MapReduce Lifting for Belief Propagation*

Babak Ahmadi [1] and **Kristian Kersting** [1,2,3] and **Sriraam Natarajan** [3]
[1] Fraunhofer IAIS, Knowledge Discovery Department, Sankt Augustin, Germany
[2] University of Bonn, Institute of Geodesy and Geoinformation, Bonn, Germany
[3] Wake Forest University, School of Medicine, Winston-Salem, USA

Abstract

Judging by the increasing impact of machine learning on large-scale data analysis in the last decade, one can anticipate a substantial growth in diversity of the machine learning applications for "big data" over the next decade. This exciting new opportunity, however, also raises many challenges. One of them is scaling inference within and training of graphical models. Typical ways to address this scaling issue are inference by approximate message passing, stochastic gradients, and MapReduce, among others. Often, we encounter inference and training problems with symmetries and redundancies in the graph structure. It has been shown that inference and training can indeed benefit from exploiting symmetries, for example by lifting loopy belief propagation (LBP).That is, a model is compressed by grouping nodes together that send and receive identical messages so that a modified LBP running on the lifted graph yields the same marginals as LBP on the original one, but often in a fraction of time. By establishing a link between lifting and radix sort, we show that lifting is MapReduce-able and thus combine the two orthogonal approaches to scaling inference, namely exploiting symmetries and employing parallel computations.

Introduction

Machine learning thrives on large datasets and models, and one can anticipate substantial growth in the diversity and the scale of impact of machine learning applications over the coming decade. Such datasets and models originate for example from social networks and media, online books at Google, image collections at Flickr, and robots entering the real life. And as storage capacity, computational power, and communication bandwidth continue to expand, today's "large" is certainly tomorrow's "medium" and next week's "small". This exciting new opportunity, however, also raises many challenges. One of them is scaling inference within and training of graphical models. Statistical learning provides a rich toolbox for scaling with techniques such as inference by approximate message passing, stochastic gradients, and MapReduce, among others. Often, however, we face inference and training problems with symmetries and redundancies in the graph structure. A prominent example are relational models, see (De Raedt et al. 2008; Getoor and Taskar 2007) for overviews, that tackle a long standing goal of AI, namely unifying first-order logic — capturing regularities and symmetries — and probability — capturing uncertainty. They often encode large, complex models using few weighted rules only and, hence, symmetries and redundancies abound.

Employing symmetries in graphical models to speed up probabilistic inference, called lifted probabilistic inference, has recently received a lot of attention, see e.g. (Kersting 2012) for an overview. Message-passing approaches to inference can be very efficient and lifted approaches make these scale to even larger problem instances (e.g. (Singla and Domingos 2008; Kersting, Ahmadi, and Natarajan 2009)). To scale probabilistic inference and training, Gonzalez *et al.* (2009) present algorithms for parallel inference on large factor graphs using belief propagation in shared memory as well as the distributed memory setting of computer clusters. Although, Gonzalez *et al.* report on map-reducing lifted inference within MLNs, they actually assume the lifted network to be given. As of today, however, these two approaches have been completely distinct and orthogonal dimensions in the scaling of inference. None of the lifted inference approaches have been shown to be MapReduce-able. Moreover, many of them have exponential costs in the treewidth of the graph, making them infeasible for most real-world applications.

We present the first MapReduce lifted belief propagation approach. More precisely, we establish a link between color-passing, the specific way of lifting the graph, and radix sort, which is well-known to be MapReduce-able. Together with Gonzalez *et al.* (2009) MapReduce belief propagation approach, this overcomes the single-core limitation for lifted inference Our experimental results show that MapReduced lifting scales much better then single-core lifting.

We proceed as follows. After briefly touching upon graphical models and factor graphs in particular, we recap lifted belief propagation. We then show how to scale message-passing inference in two dimensions, namely by MapReduce in addition to the lifting. Finally, we experimentally evaluate the approach before we conclude.

*This work is part of an article that has been accepted for MLJ.

