# A Hypergraph-Partitioned Vertex Programming Approach for Large-scale Consensus Optimization

Hui Miao*, Xiangyang Liu†, Bert Huang*, Lise Getoor*
*Dept. of Computer Science, †Dept. of Electrical & Computer Engineering
University of Maryland, College Park, USA
{hui, bert, getoor}@cs.umd.edu*, xyliu@umd.edu†

***Abstract*—In modern data science problems, techniques for extracting value from big data require performing large-scale optimization over heterogenous, irregularly structured data. Much of this data is best represented as multi-relational graphs, making vertex-programming abstractions such as those of Pregel and GraphLab ideal fits for modern large-scale data analysis. In this paper, we describe a vertex-programming implementation of a popular consensus optimization technique known as the *alternating direction method of multipliers* (ADMM) [1]. ADMM consensus optimization allows the elegant solution of complex objectives such as inference in rich probabilistic models. We also introduce a novel hypergraph partitioning technique that improves over the state-of-the-art vertex programming framework and significantly reduces the communication cost by reducing the number of replicated nodes by an order of magnitude. We implement our algorithm in GraphLab and measure scaling performance on a variety of realistic bipartite graphs and a large synthetic voter-opinion analysis application. We show a 50% improvement in running time over the current GraphLab partitioning scheme.**

***Keywords*-consensus optimization; large-scale optimization; partitioning methods; vertex programming;**

## I. Introduction

Large-scale data often contains noise, statistical dependencies, and complex structure. To extract value from such data, we need both flexible, expressive models and scalable algorithms to perform reasoning over the data. In this paper, we show how a general class of distributed optimization techniques can be implemented efficiently on graph-parallel abstraction frameworks.

Consensus optimization using *alternating direction method of multipliers* (ADMM) is a recently popularized general method for distributed large-scale optimization [1]. The dual optimization problem is decomposed into simple subproblems to be solved in parallel. The combination of the decomposition and the fast convergence of the method of multipliers makes it suitable for many problems such as inference in graphical models [2], [3] and popular machine learning algorithms [4], [5].

*Vertex programming* is an efficient graph-parallel abstraction for distributed graph computation. Pregel [6] and GraphLab [7] are recently proposed frameworks for parallelizing graph-intensive computation. These frameworks adopt a vertex-centric model to define independent programs on each vertex. Experiments show they outperform the MapReduce abstraction by one to two orders of magnitude in machine learning and data mining algorithms [8].

In this paper, we first investigate the bipartite topology of general ADMM-based consensus optimization, present its vertex-programming formulation, and develop a scalable, parallel algorithm. Previously, Boyd et al. [1] discussed the MapReduce implementation of ADMM with a single global consensus variable node, which is a special case of our setup. Secondly, we propose a novel partitioning scheme that uses the characteristics of the computation graph and a hypergraph interpretation of the bipartite data graph. Our partitioning can reduce the number of replicated vertices by an order of magnitude over the current GraphLab partitioning scheme [7], reducing communication cost accordingly, and halving the running time. Our partitioning strategy is of independent interest in vertex-programming algorithm design, since it can be used for any problem that decomposes into a bipartite computation graph, such as belief propagation in factor-graphs.

## II. Motivation

Before describing our proposed algorithm (Section IV), we begin with a simple illustrative example of the kind of problems that can be solved using ADMM. We consider the task of analyzing voting preferences of individuals connected by various relationships in a social network. The problem can be cast as a probabilistic inference problem, and the approach that we take here is to define the model using *probabilistic soft logic* (PSL) [9]. PSL is a general-purpose language for describing large-scale probabilistic models over continuous random variables using weighted logical rules.

A PSL program consists of a set of logical rules with conjunctive bodies and disjunctive heads (negations allowed). Rules are labeled with non-negative weights. The program in Fig. 1 encodes a simple model to predict voter behavior using information about a voter (registration) and their social network described by different types of links indicating various social relationships, such as Friend and Spouse.

