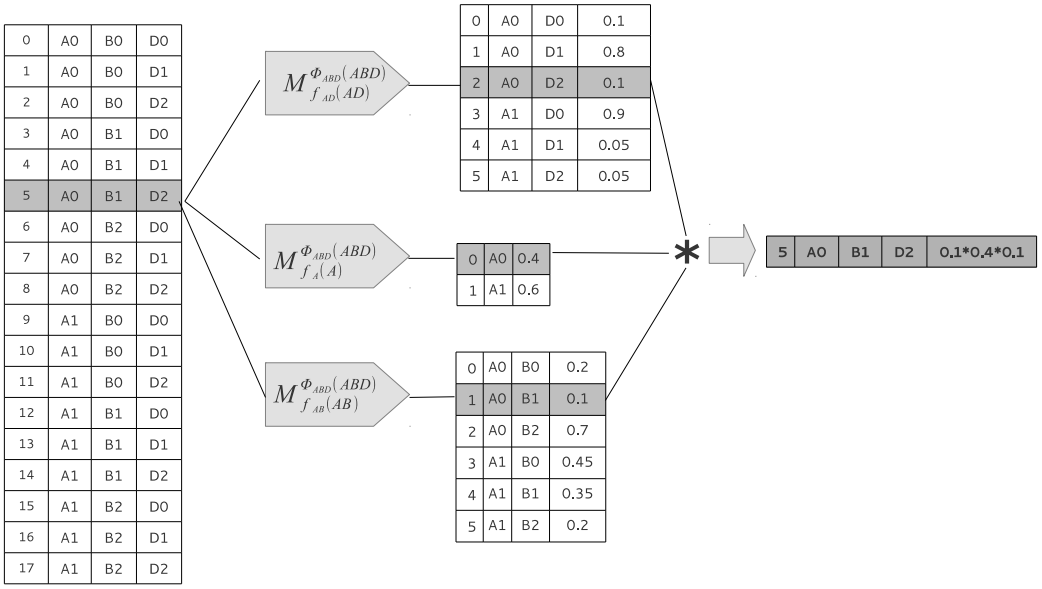


Figure 3.3: BePadoop's clique representation. We represent clique's factor C_{ABD} using the three assigned factors(f_{AD}, f_A, f_{AB}) and the respective mappings ($M^{\phi_{ABD}(ABD)}_{f_{AD}}$, $M^{\phi_{ABD}(ABD)}_{f_A}$, $M^{\phi_{ABD}(ABD)}_{f_{AB}}$). The Cells in light gray show the example for computing the value at index 5.

let us compute the values of A,B,D variables in ϕ_{C_ABD} for index 5 using Eq. 3.1:
For A we have:

$$val(\phi_{C_{ABD}}, A, 5) = \left\lfloor \frac{(5 \mod (dom(A) * strd(T^A_{\phi_{C_{ABD}}})))}{strd(T^A_{\phi_{C_{ABD}}})} \right\rfloor \Rightarrow$$

$$val(\phi_{C_{ABD}}, A, 5) = \left\lfloor \frac{(5 \mod (2 * 6))}{6} \right\rfloor \Rightarrow$$

$$val(\phi_{C_{ABD}}, A, 5) = \left\lfloor \frac{5}{6} \right\rfloor = 0$$

For B:

$$val(\phi_{C_{ABD}}, B, 5) = \left\lfloor \frac{(5 \mod (dom(B) * strd(T^B_{\phi_{C_{ABD}}})))}{strd(T^B_{\phi_{C_{ABD}}})} \right\rfloor \Rightarrow$$

$$val(\phi_{C_{ABD}}, B, 5) = \left\lfloor \frac{(5 \mod (2 * 3))}{3} \right\rfloor \Rightarrow$$

$$val(\phi_{C_{ABD}}, B, 5) = \left\lfloor \frac{5}{3} \right\rfloor = 1$$

and, finally for D:

$$val(\phi_{C_{ABD}}, D, 5) = \left\lfloor \frac{(5 \mod (dom(D) * strd(T^{D}_{\phi_{C_{ABD}}})))}{strd(T^{D}_{\phi_{C_{ABD}}})} \right\rfloor \Rightarrow$$
$$val(\phi_{C_{ABD}}, D, 5) = \left\lfloor \frac{(5 \mod (3 * 1))}{1} \right\rfloor \Rightarrow$$
$$val(\phi_{C_{ABD}}, D, 5) = \left\lfloor \frac{2}{1} \right\rfloor = 2$$

Index-mappers in order to be able to compute the mapping of one index from the clique factor $\phi_i(X_{C_i})$ to an index of factor f_j, must also calculate the strides of each variable in f_j.The following equation is used by index-mappers to compute the mapped index, $m^{indx}_{f_j}$, in the factor f_j given an $index$ of a clique's factor (ϕ_i):

$$m^{index}_{f_j} = \sum_{x_l \in X_{f_j}} val(\phi_i(X_{C_i}), x_l, index) * strd(T^{x_l}_{f_j}) \tag{3.2}$$

To conclude our example of Fig 3.3, the index-mapper of factor $f_{AD}(AD)$ for the given index, 5, computes:

$$m^{5}_{f_{AD}} = val(\phi_{C_{ABD}}, A, 5) * strd(T^{A}_{f_{AD}}) + val(\phi_{C_{ABD}}, D, 5) * strd(T^{D}_{f_{AD}}) \Rightarrow$$
$$m^{5}_{f_{AD}} = 0 * 3 + 2 * 1 = 2$$

In the same manner, we compute 0 as mapped index for $f_A(A)$ and 1 for factor $f_{AB}(AB)$. At the beginning of index-mapper's initialization, for each variable x_l present in the destination factor f_j we pre-compute the stride of x_l both in the clique's factor ϕ_i and in f_j. Then, when we want to compute an index in f_j, we use these quantities and Eq. 3.2 to compute the required index. Of course, we use our clique representation only when the required space is less than the naive approach. The space complexity of an index mapper for a clique C_i and a destination factor f_j is linear in the number of variables of f_j, as we keep three numbers for each variable $x_l \in X_{f_j}$. The computational complexity of computing one mapped index is $O(|X_{f_j}|)$ and the computational complexity of the initialization of an index-mapper is $O(|X_{C_i}|)$, since computing a variable's stride requires the product over the variables' domain size with greater index.

To conclude, let us discuss the advantages of our proposal. Fig. 3.4 shows the space requirements of the naive approach, namely generating all clique's potentials, compared to our approach using index-mappers. We use almost half the space of the naive approach for this small junction tree. In general, in order to represent a junction tree (omitting the messages), we need space to store all the factors from the original

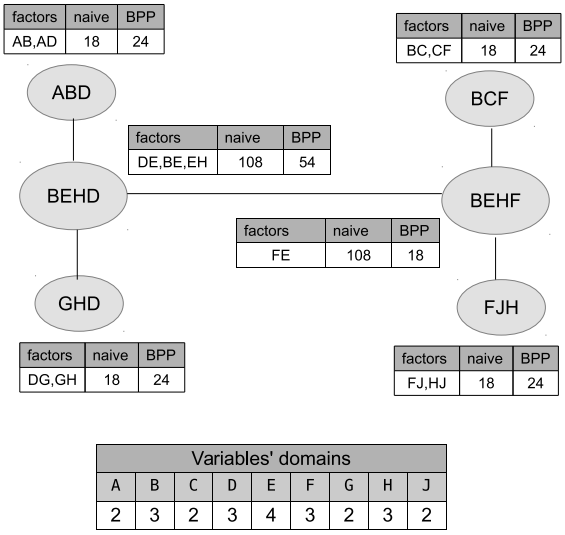


Figure 3.4: The junction tree with the space requirements for both approaches: naive and BePadoop(BPP). Each table is divided into three parts: the leftmost contains the factors that have been assigned to this clique, in the middle there is the naive approach and the rightmost section is the space requirements of the representation proposed by BePadoop.

