Design Efficient Distributed Gibbs Sampler for Factor Graphs

May 18, 2018

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 design a highly efficient distributed Gibbs sampling framework based on "semantic partitioning" techniques. For a star-like graph, our implementation achieves near-linear strong parallel scaling in specific settings (up to 3.4X on a 4 workers cluster on AWS). Also, our design never manifests whole graphical model on one machine during computation, it is memory friendly and able to scale gracefully up to arbitrarily large datasets.

Contents

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 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. Many recent work[6, 8, 7] takes *divide-and-conquer* approach, in which graph is split into small pieces, with MCMC approach performed parallel on different partitions, 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.[9] propose *asynchronous Gibbs sampling*, which enables 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 [3].

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[1] propose a novel semantic partition method called "SPARTICUS"[1], 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[4]. Using this partition strategy, they are able to achieve accurate samples without communications between workers in distributing settings.

This paper mainly focusing on designing highly efficient Gibbs sampling framework on the graph partitioned by SPARTICUS.

2 Structure

In Section 3, we briefly describe the "SPARTICUS" techniques and its motivation. In Section 4, we detail the meaning of different partitioning classes, and specify our algorithm. Section 5 describes how we implement the algorithm in object oriented ways. Following by Section 6, in which we illustrate four different partition strategies, `BDC`, `BFD`, `AGC` and `AED`, performing experiments on each of them, discussing possible following up improvements. Lastly, we conclude the paper in Section 7.

change paper structure

3 Semantic Partitioning

The key idea of SPARTICUS is to take user specified higher-level information about the structure of the graphical model into consideration. 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.

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

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 topic popular. The factor template can be `DocAbout(DocId, TopicId)` $\to$ `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.

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 can be useful, for more details of each class, check [2].

Back to the example we mentioned in Section 3.1, here is one way to partition the graph, 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(...)`$\to$`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 6.1.

4.2 Distributed Sampling algorithm

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

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 owns, sending the variables and factors back to the master; Third, master receive transmissions from workers, perform Gibbs sampling on variables it owns.

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 6.6, how to solve it is not a main topic of this paper.

5 Framework hierarchy design

Section 4.2 illustrate how the distributed algorithm looks like, we implement this algorithm with `C++14`, using `MPI` from `Boost` library to transmit messages between master and worker machines. The whole framework was designed in an object oriented way. In this section we will provide detailed class hierarchy of the sampling framework, the source code locates on `https://github.com/qzshadow/Numbskull`.

Table 1: Meaning of different partition class

Partition class	Applies to	Owned by	Cached on	Description
A	variables, factors, weights	Master	—	**Master only** Components are distributed only only the master node.
B	variables	Master	All Workers	**Master with assignment on workers** Variables are owned by (written by) the master node, and also have a (read-only) cached assignment on each worker node.
C_T	variables, factors, weights	Worker$_T$	—	**Worker only** Components are distributed onto only a particular worker node, based on the type (or types) T.
D_T	variables, factors	Worker$_T$	Master	**Worker with assignment on master** Components are owned by a particular worker node, based on the type (or types) T. Variables also have a (read-only) cached assignment on the master node. Similarly, factors also have a (read-only) partially evaluated assignment on the master node.
E_T	factors	Master	Worker$_T$	**Master with assignment on worker** Factors are owned by (are fully represented on) the master node, and also have a (read-only) partially evaluated assignment on a particular worker node, based on the type (or types) T.
F_T	factors, weights	Master, Worker$_T$	—	**Master and worker** Components are shared by (and are fully represented on) the master node and a particular worker node, based on the type (or types) T.
G_T	factors	—	Master, Worker$_T$	**Master and worker** Factors are partially represented on both the master node and a particular worker node, based on the type (or types) T.