0.5: REGISTEREDAS*(A, P)* → VOTES*(A,P)*,
0.5: VOTES*(A, P)* ∧ FRIEND*(B, A)* → VOTES*(B,P)*,
1.8: VOTES*(A, P)* ∧ SPOUSE*(B, A)* → VOTES*(B,P)*,
0.05: VOTES*(A, P)* ∧ BOSS*(B, A)* → VOTES*(B,P)*,
0.1: VOTES*(A, P)* ∧ MENTOR*(B, A)* → VOTES*(B,P)*,
0.7: VOTES*(A, P)* ∧ OLDERRELATIVE*(B, A)* → VOTES*(B,P)*.

Figure 1. Political social network voting program written in probabilistic soft logic. Additionally, the VOTES predicate is constrained to have total truth value of 1.0, to preserve mutual exclusivity of voting preference.

Consider any constants for persons, $a$ and $b$ and party $p$ instantiating logical terms $A$, $B$, and $P$ respectively. The first rule encodes the correlation between voter registration and party preferences, which tend to be aligned but are not always. The next rule states that if $a$ is a friend of $b$ and votes for party $p$, there is a chance that $b$ votes for party $p$ as well, and the second rule makes the same statement for spouses. The rule weights indicate that spouses are more likely to vote for the same party than friends. The resulting probabilistic model will combine all of these influences and include the implied structured dependencies. PSL can also include constraints on logical atoms, such as mutual exclusivity of voting preferences VOTES.

The engine behind PSL compiles the logical program into a continuous-variable representation known as a *hinge-loss Markov random field* (HL-MRF) [3], [5]. Like many probabilistic graphical models, inference in HL-MRFs can be distributed and solved using consensus optimization. In HL-MRFs, inference of the most-probable explanation (MPE) is a convex optimization, and HL-MRFs are particularly well-suited for consensus optimization. We defer to previous papers for the mathematical formalisms of HL-MRFs, and here mainly discuss the general ADMM-based consensus optimization, which has many applications beyond PSL.

## III. PRELIMINARIES

### A. ADMM-Based Consensus Optimization

Consensus optimization simplifies the solution of a global objective by decomposing a complex objective into simpler subproblems over local copies of the variables and constraining each local copy to be equal to a global *consensus variable*. The general form of the consensus optimization is

$$\min_{x_1,\ldots,x_N} \quad \sum_{i=1}^{N} \phi_i(x_i)$$
$$\text{subject to} \quad x_i - \vec{X}_i = 0, i = 1, 2, ..., N,$$

where $x_i$ with dimension $n_i$ is the local variable vector on which the $i$th subproblem depends and $\phi_i$ is the objective function for the $i$th subproblem. Let $\vec{X}_i$ denote the global consensus variable vector that local variable $x_i$ should equal.

Relaxing the global equality constraints by the augmented Lagrangian [1], the ADMM-based solution becomes:

$$x_i^{k+1} \leftarrow \underset{x_i}{\operatorname{argmin}} \left( \phi_i(x_i) + \lambda_i^k \cdot x_i + \frac{\rho}{2} \left\| x_i - \vec{X_i}^{\,k} \right\|_2^2 \right), \forall i$$
$$\lambda_i^{k+1} \leftarrow \lambda_i^k + \rho \left( x_i^{k+1} - \vec{X_i}^{\,k} \right), \forall i$$
$$X_l^{k+1} \leftarrow \frac{1}{N_l} \sum_{M(i,j)=l} (x_i)_j^{k+1}, \forall l$$

where superscript $k$ represents the iteration, $N_l$ is the number of local copies of the $l$th entry of global consensus variable, $\lambda_i$ is the vector of Lagrange multipliers for $i$th subproblem, $M(i, j)$ is the corresponding global consensus entry for $j$th dimension of local variable $x_i$, $X_l$ denotes the $l$th entry of global consensus variable, and $\rho$ is a step size parameter. The update of $X^{k+1}$ can be viewed as averaging local copies in subproblems. Throughout the remainder of the paper, we refer to ADMM-based consensus optimization as ACO.

The above equation shows that each of the subproblems can be solved independently. This general form of consensus optimization defines a bipartite structure $G(S, C, E)$ where $S$ denotes the set of subproblems containing local variables $\{x_i | i = 1, 2, \ldots, N\}$, $C$ represents the set of consensus variable entries, and $E$ expresses the dependencies. Each subproblem $\phi_i(x_i)$ is connected to its dependent consensus variables, while each consensus variable $X_j$ is connected to subproblems containing its local copies. This bipartite dependency graph makes ADMM computation well-suited for vertex-processing parallelization.

