Efficient Distributed Gibbs Sampling for Factor Graphs

Zhou Qin [1] **Chris De Sa** [1]

Abstract

Probabilistic inference on factor graphs is a popular tool in formulating many machine learning tasks. Most inference engines use Gibbs sampling to avoid intractable integrals on calculating posterior probability.

In this paper, we present SPARTICUS, an efficient implementation for distributed Gibbs sampling algorithms for different types of factor graphs partitioned using "semantic partitioning" techniques. For a star-like graph, our implementation achieves near-linear strong parallel scaling (up to 6.8X on a 8 workers cluster on AWS). For a more general bipartite-like graph, our implementation focus on how to decrease message exchange while also achieves nearly linear parallel scaling (up to 4.5X on 8 workers). Also, our algorithm never manifests whole graphical model during computation, it is memory friendly and able to scale gracefully up to arbitrarily large datasets.

1. Introduction

The goal of Bayesian inference is to maintain a full posterior probability distribution over a set of random variables. However, computation for posterior probability often involves integrals which in most case intractable. Thus most existing probabilistic engine use sampling methods to perform learning and inference efficiently. the most commonly used sampling techniques is Gibbs sampling, a lightweight Markov Chain Monte Carlo sampling technique without any user-tunable parameters.

As datasets grow in size, many of these sampling tasks become too large to fit in RAM on a single machine, thus distributed Gibbs sampling techniques have drawn growing interests in recent years. Distributed Gibbs sampling method generally need to partitioning a factor graph to different worker machines. Much recent work(Neiswanger et al., 2013; Srivastava et al., 2015; Scott et al., 2016) takes a *divide-and-conquer* approach, in which graph is broken into small pieces, with MCMC approach performed parallel in different workers, and the result averaged together to produce samples. This approach "solve" the problem by *not communicating* on workers at all, thus the sample is biased. Alexander etc.(2015) propose *asynchronous Gibbs sampling*, which enable communicates while removes synchronization barriers, instead of sampling all the variables conditional on the most recent values, each worker is sampling conditional on the most recent values that it knows about. They proved the convergence of this approach in the paper, while mixing time is undetermined. The analysis of Mixing time and bias for asynchronous Gibbs is latter detailed illustrated in (De Sa et al., 2016).

To make network communication between workers as less as possible, the most ideal partition strategy is to group related variables into same machine, in *divide-and-conquer* approach mentioned above, a partition algorithm such as connected components discovery is necessary, which itself might be highly time consuming. Chris(2017) propose a novel partition method called "semantic partitioning", which assign variables in a factor graph to machines based on their meaning. They do this using a type-based analysis of a high-level description of a factor graph, known as a factor graph template(Koller & Friedman, 2009). Using this partition strategy, we are able to achieve accurate samples without communication between workers in distribution settings.

The main contribution for this paper are

- We describe SPARTICUS[1], an efficient implementation of distributed Gibbs sampling for both inference and learning without ever manifesting the entire model on a single machine.
- We analysis in great details about two types of graph `BDC` and `BFD`, describes how to modify SPARTICUS, making it efficient in time and space consumption.

[1]Department of Compute Science, Cornell University. Correspondence to: Zhou Qin <zq32@cornell.edu>.

Proceedings of the 34^{th} International Conference on Machine Learning, Sydney, Australia, PMLR 70, 2017.

[1]SPARTICUS: **S**emantic **PART**itioning for **I**nference on **C**lusters **U**sing **S**ampling

2. Structure

In Section 3, we will briefly describe the "Semantic Partitioning" techniques and its motivation. In Section 4, we detailed the meaning of different partitioning classes, and steps for SPARTICUS algorithm. Following by Section 5, in which we provide two different types of graphs, `BDC` and `BFD`, illustrate how to modify the SPARTICUS algorithm, making it efficient in terms of time and space consumption. Lastly, we conclude the paper, showing possible following up directions in Section 6 and Section 7.

3. Semantic Partitioning

The key idea of Semantic Partitioning is taking into account higher-level information about the structure of the graphical model specified by a user. Most probabilistic engines allow users to specify probabilistic graphical models declaratively by defining a series of *factor graph templates*. Given an input database of entities these templates are used to materialize the underneath factor graph. Semantic partitioning combines the domain knowledge in factor templates with entity statistics from the input database to obtain an intelligent partitioning of the grounded factor graph.