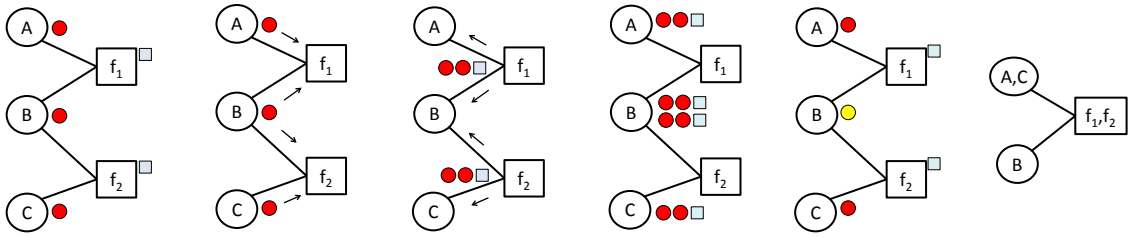

Figure 1: From left to right, steps of compressing a factor graph assuming no evidence. The shaded/colored small circles and squares denote the groups and signatures produced while lifting. the resulting compressed factor graph is shown on the right.

Lifted Belief Propagation

Let $\mathbf{X} = (X_1, X_2, \ldots, X_n)$ be a set of n discrete-valued random variables and let x_i represent the possible realizations of random variable X_i. Graphical models compactly represent a joint distribution over $\mathbf{X}$ as a product of factors (Pearl 1991), i.e., $P(\mathbf{X} = \mathbf{x}) = \frac{1}{Z} \prod_k f_k(\mathbf{x}_k)$. Here, each factor f_k is a non-negative function of a subset of the variables $\mathbf{x}_k$, and Z is a normalization constant.

Graphical models can be represented as factor graphs. A factor graph is a bipartite graph that expresses the factorization structure of the joint distribution. It has a variable node (denoted as a circle) for each variable X_i, a factor node (denoted as a square) for each f_k, with an edge connecting variable node i to factor node k if and only if X_i is an argument of f_k. Belief propagation makes local computations only. It makes use of the graphical structure such that the marginals can be computed much more efficiently. The computed marginal probability functions will be exact if the factor graph has no cycles, but the BP algorithm is still well-defined when the factor graph does have cycles. Although this loopy belief propagation has no guarantees of convergence or of giving the correct result, in practice it often does, and can be much more efficient than other methods (Murphy, Weiss, and Jordan 1999).

Although already quite efficient, many graphical models produce inference problems with a lot of additional regularities reflected in the graphical structure but not exploited by BP. Probabilistic graphical models such as MLNs are prominent examples. Lifted BP performs two steps: Given a factor graph G, it first computes a compressed factor graph $\mathfrak{G}$ and then runs a modified BP on $\mathfrak{G}$. We will now discuss each step in turn using fraktur letters such as $\mathfrak{G}$, $\mathfrak{X}$, and $\mathfrak{f}$ to denote compressed graphs, nodes, and factors.

Step 1 – Compressing the Factor Graph: Essentially, we simulate BP keeping track of which nodes and factors send the same messages, and group nodes and factors together correspondingly. Let G be a given factor graph with Boolean variable and factor nodes. Initially, all variable nodes fall into three groups (one or two of these may be empty), namely known true, known false, and unknown. For ease of explanation, we will represent the groups by colored/shaded circles, say, magenta/white, green/gray, and red/black. All factor nodes with the same associated potentials also fall into one group represented by colored/shaded squares. For an example factor graph the lifting procedure is shown in Fig. 1. As shown on the left-hand side, assuming no evidence, all variable nodes are unknown, i.e., red/dark. Now, each variable node sends a message to its neighboring factor nodes saying "I am of color/shade red/black". A factor node sorts the incoming colors/shades into a vector according to the order the variables appear in its arguments. The last entry of the vector is the factor node's own color/shade, represented as light blue/gray square in Fig. 1. This color/shade signature is sent back to the neighboring variables nodes, essentially saying "I have communicated with these nodes". The variable nodes stack the incoming signatures together and, hence, form unique signatures of their one-step message history. Variable nodes with the same stacked signatures, i.e., message history can be grouped together. To indicate this, we assign a new color/shade to each group. In our running example, only variable node `B` changes its color/shade from red/black to yellow/gray The factors are grouped in a similar fashion based on the incoming color/shade signatures of neighboring nodes. Finally, we iterate the process. As the effect of the evidence propagates through the factor graph, more groups are created. The process stops when no new colors/shades are created anymore.