graphical model plus one index mapper for each factor, thus the overall space requirements for representing a junction tree are linear in the factors of the original graphical model and the total number of factors variables, since each index-mapper keeps three integers for each variable in a factor. Let F be all the factors in the original graphical model, $|f_j|$ the size of factor f_j and $|X_{f_j}|$ the number of variables in factor f_j, then the total space requirements of our represnetation is O($\sum_{fj \in F} |fj| + \sum_{f_j \in F} |X_{f_j}|$). On the other hand, we increase the computational complexity, but this does not affect the dominant factor in the computational complexity of our problem. Normally, in order to compute an outgoing message for a clique v_i we do $deg(v_i)$ multiplications for each value in $\phi_i(X_{C_i})$. The total number of values in a clique C_i is $\prod_{x_l \in X_{C_i}} dom(x_l)$, so the overall complexity is $O(|deg(v_i)| * \prod_{x_l \in X_{C_i}} dom(x_l))$. By using our approach, we must calculate on-the-fly each value, which we showed above that for each factor is done in $O(|X_{f_j}|)$, as a result we have that to compute a single value of $\phi_i(X_{C_i})$ we need $O(\sum_{f_j \in F_i} |X_{f_j}|$ operations. However, this may increase the overall complexity but the main source of the high computational cost remains the $\prod_{x_l \in X_{C_i}} dom(x_l)$ factor for large cliques.

3.4 Lazy Computed Messages

Messages convey information about the joint distribution of common variables between two cliques,and, as a consequence, messages have also exponential space requirements. Large cliques tend to have large messages, thus even though our alternative clique representation enable us to represent large cliques, message size remains a significant problem. Our alternative representation gives us the opportunity to use a similar idea during message computation in order to save space. In general, a message is smaller in terms of size than a clique, but by using our clique representation this may no longer be the case. Instead of computing and generating the complex message distribution, we could transfer all the information (clique's potential and messages) that generate the message and again calculate the required values on-the-fly. Obviously, this representation cannot be used with all messages, thus we use it only when the message size is larger than the information that is involved in the computations. We define a message $m_{i\to j}$ from a vertex v_i to a vertex v_j as a lazy computed message iff it satisfies the following condition:

$$\sum_{k\in N(i)\setminus j} |m_{k\to i}(S_{k,i})|_{real} + |\phi_i(X_{C_i})|_{real} < |m_{i\to j}(S_{i,j})| \tag{3.3}$$

where N(i) are the neighbors of the vertex v_i. We should point out that on the right side of the inequality we use the theoretical size of the message, which is equal to $\prod_{x_l\in S_{i,j}} dom(x_l)$, where $S_{i,j}$ are the variables present in message $m_{i\to j}$. On the other hand, on the left side of the inequality we use the actual representation size i.e. the actual size needed to represent clique potential and messages. Note that some incoming messages may be lazily computed, and cliques can be represented through the model factors.

In order to understand how we compute a message value on-the-fly, let us give some useful insights on point-wise multiplication. Fig 3.5 depicts a point-wise multiplication of three tables A,AB,AD. One value index from ABD matches to exactly one (index) of the smaller tables (striped cells). On the other hand, one cell of the smaller table maps to several values of the bigger table (light gray cells). Thus, we introduce a new component, called index mapper iterator, that maps message indices to the set of indices in the clique potential to enable the efficient to enable the efficient computation of clique output messages.

The best way to demonstrate the internals of an index-mapper iterator is by a simple example. In Fig 3.5, let ABD be the clique and the smaller tables to be its messages. Additionally, AD is a lazily computed message and we want to compute the value at index 3 ($\{A_1, D_0\}$). The values that $\{A_1, D_0\}$ maps are $\{A_1, B_0, D_0\}$, $\{A_1, B_1, D_0\}$,

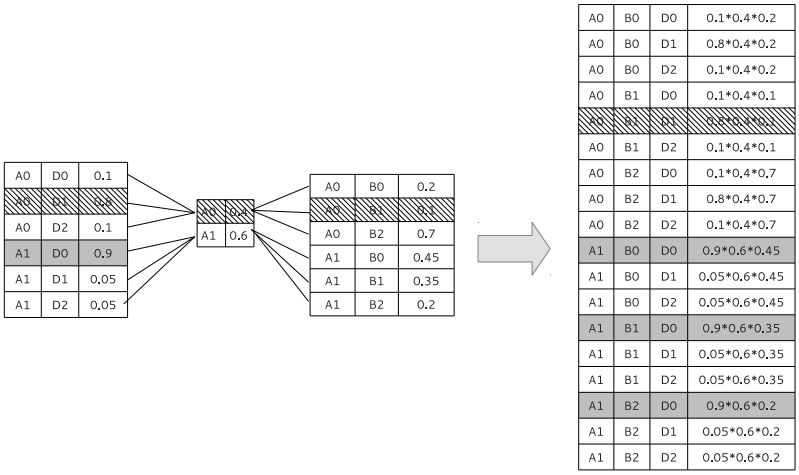


Figure 3.5: Point-wise multiplication between three factors A,AB,AD , which result in the ABD).

$\{A_1, B_2, D_0\}$. We observe that the common variables (A,D) are fixed to the values dictated by the message index, while the remaining "free" variables (B) take all the values of their domain. We iterate over the set of matching indices by finding the first matching index and then systematically computing the next in order index. Using Eq. 3.1, we can compute the index of each variable for the given index, i.e., A_1, D_0. Then, in order to compute the first matching index we multiply each variable value with its respective stride in the clique potential and sum those values. In our example, the stride of A is 9, D's 1 and B's 3, so the first matching index is equal to $1*9+0*1 = 9$, which is the index of $\{A_1, B_0, D_0\}$. The next value in order is $\{A_1, B_1, D_0\}$. The value can be computed by adding to the first matching index the result of the multiplication of B's value and its stride ($9+1*3 = 12$). Fig 3.6 shows how the index mapper iterator works for our example.

More formally, consider a clique C_i with variables X_{C_i}, a lazily computed message $m_{i\to j}$ of C_i with variables $S_{i,j} \subset X_{C_i}$, an index $n_{m_{i\to j}}$ of $m_{i\to j}$ and the index mapper iterator that generates the set of indices $N^{C_i}_{m_{i\to j}}$ of C_i given the $n_{m_{i\to j}}$. Let $U = X_{C_i} \backslash S_{i,j}$ denote the set of "free" variables. We find the first matching index by decoding the common variables and summing the result of the multiplication of the decoded variable values for the $n_{m_{i\to j}}$ with their respective strides in ϕ_{C_i}. Using a vector V_u, which tracks the current values of the variables in U, we iterate over all the possible U's values and for each such value, we carry out the same process as for the first matching index and the result is added to the first matching index. In this moment we generate all the $\prod_{u\in U} dom(u)$ matching indices and their corresponding contributions to $m_{i\to j}$. Since we use the strides of the clique's variables and the vector V_u which, in worst case, is in the same order as the number of clique variables, the space complexity is O($|X_{C_i}|$).

Finally, the computational complexity for generating one index is linear in the number of variables in U but, of course, the overall complexity for generating the $N_{m_{i\to j}}^{C_i}$ is O($\prod_{u\in U} dom(u)$).

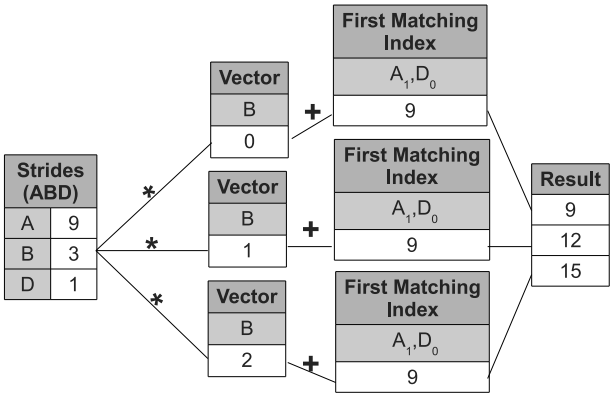


Figure 3.6: Calculating the set of indexes that map to A_1, D_0. The vector with the set of "free" variables in the clique takes all the possible values, which are decoded to values multiplied by their stride and then added to the first matching index to compute one index in the result set.