This technique first analyzes how variables are instantiated from the template in order to extract the semantics of the variable. then partitioning variable with similar or related meanings onto same worker. For example, suppose we are interested in a graphical model whose variables are facts about sentence such as which their topics are, In this settings, a bool variable that represents the statement "Obama is the president" could be co-located with a bool variable representing the statement "Obama lives in the White House" since both of these are statement about the same entity, so these two variables are placed to the same worker machine.

3.1. Type-based Partitioning

The idea of semantic partitioning can be implement in different ways, we will introduce a simple but one called *Type-based Partitioning*.

Considering such an example, suppose we have a variable template `DocAbout(DocId, TopicId)` indicating whether a document is about a topic. And we also have a variable template `IsPopular(TopicId)` indicating whether a topis popular. The factor template can be `DocAbout(DocId, TopicId)` $\rightarrow$ `IsPopular(TopicId)`. In this example, we can separate the graph by type *TopicId*, making the variable contains same TopicId to be assigned to the same worker machine. As you can imagine, during the sampling process, workers are totally independent with each other, so they can perform sampling simultaneously.

Table 1. Meaning of different partition class

Partition class	Applies to	Owned by	Cached on
A	variables, factors, weights	Master	—
B	variables	Master	All Workers
C_T	variables, factors, weights	Worker$_T$	—
D_T	variables, factors	Worker$_T$	Master
E_T	factors	Master	Worker$_T$
F_T	factors, weights	Master, Worker$_T$	—
G_T	factors	—	Master, Worker$_T$

4. Sampling

4.1. Partioned Grounding

Once a type-based partitioning has been constructed, we are given, for each template variable, template factor and template weight, a particular *partition class*. partition class means how we can assign variables, factors and weight to different machines.

For template variables, the partition class determines

- The **unique** machine owns (i.e. samples) the variable;
- The machines that have cached (for temporary usage) the variable;

For template factors, the partition class determines

- On which machines the factor is fully manifested;
- On which machines the factor is partially manifested (with partial cached assignment);

For template weights, the partition class determines simply which machines own the weight, there is no notion of full versus partial manifestation for weights. Table 1 shows different types of partition classes we define, for more details of each class, check (De Sa, 2016).

Back to the example we mentioned in Section 3.1, we partition the graph in this way, all the variables `IsPopular(TopicId)` belongs to class B. we partition variables `DocAbout(DocId, TopicId)` using partition class $C_{TopicId}$, and we partition factors `DocAbout(...)`$\rightarrow$`IsPopular(...)` using partition class $D_{TopicId}$. This partition forms a star-like distributed graph, we will talks about how to design algorithm to efficient sampling from this latter in Section 5.1.

4.2. Sampling algorithm

With these partitioning classes, we can assign variables, factors, and weights to different machines. Thus we can design an distributed Gibbs sampling framework shows in Algorithm 1.

Algorithm 1: SPARTICUS algorithm

1 **while** *Not mix* **do**
2 Master broadcasts its assignment of all $\boldsymbol{B}$-key variables to the workers
3 Master computes partial evaluation of all $\boldsymbol{E_i}$-key and $\boldsymbol{G_i}$-key factors, and transmits them to worker i
4 Workers receive all transmissions from master, and update their cached $\boldsymbol{B}, \boldsymbol{E_i}, \boldsymbol{G_i}$ things.
5 Each worker runs a single pass of Gibbs sampling on the variable it owns.
6 Each worker transmits its assignment of all $\boldsymbol{D_i}$-key variables to the master
7 Each worker computes partial evaluation of all $\boldsymbol{D_i}, \boldsymbol{G_i}$-key factors, transmits them to the master.
8 Master receives all transmissions from worker, and updates its cached assignments of $\boldsymbol{D_i} - key$ and $\boldsymbol{G_i} - key$ things
9 Master then runs a asingle pass of Gibbs sampling on the variable it owns.
10 Output current state of all variables.