Algorithm 1: Distributed inference via synchronous Gibbs sampling algorithm

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

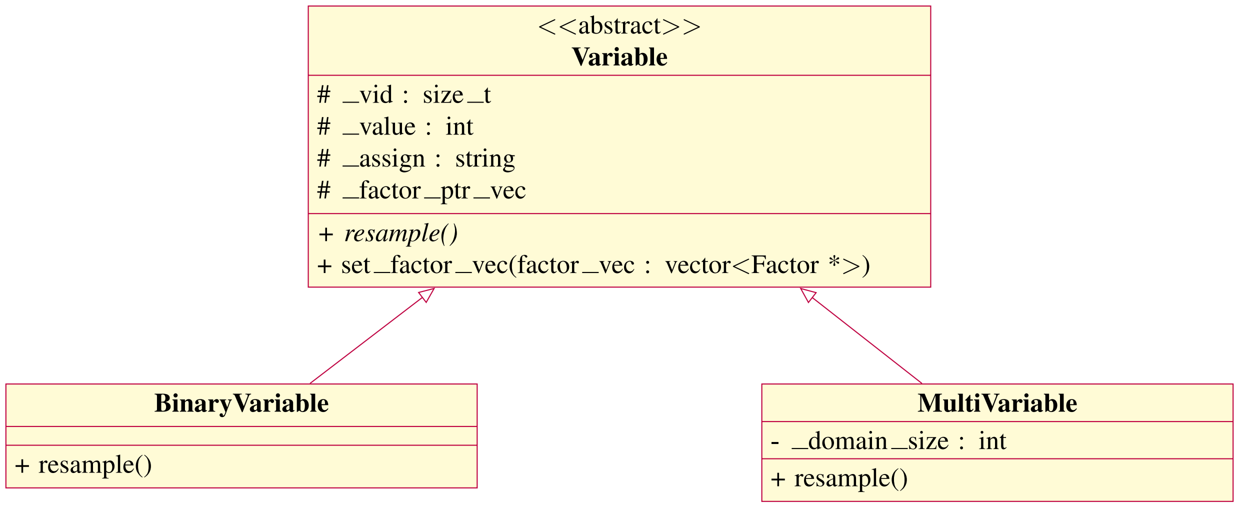


Figure 1: UML graph for Variable class hierarchy

5.1 Variable

For now, we support two types of Variables. `BinaryVariable` and `MultiVariable`, both of them inherit from abstract class `Variable`, `MultiVariable` class contains an extra field `_domain_size` to indicate how many values this variable can draw. Figure 1 shows the class hierarchy.

5.2 Factor

In the distributed settings, a variable in the master machine generally connections to other variables distributed on different machines. Factors are used to model these connections.Figure 2 shows the class hierarchy.

For example, suppose we have `var0` on master machine, `var1, var2` on worker1, `var3, var4` on worker2. new value for `var0` might be generate (sampled) using the formula

$$\texttt{var0}_{new} = \texttt{var0} \oplus_0 (\texttt{var1} \oplus_1 \texttt{var2}) \oplus_0 (\texttt{var3} \oplus_2 \texttt{var4}) \tag{1}$$

here $\oplus$ means `AndFactor` and subscript i means where the computation performed. On worker1, we have a `AndFactor` $\oplus_1$, which perform a `partial_eval` on `var1` and `var2`, generating a partial evaluation for $\texttt{var0}_{new}$, then send this partial evaluation to master. On the master, we have a `PartialAndFactor` to temporarily save this value, then a `resample` will be performed on `var0` to generate a new value based on these partial evaluations.

5.3 Edge

Edges perform a transformation for variable value before use it directly in Factor evaluation. The transformation might not be necessary, in this case, we will use an `IdentityEdge` to forward to variable value. Figure 3 shows the class hierarchy.

5.4 FactorGraph

`FactorGraph` class is responsible for build the partitioned subgraph on each machine, which will assign memories for variables, edges, and factors manifested on this machine and recycle memories when program ends. Figure 4 shows the class hierarchy. `gibbs` function will perform sampling for each variable on this machine. The logic of Algorithm 1 was implemented here.

5.5 Delegation sampling model

In our implementation, we use "delegation" design model to perform sampling. As you might have noticed in the class hierarchy, class `Variable` contains a protected field `vector<Factor *>`, which indicates all the factors