### B. Vertex Programming Frameworks

Recent development of vertex programming frameworks, such as Pregel [6] and GraphLab [8], are aimed at improving the scalability of graph processing. Vertex-centric models execute user-defined functions on each vertex independently and define the order of execution of vertices. Pregel and GraphLab have superior computational performance over MapReduce for many data mining and machine learning algorithms, such as belief propagation, Gibbs sampling, and PageRank [6], [8]. In this paper, we implement our synchronous vertex programs in GraphLab.

Gonzalez et al. [7] propose the *gather-apply-scatter* (*GAS*) abstraction that describes common structures of various vertex programming frameworks. In the vertex programming setting, the user defines a data graph with data structures representing vertices, edges, and messages. The user provides a vertex program associated with each vertex. The GAS model abstracts the program into three conceptual phases of execution on each vertex. In the *gather* phase, each vertex is able to aggregate neighborhood information, which can be pushed or pulled from adjacent nodes. The aggregation during this phase is user-defined, but must be commutative and associative. In the *apply* phase, each vertex can use its aggregated value to update its own associated data. Finally in the *scatter* phase, each vertex either sends messages to its neighbors or updates other vertices or edges in its neighborhood via global state variables.

## IV. Distributed ADMM-based consensus optimization implementation (ACO)

In order to implement a graph algorithm using vertex programming, one needs to define the data graph structure and a vertex program that defines the computation.

### A. Data Graph and Data Types

In ADMM-based consensus optimization, the computation graph is based on the dependencies between subproblems and consensus variables in the bipartite dependency graph $G(S, C, E)$ in which any consensus variable has a degree of at least two, since any local variables that only appear in one subproblem do not need consensus nodes. We use different data types for subproblem vertices and consensus-variable vertices, denoted *sub* and *con*, respectively. For each subproblem vertex $v_i \in S$, we maintain the involved local variables in $x_i$ and associated Lagrange multiplier $\lambda_i$, both of which are $n_i$ dimension vectors. Each $v_i \in S$ also stores a vector $\vec{X}_i$ of dimensionality $|E_{\mathbf{v}_i}|$ for holding the dependent consensus variable values. For each consensus variable $v_j \in C$, we only store its current value.

### B. ACO Vertex Program

We use the GAS abstraction introduced in Section III-B to describe our ADMM-based consensus optimization implementation, ACO, shown in Alg. 1. In each iteration, we define a temporary consensus-variable key-value table `consensus_var`, where the key is a consensus variable's global unique ID, i.e., `consensus_var`$[id(\vec{X})] \rightarrow \vec{X}$. We also define a program variable `local_copy_sum` to aggregate the sum of each local copy of a consensus variable.

As shown in Alg. 1, in the gather, apply and scatter stages, we alternate computation on the subproblem nodes $S$ and the consensus nodes $C$. We describe the computation for each type and the termination condition below.

```
1  ACO Algorithm
2  // gather neighbor information
3  gather(v_i, (v_i, v_j), v_j):
4    if v_i.type == sub
5      consensus_var[id(v_j.X_j)] ← v_j.X_j^k
6    else
7      local_copy_sum += v_j.x_j^k[id(v_i.X_i)]
8  // update the vertex data of v_i
9  apply(v_i, sum_result):
10   // get consensus_var new value, solve
       objective, update multiplier
11   if v_i.type == sub
12     v_i.X_i ← consensus_var
13     v_i.x_i ← argmin_{x_i}(φ_i(x_i) + v_i.λ_i · x_i + (ρ/2) ||x_i − v_i.X_i||_2^2)
14     v_i.λ_i ← v_i.λ_i + ρ(v_i.x_i − v_i.X_i)
15   // average the sum of each local copy
16   else
17     v_i.X_i ← local_copy_sum/degree(v_i)
18 // update neighborhood
19 scatter(v_i, (v_i, v_j), v_j):
20   // notice consensus node the value change
21   if v_i.type == sub
22     notify(v_j)
23   else
24     if (convergence_check() == false)
25       notify(v_j)
```