The main body contains three parts. First, master send evaluation of variables and factors to worker; Second, workers receive transmissions from master and perform Gibbs sampling on variables it own, sending the variables and factors back to the master; Third, master receive transmissions from workers, perform Gibbs sampling on variables it own.

clearly there are some "concurrency barriers" in this algorithm, master has to waiting all the transmissions from workers to perform Gibbs sampling, thus decrease the performance of the system, we will discuss some possible solutions to handle this problem in Section 6, but how to solve it is not a main topic of this paper.

5. Case study

After separation, the graph might contains different types of sub factor graphs. In this section, we will takes two types of this sub factor graphs, illustrate how to specify SPARTICUS, making it efficient on different types of graphs.

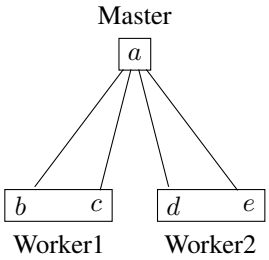

Figure 1. An example of BDC graph. The variable a belongs to partition class B, variables b, c belongs to partition class C_1, variables d, e belongs to partition class C_2, factors $\phi(a, b), \phi(a, c)$ belongs to partition class D_1, factor $\phi(a, d), \phi(a, e)$ belongs to partition class D_2, factors are not shown in the graph. This type of graph is called BDC graph according to its partition classes

5.1. BDC Graph

BDC graph is a type of factor graph which contains partition class B, C, D, since nodes assigned to Master is generally much less than to Workers, so it is a star-like graph. Figure 1 provides an simple example of this type of graph. Variable a belongs to partition class B, owned by master; Variables b, c belong to partition class C_1, factors $\phi(a, b), \phi(a, c)$ belong to D_1, they are owned by Worker1; Variables d, e belong to partition class C_2, factors $\phi(a, d), \phi(a, e)$ belong to partition class D_2, they are owned by Worker2.

Although the structure of this graph is simple, the BDC graph is actually pretty common in production environment, as you can check the example we made in Section 3.1 is an instance of BDC type graph. One of the advantage of this graph is that Worker$_T$ owns variables and all factors related to these variables, thus they are independent with each other, so it is well suitable for distributed computing.

5.1.1. Algorithm

In this section, we will specify SPARTICUS algorithm for BDC type of graph, to see how we make Gibbs sampling efficient. The REDUCTION in Line 5 is used to minimize transmission cost from workers to master, the transmission is generally through the Network, thus it is important to make it as less as possible. We will use the graph shown in Figure 1 to illustrate how we perform the REDUCTION.

When runing Line 5, suppose the sample value for variable b, c on Worker1 is b_s, c_s, and the sample value for variable d, e on Worker2 is d_s, e_s, then we need to sample a on Master. Because Master own on factors, so we need to calculate the (partial) factor value for a on Workers. More specifically, Based on $b_s, \phi(a, b)$, we calculate the factor value for all the possible values that a can take, if a is binary, we get the factor value for $a = 0$ and $a = 1$, suppose they

Algorithm 2: SPARTICUS for `BDC` graph

1 **while** *Not mix* **do**
2 Master broadcasts its assignment of all **B-key** variable to the workers
3 Workers receive all transmissions from master, and update their cached **B-key** variables.
4 Each worker runs a single pass of Gibbs sampling on the **C-key** variable it owns.
5 Each worker computes partial evaluation of all **D-key** factors, perform a REDUCTION, then transmits them to the master.
6 Master receives all transmissions from worker, and updates its cached assignment of **D-key** things if necessary.
7 Master perform a REDUCTION for partial evaluation of factors again, then runs a asingle pass of Gibbs sampling on the **B-key** variable it owns.
8 Output current state of all variables.

are ϕ_b^0, ϕ_b^1 respectively. Similarly, Based on $c_s, \phi(a, c)$, we get ϕ_c^0, ϕ_b^1; On Worker2, we get $\phi_d^0, \phi_d^1, \phi_e^0, \phi_e^1$. We can transform all these values to Master to sample a, however, as mentioned before, the transmission through network is quite expensive, thus we need to figure out a way to reduce it as much as possible.