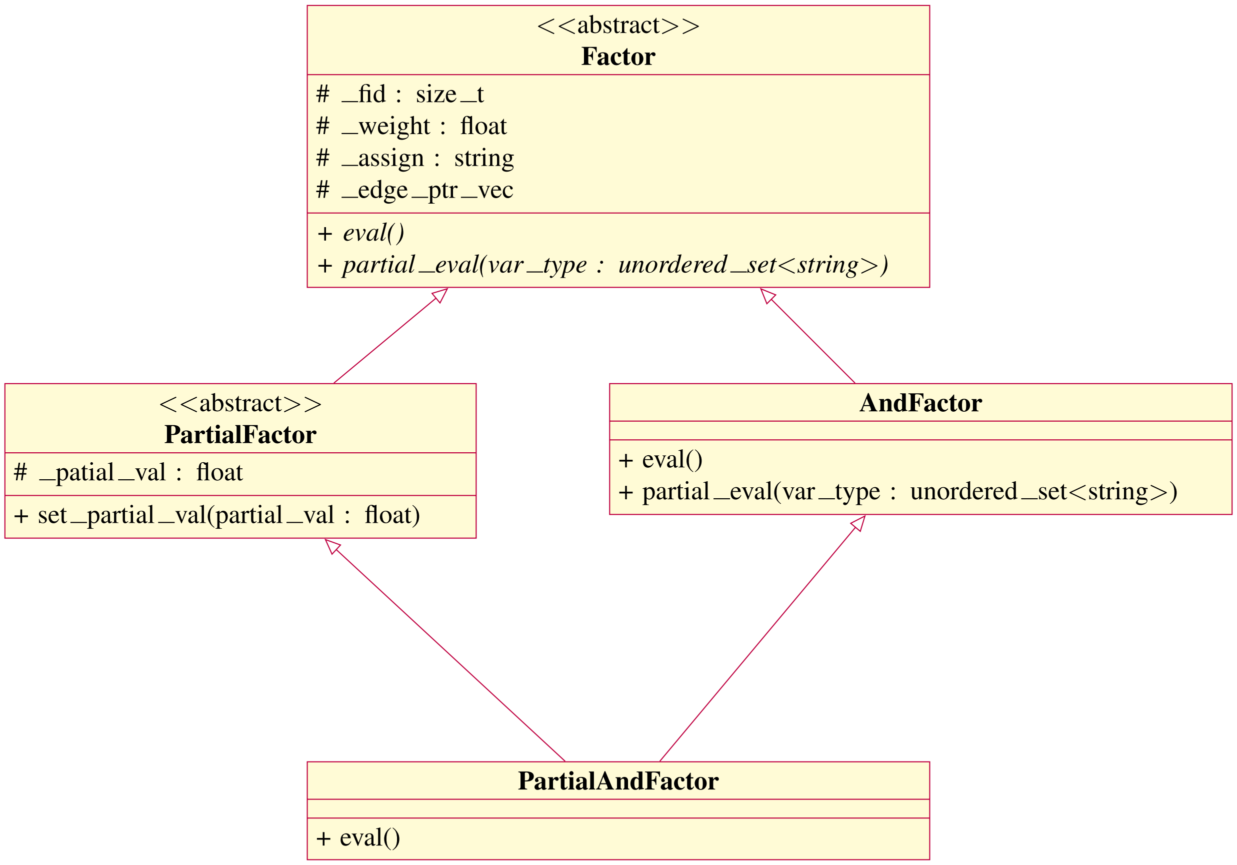


Figure 2: UML graph for Factor class hierarchy

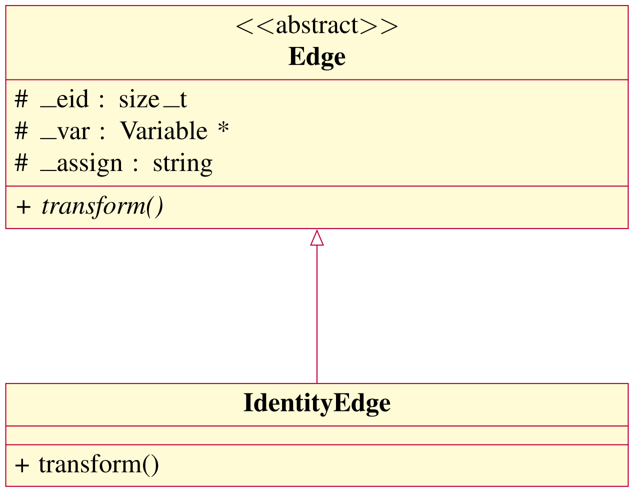
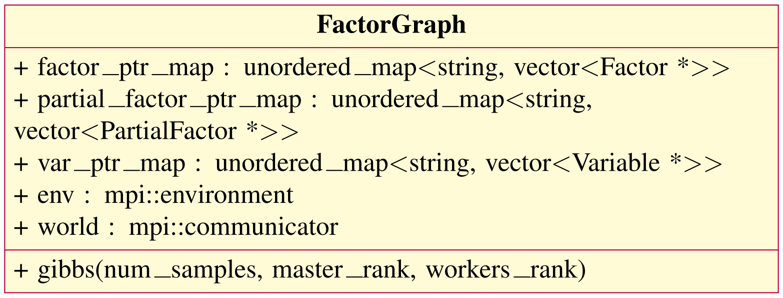