3.5 Belief Propagation in Hadoop

BePadoop is an efficient algorithm for exact belief propagation on Hadoop, which uses two MapReduce jobs in parallel. By Theorem 1, vertices with equal rank can be processed independently of each other. As a result, we can structurally parallelize BP by splitting each set of vertices belonging to the same rank into independent subsets (structural parallelization). Moreover, we can parallelize the computations for producing the outgoing messages for each clique by dividing the clique potential into smaller parts and calculating parts of the outgoing message in parallel (computation parallelization). Then, by merging the partial results, we can compute the final outgoing message. Both types of parallelization have benefits for certain scenarios, thus BePadoop uses a hybrid solution, where both techniques are exploited.

Structural parallelization is implemented by the Message Computation (MC) job and message computation parallelization is implemented by the Parallel Message Computation (PMComp) job. Note that PMComp is employed only on outlier cliques. In Fig. 3.7, a high level overview of BePadoop is shown: We start by processing all the vertices with rank 1 (leaf vertices). During each iteration we use MC to compute all BP messages of that rank. At the same time, all outlier cliques (which were detected during the initialization), are processed by PMComp. Finally, after we have processed

all the ranks, we construct the calibrated junction tree and we are ready to answer probabilistic queries. In the rest of this chapter, we discuss the initialization of BePadoop followed by the presentation of the two MapReduce jobs .

3.6 BePadoop Initialization

Initialization is an integral part of BePadoop, since it is responsible for the pre-processing of the junction tree. All the steps shown in Alg. 2 are implemented using MapReduce jobs. We proceed by extracting the graph of the junction tree, and then ranking the junction tree both forward and backward lines (1-5). Then, we split the junction tree by rank and we partition each rank into K partitions. Each vertex v_i is assigned to partition k_l according to the following formula: $k_l = \textit{HashKey}(v_i.rank + v_i.id) \bmod K$. The Hashkey is produced using the md5 cpyptographic hash function. Partitioning is important in order to better utilize cluster resources. Each partition is stored in a separate file. Moreover, as we compute the partitions we keep some statistics (average partition workload, total rank workload and average vertex workload), that will be used to find outlier cliques. The last step before the main phase of BePadoop intialiazation is one MapReduce job which detects outlier cliques and separates them from the rest cliques of the partition to be processed differently. The FindOutlierCliques job (line 7) reads each partition and uses the statistics produced by the previous job to decide whether a clique can be characterized as outlier. An important sidenote is that there are cases when the partitions are not balanced in terms of computational cost. To deal with such scenarios, we extend the notion of outlier cliques: If adding a clique C_i to a partition k_l results in exceeding the average partition workload, then C_i is characterized as outlier. As a result, in the end of the initialization we have split the junction tree by rank and each rank is partitioned and stored in separate files. Moreover, these partitions are further divided into two parts, one that will be processed by the MC job and another that will provide the input of the PMComp job. We are now ready to get into the two main BePadoop MapReduce Jobs.

3.6.1 Message Computation Job (MC)

Message Computation exploits Theorem 1 and processes independently vertices of the same rank. At iteration i, MC reads all the vertices with rank equal to i and the respective incoming messages. During the map phase, we compute the outgoing messages of each vertex and output them to reducers with a composite key. The key is composed of a MD5 key, which is generated in the same way as for the vertices, together with the rank and the id of the destination vertex. We use the MD5 key to store the message

Algorithm 2: BePadoop Initialization

Input: Graphical model g
Output: Fully Ranked Graphical Model Splitted by rank

1 $g_l \leftarrow$ `ExtractGraph` (g);
2 $g_l \leftarrow$ `ForwardRank` (g_l);
3 $roots[] \leftarrow$ `FindRoots` (g_l);
4 $g_l \leftarrow$ `BackwardRank` $(g_l, roots)$;
5 $g_{up} \leftarrow$ `UpdateGraphModel` (g_l, g);
6 $splits[] \leftarrow$ `SplitIntoPartsByRank` (g_{up});
7 $mcPartitions, outlierPartitions \leftarrow$ `FindOutlierCliques` (g_{up});
8 return splits ;

for the appropriate partition. Furthermore, we use rank and vertex id to totally order messages, since we need the messages to be sorted by vertex id when stored to exploit a similar idea as in the sort-merge goin algorithm. We provide more details on this in the next paragraph.

Each reducer takes as input lists of messages, destined for the same partition, ordered by both the rank and id of the destination vertex. Reducers pack messages with the same destination id and rank into an array and append those arrays into the respective message file. We need messages to be sorted, because we want to exploit the fact that partition files are already sorted by vertex id (a byproduct of the initialization procedure). Thus, we can exploit a similar idea as in the sort-merge join algorithm, where both relations are sorted by the join attribute prior the join phase, so that interleaved linear scans will encounter joinable sets of tuples at the same time. That is also the case in our problem, since partition files are sorted by vertex id and we ensure that message files are also sorted by the destination vertex id. We can exploit the same idea in order to read both the partition vertex file and the respective message files only once during a given iteration.

In Alg. 3 the pseudo-code for the map function of each mapper task is shown. First, the mapper reads the messages for the vertex v_i (line 1). Next it initializes the outgoing messages. Since we must iterate through all the possible values of the clique potential, we need an efficient way to determine for each value (index) of the clique potential the respective value (index) of each incoming and outgoing message. This is achieved through the aforementioned index-mapper components, which are initialized for both incoming and outgoing message in lines (2-4). In line 5, we find all the lazy computed outgoing messages. Then, the calculation of the outgoing messages is initiated. We iterate over all possible values of the clique potential and multiply each clique value with the respective values of all incoming messages. After we have calculated the pointwise product for a certain index of the clique distribution, we add the result (line 12)

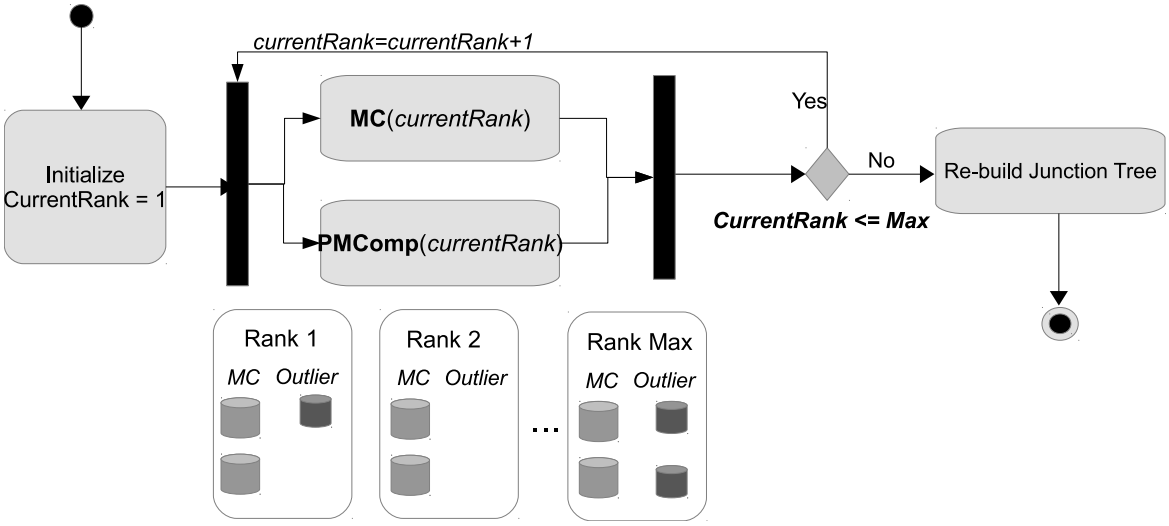


Figure 3.7: High level overview of BePadoop BP. At each iteration we use two jobs in parallel (MC,PMComp) to compute outgoing messages until we have processed vertices with maximum ranks.

to all outgoing messages. Notice that during the backward rank, to produce the correct message values, which are in accordance with Eq. 2.4, we must divide the value that we add to each outgoing message with the appropriate value of the incoming message, which has been produced during the forward pass along the same edge. Consider the junction tree in Fig. 3.1a during the forward passC_{BDEH} received two messages one from C_{ABD} and another from C_{GHD}. During the backward pass, when we calculate the messages$m_{BDEH \to ABD}$ and $m_{BDEH \to GHD}$, we must divide with the appropriate values of the two incoming messages produced during the forward rank namely,$m_{ABD \to BDEH}$ and $m_{GHD \to BDEH}$. Before we output the messages (lines 18, 19), we generate the two appropriate composite keys shown in lines 14 - 17.