Luckily, the Master calculation the probability of a take value v in such way:

$$p(v) = exp(\sum_{\phi^v \in \Phi^v} w_{\phi^v} \cdot \phi^v) \tag{1}$$

where w_ϕ is the weight for each factor ϕ. Φ^v is the set of all related ϕs in which variable takes value v. For example, if $v = 0$, then $\Phi^v = \{\phi_b^0, \phi_c^0, \phi_d^0 \phi_e^0\}$, the master calculated all possible normalized $p(v)$, namely $p(0), p(1)$, then sample variable based on that. As we can see from Equation (1), what is important for Master is just the weighted summation of factors on each Worker. So we perform a `reduction` here by taking weighted summation on each Worker.

When all the partial factor value transmitted to Master, we do a REDUCTION again on the Master node as shown in Line 7, in our example, to calculate $p(0)$, we receive partial sum $s_1 = w_{\phi_b^0} \cdot \phi_b^0 + w_{\phi_c^0} \cdot \phi_c^0$ from Worker1, and partial sum $s_2 = w_{\phi_d^0} \cdot \phi_d^0 + w_{\phi_e^0} \cdot \phi_e^0$ from Worker2, we sum up s_1 and s_2, calculate `exp` to this sum, as shown in Equation (1), to get $p(0)$. We do the something to calculate $p(1)$, and then Master sample a based on $p(0), p(1)$, finishing one loop of the algorithm.

Table 2. Detailed configuration of experiments runs for `BDC` graph, # means number of, each experiment has only one variable on the Master machine, and each experiment samples 1000 times for each variable. Last two columns are the running time for each experiment, Total job time was the accumulate computation time while Wall time was the actual time algorithm takes

Index	#Workers	#Variable on each Worker	Total job time	Wall time
1	1	4800	290	306
2	2	2400	348	188
3	4	1200	466	130
4	6	800	590	113
5	8	600	717	105

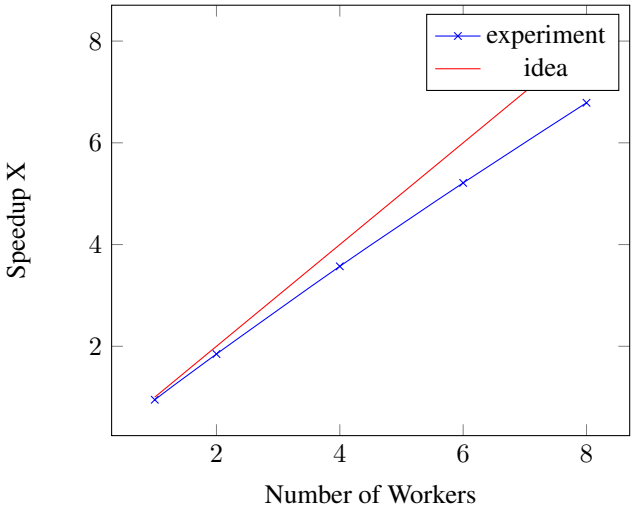

Figure 2. Speedup ratio (Wall time divided by Total job time) plot of SPARTICUS on `BDC` graph

5.1.2. EXPERIMENTS

We perform our experiment on **AWS**, with it's `t2.micro` instance, this type of machine has 1 (virtual) cpu, 1 GB memory, settled in the normal network environment.

We perform 5 comparison experiments with different number of workers, all of them have only one variable at master machine, total variable number of workers is 4800. Depends on how many workers in each experiment, we separate these variable evenly on each worker. We perform 1000 rounds sampling for each variable, so in each experiment, we actually sample $1000 \times 4800 = 4.8M$ variables. Table 2 shows the detailed configuration for each experiment. The speed up ratio plotted in Figure 2 shows that our implementation achieves near-linear strong parallel scaling.

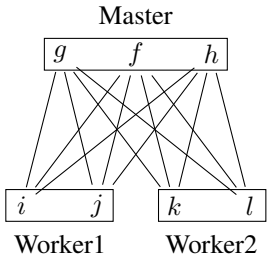

Figure 3. An example of BFD graph. Variables g, f, h belong to partition class B, variables i, j belong to partition class D_1, variables k, l belong to partition class D_2, factors $\phi(i, \cdot), \phi(j, \cdot)$ belong to partition class F_1, factors $\phi(k, \cdot), \phi(l, \cdot)$ belong to partition class F_2, "$\cdot$" here means any variable from Master (i.e. g, f, h). This type of graph is called BFD graph according to its partition classes.