Figure 3: UML graph for Edge class hierarchy



FactorGraph

+ factor_ptr_map : unordered_map<string, vector<Factor *>>
+ partial_factor_ptr_map : unordered_map<string, vector<PartialFactor *>>
+ var_ptr_map : unordered_map<string, vector<Variable *>>
+ env : mpi::environment
+ world : mpi::communicator

+ gibbs(num_samples, master_rank, workers_rank)

Figure 4: UML graph for FactorGraph class hierarchy

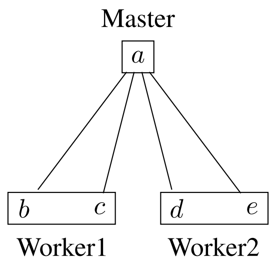


Figure 5: A commonly seen star-like graph.

this variable connects to; class `Factor` contains a protected field `vector<Edge *>`, which indicates all the edges this factor connects to; class `Edge` contains a protected field `Variable *`, which indicates the variable this edges corresponds to.

When we want to perform a resample for a variable, saying `Var0` in Equation (1), we need to get the current value of `var1, var2, var3, var4`. Distributed settings make the problem became nontrivial, because we can not send these variables directly to master machine, in which case all the computation will be performed on master.

In our framework, the delegation model works as following, for `var0`, it asks each `Factor` it connect to to send a partial evaluation, each `Factor` will ask each `Edge` it connect to to send a transformed value, each `Edge` will ask each `Variable` it connect to to send its most recent value. By doing partial evaluation of `Factor` on workers, computation works are dispersed to workers, thus increase the system efficiency.

6 Case study

Different partition strategies can be performed even on the same graph. In this section, we will take a commonly seen star-like graph (Figure 5) and partition it with different strategies, illustrating how to specify Algorithm 1 for each types of partitions.

6.1 BDC partition

For `BDC` partition, variables on master are assigned to B, variables on workers are assigned to C, factors connects these variables are assigned to D. See Figure 6 for detailed assignment.

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

6.1.1 Algorithm

Algorithm 2 specifies the algorithm to perform Gibbs sampling on this graph partitioned in `BDC` type.

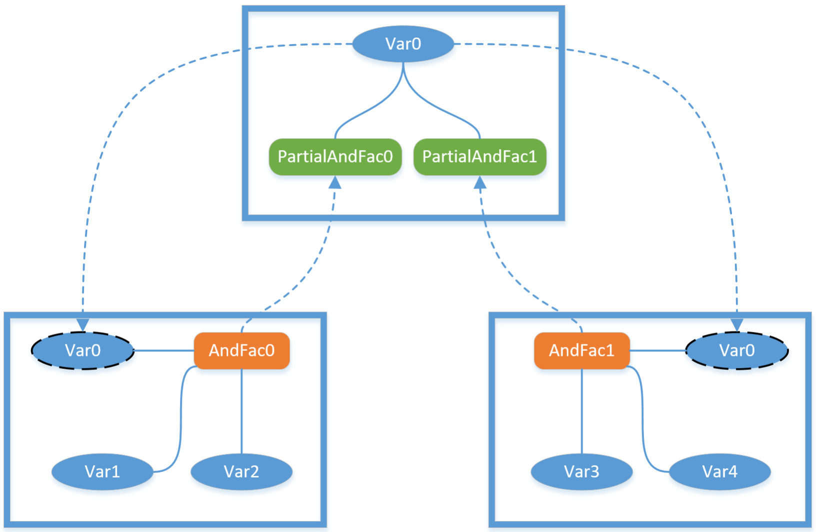


Figure 6: BDC partition, `var0` belongs to partition class B, owned by master; `var1, var2` belong to partition class C_1, factors `AndFac0` belong to D_1, they are owned by worker1; `var3, var4` belong to partition class C_2, factors `AndFac1` belong to partition class D_2, they are owned by Worker2.

