Scalable Overlapping Community Detection

Ismail El-Helw*, Rutger Hofman*, Wenzhe Li†, Sungjin Ahn‡, Max Welling§ and Henri Bal*
*Vrije Universiteit Amsterdam, The Netherlands
Email: {ielhelw,rutger,bal}@cs.vu.nl
†University of Southern California, USA
Email: wenzheli@usc.edu
‡University of Montreal, Canada
Email: sungjia@ics.uci.edu
§University of Amsterdam, The Netherlands
Email: m.welling@uva.nl

***Abstract*—**

Recent advancements in machine learning algorithms have transformed the data analytics domain and provided innovative solutions to inherently difficult problems. However, training models at scale over large data sets remains a daunting challenge. One such problem is the detection of overlapping communities within graphs. For example, a social network can be modeled as a graph where the vertices and edges represent individuals and their relationships. In this paper, we present and evaluate an efficient parallel and distributed implementation of a Stochastic Gradient Markov Chain Monte Carlo algorithm that solves the overlapping community detection problem. We show that the algorithm can scale and process graphs consisting of billions of edges and tens of millions of vertices on a compute cluster of 65 nodes.

***Keywords*-Distributed computing; Parallel programming; High performance computing; Performance analysis; Machine learning;**

I. Introduction

The tremendous amount of data that we generate through our daily applications such as social networking services, online shopping, and news recommendations, provides us with an opportunity to extract hidden but useful, even invaluable information. Realizing this opportunity, however, requires a significant amount of effort because traditional machine learning algorithms often become extremely inefficient with large amounts of data.

There have been two main approaches to this issue; machine learning researchers have developed new scalable algorithms [1], [2], while systems and networking researchers have worked on developing new generic infrastructure systems which can be leveraged to construct machine learning algorithms more efficiently [3], [4]. However, it is clear that in most cases we see the best performance by carefully integrating both of these approaches in one system.

One such big data problem is analyzing large graphs such as social networks where it is not unusual to see a network consisting of billions of edges and tens of million of vertices [5]. In particular, we are interested in the overlapping community detection problem [6], where the goal is to learn the probability distribution of each vertex to participate in each community, given a set of vertices, the links between them (which are usually very sparse), and the number of latent communities. A community can be seen as a densely connected group of vertices that are only sparsely connected to the rest of the network.

This problem is modeled by the mixed membership stochastic blockmodels (MMSB) [7] and in this paper we are particularly interested in a variant of MMSB, called assortative MMSB (a-MMSB[1]) [8].

The MMSB model is a probabilistic graphical model [9] that represents a convenient paradigm for modeling complex relationships between a potentially large number of random variables. Bayesian graphical models, where we define priors and infer posteriors over parameters, also allow us to quantify model uncertainty and facilitate model selection and averaging. However, an increasingly urgent question is whether these models and their inference procedures will be up to the challenge of handling very large graphs.

There have been two main recent advances in this direction of scalable Bayesian inference: methods based on stochastic variational Bayes (SVB) [8], [10], [11] and stochastic gradient Markov chain Monte Carlo (SG-MCMC) [12], [13], [14], [15]. Both methods have the important property that they only require a small subset of the data for every iteration. In other words, they can be applied to (infinite) streaming data.

In this paper, we are particularly interested in the SG-MCMC method applied to the a-MMSB model introduced in [16], which turned out to be faster and more accurate than the SVB method. Although [16] proposed a scalable algorithm for the problem, there is still room to improve the performance further by considering a custom high-performance implementation. To this end, we propose a design of a parallel and distributed system specifically tailored to solve the a-MMSB problem. In particular, we use

[1] Although we work on a-MMSB for simplicity, it is also straightforward to apply the proposed method to the general MMSB model.

a mixture of OpenMP, MPI and RDMA in order to efficiently scale and accelerate the algorithm's computation.