Algorithm 3: SPARTICUS for BFD graph

while *Not mix* **do**
- Master broadcasts its assignment of all **B-key** variable to the workers.
- Workers receive all transmissions from master, and update their cached assignments of **B-key** variables.
- Each Worker runs a single pass of Gibbs sampling on the variable it owns.
- Each worker transmits its assignment to all $\mathbf{D}_i$**-key** variables to the master.
- Master receive all transmissions from workers, and updates its cached assignment of $\mathbf{D}_i$ variables.
- Master runs a single pass of Gibbs sampling on the variables it owns.
- Output current state of all variables.

5.2. BFD Graph

BFD graph is a type of factor graph which uses partition class B, F, D, it is a fully connected bipartite graph whose both sides contains multiple variables. Figure 3 provides an simple example of this type of graph. Variables g, f, h belong to partition class B, owned by Master. Variables i, j belong to partition class D_1, owned by Worker1. Variables k, l belong to partition class D_2, owned by Worker2. Factors $\phi(i, \cdot), \phi(j, \cdot)$ belong to partition class F_1 owned by Worker1 and Master. Factors $\phi(k, \cdot), \phi(l, \cdot)$ belong to partition class F_2 owned by Worker2 and Master. "$\cdot$" here means any variable from Master (i.e. g, f, h). Following section we will describe specific SPARTICUS algorithm for BFD type of graph.

5.2.1. Algorithm

The key idea to perform efficient sampling in the distributed environment is making transmissions as less as possible, since there can be $O(n^2)$ edges in the graph, transmitting partial factor values among Master and Workers is much expensive than BDC type graph, thus we cache factors on Master and Workers, sampling variables independently on Master and Workers. See Algorithm 3 for details.

The algorithm is simple and self explanatory. Using the Figure 3 as an example, in one round of Gibbs sampling, first Master transmit all assignment of g, f, h to Workers, because Worker1 has all factors related to the variables it owns, it runs Gibbs sampling for variables i, j immediately. Simultaneously, Worker2 run Gibbs sampling for variables k, l. then Workers transmit the assignment of i, j, k, l to Master, Master also has all factors, it then runs Gibbs sampling for variables g, f, h.

Table 3. Detailed configuration of experiments runs for BFD graph, # means number of. The number of variables on Master is one percent of number of variables on each worker. and each experiment samples 100 times for each variable. Last two columns are the running time for each experiment, Total job time was the accumulate computation time while Wall time was the actual time algorithm takes

Idx	#Workers/ #Variable on each Worker	#Variable on Master	Total job time	Wall time
1	1/4800	48	34	67
2	2/2400	24	35	48
3	4/1200	12	37	23
4	6/ 800	8	45	15
5	8/ 600	6	55	12

5.2.2. Experiments

We perform the experiments on the same machines we described above, 5 comparsion experiments were conducted, each with different number of workers, number of variables on master, number of variable on worker. Total variable number of workers is 4800, they are separated evenly on each worker. We perform 100 rounds sampling for each variable, thus we sample $100 \times 4800 = 0.48M$ variables. Table 3 shows the detailed configuration for each experiment. As Figure 4 shows, our implementation for BFD type graph cannot achieve a strong parallel scaling as BDC does. That's mainly depends on the fact of BFD architecture, one reason is the "concurrency barriers", master has to waiting all the transmissions from workers to perform Gibbs sampling. More importantly, The amount of calculation on Master is nearly equals to the total amount of calculations