Algorithm 2: Specified Algorithm for `BDC` partition

```
1 while $num\_samples > 0$ do
2     Master broadcasts its assignment of var0 to the workers
3     Workers receive var0 from master, and update their cached value
4     worker1 runs a single pass of Gibbs sampling on var1, var2 by calling the resample function;
        worker2 runs a single pass of Gibbs sampling on var3, var4
5     worker1 computes partial evaluation on AndFac0 by calling partial_eval function; worker2
        computes partial evaluation on AndFac1, then they transmit these two partial evaluation values to the
        master.
6     Master receives partial evaluation values from worker, and updates its PartialAndFac0,
        PartialAndFac1 by calling set_partial_val function
7     Master runs a single pass of Gibbs sampling on var0 it owns.
8     Output current state of all variables; $num\_samples - 1$
```

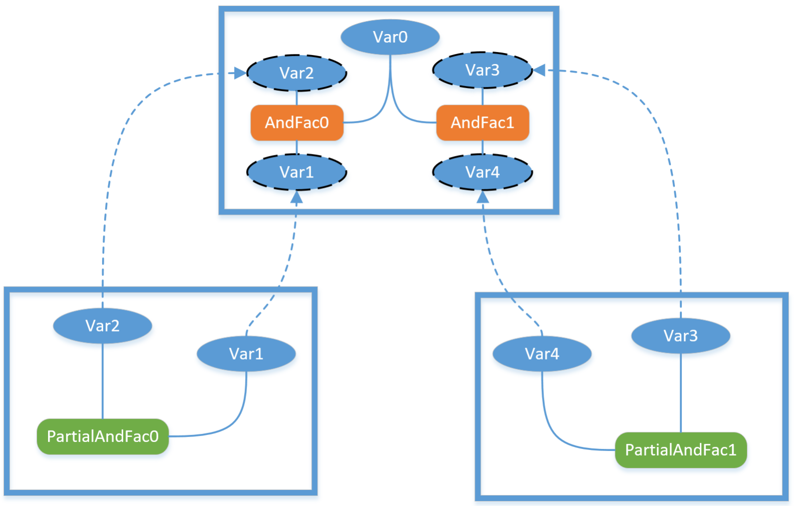


Figure 7: AED partition, `var0` belongs to partition class A, factor `AndFac0, AndFac1` belong to partition class E, they are owned by master; `var1, var2` belongs to partition class D_1, `var3, var4` belongs to partition class D_2

6.1.2 *Unary Partial Factor Optimization

6.2 AED Partition

For `AED` partition, variables on master are assigned to A, variables on workers are assigned to D, factors connects these variables are assigned to E. See Figure 7 for detailed assignment.

6.2.1 Algorithm

Algorithm 3 specifies the algorithm to perform Gibbs sampling on this graph partitioned in `AED` type.

Algorithm 3: Specified Algorithm for `AED` graph

```
1 while num_samples > 0 do
2     Master computes partial evaluation on AndFac0, AndFac1 by calling partial_eval, and transmit
      evaluation value to worker1, worker2 respectively.
3     Worker1 receive partial evaluation value from master, and update cached value in PartialAndFac0 by
      calling set_partial_val; worker2 receive partial evaluation value from master, and update cached
      value in PartialAndFac1
4     worker1 runs a single pass of Gibbs sampling on var1, var2 by calling the resample function;
      worker2 runs a single pass of Gibbs sampling on var3, var4
5     Worker1 transmit its assignment of var1, var2 to master, Worker2 transmit its assignment of var3,
      var4 to master.
6     Master runs a single pass of Gibbs sampling on Var0
7     Output current state of all variables; $num\_samples - 1$
```

6.3 AGC Partition

For `AGC` partition, variables on master are assigned to A, variables on workers are assigned to C, factors connects these variables are assigned to G. See Figure 8 for detailed assignment.