Achieving this goal necessitated overcoming several challenges. First, the algorithm's state grows rapidly with larger graphs and number of latent communities. Since the full state information is too large to fit in a single machine's memory, it is partitioned and distributed across a cluster of machines. Second, to access the full state, each cluster node must read remote memory hosted by its peers. We leveraged RDMA to limit the high latency of such operations and increase the communication bandwidth. To reduce the latency further, we pipelined the algorithm's computations such that data can be fetched in advance over the network. Finally, the algorithm's computation is effectively distributed across the cluster nodes and parallelized further within each node by exploiting their multi-core CPUs.

The remainder of this paper is organized as follows. Section 2 provides an overview of the algorithm and its theoretical foundation. The design and implementation of the parallel and distributed solution is presented in Section 3. Section 4 evaluates the efficacy of the system and analyzes its performance. Finally, Section 5 provides concluding remarks.

II. Background

A. Assortative Mixed-membership Stochastic Blockmodels (a-MMSB)

The assortative mixed membership stochastic blockmodel (a-MMSB) [8] is a special case of MMSB [7] that models the group structure in a network of N vertices. Consider a set $\mathcal{V}$ containing all the vertices in the graph, and a set $\mathcal{E}$ containing all the linked edges between pairs of vertices. Each vertex a in the vertex set $\mathcal{V}$ has a K-dimensional probability distribution π_a of participating in the K members of the community set $\mathcal{K}$. For every possible peer b in the network, each vertex a randomly draws a community z_{ab}. If a pair of vertices (a, b) in the edge set $\mathcal{E}$ are in the same community, i.e., $z_{ab} = z_{ba} = k$, then they have a significant probability β_k to connect, i.e., $y_{ab} = 1$. Otherwise this probability is small. Each community has its connection strength $\beta_k \in (0, 1)$ which reflects how likely its members are linked to each other. The whole generative process of a-MMSB is then described by:

1) For each community k, draw community strength $\beta_k \sim \text{Beta}(\eta)$
2) For each vertex a, draw community memberships $\pi_a \sim \text{Dirichlet}(\alpha)$
3) For each pair of vertices a and b,
    a) Draw interaction indicator $z_{ab} \sim \pi_a$
    b) Draw interaction indicator $z_{ba} \sim \pi_b$
    c) Draw link $y_{ab} \sim \text{Bernoulli}(r)$, where $r = \beta_k$ if $z_{ab} = z_{ba} = k$, and $r = \delta$ otherwise.

η and α are parameters to the Beta and Dirichlet distribution functions. δ (usually small) is a parameter for the algorithm.

B. Stochastic Gradient Markov Chain Monte Carlo

The algorithm we are working on in this paper is based on stochastic gradient Langevin dynamics (SGLD) [12]. SGLD applies the following update rule to obtain samples from a posterior distribution $p(\theta|\mathcal{X}) \propto p(\mathcal{X}|\theta)p(\theta)$ of N i.i.d. data points $\mathcal{X} = \{x_i\}_{i=1}^N$:

$$\theta^* \leftarrow \theta + \frac{\epsilon_t}{2}\left(\nabla_\theta \log p(\theta_t) + N\bar{g}(\theta; \mathcal{D}_n)\right) + \xi_t, \qquad (1)$$

where $\xi_t \sim \mathcal{N}(0, \epsilon_t)$ with ϵ_t the step size, $\mathcal{D}_n$ a mini-batch of size n sampled from $\mathcal{X}$, and $\bar{g}(\theta; \mathcal{D}_n)$ the mean stochastic gradient, i.e., $\frac{1}{|\mathcal{D}_n|}\sum_{x \in \mathcal{D}_n} \nabla_\theta \log p(x|\theta)$. As the step size goes to zero by a schedule satisfying $\sum_{t=1}^{\infty} \epsilon_t = \infty$ and $\sum_{t=1}^{\infty} \epsilon_t^2 < \infty$, SGLD samples from the true posterior distribution. One benefit of using SGLD is that we do not need the Metropolis-Hastings (MH) accept-reject tests since the rejection probability goes to zero as the step size tends to zero. Although in practice we often do not reduce the stepsize to zero to obtain better mixing (thus resulting in some bias), we obtain good performance by drawing many more samples per unit time.