Alg. 4 shows the reducer of the MC job. Each reducer uses an array to store messages with the same destination id and a variable to keep track of the current output rank. As long as the messages have the same destination rank and vertex id, we add them to the aforementioned array (line 29). If we encounter a message with different destination vertex id, we flush the array to a file (lines 23 - 27). Lines 12 - 19 denote that if we encounter a message with different destination rank, we flush the remaining messages if they have not already been flushed and we create a new file for that rank.

3.6.2 Parallel Message Computation (PMComp)

MC utilizes cluster resources efficiently if all cliques have almost equal size but, when outlier cliques exist, MC will be significantly slowed down as the time to process those

Algorithm 3: Map function of the Mapper of MC Job. This function is executed until the whole Mapper's input is consumed

Input: rank r,partitionNumber p ,vertex v_i
Output: Outgoing messages

```
1  vertex, incomingMsgs ←ReadMessages (r, p, v_i.id) ;
2  outgoingMsgs ← InitializeOutgoingMessages () ;
3  inMsgmaps ← InitializeIndexMappers (clique, incomingMsgs) ;
4  outMsgmaps ← InitializeIndexMappers (clique, outgoingMsgs) ;
5  lazymsg ← FindLazyComputedMessages (outgoingMsgs,vertex) ;
6  for i = 0...Clique.factor.size do
7      tmpvalue ← 1 ;
8      for msgindex = 0...incomingsMsgs.length do
9          tmpvalue *= incomingMsgs[msgindex] .
           getValueByIndex(inMsgmaps[msgindex][i]) ;
10     for msgindex = 0...outgoingMsgs.length do
11         updateValue ← tmpvalue / valueofcounterMessage ;
12         outgoingMsgs[msgindex] .
           addValueToIndex(updateValue,outMsgmaps[msgindex][i]) ;
13 for msg ∈ outgoingMsgs ∪ lazymsg do
14     MD5Key1 ←GetMDKey
       (msg.destinationForwardRank,msg.destinationId) ;
15     MD5Key2 ←GetMDKey (msg.srcBackwardRank-1,msg.destinationId) ;
16     CompositeKey1 ← InitCompositeKey (
       MD5Key1,msg.destinationForwardRank ,msg.destinationId );
17     CompositeKey2 ← InitCompositeKey (
       MD5Key2,msg.srcBackwardRank-1, msg.destinationId );
18     output (CompositeKey1,msg) ;
19     output (CompositeKey2 ,msg) ;
```

Algorithm 4: Reducer of MC Job

Input: compositeKey k, list of messages L
Output: Sorted By destination vertex id message files

```
outputMessages[] ← empty ;
outputFile ← null ;
currentDestinationVertex ← −1 ;
for msg ∈ L do
    if outputRank ≠ key.rank then
        {
        outputRank ← key.rank ;
        if outputMessages ≠ empty then
            outputFile.flush(otuputMessages) ;
        outputFile ←initializeOutputFile (outputRank) ;
    if currentDestinationVertex ≠ msg.destinationId then
        {
        outputFile.flush(otuputMessages) ;
        outputMessages[] ← empty ;
    ouptutMessages.add(msg);
```

cliques will dominate the processing time for other mappers. Thus, the processing of the whole rank will be slowed down. PMComp parallelizes the computations for outlier vertices. More specifically,at iteration i, PMComp reads outlier cliques with rank i and splits them into smaller pieces. PMComp mappers are almost identical with the MC mappers. Apart from the fact that PMComp produce partial messages, the only other difference is that they iterate over the split indices and not over the whole clique potential. Moreover, PMComp reducers, instead of just appending messages to the output array containing messages with the same destination vertex, they also merge partial messages in order to create the final message. Note that we cannot simply replace MC with PMComp -the overhead of splitting vertices and merging the partial message makes such an approach impractical.

3.6.3 Complexity

Let us discuss the computational complexity of BePadoop main processing steps. Exact inference is an NP-hard problem, thus, we cannot avoid exponential time complexity. Here, we discuss the complexity of one iteration (rank) for each job. MC's complexity over a set of cliques C_{rank} and messages E_{rank} is $O(E_{rank}\log(E_{rank}) + \sum_{C_i \in C_{rank}}(|\phi_{C_i}|))$, which is the complexity of sorting the keys of messages that are

transmitted from mappers to reducers for the given input rank added to the sum of all the clique sizes at that rank. Since, in most cases, $\sum_{C_i \in C_{rank}}(|C_i|) >> E_{rank}(\log E_{rank})$, the computational complexity of MC could be simplified to $O(\sum_{C_i \in C_{rank}}(|\phi_{C_i}|))$.

PComp computes partial messages, which are passed from mappers to reducers and finally merged to the reducers, so the complexity is O($\sum_{C_i \in C_{outlier}}(|\phi_{C_i}|)) + M * E_{outlier} log(E_{outlier}) + M * E_{outlier}$), where M is the number of machines we split each clique.As a consequence, following the same logic with MC PComp's complexity can be simplified to O($\frac{\sum_{v_i \in V_{outlier}}(|v_i|))}{M}$).

In terms of space space complexity note that the space requirements for a junction tree using our clique representation (without the messages) is linear in the factors of the original graphical model. Unfortunately, not all messages can take advantage of the lazy computation technique, thus, in the worst case scenario, we have exponential space requirements. In general, the information produced by BePadoop has exponential complexity, due to the messages' space requirements.

Finally, we discuss the communication cost of BePadoop in terms of messages (Msg) and cliques (C) that we read and write from mappers and reducers. MC's mappers output each message only once but read it twice. On the other hand, MC reducers output the produced messages twice. Thus, since vertices are used twice (in the forward and backward pass), the total communication cost for all iterations is $2C + 5Msg =$ O($C + Msg$).

The communication cost for PComp is very similar to MC. We read the outlier cliques twice. We create splits equal to the number of the available map tasks and each map task creates a partial message that has the same space requirements as the final message. As a result, the total communication cost is the number of mappers multiplied by the number of the outgoing messages $Msg_{outlier}$ of outlier cliques added to the size of outlier cliques. The number of the outlier cliques is expected to be almost 3% of the total number of vertices. Additionally, we write each message twice, so in worst case this would be $3\% * C + Msg_{outlier} * mappers + 2 * Msg_{outlier} =$ O($C + Msg_{outlier} * mappers$).

Finally, we consider the communication cost of the initialization of our approach. During initialization, we only pass from the mappers to the reducers the whole junction tree three times during the last three MapReduce jobs. Additionally, during the ranking we use only the graph of the junction tree, which most of the times is orders of magnitude smaller than the junction tree with the probabilistic notation. One sidenote is that during the initialization there are no messages, so we read and write only the cliques.

It must be stressed that our main algorithm communication cost is independent of the number of iterations, since during each iteration only part of the junction tree is processed. Tables 3.1,3.2 summarize the time and communication complexities of BePadoop's jobs.