The final compressed factor graph $\mathfrak{G}$ is constructed by grouping all nodes with the same color/shade into so-called *clusternodes* and all factors with the same color/shade signatures into so-called *clusterfactors*. In our case, variable nodes `A`, `C` and factor nodes f_1, f_2 are grouped together, see the right hand side of Fig. 1. Clusternodes (resp. clusterfactors) are sets of nodes (resp. factors) that send and receive the same messages at each step of carrying out BP on G. It is clear that they form a partition of the nodes in G.

Now we can run BP with minor modifications on the compressed factor graph $\mathfrak{G}$.

Step 2 – BP on the Compressed Factor Graph: Recall that the basic idea is to simulate BP carried out on G on $\mathfrak{G}$. An edge from a clusterfactor $\mathfrak{f}$ to a cluster node $\mathfrak{X}$ in $\mathfrak{G}$ essentially represents multiple edges in G. Let $c(\mathfrak{f}, \mathfrak{X}, p)$ be the number of identical messages that would be sent from the factors in the clusterfactor $\mathfrak{f}$ to each node in the clusternode $\mathfrak{X}$ that appears at position p in $\mathfrak{f}$ if BP was carried out on G. The message from a clustervariable $\mathfrak{X}$ to a clusterfactor $\mathfrak{f}$ at position p is $\mu_{\mathfrak{X}\to\mathfrak{f},p}(x) =$

$$\mu_{\mathfrak{f},p\to\mathfrak{X}}(x)^{c(\mathfrak{f},\mathfrak{X},p)-1} \prod_{\mathfrak{h}\in \mathrm{nb}(\mathfrak{X})} \prod_{\substack{q\in P(\mathfrak{h},\mathfrak{X}) \\ (h,q)\neq(f,p)}} \mu_{\mathfrak{h},q\to\mathfrak{X}}(x)^{c(\mathfrak{h},\mathfrak{X},q)} ,$$

Algorithm 1: MR-CP: MapReduce Color-passing

repeat
  `// Color-passing for the factor nodes`
  forall $f \in G$ **do in parallel**
    $s(f) = (s(X_1), s(X_2), \ldots, s(X_{d_f}))$, where $X_i \in nb(f), i = 1, \ldots, d_f$
  Sort all of the signatures $s(f)$ in parallel using **MapReduce**;
  Map each signature $s(f)$ to a new color, s.t. $col(s(f_i)) = col(s(f_j))$ iff $s(f_i) = s(f_j)$
  `// Color-passing for the variable nodes`
  forall $X \in G$ **do in parallel**
    $s(X) = (s(f_1), s(f_2), \ldots, s(f_{d_X}))$, where $f_i \in nb(X), i = 1, \ldots, d_X$
  Sort all of the signatures $s(X)$ in parallel using **MapReduce**;
  Map each signature $s(X)$ to a new color, s.t. $col(s(X_i)) = col(s(X_j))$ iff $s(X_i) = s(X_j)$
until *grouping does not change* ;

where $\mathrm{nb}(\mathfrak{X})$ denotes the neighbor relation in the compressed factor graph $\mathfrak{G}$ and $P(\mathfrak{h}, \mathfrak{X})$ denotes the positions nodes from $\mathfrak{X}$ appear in $\mathfrak{f}$. The $c(\mathfrak{f}, \mathfrak{X}, p)-1$ exponent reflects the fact that a clustervariable's message to a clusterfactor excludes the corresponding factor's message to the variable if BP was carried out on G. The message from the factors to neighboring variables is given by $\mu_{\mathfrak{f},p\to\mathfrak{X}}(x) =$

$$\sum_{\neg\{\mathfrak{X}\}} \left(\mathfrak{f}(\mathbf{x}) \prod_{\mathfrak{Y}\in \mathrm{nb}(\mathfrak{f})} \prod_{q\in P(\mathfrak{f},\mathfrak{Y})} \mu_{\mathfrak{Y}\to\mathfrak{f},q}(y)^{c(\mathfrak{f},\mathfrak{Y},q)-\delta_{\mathfrak{X}\mathfrak{Y}}\delta pq} \right) ,$$

where $\delta_{\mathfrak{X}\mathfrak{Y}}$ and δpq are one iff $\mathfrak{X} = \mathfrak{Y}$ and $p = q$ respectively. The unnormalized belief of $\mathfrak{X}_i$, i.e., of any node X in $\mathfrak{X}_i$ can be computed from the equation

$$b_i(x_i) = \prod_{\mathfrak{f}\in \mathrm{nb}(\mathfrak{X}_i)} \prod_{p\in P(\mathfrak{f},\mathfrak{X}_i)} \mu_{\mathfrak{f},p\to\mathfrak{X}_i}(x_i)^{c(\mathfrak{f},\mathfrak{X},p)} .$$