SGLD originated from the Langevin Monte Carlo (LMC) [17], where unlike SGLD the gradient is computed exactly by using all data points. Then, Metropolis-Hastings accept-reject tests are applied. Comparing to LMC, SGLD only requires to process a mini-batch $\mathcal{D}_n$ at each iteration and ignores the MH test, and thus the computation complexity substantially reduces from $\mathcal{O}(N)$ to $\mathcal{O}(n)$.

Stochastic gradient Riemannian Langevin dynamics (SGRLD) [13] is a subclass of SGLD which is developed to efficiently sample from the probability simplex. By applying Riemannian geometry [17] and using the mini-batch-based estimator in Eqn. 1, it achieved state-of-the-art performance for latent Dirichlet allocation (LDA) [18]. In particular, for a K-dimensional probability simplex π, it uses the *expanded-mean* re-parameterization trick, where the probability of a category k is given by $\pi_k = \theta_k / \sum_{j=1}^K \theta_j$ with $\theta_k \sim \text{Gamma}(\alpha, 1)$ and α a hyperparameter of the Dirichlet distribution $p(\pi|\alpha)$. Then, the update rule of SGRLD is:

$$\theta_k^* \leftarrow \left|\theta_k + \frac{\epsilon_t}{2}\left(\alpha - \theta_k + \frac{N}{|\mathcal{D}_n|}\sum_{d \in \mathcal{D}_n} g_d(\theta_k)\right) + (\theta_k)^{\frac{1}{2}}\xi_t\right|, \quad (2)$$

here $g_d(\theta_k)$ is the gradient of the log posterior w.r.t. θ_k on a data point $d \in \mathcal{D}_n$.

C. Scalable MCMC for a-MMSB

This section describes the SG-MCMC algorithm for a-MMSB. Please refer [16] for more details. The algorithm iterates updating local parameter π and global parameter β. Since both parameters lie on the probability simplex, SGRLD is applied to make the sampling process more efficient. Here, the parameters ϕ and θ are used to re-parameterize π and β respectively. After updating ϕ and θ, we can obtain π and β by normalizing ϕ and θ, respectively.

That is, $\beta_k = \theta_{k1}/(\theta_{k0} + \theta_{k1})$ and $\pi_{ak} = \phi_{ak}/\sum_{j=1}^{K}\phi_{aj}$. In the following, we briefly sketch the iterative update steps.

Sampling global parameters: The update rule for global parameter θ is

$$\theta^*_{ki} \leftarrow \left|\theta_{ki} + \frac{\epsilon}{2}\left(\eta - \theta_{ki} + h(\mathcal{E}_n)\sum_{(a,b)\in\mathcal{E}_n} g_{ab}(\theta_{ki})\right) + (\theta_{ki})^{\frac{1}{2}}\xi_{ki}\right|, \tag{3}$$

where the gradient in θ

$$g_{ab}(\theta_{ki}) = \frac{f^{(y)}_{ab}(k,k)}{Z^{(y)}_{ab}}\left(\frac{|1-i-y|}{\theta_{ki}} - \frac{1}{\sum_j \theta_{kj}}\right), \tag{4}$$

here $\mathcal{E}_n$ is a mini-batch of n_t vertex pairs sampled from $\mathcal{E}$; $h(\mathcal{E}_n)$ is a weight factor to scale the effect of a mini-batch towards the full network; and

$$f^{(y)}_{ab}(k,l) = \begin{cases} \beta_k^y(1-\beta_k)^{(1-y)}\pi_{ak}\pi_{bk}, & \text{if } k = l \\ \delta^y(1-\delta)^{(1-y)}\pi_{ak}\pi_{bl}, & \text{if } k \neq l. \end{cases}$$

$Z^{(y)}_{ab}$ is the normalization constant which we can compute in $\mathcal{O}(K)$ time [16].