Job	Computational
MC	O($E_{rank}\log(E_{rank}) + \sum_{C_i \in C_rank}(\|\phi_{C_i}\|)$)
PComp	$(\sum_{C_i \in C_{outlier}}(\|\phi_{C_i}\|)) + M(E_{outlier} log(E_{outlier}) + E_{outlier})$
Init	O($ClogC$)
BePadoop	O($\sum_{C_i \in C}(\|\phi_{C_i}\|) + ElogE$)

Table 3.1: The computational complexity for the two main jobs of Bepadoop algorithm, BePadoop's initialization and Bepadoop.

Job	Communication
MC	O($C_{rank} + Msg_{rank}$)
PComp	O($C_{outlier} + E_{outlier} * mappers$)
Init	O(C)
BePadoop	O($C + Msg$)

Table 3.2: The communication complexity for the two main jobs of Bepadoop algorithm, BePadoop's initialization and Bepadoop.

Chapter 4

Related Work

As the MapReduce framework successfully brought parallel programming to a wider audience, a large number of algorithms have been ported (including, e.g., search query, information retrieval, approximate inference and several graph algorithms. On the other hand, new frameworks have been proposed for developing parallel graph algorithms, given the nontrivial difficulties in the efficient implementation of graph algorithms in the MapReduce framework. Nevertheless, in the recent years, some design patterns for developing graph algorithms have emerged. In this chapter, we start by presenting the related work on inference in MapReduce. Then, we discuss some design patterns for graph algorithms and how they have been exploited in BePadoop. Finally, we conclude by introducing the proposed frameworks for developing graph algorithms and we give a brief explanation why we decided to implement BePadoop over Hadoop.

4.1 MapReduce inference Algorithms

In [13] the authors introduce Generalized Iterative Matrix-Vector multiplication (GIM-V), a generalization of normal matrix-vector multiplication $M \times v = \acute{v}$, where M is a n by n matrix,v a vector of size n and $\acute{v} = \sum_{j=1}^{n} m_{i,j}v_j$. GIM-V generalizes a matrix-vector multiplication by introducing three methods:

1. combine2: multiply $m_{i,j}$ and v_j.

2. combineAll: summarize the n multiplication results for node i.

3. assign: overwrite previous value v_i with new result to make $\acute{v_i}$.

Ha-LFP[12] is a system that use GIM-V in order to implement an approximate belief propagation algorithm for pairwise Markov Random Fields. In contrast to BePadoop

that employs ranking to minimize the communication cost between mappers and reducers, HadoopBP uses the whole graphical model as input to each iteration. This naturally introduces an extra communication bottleneck between mappers and reducers. As explained earlier, BePadoop uses only the active portion of junction tree at each iteration that minimizes the communication between mappers and reducers, which can be a crucial performance factor for large junction trees.

In [18], various parallel exact inference algorithms on multicore systems using MapReduce are presented. To our knowledge, these are the only other exact inference algorithms that use the MapReduce abstraction and are thoroughly described and presented[1]. The most efficient algorithm in [18] is a level-based MapReduce algorithm. The level l of each clique is the number of of edges on the path from it to the root of the junction tree. In each iteration, all the nodes at a certain level l are the input for the map phase, where the outgoing messages are computed and during the reduce phase the vertices of level $l-1$ ($l+1$) are updated during the forward (respectively backward) pass. Since the level-based algorithm is designed for multicore systems there are certain disadvantages when implemented in the Hadoop setting. First of all, by randomly choosing a root node the number of iterations for a junction tree is not minimized, which implies addtional communication cost. Moreover, by updating at each iteration the vertices that receive messages, we at least double the computational cost since each vertex is processed at least four times; two when it sends message and two when it receives messages. Additionally, BePadoop handles potential bottleneck (outlier) cliques more efficiently by parallelizing the computation of their messages.

4.2 Graphs and MapReduce

The MapReduce framework is widely accepted as a *de facto* tool for processing data qt large scale. As a result, there have been a wide number of design proposals for implementing algorithms on. Filtering is a method introduced in [15] which dictates a design for implementing graph algorithms in the MapReduce framework. The main idea behind filtering is to reduce the size of input for consecutive iterations until the problem size fits into the memory of a single machine. Assuming the memory per machine is super-linear in the number of vertices, the algorithms presented in the paper run in a constant number of iterations. Our approach utilizes the selective input into each iteration in a more sophisticated manner as it loads only the active vertices at each iteration; furtheremore, for junction trees with the same diameter the number of iterations in BePadoop remains the same independent of the number of the cliques. On the other hand, the iterations required by filtering depend on the relation of the total

[1] Another exact MapReduce inference algorithm has been presented in Hadoop Summit 2010 by Alexander Kozlov but there are no details about the implementation of the algorithm, except for the presentation slides.

available memory with the input size.

In [16], the authors give effective ways for optimizing the implementation of graph algorithms. Apart from the use of combiners and in-mapper combining, which suggests that mappers should emit aggregated results in order to reduce the number of key-value pairs shuffled across the network, there is Schimmy, a design pattern that separates the usual two data flows for a graph algorithm in MapReduce (one for the graph structure and the other for the messages exchanged). This is achieved by partitioning the input graph into n files, such that the files are sorted by vertex id (done by a simple MapReduce job). Then, a similar idea used in parallel sort-merge join algorithm could be used when two relations are sorted, so that interleaved linear scans will encounter joinable sets of tuples at the same time. The only data exchanged between mappers and reducers are the messages between vertices. BePadoop expands this idea by partitioning each rank, without the need to read the whole junction tree in each iteration.

4.3 Graph Frameworks

After the wide success of Mapreduce, a lot of work has been put to develop programmatic frameworks that provide better computation abstractions for developing graph algorithms for large datasets. Implementing graph algorithms has different porgrammatic requirements than those of batch data processing. As a result, Google has introduced the Pregel sysytem designed to address all the shortcomings of MapReduce regarding the development of graph algorithms. After Pregel, some open source alternatives have been published (GoldenOrb and Girraph), which provided an open source implementation of Pregel. Apart from Pregel-like systems, there have been frameworks with different philosophy, but with the same goal of providing an easy to use and simple programming model for distributed graph processing. All these systems are discussed in the rest of this section.

4.3.1 Pregel

Pregel [19] is a bulk synchronous message passing, fault tolerant programming framework designed by Google with a more vertex centric approach than the MapReduce's data-flow approach. Pregel is similar to MapReduce in respect that users use local operations that are independent of each other and the system is responsible to combine these independent actions in order to lift computation to large datasets. Despite the higher level similarities, Pregel is focused on addressing the issues that arise when developing graph algorithms, contrary to the MapReduce framework which is primarily inteded for batch data processing.

Pregel's design is influenced by Bulk Synchronous Parallel model, which defines a computation as an iterative procedure of supersteps. A superstep consists of three ordered stages:

1. Parallel Computation where independent computations done in parallel,
2. Communicationwhere data is exchanged between the processes and,
3. Barrier synchronization, where all processes wait until the communication is finished.

In a similar way, Pregel divides the computation into a series of supersteps. In Pregel there are two types of nodes; the master node which is the coordinator of the system, and the worker nodes, that operate on the input data. The master node is responsible for initiating each superstep and the workers for processing the vertices. The graph is divided into partitions, which are distributed across the cluster. Each vertex can be active or inactive and implements a compute method. The computation in Pregel is done into series of supersteps. In each superstep, the compute method of each vertex is executed. Moreover, each vertex can send messages to arbitrary vertices of the graph, as well as alter the structure of the graph. All the messages are given as input to the vertex at the start of each superstep. Pregel's model has one major disadvantage when used on our problem because the time requirements for processing vertices is not always uniformly distributed. As a consequence, there can be a single vertex, which dominates over the rest of the dataset in terms of time execution. This in Pregel cannot be addressed with a straightforward manner, however, BePadoop splits the workload for those outlier vertices to all the cluster nodes by using Parallel Message Computation.