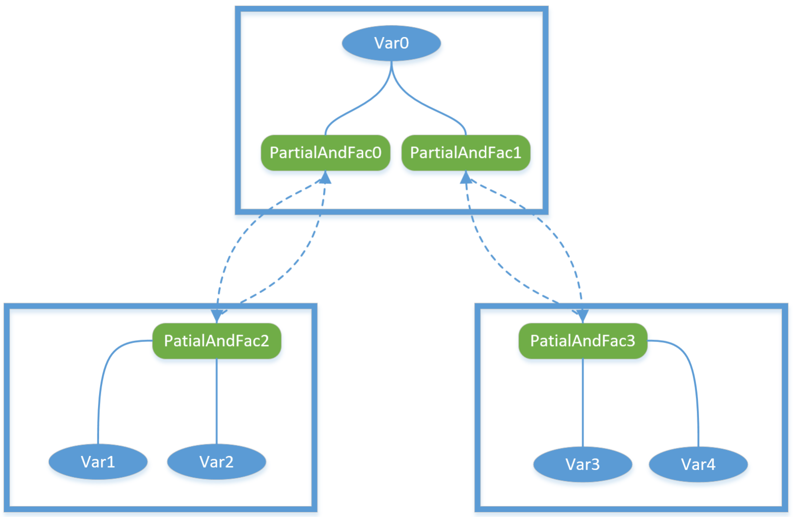


Figure 8: AGC partition, `var0` belongs to partition class A, owned by master; `var1, var2` belong to partition class C_1, partial factor `PartialAndFac0` belongs to G_1, they are owned by worker1, `var3, var4` belong to partition class C_2, factors `AndFac1` belong to partition class G_2, they are owned by worker2.

6.3.1 Algorithm

In this section, we describe how to perform Gibbs sampling on `AGC` partition graph based on Algorithm 1.

Algorithm 4: Specified Algorithm for `AGC` partition

```
1 while num_samples > 0 do
2     Master computes partial evaluation on PartialAndFac0, PartialAndFac1 by calling
        partial_eval, and transmit evaluation value to worker1, worker2 respectively.
3     Worker1 receive partial evaluation value from master, and update cached value in PartialAndFac2 by
        calling set_partial_val; worker2 receive partial evaluation value from master, and update cached
        value in PartialAndFac3
4     worker1 runs a single pass of Gibbs sampling on var1, var2 by calling the resample function;
        worker2 runs a single pass of Gibbs sampling on var3, var4
5     worker1 computes partial evaluation on PartialAndFac2 by calling partial_eval function;
        worker2 computes partial evaluation on PartialAndFac2, then transmits these two partial evaluation
        value to the master.
6     Master partial evaluation value from worker, and updates its PartialAndFac0, PartialAndFac1
        by calling set_partial_val function
7     Master runs a single pass of Gibbs sampling on var0 it owns.
8     Output current state of all variables; $num\_samples - 1$
```

6.4 BFD Graph

For `BFD` graph, variables on master are assigned to B, variables on workers are assigned to D, factors connected to them are assigned to F, see Figure 9 for detailed assignment.

6.4.1 Algorithm

In this section, we describe how to perform Gibbs sampling on `BFD` partition graph based on Algorithm 1.

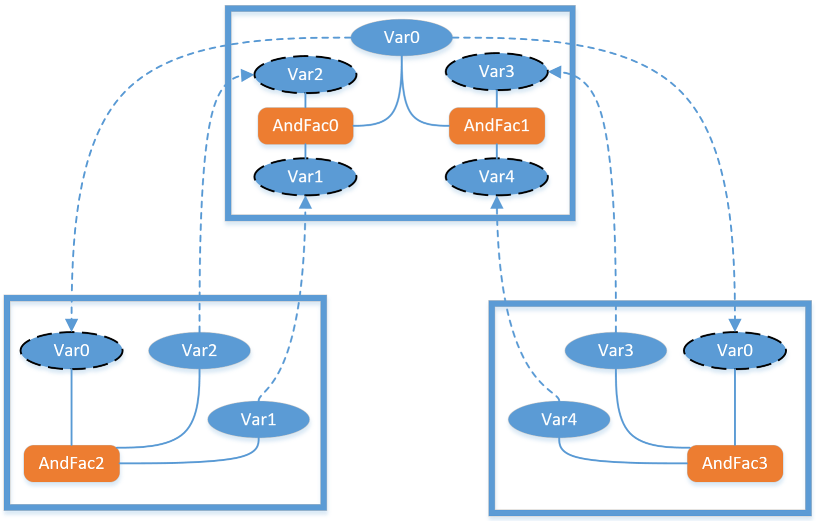


Figure 9: BFD partition, `var0` belongs to partition class B, factors `AndFac0, AndFac1` belong to partition class F, they are owned by master; `var1, var2` belong to partition class D_1, factors `AndFac2` belong to F_1, they are owned by worker1; `var3, var4` belong to partition class D_2, factors `AndFac3` belong to partition class F_2, they are owned by worker2.