Sampling local parameters: The update rule for the local parameters ϕ is

$$\phi^*_{ak} \leftarrow \left|\phi_{ak} + \frac{\epsilon}{2}\left(\alpha - \phi_{ak} + \frac{N}{|\mathcal{V}_n|}\sum_{b\in\mathcal{V}_n} g_{ab}(\phi_{ak})\right) + (\phi_{ak})^{\frac{1}{2}}\xi_{ak}\right|, \tag{5}$$

where the gradient in ϕ

$$g_{ab}(\phi_{ak}) = \frac{f^{(y)}_{ab}(k)}{Z^{(y)}_{ab}\phi_{ak}} - \frac{1}{\sum_j \phi_{aj}}. \tag{6}$$

Here, $\mathcal{V}_n$ is the neighbor set for a mini-batch node, another random mini-batch of n nodes sampled from $\mathcal{V}$. Note that $|\mathcal{V}_n| \ll |\mathcal{V}| = N$.

As is common in machine learning problems, the edges of the graph are divided into two sets, the training set and a held-out set $\mathcal{E}_h$. As the performance metric we use perplexity, which is defined as the exponential of the negative average log-likelihood of the held-out set $\mathcal{E}_h$. Given a collection of T samples of the model parameters $\{\beta_t\}$ and $\{\pi_t\}$, the averaged perplexity on the held-out test set $\mathcal{E}_h$ is

$$\begin{aligned} &\text{perp}_{\text{avg}}(\mathcal{E}_h|\{\beta_t\},\{\pi_t\}) \\ &= \exp\left(-\frac{\sum_{(a,b)\in\mathcal{E}_h}\log\{(1/T)\sum_{t=1}^{T} p(y_{ab}|\beta_t,\pi_t)\}}{|\mathcal{E}_h|}\right) \end{aligned} \tag{7}$$

The perplexity is not evaluated at every iteration, but at regular intervals.

The pseudo-code of the sequential algorithm is presented in Algorithm 1.

Algorithm 1 Sequential version of SG-MCMC for a-MMSB

```
Initialize $\pi, \beta, \phi, \theta$
while sampling do
    Sample a mini-batch of vertex pairs, $\mathcal{E}_n$, from $\mathcal{E}$
    for each vertex in $\mathcal{E}_n$ do
        Sample a mini-batch of vertices, $\mathcal{V}_n$, from $\mathcal{V}$
        Update $\phi_a$ using Eqn 5
        Obtain $\pi_a$ from $\phi^*_a$
    end for
    for $k = 1, \dots, K$ do
        Update $\theta_k$ using Eqn 3
        Obtain $\beta_k$ from $\theta^*_k$
    end for
end while
```

D. Related work on parallel community detection

Among previous work on parallelizing MMSB algorithms, we mention the Online Tensor approach [19] on GPUs, and our previous parallel implementation of SG-MCMC on GPUs and multi-core CPUs (submitted for publication). There are a few projects that did a distributed implementation for community detection on very large network graphs. In contrast to our work, they all detect non-overlapping communities: [20] investigate a large number of networks; [21] investigate hierarchical stochastic blockmodels; [22] use multi-core machines.

symbol	type	size	description
$\mathcal{K}$		K	set of communities
$\mathcal{V}$	{vertex}	N	vertices in the graph
$\mathcal{E}$	{edge}		linked edges in the graph
E	{edge}		$\mathcal{V} \times \mathcal{V}$: linked and nonlinked edges
$\mathcal{E}_h$	{edge}		held-out subset of the graph
$\mathcal{E}_n$	{edge}		sampled mini-batch of edges in E
M			number of vertices in $\mathcal{E}_n$
$\mathcal{V}_n$	{vertex}		sampled neighbor set for a vertex in $\mathcal{E}_n$
θ	float vector 2-D	$K \times 2$	global latent variables for community strength
β	float vector	K	$\beta[k] = \theta[k][0]/\sum_j \theta[k][j]$ is the community strength
ϕ	float vector 2-D	$N \times K$	local latent variables; $\phi[i][k]$ reflects probability that vertex i is in community k
π	float vector 2-D	$N \times K$	$\pi[i][k]$ is probability that i is in k $\pi[i][k] = \phi[i][k]/\sum_j \phi[i][j]$