Algorithm 1. The ACO vertex program for ADMM-based Consensus Optimization on vertex $\mathbf{v}_i$ at iteration $k+1$

*1) Subproblem Nodes:* In the gather phase of the $(k+1)^{th}$ iteration, each subproblem node $v_i \in S$ reads the consensus variables updated in the $k^{th}$ iteration in its neighborhood. We store each consensus variable in the key-value table `consensus_var`. The commutative and associative aggregation function combines the key-value tables. We use it to solve the optimization in subproblem in line 13, and the $x_i$ vector is updated to the solution. Note that the subproblem solver is application-specific. In the scatter, the subproblem always notifies dependent consensus nodes.

*2) Consensus Variable Nodes:* Consensus variable nodes $v_j \in C$ behave differently. Each aggregates the sum of all local copies from the subproblem nodes in the gather phase, then updates itself with the average in the apply phase. In the scatter phase, convergence conditions are used to determine whether related subproblems need to be run again.

*3) Termination Conditions:* One possible criterion for convergence is the global primal and dual residual of all consensus variables and their local copies. At the superstep, an aggregator can be used to calculate residuals across all consensus variables. If both primal and dual residuals are small enough, then we reach global convergence and the program stops. If not, all subproblems are scheduled again in the next iteration. However, this global convergence criterion has two disadvantages: first, the use of aggregator brings overhead as it needs to aggregate information from all machines; second, some consensus variables and corresponding local copies do not change much and the subproblem counterparts are still scheduled to run, getting the same solution, thus wasting computation resources.

Instead, our proposed convergence criterion measures local convergence. In this local criterion, each consensus vertex calculates both primal and dual residuals using its dependent local copies only; if both of them are small, it does not notify connected subproblem nodes to run in the next iteration. For a particular subproblem node, if none of the connected consensus variables notifies it, it skips the next iteration, thus saving computation. A skipped subproblem node will run again if some dependent consensus variable is updated and the local convergence criterion is not met.

## V. Hypergraph partitioning

Next we introduce a novel hypergraph-based partitioning scheme (Hyper) that is better suited for ACO than current state-of-the-art approaches, e.g., in GraphLab's PowerGraph [7]. The primary factors for efficient implementation of

distributed graph algorithms are load balancing and communication cost. Vertex programming frameworks optimize these by distributing data according to balanced $p$-way cuts of their graphs. In our work, we use a vertex-cut formulation in the same manner as GraphLab, which has been shown to produce lower communication overhead than Pregel's hash-based random edge-cut strategy [7], [10].

### A. Problem Definition and Notation

Let $G(V, E)$ be a graph, $\beta$ be the imbalance parameter. For any $v \in V$, let $A(v)$ denote the subset of $M$ machines that vertex $v$ is assigned to. Then the *balanced $p$-way vertex cut problem* for $M$ machines is defined as

$$\begin{aligned} \min_{A} \quad & \frac{1}{|V|} \sum_{v \in V} |A(v)| \\ \text{subject to} \quad & |\{e \in E | A(e) = m\}| \leq \beta \frac{|E|}{M}, \forall m \end{aligned} \tag{1}$$

where $m \in \{1, \ldots, M\}$. The objective corresponds to the *replication factor*, and the constraint corresponds to a limit on the edges that can be assigned to any one machine.

### B. Intuition and Partitioning in State-of-the-Art Framework

The current best strategy used in GraphLab is a sequential greedy heuristic algorithm [7], which we refer to as GREEDY. Multiple machines process sets of edges one by one and place each of them into a machine, where the placement $A(v)$ is maintained across multiple machines. When a machine places an edge $(u, v)$, the GREEDY strategy follows heuristic rules: if both $A(u)$ and $A(v)$ are $\emptyset$, edge $(u, v)$ is placed on the machine with the fewest assigned edges; if only one of $A(u)$ and $A(v)$ is not $\emptyset$, say $A(u)$, then $(u, v)$ is put in one machine in $A(u)$; if $A(u) \cap A(v) \neq \emptyset$, then $(u, v)$ is assigned to one of the machines in the intersection; the last case is both $A(u)$ and $A(v)$ are not $\emptyset$, but $A(u) \cap A(v) = \emptyset$, then $(u, v)$ is assigned to one of the machines from the vertex with the most unassigned edges.