4.3.2 GraphLab,PowerGraph

GraphLab is an asynchronous distributed shared-memory framework, in which vertices can access information on adjacent vertices and the respective edges. PowerGraph is a generalization which combines the advantages of the models of both Pregel and GraphLab, by dividing the computation of each vertex into three phases. The first phase is gather, where information about adjacent vertices and edges is collected through a generalized sum. The resulting sum is used in the apply phase to update the vertex and finally the scatter phase that updates the data on adjacent edges. In spite of the fact that describing BePadoop in both abstractions would be simpler we could not effectively handle the outlier cliques, which is an important requirement of our problem. In general, these frameworks work very well when the vertex compute function is relatively small or the computation can be factorized, however, when dealing with exact inference the exponential computational cost causesg these assumptions to be violated.

Chapter 5

Experimental Evaluation

To verify the effectiveness of BePadoop, we devised a wide range of experiments on synthetic and real datasets. We compare BePadoop with a naive implementation of our algorithm, where ranking is not used, so we can not take advantage of the parallel computation. The naive implementation takes at each iteration i as input the whole junction tree and all the messages produced up to i. The cliques that are ready output their outgoing messages. The naive algorithm terminates, when all vertices have sent all their messages. Apart from the naive approach, we also compare BePadop with the aforementioned algorithm described in [18][1]. To our knowledge, there is not another exact inference algorithm for MapReduce paradigm. Since the algorithm is designed for multicore systems, we ported the level based algorithm to the Apache Hadoop MapReduce. In order to compare the three approaches we use the time required for the calibration of the junction tree (time for the forward and backward pass). All the experiments were conducted on a cluster with 18 machines, equipped with the Intel Xenon X3323 processor, 4 GB RAM and high speed ethernet connection (1GB/s). We used the 1.0.3 version of Apache Hadoop and the 1.6.0_16 version of Oracle Java.

5.1 Sensitivity Analysis

The first set of experiments that we conducted was to evaluate the sensitivity of BePadoop on different junction tree parameters:

1. domain size of each variable $dom(x_i)$
2. number of variables in a clique (treewidth) w
3. number of cliques in a junction tree N

[1] From here on we refer to this algorithm in short as Phoenix, which is the name of the framework that was developed on.

4. the degree of each clique $deg(v_i)$

5. diameter of the junction tree d

We generated synthetic datasets with various values for the aforementioned parameters, and, unless stated otherwise, we used the following default parameter values: $dom(x_i) = 4$, $d = 8$, $w = 10$, $deg(v_i) = 100$ and $N \simeq 100000$. The variables of each message are randomly selected from the powerset of the clique variables. Finally, we conducted our analysis using all four possible configurations of BePadoop, i.e. using lazy messages/parallel computation or not.

Sensitivity Analysis Results

Let us begin with the domain size of the variables, Fig. 5.1 (a) depicts BePadoop's running time when we increase the domain size of each variable. BePadoop exhibits exponential increase in time as the domain size increases, which is a direct consequence of the computational complexity for computing a message. Consider a clique C_i with degree $deg(C_i)$ and w variables with domain equal to k, then, the computational complexity for producing one message equals to $deg(C_i) * k^w$. For small domain sizes, there is not significant difference in the running time between the four configurations of BePadoop. On the other hand, for larger domain sizes using lazily computed messages affects, negatively, the performance of BePadoop. As we discussed in 3.4, there is an increase on the computational complexity when a clique uses lazily computed messages to produce outgoing messages. Note that using the parallel message computation job results in a 14% speedup when using lazy messages.There is a similar exponential increase in the running time when the number of variables in a clique is increased Fig. 5.1 (b) denotes the BePadoop's running time for 6,8,10 variables in every clique of the junction tree. In this case, there are no indicative differences between the four configurations.

Fig. 5.1 (c) depicts the running time for junction trees with different diameters. Since, the number of iterations of our algorithm is directly associated with the diameter of the junction tree as the diameter of the junction tree increases so is the running time. Notice that using lazily computed messages results in approximately 12% speed up.

Next, we examine the degree of each vertex, Fig. 5.1 (a), since we increase the number of iterations that we must do over the clique potential to produce a message $(deg(C_i) * k^w)$ there is a linear increase in running time. The results of these experiments provide us with some informative conclusions. First, greater degree in a vertex implicates a higher probability to have large messages, thus, there would be more lazily computed messages. As a consequence, there is a speed up in the overall process when having a moderate number of lazy messages. Additionally, parallel message computation improves the overall performance and in certain cases there is a significant reduce in the running time (notice the experiments for degree 1500 and lazy off). Finally, we

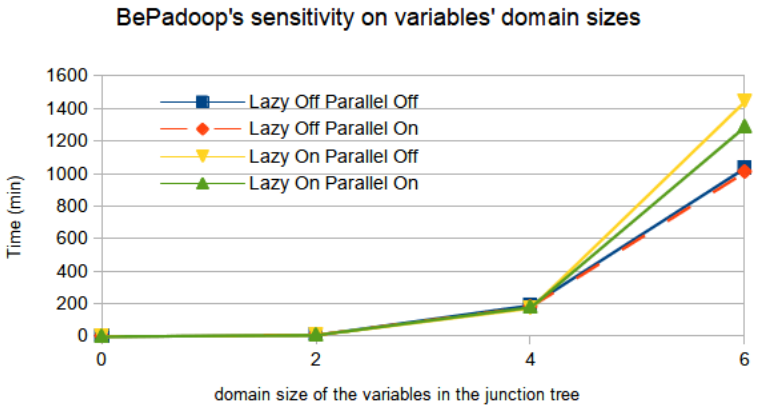
discussthe effect of the total number of nodes. Increasing the total number of nodes, results in a growth in the number of nodes processed by each rank, as a consequence, the total running time will increase. Fig 5.1 (b) shows BePadoop's running time with different total number of cliques, where we have an upsurge in the running time for 250000 cliques, where the volume of data shuffled between the map and reduce phases cannot be handled by the cluster we used to conduct the experiments. To conclude our sensitivity analysis, we can safely state that in most cases using parallel computation and lazily computed messages is beneficial to BePadoop and in some cases result in significant improvement on the performance.

5.2 Comparison to other approaches and Scale up

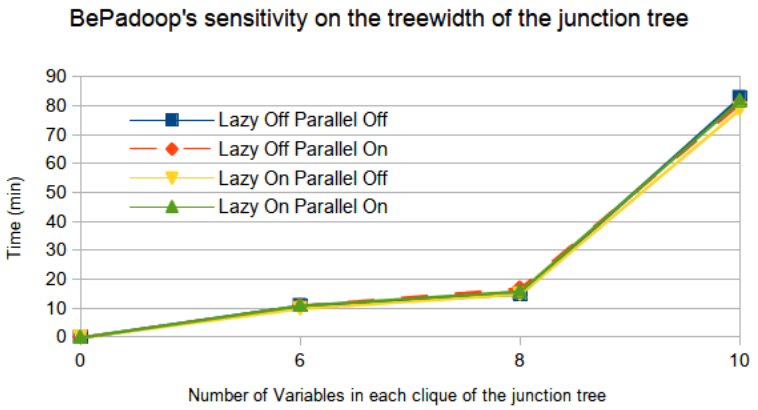
We also compared BePadoop with two other approaches, a naive version of BePadop and the Hadoop MapReduce version of the Phoenix algorithm. We used synthetic and real datasets to compare the three approaches. The following default configuration was used to generate the synthetic datasets 100000 nodes, diameter $d = 8$, degree $deg(v_i) = 100$, 10 variables in each clique and domain size for each variable v_i equal to 3. We must stress that we generated smaller datasets than our senisitivity analysis because for larger datasets both the naive and the Phoenix algorithm either could not be run due to memory limitations or the overhead of the data shuffling stage made it impractical. Fig 5.3b depicts the running time of the three algorithms on five datasets. The first two datasets are synthetic. The former dataset was generated with 6 variables per clique and the latter with variable domain size equal to 2.