Table I
DEFINITION OF MOST IMPORTANT SYMBOLS

III. System Design

The SG-MCMC algorithm described in the previous section has an abundance of opportunities for parallelism, for the multi-threaded, shared-memory type as well as for distributed-memory parallelism. The benefit of multi-threaded parallelism is speed-up of the computation. A distributed implementation additionally allows to store data in the collective memory of the cluster machine and increases memory bandwidth which scales with the number of machines. The downside of a distributed implementation is that it requires considerably more programming effort.

This section describes how we parallelize the algorithm, which had been implemented sequentially in C++. In most places, the usage of multi-threaded parallelism is straightforward, therefore, we will discuss details only where appropriate. The distributed design follows a master-worker paradigm, where in essence the master controls the parallel operations and the workers perform the calculations. For thread parallelism, we annotate the program with OpenMP [23]. For distributed communication, we use MPI [24].

A. Data distribution

The largest data structures of the algorithm are $\mathcal{E}$, $\mathcal{E}_h$, π and ϕ, see Table I. For the largest dataset in this paper, com-Friendster, $\mathcal{E}$ has 1.8 billion undirected edges. In our representation with directed edges, this takes up 13.5GB. Our design lets $\mathcal{E}$ reside only at the master's. We observe that the calculations in the update stages only require the subset of $\mathcal{E}$ that is touched by the mini-batch vertices, so the master scatters that subset to the workers together with the scattering of the mini-batch vertices. This way, we trade a reduction in memory usage at the workers against limited communication costs. In contrast, $\mathcal{E}_h$ is be partitioned statically over all machines for the parallel perplexity calculation.

π and ϕ are 32-bit float arrays of size $K \times N$. For our largest distributed experiment, com-Friendster with N=64M and K=12K, each requires 3TB. We decided to store only π and $\sum \phi$, and recalculate ϕ from these whenever appropriate. Here, we trade a substantial gain in memory usage against some computation. π is partitioned across the workers, and π values are accessed via a DKV (distributed key-value) store.

The distribution within the graph of the vertices of the mini-batch as well as the neighbor sets is completely random. That means that there is no locality in π access patterns, especially in the accesses to the neighbor sets $\mathcal{V}_n$. Hence, there is no opportunity to exploit data locality through caching of π.

B. RDMA Distributed Key-Value Store

Modern network technology allows access to the memory of remote machines through the network cards (NICs) without involvement from the remote host altogether [25], [26]. This has opened up the development of extremely efficient remote memory services, often in the form of DKV stores. This project stores π in a DKV store in the collective memory of the cluster, where $\pi[i] + \sum \phi[i]$ is the value for key i. We decided not to use any of the existing implementations of RDMA DKV stores, and instead build our own DKV store on top of the Infiniband ib-verbs API. The main consideration is that our use case is unusually simple in a number of important aspects. The KV layout is static in the sense that there are no inserts or deletes after the initial population, which allows a static partitioning of KV pairs over the machines. The values are all of the same size, a vector of $K + 1$ floats. The access pattern is well controlled because the computation is partitioned into stages that are separated by barriers. A stage either reads values or updates values. The algorithm ensures that updates are always targeting unique elements, therefore, there are no read/write or write/write hazards. The existing DKV stores (RamCloud [27], FaRM [28], and reputedly the fastest, Herd [29]) incur overhead for insert/delete flexibility, concurrency control, or variable-sized values. Our use case allows us to do any operation in exactly one RDMA read or RDMA write. Herd argues that replacing RDMA reads by an RPC to the server, followed by an unreliable write, is faster. However, they show this only holds for small payload packets, up to 256B. A π packet in our application is typically thousands to hundreds of thousands of 4-byte floats.