Evidence is incorporated by setting $\mathfrak{f}(\mathbf{x}) = 0$ for states $\mathbf{x}$ that are incompatible with it. Again, different schedules may be used for message-passing. If there is no compression possible in the factor graph, i.e. there are no symmetries to exploit, there will be only a single position for a variable $\mathfrak{X}$ in factor $\mathfrak{f}$ and the counts $c(\mathfrak{f}, \mathfrak{X}, 1)$ will be 1. In this case the equations simplify to the standard BP equations. Indeed lifted belief propagation as introduced is an attractive avenue to scaling inference It can render large, previously intractable probabilistic inference problems quickly solvable by employing symmetries to handle whole sets of indistinguishable random variables. With the availability of affordable commodity hardware and high performance networking, however, we have increasing access to computer clusters providing an additional dimension to scale lifted inference. We now show how we can distribute the color-passing procedure for lifting message-passing using the MapReduce framework (Dean and Ghemawat 2008).

Algorithm 2: Map Function for Hadoop Sorting

Input: Split S of the network
Output: Signatures of nodes $X_i \in S$ as key value pairs $(s(X_i), X_i)$
forall $X_i \in S$ **do in parallel**
  $s(X_i) = (s(f_1), s(f_2), \ldots, s(f_{d_X}))$, where $f_j \in nb(X_i), j = 1, \ldots, d_{X_i}$
return *key-value pairs* $(s(X_i), X_i)$

Algorithm 3: Reduce Function for Hadoop Sorting

Input: Key-value pairs $(s(X_i), X_i)$
Output: Signatures with corresponding list of nodes
Form list L of nodes X_i having same signature
return *key-value pairs* $(s(L), L)$

MapReduce Lifting

The *MapReduce* programming model allows parallel processing of massive data sets It is basically divided into two steps, the *Map*- and the *Reduce*-step. In the *Map*-step the input is taken, divided into smaller sub-problems and then distributed to all worker nodes. These smaller sub problems are then solved by the nodes independently in parallel. Alternatively, the sub-problems can be further distributed to be solved in a hierarchical fashion. In the subsequent *Reduce*-step all outputs of the sub-problems are collected and combined to form the output for the original problem. Now each iteration of color-passing basically consists of three steps:

1. Form the colorsignatures
2. Group similar colorsignatures
3. Assign a new color to each group

for variables and factors, respectively. We now show how each step is carried out within the MapReduce framework. Recall that color-passing is an iterative procedure so that we only need to take care of the direct neighbors in every iteration, i.e. building the colorsignatures for the variables (factors) requires the variables' (factors') own color and the color of the neighboring factors (variables). *Step-1* can thus be distributed by splitting the network into k parts and forming the colorsignatures within each part independently in parallel. However, care has to be taken at the borders of the splits. Fig. 2 shows the factor graph from our previous example and how it can be split into parts for forming the colorsignatures for the variables (Fig. 2 **(left)**) and the factors (Fig. 2 **(right)**). To be able to correctly pass the colors in this step we have to introduce copynodes at the borders of the parts.[1] In the case of the node signatures for our earlier example factor graph (Fig. 2 **(left)**), both f_1 and f_2 have to be duplicated to be present in all signatures of the variables. The structure of the network does not change during color-passing, only the colors may change in two subsequent iterations and have to be communicated. To reduce communication cost we thus split the network into parts before we

[1]Note that in the shared memory setting this is not necessary. Here we use MapReduce , thus we have to introduce copynodes.