Unfortunately, this GREEDY strategy does not work well with bipartite ACO graphs. In ACO and other similar problem structures, subproblem nodes tend to have much lower degree than consensus nodes. Because the last heuristic in the greedy scheme is biased to large degree nodes, GREEDY tends to place subproblem nodes onto different machines. On the other hand, in practice, large-scale ACO involves millions of consensus variables, so computing a high-quality partitioning is more important than fast sequential partitioning. Once partitioned, the same topology may be reused multiple times, for example when performing parameter optimization.

### C. Hypergraph-based Partition for ACO

A large-scale ACO bipartite graph $G(S, C, E)$ exhibits three properties which do not fit well with the greedy heuristics. **a)** The subproblem optimization in $v_s \in S$ is much more expensive than the simple averaging in consensus varaible $v_c \in C$. Thus cutting a subproblem vertex into multiple machines results in solving the subproblem optimization multiple times; **b)** The consensus variable degree follows a power-law degree distribution, while subproblem degree distribution is centered around some small number; and **c)** $|S|$ is usually much larger than $|C|$. In large-scale ACO problems, such degree distributions are common when variables involved in subproblems correspond to objects in the real world, especially for applications on social and natural networks. On the other hand, due to the utility of the ADMM decomposition, the original optimization problem decomposes into small subproblems that are each easy to solve, thus the degree of each subproblem node is small. These properties of the bipartite ACO graph motivate us to split only the consensus variables $C$ instead of partitioning over the whole set $S \cup C$, to avoid solving subproblems repeatedly and also to reduce the problem size.

Cutting $C$ with the balanced edge placement constraint in $G(S, C, E)$ can be transformed to a hyperedge cut problem with a balanced vertex placement constraint in the hypergraph $H(V_S, E_C)$, where $V_S$ corresponds to $S$, $E_C$ is a set of hyperedges, and each $e \in E_C$ represents the set of all related subproblems of a consensus variable. Hypergraph partitioning is a well-studied area and there are efficient packages avialable, e.g. *hMETIS* [11]. Our proposed hypergraph partitioning formulation is novel in the bipartite graph structure in vertex programming setting. Hypergraph-based general graph partitioning is studied in [10], [12].