Prior to the discussion of the results, let us first state some preliminaries which help to understand better the results. The Phoenix algorithm was designed for multicore systems with shared memory across CPUs. Additionally, all distributions are represented as full tables, thus, the representation of the junction tree is larger than BePadoop's representation,consequently, Phoenix shuffles more data between mappers and reducers. Moreover, a random junction tree vertex is chosen to be the root of the tree which results in a suboptimal number of iterations. On the other hand, the Naive approach uses BePadoop's clique representations, but reads the whole input at each iteration. As a consequence, there is higher communication cosst. Moreover, at each iteration checking if a clique is ready to send a message, results in redundant computations.

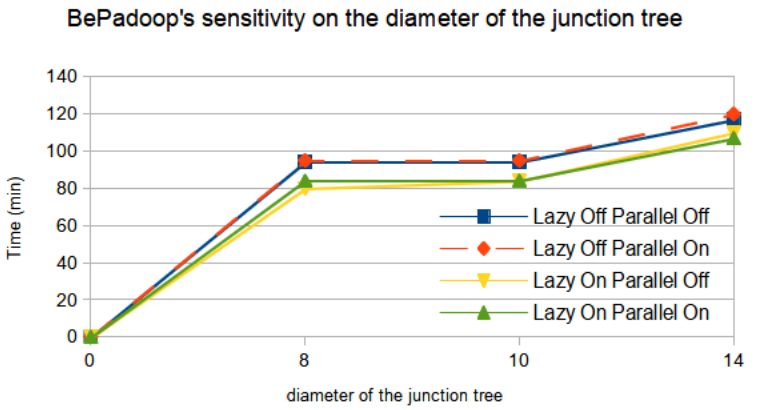
Furthermore, we compared the three approaches using three real datasets. The first real dataset is a snapshot of the Slashdot Zoo social graph with 78000 vertices and 400000 edges. We sampled a subgraph from the dataset and then we created pairwise Markov Random Field graphical from that subgraph. For each vertex in the graph we assigned a new random variable with domain size 4 and we also introduced for each edge in the graph a factor for the interaction of two random variables. Then we constructed the junction tree using the Huging algorithm. The Slashdot Zoo sampled



(a) Running time for various domain sizes for the variables

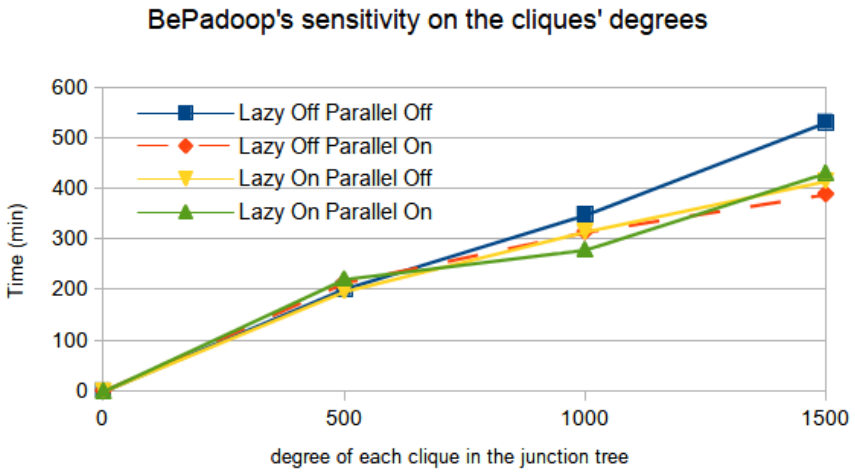


(b) Running time for different number of vertices in the junction tree

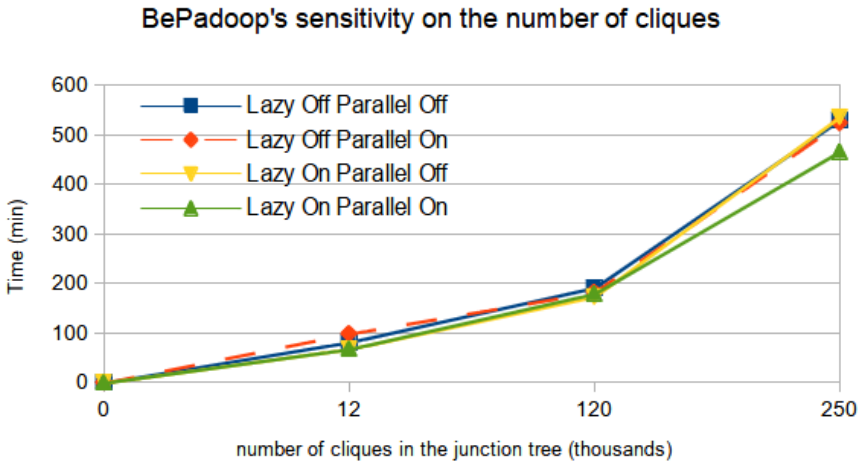


(c) Running time for different diameter of the junction tree

Figure 5.1: Sensitivity Analysis for BePadoop for domain sizes, number of variables in a clique and diameter



(a) Running time for different degrees of the vertices in the junction tree



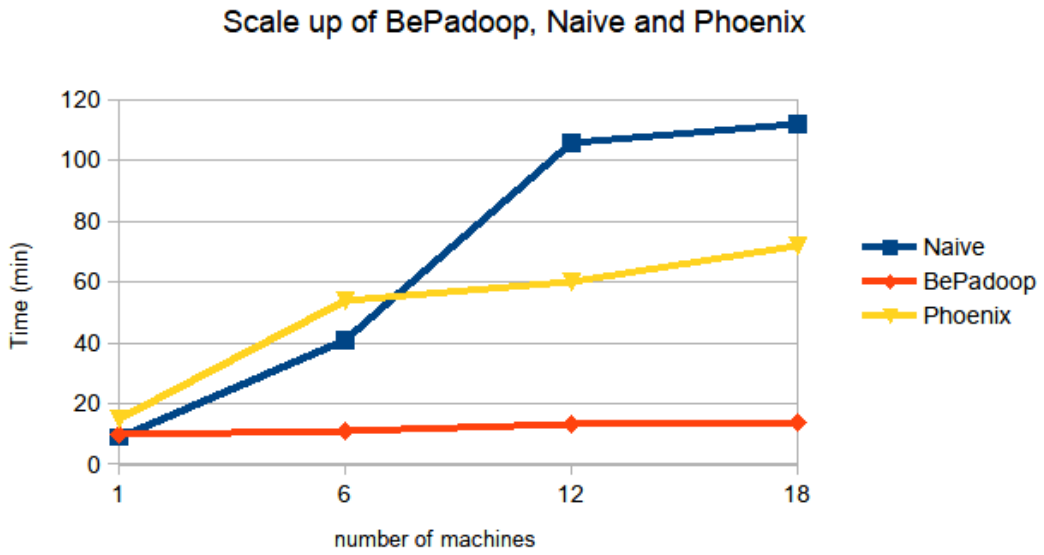
(b) Running time for different total number of nodes

Figure 5.2: Sensitivity Analysis for BePadoop for different degrees of the vertices and total number of nodes