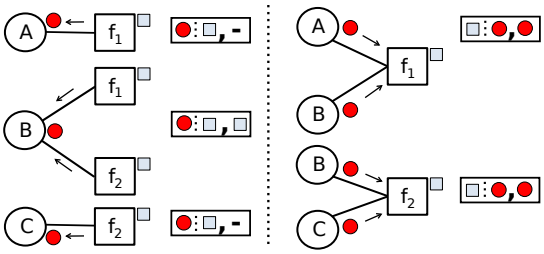

Figure 2: The partitions of color-passing for the variable **(left)** and factor **(right)** signatures and the corresponding colorsignatures that have to be sorted. Copynodes (-factors) are introduced at the borders to form correct neighborhoods.

start color-passing and use this partitioning in all iterations.[2]

The next step that has to be carried out is the sorting of the signatures of the variables (factors) within the network (*Step-2*). These signatures consist of the node's own color and the colors of the respective neighbors. Fig. 2 also shows the resulting colorsignatures after color-passing in the partitions. For each node (Fig. 2 **(left)**) and factor (Fig. 2 **(right)**) we obtain the colorsignatures given the current coloring of the network. These colorsignatures have to be compared and grouped to find nodes (factors) that have the same one-step communication pattern. Finding similar signatures by sorting them can be efficiently carried out by using radix sort (Cormen et al. 2001) which has been shown to be MapReduce-able (Zhu et al. 2009). Radix sort is linear in the number of elements to sort if the length of the keys is fixed and known. It basically is a non-comparative sorting algorithm that sorts data with grouping keys by the individual digits which share the same significant position and value. The signatures of our nodes and factors can be seen as the keys we need to sort and each color in the signatures can be seen as a digit in our sorting algorithm. Indeed, although radix sort is very efficient — the sorting efficiency is in the order of edges in the ground network — one may wonder whether it is actually well suited for an efficient implementation of lifting within a concrete MapReduce framework such as Hadoop. The Hadoop framework performs a sorting between the *Map*- and the *Reduce*-step. The key-value pair that is returned by the mappers has to be grouped and is then sent to the respective reducers to be processed further. This sorting is realized within Hadoop using an instance of quick sort with an efficiency $\mathcal{O}(n \log n)$. If we have a bounded degree of the nodes in the graph as in our case, however, this limits the length of the signatures and radix sort is still the algorithm of choice. Moreover, our main goal is to illustrate that — in contrast to many other lifting approaches — color-passing is naturally MapReduce-able. Consequently, for the sake of simplicity, we stick to the Hadoop internal sorting to group our signatures and leave a highly efficient MapReduce realization for future work. Alg.2 and Alg. 3 show the map and the reduce function respectively. In the map phase we form the signatures of all nodes (factors) in a split of the graph and reduce a key-value pair of $(s(X_i), X_i)$ which are the signature and the id of the node (factor). These are then grouped by the reduce function and a pair $(s(L), L)$ for all nodes having the same signature, i.e. $s(L) = s(X_i)$ for all $X_i \in L$. In practice one could gain additional speed-up the if the sorting of Hadoop is avoided, for example by realizing a *Map*-only task using a distributed ordered database like *HBase* [3]. The signatures that are formed in *Step-1* would be written directly into the database as $(s(X_i)\_X_i, X_i)$, i.e. for each node an entry with a key that is composed of the signature and the node's id is inserted and the groupings could be read out sequentially for the next step. The *Map*-phase of (*Step-2*) could also be integrated into the parallel build of the signatures (*Step-1*), such that we have one single *Map*-step for building and sorting the signatures.

Finally, reassigning a new color (*Step-3*) is an additional linear time operation that does one pass over the nodes (factors) and assigns a new color to each of the different groups. That is, we first have to build the color signatures. The splits of the network are the input and are distributed to the mappers. In the example shown in Fig. 3 there is one line for every node and its local neighborhood, i.e. ids of the factors it is connected to. The mappers take this input and form the signatures of the current iteration independently in parallel. These signatures are then passed on to MapReduce sort. The mappers in the sorting step, output a tuple of key-value pairs consisting of the the signature and the *id* of the respective node. [4] These are then grouped by the reduce operation. All nodes having the same signatures are returned in a list.

Taking the MapReduce arguments for (*Step-1*) to (*Step-3*) together this proves that color-passing itself is MapReduce-able. Together with the MapReduce-able results of Gonzalez *et al.* (Gonzalez et al. 2009) for the modified belief propagation, this proves the following Theorem.

Theorem 1. *Lifted belief propagation is MapReduce-able.*

Moreover, we have the following time complexity, which essentially results from running radix sort h times.