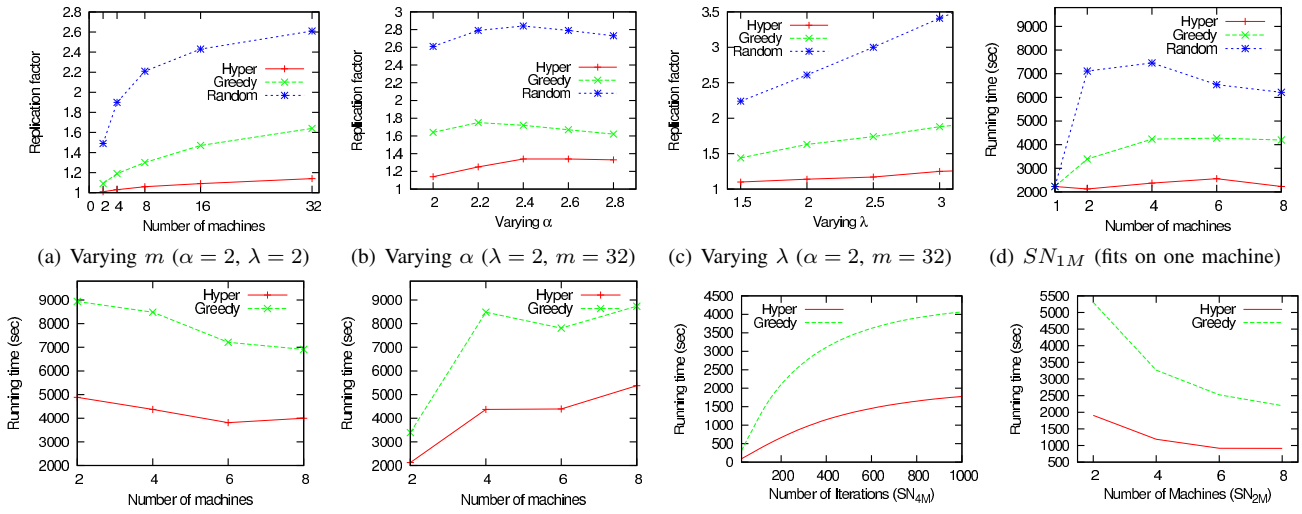
## VI. EXPERIMENTS

In this section, we first compare our hypergraph partitioning vertex-cut technique HYPER with the greedy vertex-cut GREEDY and hash-based random partitioning RANDOM. We then present the evaluation of our ACO implementation using GraphLab on the large-scale social network analysis problem introduced in Section II. In all experiments, we use *hMetis* [13] with unbalanced factor $\beta = 2$ and use the *sum of external degree* objective to perform the hyperedge cut.

### A. Evaluation of Partitioning Strategies

*1) Dataset Description:* We begin by studying the effect of different partitioning strategies on the replication factor of the vertex program. For more details, please refer to our technical report [14]. We generate graphs with a power-law degree distribution for consensus variable vertices with shape parameter $\alpha$ ($\alpha \in \{2, 2.2, 2.4, 2.6, 2.8\}$) and a Poisson distribution over the degree of subproblems with parameter $\lambda$ ($\lambda \in \{1.5, 2, 0, 2.5, 3.0, 3.5\}$). We also fix the number of consensus variables to $100,000$.

*2) Replication Factor Results with Synthetic Data:* Given the generated dataset, we vary the number of machines (partitions) $m \in \{2, 4, 8, 16, 32\}$ and measure the replication factor $RF = \frac{1}{|V|} \sum_{i=1}^{N} |A(v)|$ of each scheme in Fig. 2. In general, in all generated datasets, HYPER always has a smaller replication factor than GREEDY and RANDOM.




(a) Varying $m$ ($\alpha = 2$, $\lambda = 2$) (b) Varying $\alpha$ ($\lambda = 2$, $m = 32$) (c) Varying $\lambda$ ($\alpha = 2$, $m = 32$) (d) $SN_{1M}$ (fits on one machine)

(e) $SN_{2M}$ (f) Weak scaling with increasing size (g) Run time of iterations ($SN_{4M}$) (h) Execution Time ($SN_{2M}$)

Figure 2. Performance of different partitioning scheme. Scaling analysis under both full and partial convergence

In the worst case, GREEDY replicates around $17\times$ more vertices than HYPER ($\alpha = 2, \lambda = 2.5, m = 2$), and always replicates $1.6\times$ more ($\alpha = 2.8, \lambda = 3.5, m = 32$). In Fig. 2(a), we vary $m$ to show how replication factor grows when the number of machines increases. The results show that HYPER is less sensitive to the number of machines than the other schemes and scales better in practice.

Next we fix $m = 32$, vary $\alpha$ and $\lambda$ to study the partitioning performance based on different bipartite graph topologies. Recall that parameter $\alpha$ determines the power-law shape, as shown in Fig. 2(b), the larger $\alpha$ is, the smaller the maximum degree of the consensus nodes become, and the difference between $|S|$ and $|C|$ is smaller, e.g., when $\alpha = 2.8, \lambda = 2$ in the plot, $|S|/|C|$ is only 1.78. In this case, cutting the consensus nodes in HYPER provides less improvement over GREEDY. On the other hand, in Fig. 2(c), when $\lambda$ increases, each subproblem has more variables, $|S|$ decreases, and replication factor increases. HYPER tends to be less sensitive to $\lambda$ than GREEDY.

In summary, our proposed hypergraph-based vertex-cut scheme outperforms the GREEDY scheme provided in the state-of-the-art GraphLab implementation [7], [15] for realistic bipartite graph settings. Especially when the two types of nodes in the bipartite graph are imbalanced, which is typically the case in large-scale consensus optimization, HYPER can generate much higher quality partitions.