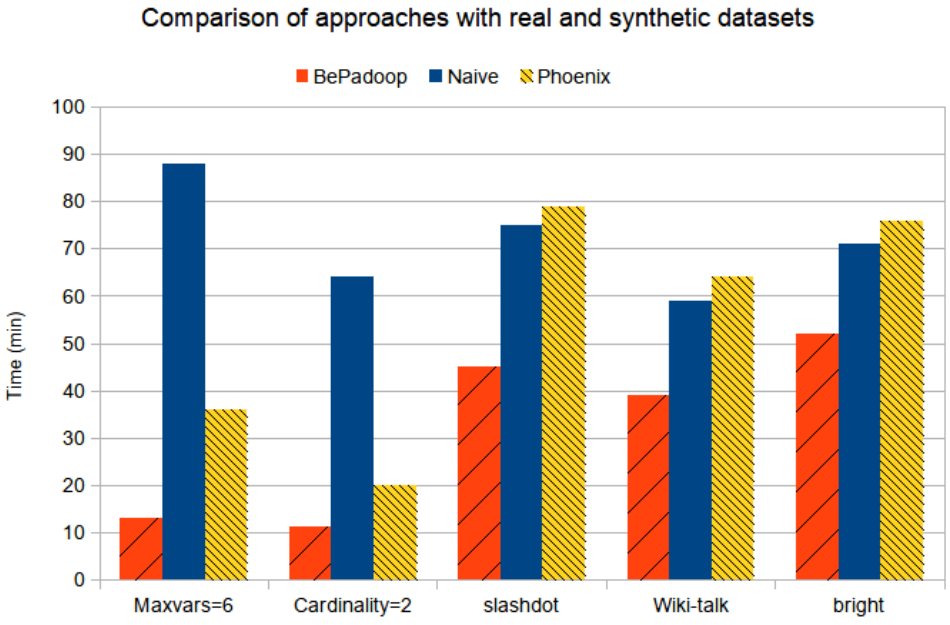
graph contained 34000 vertices and 60000 edges and the resulting junction tree with the maximum number of vertices has almost 6000 thousands cliques and diameter 44. We followed the same procedure for a graph depicting edits of users to the wikipedia talk pages with 2 million vertices and 5 million edges where the sampled subgraph contained 71300 vertices and 88812 edges and its junction tree 7000 cliques and diameter equal to 53. The last dataset was from Brightkite, a location-based social networking service provider, where users shared their locations by checking-in. The sample had 33000 vertices and 50000 edges and the induced junction tree contained 9000 cliques and had diameter of 62.

The last three datases in Fig. 5.3b denote the time required for the calibration of the junction trees by the three approaches. The results are similar with the results on the synthetic datasets, except for the fact that the naive performance has significantly improved. BePadoop still remains the most efficient of the three, on the other hand, the Phoenix algorithm is slightly the slowest between the three for two main reasons, first there is higher communication cost as all the distributions are represented as tables and second the number of iterations is larger than both naive and BePadoop. On the other hand, both the large diameter of the junction tree and the small number of cliques favor the Naive approach as at each iteration the overhead for reading all the cliques and its messages is less than the cost of deploying one MapReduce job, as a result, Naive is faster than the Phoenix. However, BePadoop is at least 25% faster than the Naive approach as it better utilizes the cluster resources with less communication and the parallel computation and almost always 40% more efficient than the Phoenix algorithm. In conclusion, the results of the comparison between the three approaches are in favor of BePadoop, as it better utilizes cluster resources and minimizes the communication cost between the mappers and the reducers as well as the data read in during each iteration. Moreover, BePadoop minimizes the number of iterations which is crucial for MapReduce algorithms.

Finally, we synthesized a set of datasets in order to evaluate the scalability of the three algorithms. Fig 5.3a shows the results of these experiments, where our algorithm has almost flat scale up as we increase the input size and increase respectively the number of available machines. The time required by BePadoop for the junction tree calibration is almost the same for all the four datasets. On the other hand, the naive version of BePadoop exhibits the same problems with large input as before and does not scale up well. Additionally, Phoenix algorithm exhibits an almost linear increase as the size of the input increases.



(a) BePadoop scale up diagram



(b) Comparison of BePadoop with the Naive approach and MapReduce algorithm for multicore systems

Chapter 6

Future Work and Conclusion

BePadoop gives rise to new interesting problems that must be addressed such as how to effectively compress a message distribution in order to surpass the memory limits of a cluster. Additionally, we research how to approximate the message distribution representations without having a large penalty to the overall inference solution. Thus, we could develop an approximate inference algorithm that takes advantage of BePadoop's optimal number of iterations and the performance of an approximate algorithm.

To conclude, in this thesis, we presented the basics for exact inference and we introduced BePadoop, a novel approach for exact inference on Hadoop MapReduce that scales up well and presented solutions for some crucial problems that impede the exact inference on large datasets i.e. clique representation, memory limitations, messages size, communication cost. We analysed the sensitivity of BePadoop for various parameters of the junction tree and we showed that BePadoop's optimizations not only do not introduce a computational overhead in the already difficult problem of exact inference but in most cases improve the running time of BePadoop. Finally, we compared BePadoop with two other algorithms and there were cases that BePadoop was an order of magnitude faster than the other two approaches.

Bibliography

[1] Apache mahout,http://mahout.apache.org/. 1

[2] A. Chechetka and C. Guestrin. Focused belief propagation for query-specific inference. In *In Artificial Intelligence and Statistics (AISTATS)*, May 2010. 9

[3] G. F. Cooper. The computational complexity of probabilistic inference using bayesian belief networks. *Artificial Intelligence 42, 393-405*, 1990. 1

[4] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51:107–113, Jan. 2008. 1, 10

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

[6] J. Gonzalez, Y. Low, and C. Guestrin. Residual splash for optimally parallelizing belief propagation. In *International Conference on Artificial Intelligence and Statistics*, pages 177–184, 2009. 1

[7] 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. 9

[8] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 17–30, 2012. 1

[9] J. E. Gonzalez, Y. Low, C. Guestrin, and D. O'Hallaron. Distributed parallel inference on large factor graphs. In *Proceedings of the Twenty-Fifth Conference on Uncertainty in Artificial Intelligence*, pages 203–212. AUAI Press, 2009. 1

[10] F. V. Jensen. *An introduction to Bayesian networks*, volume 210. UCL press London, 1996. 4

[11] M. I. Jordan. An introduction to probabilistic graphical models, 2002. 3

[12] U. Kang, D. H. Chau, and C. Faloutsos. Mining large graphs: Algorithms, inference, and discoveries. In *Data Engineering (ICDE), 2011 IEEE 27th International Conference on*, pages 243–254. IEEE, 2011. 1, 33

[13] U. Kang, C. E. Tsourakakis, and C. Faloutsos. Pegasus: A peta-scale graph mining system- implementation and observations, 2009. 33

[14] D. Koller and N. Friedman. *Probabilistic graphical models: principles and techniques*. The MIT Press, 2009. 3

[15] S. Lattanzi, M. B., S. Suri, and S. Vassilvitskii. Filtering: A method for solving graph problems in mapreduce, 2011. 34

[16] J. Lin and M. Schatz. Design patterns for efficient graph algorithms in mapreduce. In *Proceedings of the Eighth Workshop on Mining and Learning with Graphs*, MLG '10, pages 78–85, New York, NY, USA, 2010. ACM. 35

[17] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. Graphlab: A new framework for parallel machine learning. *arXiv preprint arXiv:1006.4990*, 2010. 1

[18] N. Ma, Y. Xia, and V. Prasanna. Parallel exact inference on multicore using mapreduce. In *Computer Architecture and High Performance Computing (SBAC-PAD), 2012 IEEE 24th International Symposium on*, pages 187 –194, oct. 2012. 1, 34, 37

[19] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing - "abstract". In *Proceedings of the 28th ACM symposium on Principles of distributed computing*, PODC '09, pages 6–6, New York, NY, USA, 2009. ACM. 1, 35

[20] R. J. 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, 1998. 1

[21] J. Pearl. *Probabilistic reasoning in intelligent systems: networks of plausible inference*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1988. 9

[22] D. Roth. On the hardness of approximate reasoning. *Artif. Intell. 82, 273-302*, 1996. 1

[23] Y. Xia and V. K. Prasanna. Node level primitives for parallel exact inference, 2007. 1

[24] J. S. Yedidia, W. T. Freeman, Y. Weiss, et al. Generalized belief propagation. In *NIPS*, volume 13, pages 689–695, 2000. 1