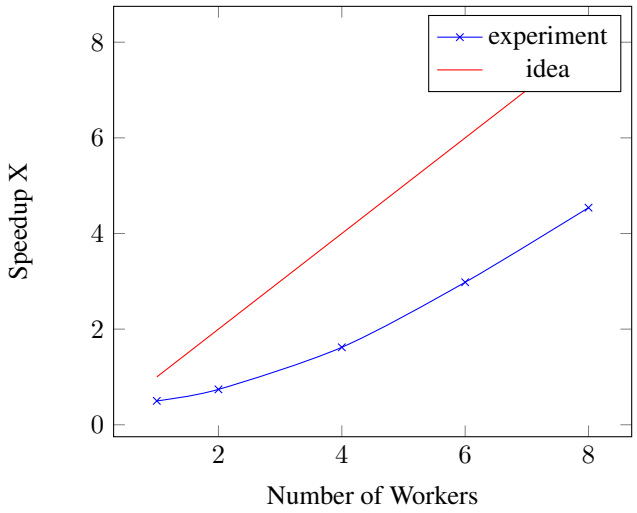

Figure 4. Speedup ratio (Wall time divided by Total job time) plot of SPARTICUS on BFD graph

on all Workers, thus when Workers finish their computation, Master still working on that, the Workers will be idle until Master finish its computation.

But it does not mean BFD is unnecessary in our implementation, we will analysis its space advantage in Section 5.3.2.

5.3. Why efficient

In the description above, we keep emphasizing that the algorithm is efficient for different types of graphs, the efficiency reflect in two aspects, time and space consumption. In this section, we will discuss why it's efficient, why we need so many partition classes. We still use the example graph shows in Figures 1 and 3.

5.3.1. Turn BDC into BFD?

Clearly, BDC graph is a special case of BFD graph where Master machine only contains very less variables. How about we use Algorithm 3 prepared for BFD graph to sample variables on BDC graph? It still generates correct result and we will compare them on the amount of message transmitted and space consumption.

The analysis is based on the left graph of Figure 5, we show what message need to be transmitted for both algorithm on Table 4. As you can see, messages transmitted on this graph is same for two algorithms.

However, when we consider the space consumption, notice for Algorithm 3, there are two copys of factors need to be cached on our system, one for Master, the other for Workers. While for Algorithm 2, only one copy of factors need to be stored on our system, they are stored on Workers. So

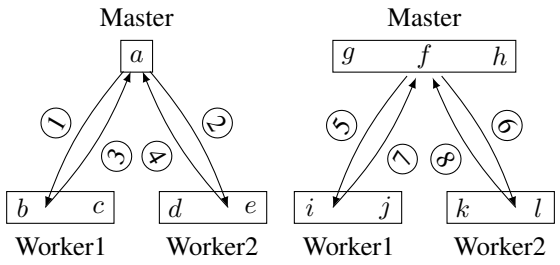

Figure 5. Message transmission of BDC (left) and BFD (right) graph

Table 4. Detailed transmission needed by different algorithms on left BDC graph of Figure 5, v:v_s is the key-value pair transmitted in our protocol, v is variable name, v_s is the assignment for v at this round. For BDC graph, our algorithm need to do some reductions, v^0 means the partial factor value when $v = 0$, we use _ to represent this unknown partial factor value.

	Algorithm 2 design for BDC	Algorithm 3 design for BFD
① ②	a:a_s	a:a_s
③	a^0:_, a^1:_	b:b_s, c:c_s
④	a^0:_, a^1:_	d:d_s, e:e_s

in terms of space consumption, Algorithm 2 is much better, that's why we want to use it for BDC graph.

Due to our partition class criteria, it is the ground truth that BDC consumes less space than BFD, how about we deal with all BFD graph with Algorithm 2 designed for BDC? We will analyze it in the following section.

5.3.2. Turn BFD into BDC?

Although In BFD graph, Master contains many variables, it is still possible to sampling them using Algorithm 2 prepared for BDC graph. We can send reduced factor value for each variables on Master. But it is not a good choice because it transmit too many messages. In practice, we might have network flow constraints, and too many transmissions will also slow down the system in network environment.

For example, consider the right graph of Figure 5, we show what message need to be transmitted for both algorithm in Table 5. As it clearly shows, using Algorithm 3, messages transmitted on this graph is much less than Algorithm 2, even if we already do some reduction works.

Algorithm 2, designed for BDC, will transmit more and more messages when the number of variables on the Master increases. In contrast, we can always control the message transmitted by scaling Workers for Algorithm 3, that's why we want to use it for BFD.

Table 5. Detailed transmission needed by different algorithms on right BFD graph of Figure 5. v:v_s is the key-value pair transmitted in our protocol, v is variable name, v_s is the assignment for v at this round. For BDC graph, our algorithm need to do some reductions, v^0 means the partial factor value when $v = 0$, we use _ to represent this unknown partial factor value.