Algorithm 5: Specified Algorithm for `BFD` graph

```
1 while $num\_samples > 0$ do
2     Master broadcasts its assignment var0 to the workers.
3     Workers receive var0, and update their cached assignments of var0.
4     Worker1 runs a single pass of Gibbs sampling for var1, var2 it owns; Worker2 runs a single pass of
        Gibbs sampling for var3, var4 it owns.
5     Worker1 transmit its assignment of var1, var2 to master, Worker2 transmit its assignment of var3,
        var4 to master.
6     Master runs a single pass of Gibbs sampling on the var0.
7     Output current state of all variables; $num\_samples - 1$
```

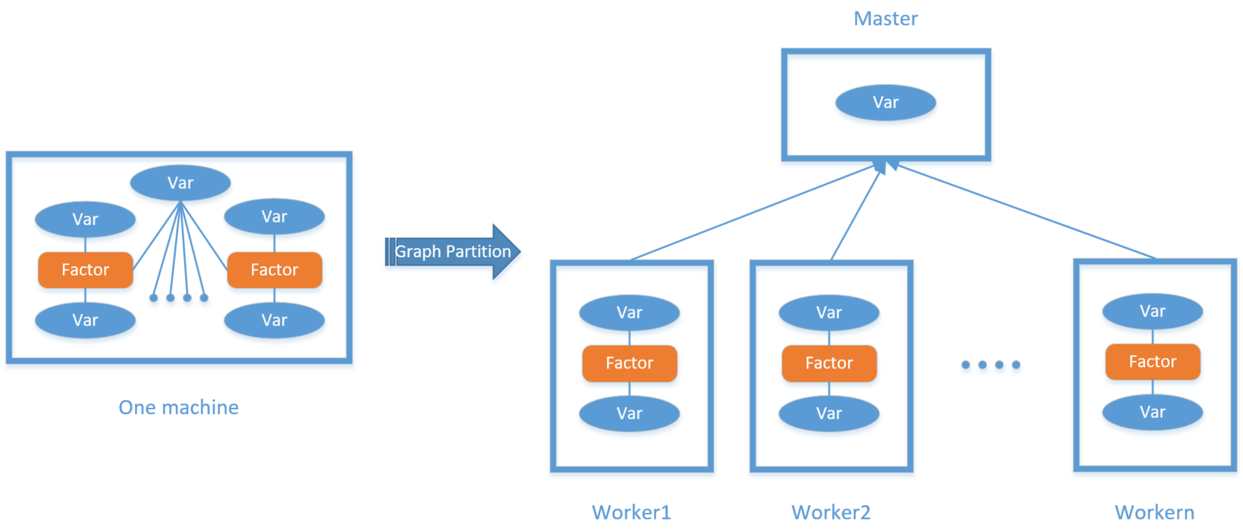


Figure 10: Experiment settings

6.5 Experiment

In the previous sections, we have studied how to partition a graph in different ways. In this section,we perform our experiment on **AWS**, with its *m5.4xlarge* instance, which contains 16 virtual cpu, 64 GB memory, settled in the normal network environment.

We set only one variable on master, then change number of variables on worker machines while keep the total number of variables in the system nearly equals to 9600. For example, if we use 2 worker machines, each machine will contains 4800 variables, if we use 4 worker machines, each machine will contains 2400 variables. By doing so, we can check the scalability of the system in terms of worker numbers. See Figure 10 for architecture.

We perform 200 rounds sampling for each variable, so in each experiment, we actually sample $200 \times 9600 \approx 2M$ variables.

We partition the graph with all the 4 partition strategies we discussed above, Figure 11 shows the result. And also, we perform Gibbs sampling on one machine by set every variable partition class A and every factor partition class F, such that the whole graph manifested on master machine. The running time for this setting takes 37.8 seconds.

6.6 Discussion and follow up improvements

From Figure 11, It can be seen that the framework scales pretty well when the machine numbers are reasonably small (less than 4 in this case). when there are more than 4 worker machines, the communication cost surpass the computation cost so running time does not decrease linearly anymore.

Another Interesting sight is that the 4 different partition strategies seems does not make much different in terms of running time. That mainly caused by our implementation so far, we do not combine `partial_val` send by the same type of factors, so every `PartialFactor` corresponds to only one `Factor` between master and worker, `BDC` type graph will be much more efficient than `BFD` graph if the one-to-many correspondence between `PartialFactor` and `Factor` implemented.