Theorem 2. *The runtime complexity of the color-passing algorithm with h iterations is $\mathcal{O}(hm)$, where m is the number of edges in the graph.*

Proof. Assume that every graph has n nodes (ground atoms) and m edges (ground atom appearances in ground clauses). Defining the signatures in step 1 for all nodes is an $\mathcal{O}(m)$ operation. The elements of a signature of a factor are $s(f) = (s(X_1),\ s(X_2), \ldots, s(X_{d_f}))$, where $X_i \in nb(f)$, $i = 1, \ldots, d_f$. Now there are two sorts that have to be carried out. The first sort is within the signatures. We have to sort

[2] Note that how we partition the model greatly affects the efficiency of the lifting. Finding an optimal partitioning that balances communication cost and CPU-load, however, is out of the scope of this paper. In general, the partitioning problem for parallelism is well-studied (Chamberlain 1998), there are efficient tools, e.g. `http://glaros.dtc.umn.edu/gkhome/metis/parmetis/`. We show that CP is MapReduce-able and dramatically improves scalability even with a naive partitioning.

[3] http://hbase.apache.org/

[4] Here, the output of the previous step is essentially passed through. The two steps could be integrated. We keep them separate for illustration purposes of the two distinct color-passing steps.

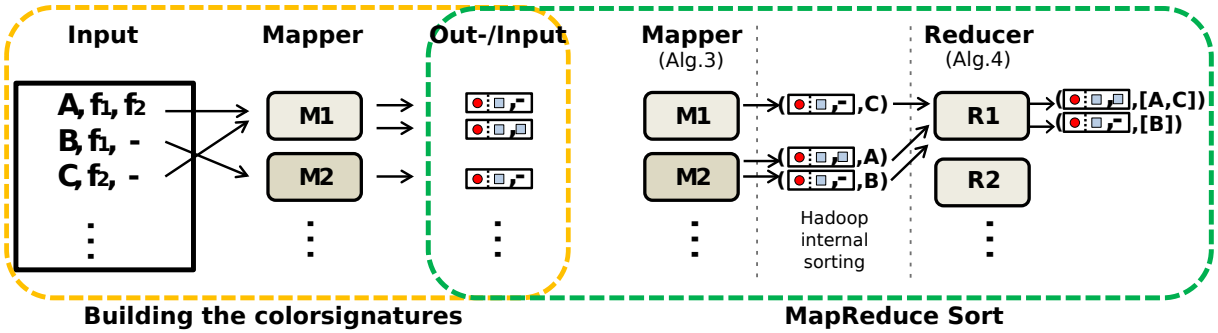

Figure 3: MapReduce jobs for *Step-1* **(yellow)** and *Step-2* **(green)** of color-passing. For building the signatures the local parts are distributed to the mappers that form the signature based on the colorings of the current iteration. The output is sorted and for each signature a list of nodes with the same signature is returned.

the colors within a node's signatures and in the case where the position in the factor does not matter, we can safely sort the colors within the factor signatures while compressing the factor graph. Sorting the signatures is an $\mathcal{O}(m)$ operation for all nodes. This efficiency can be achieved by using counting sort, which is an instance of bucket sort, due to the limited range of the elements of the signatures. The cardinality of this signature is upper-bounded by n, which means that we can sort all signatures in $\mathcal{O}(m)$ by the following procedure. We assign the elements of all signatures to their corresponding buckets, recording which signature they came from. By reading through all buckets in ascending order, we can then extract the sorted signatures for all nodes in a graph. The runtime is $\mathcal{O}(m)$ as there are $\mathcal{O}(m)$ elements in the signatures of a graph in iteration i. The second sorting is that of the resulting signatures to group similar nodes and factors. This sorting is of time complexity $\mathcal{O}(m)$ via radix sort. The label compression requires one pass over all signatures and their positions, that is $\mathcal{O}(m)$. Hence all these steps result in a total runtime of $\mathcal{O}(hm)$ for h iterations. □

Evaluation

We have just shown for the first time that lifting per se can be carried out in parallel. Thus, we can now combine the gains we get from the two orthogonal approaches to scaling inference, putting scaling lifted SRL to Big Data within reach. That is, we investigate the question whether the MapReduce lifting additionally improves scalability. We compare the color-passing procedure with a single-core Python/C++ implementation and a parallel implementation using MRJob[5] and Hadoop [6]. We partitioned the network per node (factor) and introduced copynodes (copyfactors) at the borders. The grouping of the signatures was found by a MapReduce implementation of Alg. 2 and Alg. 3.