### B. Performance of ACO for PSL Voter Model

Next We evaluate the performance of our proposed ACO algorithm on an MPI 2 (Open-MPI 1.4.3) cluster consisting of eight Intel Core2 Quad CPU 2.66GHz machines with 4GB RAM running Ubuntu 12.04 Linux. We implemented our algorithm using GraphLab 2 (v2.1.4245) [15]. For each machine in the cluster, we start one process with 4 threads (ncpus), and we use the synchronous engine which is explained in detail in [7]. Our proposed approach can be applied to other vertex programming frameworks easily as it does not use any GraphLab features beyond the synchronous GAS API.

*1) Voter Network Dataset Description:* We generate social voter networks using the synthetic generator in [5] and create a probabilistic model using the PSL program in Section II. The details of the datasets are listed in Table I. The smallest, $SN_{1M}$, fits in 4GB memory on a single machine when loaded in GraphLab; the rest of the datasets do not. In the voter PSL model, the variables corresponding to the truth of the VOTES(person,party) predicate are consensus variables, and each initialized rule maps to a subproblem. Each VOTES(person, party) appears in at most eight rules. In practice, PSL programs can be far more complex and many more subproblems can be grounded, thus the proportion may be even larger. In such cases, ACO-HYPER partitioning will even further reduce communication cost.

*2) Performance Results with PSL Inference:* We use a GraphLab vertex program that implements ACO algorithm described in Section IV and vary the partitioning schemes. We consider performance of ACO under two settings: full convergence and early stopping when one considers computation time budgets. It is important to consider the early-stopped setting since ACO is known to have very fast initial convergence and then slow convergence toward the final optimum [1]. In practice, one can stop early when the majority of variables have converged and quickly obtain a high-quality approximate solution. As shown in Fig. 3,

| Name | $\|S\|$ | $\|C\|$ | $\|E\|$ | $\|S\|/\|C\|$ |
|---|---|---|---|---|
| $SN_{1M}$ | 3,307,971 | 1,102,498 | 6,011,257 | 3.00 |
| $SN_{2M}$ | 6,656,775 | 2,101,072 | 12,107,131 | 3.17 |
| $SN_{3M}$ | 9,962,627 | 3,149,103 | 18,113,119 | 3.16 |
| $SN_{4M}$ | 13,349,751 | 4,203,703 | 24,288,223 | 3.18 |

Table I
SUMMARY OF SOCIAL NETWORK DATA SET FOR VOTER MODEL

inference in the PSL voter model converges on 99% of the consensus variables with 1,000 iterations on all datasets.

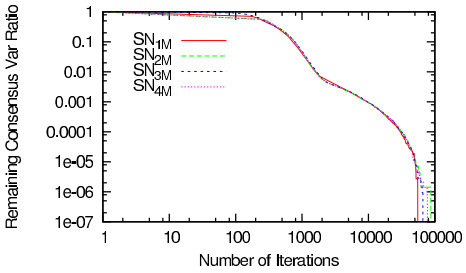



Figure 3. Convergence Rate in PSL Voter Model

**Full convergence:** As shown in Fig. 2(d) and 2(e), we first vary $m$ to show the running time and speedup under the full convergence setting. Because $SN_{1M}$ is able to fit on a single machine, communication cost overwhelms extra computation resources, and prevents distributed computation from performing better than single machine. In Fig. 2(d), ACO-GREEDY performs (2-4×) worse than the single machine setting, while ACO-HYPER has similar running with a single machine and is 2× better than ACO-GREEDY.

On larger data sets, our approach is approximately twice as fast as ACO-GREEDY (Fig. 2(e) and Fig. 2(f)). Note for the full convergence because fewer than 1% of the consensus variables are still active after 1,000 iterations, increasing the number of machines will not produce speedup in terms of computation time. In Fig. 2(f), we evaluate weak scaling, both schemes scale well on larger datasets.

**Early stopping:** Since modern computing models often include a pay-as-you-go cost, one may not benefit from waiting for the last few variables to converge. For instance, the last 1% of vertices in $SN_{2M}$ take $2/3$ of the total time for full convergence. Motivated by this, we measure the running time to complete 1,000 iterations of ACO-HYPER and ACO-GREEDY, regardless of the convergence status.