CPU occupations are about 30%-60% during the experiment, it is because we implement a synchronous Gibbs sampling framework, 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 on all Workers, thus when Workers finish their computation, Master still working on that, the Workers will be idle until Master finish its computation and transmit back informations, the “concurrency barriers” and ”idle workers” problem became the major killer for efficiency. Asynchronous Gibbs sampling described in [9, 3] can be used, and ”idle workers” problem can be ease after implementing `partial_val` combination on workers before transmit to `PartialFactors` on master.

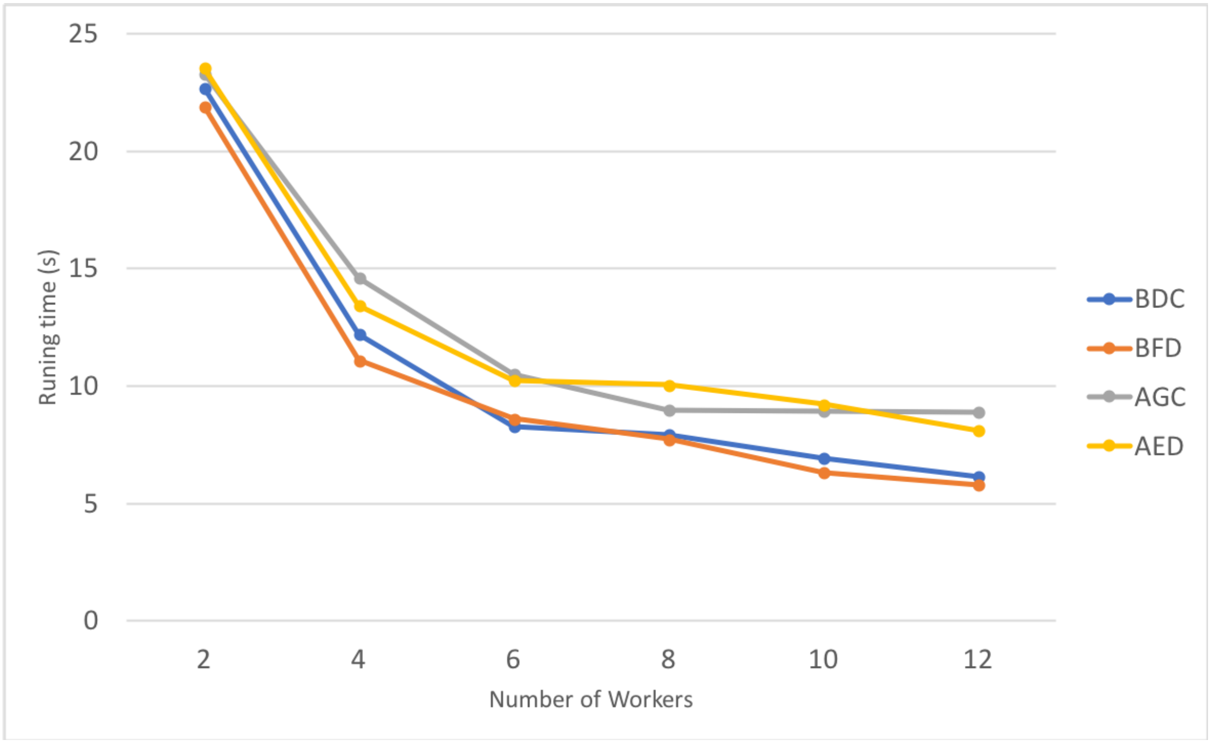


Figure 11: Running time Comparison of 4 different partition strategies in distributed settings

Because CPU occupation rate is not high, another improvement direction is multithreading techniques. DimmWitted Gibbs Sampler introduced by [10] 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. Giving the though of semantic partitioning to separate the whole graph, such that all the variables on different workers are independent with each other, our algorithm is inherently different from previous ones, the samples we get are exactly the same as sampling in sequential ways. We implement this algorithm in `C++` with `BoostMPI` library.

We elaborate 4 different graph partition strategies, running them in Amazon's cloud cluster, the comparison result shows our algorithm scales perfectly well when computation surpass transmission. Also, further improvement needs to be done to make full use of different partition strategies.

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

References

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

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

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

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

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

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

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

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

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

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