Note that for this experiment the amount of lifting is not important. Here, we compare how the lifting methods scale. Whether we achieve lifting or not does not change the runtime of the lifting computations. As Theorem 2 shows, the runtime of color-passing depends on the number of edges present in the graph. Thus, we show the time needed for lifting on synthetic grids of varying size to be able to precisely control their size and the number of edges in the network. Fig. 4 shows the results on grids of varying size. We scaled the grids from 5×5 to 500×500, resulting in networks with 25 to 250.000 variables. The ratio is 1 at the black horizontal line which indicates that at this point both methods are equal. Fig. 4 **(left)** shows the ratio of the runtime for single-core color-passing and MapReduce color-passing (MR-CP) carried out on four cores. We see that due to the overhead of MapReduce the single core methods is faster for smaller networks. However, as the number of variables grows, this changes rapidly and MR-CP scales to much larger instances. Fig. 4 **(right)** shows the running time of single-core color-passing and MapReduce color-passing. One can see that MR-CPs scales orders of magnitude better than single-core color-passing. The results clearly favor MR-CP for larger networks and affirmatively answers our question.

Conclusions

Symmetries can be found almost everywhere and have also been explored in many AI tasks. For instance, there are symmetry-aware approaches in (mixed–)integer programming (Bödi, Herr, and Joswig 2011; Margot 2010), linear programming (Herr and Bödi 2010; Mladenov, Ahmadi, and Kersting 2012), SAT and CSP (Sellmann and Van Hentenryck 2005) as well as MDPs (Dean and Givan 1997; Ravindran and Barto 2001). While there have been considerable advances in statistical relational AI, one should start investigating symmetries in general AI and machine learning approaches such as support-vector machines and Gaussian processes. In this paper have introduced lifted belief propagation and shown that lifting is MapReduce-able, thus combining two orthogonal approaches to scaling inference.

Acknowledgements: The authors would like to thank Roman Garnett, Fabian Hadiji and Mirwaes Wahabzada for helpful discussion on map-reduce. Babak Ahmadi and Kristian Kersting were supported by the Fraunhofer ATTRACT fellowship STREAM and by the European Commission under contract number FP7-248258-First-MM. Martin Mladenov and Kristian Kersting were supported by the German

[5] https://github.com/Yelp/mrjob
[6] http://hadoop.apache.org/

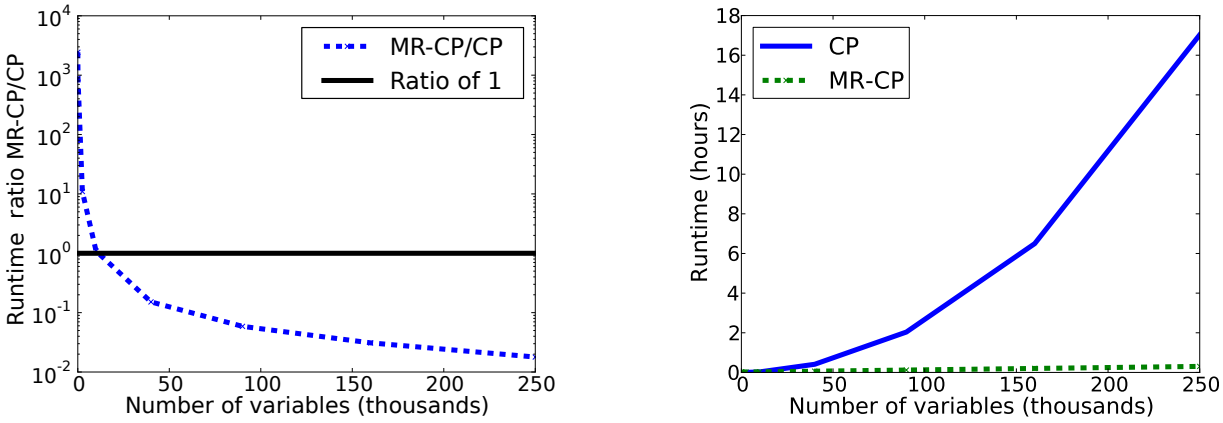

Figure 4: **(left)** The ratio of the runtime for single-core color-passing and MapReduce color-passing (MP-CP) on grids of varying size. The ratio is 1 at the black horizontal line which indicates that at this point both methods are equal. We see that due to the overhead of MapReduce for smaller networks the single core methods is a few orders of magnitude faster. However, as the number of variables grows, this changes rapidly and the MP-CP scales to much larger instances. **(right)** Runtimes for color-passing and MR-CP. One can clearly see that MR-CP scales lifting to much larger instances.