In Fig. 2(g), we show the accumulated running time of each iteration. Note because we use synchronous setting, both algorithms have the same state at the end of each iteration. ACO-HYPER performs 2-4× better than the ACO-GREEDY because of the reduced communication cost. In Fig. 2(h), we vary $m$ for $SN_{2M}$ to show the speed up. We show better returns when increasing $m$ than full convergence, and ACO-HYPER always outperforms ACO-GREEDY.

## VII. CONCLUSION

In this paper, we introduce a vertex programming algorithm for distributed ADMM-based consensus optimization. To mitigate the communication overhead of distributed computation, we provide a novel partitioning strategy that converts the ADMM bipartite computation graph into a hypergraph and uses a well-studied hypergraph cut algorithm to assign vertices to machines. Our experiments on probabilistic inference over large-scale, synthetic social networks demonstrate that our contributions lead to a significant improvement in performance. The partitioning scheme is of independent interest to practitioners in vertex programming.

## VIII. ACKNOWLEDGEMENTS

This work was supported by NSF grants IIS0746930, CCF0937094 and IIS1218488, and IARPA via DoI/NBC contract number D12PC00337. The U.S. Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright annotation thereon. Disclaimer: The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the offcial policies or endorsements, either expressed or implied of IARPA, DoI/NBC, or the U.S. Government.

## REFERENCES

[1] S. Boyd, N. Parikh, E. Chu, B. Peleato, and J. Eckstein, "Distributed optimization and statistical learning via the alternating direction method of multipliers," *Foundations and Trends in Machine Learning*, vol. 3, no. 1, 2011.

[2] N. Komodakis, N. Paragios, and G. Tziritas, "MRF optimization via dual decomposition: Message-passing revisited," in *IEEE Intl Conf. on Computer Vision (ICCV)*, 2007.

[3] S. Bach, M. Broecheler, L. Getoor, and D. O'Leary, "Scaling MPE inference for constrained continuous Markov random fields with consensus optimization," in *NIPS*, 2012.

[4] P. Ferero and A. Cano, "Consensus-based distributed support vector machines," *The Journal of Machine Learning Research*, vol. 99, pp. 1663–1707, 2010.

[5] S. Bach, B. Huang, B. London, and L. Getoor, "Hinge-loss Markov random fields: Convex inference for structured prediction," in *Uncertainty in Artificial Intelligence*, 2013.

[6] G. Malewicz, M. Austern, A. Bik, J. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: A system for large-scale graph processing," in *ACM SIGMOD*, 2010.

[7] J. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "PowerGraph: Distributed graph-parallel computation on natural graphs," in *USENIX Conference on Operating Systems Design and Implementation*, 2012.

[8] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. Hellerstein, "Distributed GraphLab: A framework for machine learning and data mining in the cloud," *Proceedings of the VLDB Endowment*, vol. 5, no. 8, pp. 716–727, 2012.

[9] A. Kimmig, S. Bach, M. Broecheler, B. Huang, and L. Getoor, "A short introduction to probabilistic soft logic," in *NIPS Workshop on Probabilistic Programming*, 2012.

[10] U. V. Catalyurek and C. Aykanat, "Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication," *IEEE Trans. Parallel and Distributed Systems*, vol. 10, no. 7, pp. 673–693, 1999.

[11] G. Karypis and V. Kumar, "Multilevel k-way hypergraph partitioning," in *ACM/IEEE Design Automation Conf.*, 1999.

[12] B. Hendrickson and T. G. Kolda, "Graph partitioning models for parallel computing," *Parallel computing*, vol. 26, no. 12, pp. 1519–1534, 2000.

[13] "hMetis v2.0pre1," http://glaros.dtc.umn.edu/gkhome/fetch/sw/hmetis/hmetis-2.0pre1.tar.gz, May 2007.

[14] H. Miao, X. Liu, B. Huang, and L. Getoor, "A hypergraph-partitioned vertex programming approach for large-scale consensus optimization," University of Maryland College Park, Tech. Rep., 2013, http://arxiv.org/abs/1308.6823.

[15] "GraphLab 2 v2.1.14245," https://graphlabapi.googlecode.com/files/graphlabapi_v2.1.4245.tar.gz, August 2012.