	Algorithm 2 design for BDC	Algorithm 3 design for BFD
⑤ ⑥	$g : g_s, f : f_s, h : h_s$	$g : g_s, f : f_s, h : h_s$
⑦	g^0:_, g^1:_ f^0:_, f^1:_ h^0:_, h^1:_	$i : i_s, j : j_s$
⑧	g^0:_, g^1:_ f^0:_, f^1:_ h^0:_, h^1:_	$k : k_s, l : l_s$

6. Discussion and Follow up works

We have described the weakness of our implementation for sampling BFD graph, "concurrency barriers" and "idle workers" problem. One possible solution is to use asynchronous Gibbs sampling described in (Terenin et al., 2015; De Sa et al., 2016), but in the general case, The SPARTICUS algorithm works simultaneously on different graphs, when one worker finish the sample work of BFD graph, it might turn into another sampling work from BDC graph right away, so "idle workers" may not be a problem in production environment.

For the time constraints, I did not perform any comparison of SPARTICUS and other parallel MCMC methods, but as far as I know, the DimmWitted Gibbs Sampler (Zhang & Ré, 2014) had reached a sampling speed of $3.6M/s$ on single machine using multi-thread techniques. Multithread combined with distribution is a promising direction for the following work.

7. Conclusion

In this paper, we focus on efficient implementation for Gibbs sampling on graph models, we propose SPARTICUS, a distributed Gibbs sampling framework for different type of graphs. We utilize semantic partitioning to separate the whole graph, the advantage of this partitioning techniques is that all the variables on different workers are independent with each other, thus our algorithm is inherently different from previous ones, the samples we get are exactly same as samples in sequential ways.

We use two study case, BDC and BFD graph to illustrate SPARTICUS can be specified to eliminate both time and space consumption for different types of graph. For BDC type of graph, which is much commonly used in production environment, our implementation achieves near-linear strong parallel scaling; For BFD type of graph, we change the implementation to decrease the amount of message transferred between Master and Workers while keep it still scalable in distributed settings.

All the related code for this paper can be check from my github repository: https://github.com/qzshadow/dispy_gibbs.git

References

Aberger, Christopher. Distributed inference for graphical models using semantic partitioning. Stanford, CA, 2017.

De Sa, Christopher. Partitioned grounding: How to ground and sampling using a type-based partition. Technical report, Computer Science Department, Stanford University, 2016.

De Sa, Christopher, Olukotun, Kunle, and Ré, Christopher. Ensuring rapid mixing and low bias for asynchronous gibbs sampling. In *JMLR workshop and conference proceedings*, volume 48, pp. 1567. NIH Public Access, 2016.

Koller, Daphne and Friedman, Nir. *Probabilistic graphical models: principles and techniques*. MIT press, 2009.

Langley, P. Crafting papers on machine learning. In Langley, Pat (ed.), *Proceedings of the 17th International Conference on Machine Learning (ICML 2000)*, pp. 1207–1216, Stanford, CA, 2000. Morgan Kaufmann.

Neiswanger, Willie, Wang, Chong, and Xing, Eric. Asymptotically exact, embarrassingly parallel mcmc. *arXiv preprint arXiv:1311.4780*, 2013.

Scott, Steven L, Blocker, Alexander W, Bonassi, Fernando V, Chipman, Hugh A, George, Edward I, and McCulloch, Robert E. Bayes and big data: The consensus monte carlo algorithm. *International Journal of Management Science and Engineering Management*, 11(2): 78–88, 2016.

Srivastava, Sanvesh, Li, Cheng, and Dunson, David B. Scalable bayes via barycenter in wasserstein space. *arXiv preprint arXiv:1508.05880*, 2015.

Terenin, Alexander, Simpson, Daniel, and Draper, David. Asynchronous gibbs sampling. *arXiv preprint arXiv:1509.08999*, 2015.

Zhang, Ce and Ré, Christopher. Dimmwitted: A study of main-memory statistical analytics. *Proceedings of the VLDB Endowment*, 7(12):1283–1294, 2014.