Research Foundation DFG, KE 1686/2-1, within the SPP 1527. Sriraam Natarajan gratefully acknowledges the support of the DARPA Machine Reading Program under AFRL prime contract no. FA8750-09-C-0181. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the view of DARPA, AFRL, or the US government.

References

Bödi, R.; Herr, K.; and Joswig, M. 2011. Algorithms for highly symmetric linear and integer programs. *Mathematical Programming, Series A*.

Chamberlain, B. L. 1998. Graph partitioning algorithms for distributing workloads of parallel computations. Technical report.

Cormen, T. H.; Leiserson, C. E.; Rivest, R. L.; and Stein, C. 2001. *Introduction to Algorithms*. The MIT Press, 2 edition.

De Raedt, L.; Frasconi, P.; Kersting, K.; and Muggleton, S., eds. 2008. *Probabilistic Inductive Logic Programming*, volume 4911 of *Lecture Notes in Computer Science*. Springer.

Dean, J., and Ghemawat, S. 2008. Mapreduce: simplified data processing on large clusters. *Commun. ACM* 51(1):107–113.

Dean, T., and Givan, R. 1997. Model minimization in markov decision processes. In *Proc. of the Fourteenth National Conf. on Artificial Intelligence (AAAI-97)*, 106–111.

Getoor, L., and Taskar, B. 2007. *Introduction to Statistical Relational Learning*. The MIT Press.

Gonzalez, J.; Low, Y.; Guestrin, C.; and O'Hallaron, D. 2009. Distributed parallel inference on large factor graphs. In *UAI*.

Herr, K., and Bödi, R. 2010. Symmetries in linear and integer programs. *CoRR* abs/0908.3329.

Kersting, K.; Ahmadi, B.; and Natarajan, S. 2009. Counting belief propagation. In *UAI*.

Kersting, K. 2012. Lifted probabilistic inference. In Raedt, L. D.; Bessiere, C.; Dubois, D.; Doherty, P.; Frasconi, P.; Heintz, F.; and Lucas, P., eds., *Proc. of 20th European Conference on Artificial Intelligence (ECAI–2012)*. Montpellier, France: ECCAI. (Invited Talk at the Frontiers of AI Track).

Margot, F. 2010. Symmetry in integer linear programming. In Jünger, M.; Liebling, T.; Naddef, D.; Nemhauser, G.; Pulleyblank, W.; Reinelt, G.; Rinaldi, G.; and Wolsey, L., eds., *50 Years of Integer Programming 1958–2008: From the Early Years to the State-of-the-Art*. Springer. 1–40.

Mladenov, M.; Ahmadi, B.; and Kersting, K. 2012. Lifted linear programming. In *15th Int. Conf. on Artificial Intelligence and Statistics (AISTATS 2012)*, 788–797. Volume 22 of JMLR: W&CP 22.

Murphy, K.; Weiss, Y.; and Jordan, M. 1999. Loopy Belief Propagation for Approximate Inference: An Empirical Study. In *Proc. of the Conf. on Uncertainty in Artificial Intelligence (UAI-99)*, 467–475.

Pearl, J. 1991. *Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann, 2. edition.

Ravindran, B., and Barto, A. 2001. Symmetries and model minimization in markov decision processes. Technical report, University of Massachusetts, Amherst, MA, USA.

Sellmann, M., and Van Hentenryck, P. 2005. Structural symmetry breaking. In *Proc. of 19th International Joint Conf. on Artificial Intelligence (IJCAI-05)*.

Singla, P., and Domingos, P. 2008. Lifted First-Order Belief Propagation. In *Proc. of the 23rd AAAI Conf. on Artificial Intelligence (AAAI-08)*, 1094–1099.

Zhu, S.; Xiao, Z.; Chen, H.; Chen, R.; Zhang, W.; and Zang, B. 2009. Evaluating splash-2 applications using mapreduce. In *Proceedings of the 8th International Symposium on Advanced Parallel Processing Technologies*, APPT '09, 452–464. Springer-Verlag.