C. Implementation of distributed parallelism

This section describes the parallelization of each of the stages of the algorithm's main loop. All but the first stage justify parallelization for high values of combinations of the parameters mini-batch size $|\mathcal{E}_n|$, number of vertices in the mini-batch M, number of communities K, neighbor sample size $|\mathcal{V}_n|$, and held-out set size $|\mathcal{E}_h|$.

The first stage, mini-batch selection (line 3 in Algorithm 1), is done by the master and it is not itself parallelized. However, the distributed implementation overlaps its execution using pipeline parallelism while the workers are performing *update_phi*. The mini-batch is partitioned equally over the workers and the relevant sections of $\mathcal{E}$ are distributed together with the mini-batch subsets.

After a worker has received its subset of the mini-batch, it samples a neighbor set $\mathcal{V}_n$ for each of its mini-batch vertices, using thread parallelism (line 5 in Algorithm 1).

The next stage, *update_phi* (line 6 in Algorithm 1), is the algorithm's dominant stage, not only in calculation but also in memory accesses. It performs $M \times |\mathcal{V}_n| \times K$ operations. It can be fully parallelized because it is a data-parallel operation over each of the mini-batch vertices. *update_phi* loads the π values for the local mini-batch vertices and their neighbors from the DKV store. The updates to ϕ for the mini-batch vertices are calculated independently using thread parallelism.

These updates are used by *update_pi* (line 7 in Algorithm 1) locally, besides the value of π/ϕ for the mini-batch vertices. The update to π requires $M \times K$ operations, and it is done in parallel over mini-batch vertices. Because of memory consistency, this stage awaits completion of *update_phi* with an MPI barrier. After the calculation, the updated values of $\pi + \sum \phi$ are written through the DKV store.

update_beta (lines 10 and 11 in Algorithm 1) requires $|\mathcal{E}_n| \times K$ operations. It is preceded by an MPI barrier to ensure that up-to-date π values are read. *update_beta* is split into four steps. The first stage partitions the mini-batch across machines to calculate contributions to $g_{ab}(\theta)$. The values of π for the local mini-batch vertices are loaded from the DKV store. These calculations are done with thread parallelism over the mini-batch vertices. In the second step, a multi-threaded summation is performed to calculate the contributions per machine, followed by a distributed summation using an MPI reduce operation. The third step calculates β from θ sequentially at the master. This operation takes only K steps. The resulting β is broadcast with MPI to the workers.

The calculation of perplexity requires $|\mathcal{E}_h| \times K$ steps. Each of the machines owns a subset of $\mathcal{E}_h$. It loads up-to-date values of π for each vertex in its part of $\mathcal{E}_h$, then for each of its edges in $\mathcal{E}_h$ it calculates the contribution to the perplexity using thread parallelism. The resulting contributions are summed to the global perplexity value in two stages, with a local OpenMP reduction followed by a global MPI reduction.

Note that the calculation requirements per iteration do not depend on N or $|\mathcal{E}^*|$, the size of the network graph. However, larger graphs are expected to require more iterations to achieve convergence.

D. Pipelining of computation and loading π

The distributed design features two instances of pipeline parallelism. First, at the process level, sampling of the minibatch by the master is decoupled from *update_phi* by the workers, so the master computes the minibatch for the next iteration while the workers compute *update_phi* for the current iteration. Second, at the threads level within the workers, loading of π and *update_phi* are split up into chunks, and the calculation in *update_pi* for the current chunk is done simultaneously with loading π for the next chunk. The rationale for implementing pipeline parallelism in these places of the program, is that loading π is the dominant contribution within the dominant stage *update_phi*, as will be shown in Section IV-C. Since loading π is the performance bottleneck of the distributed algorithm, performing other work in parallel shortens the critical path.

IV. Evaluation

This section presents an in-depth empirical analysis of the design discussed in Section III. To this end, we assess the scalability and efficiency of the solution using multiple criteria. First, an evaluation of the *strong scaling* behavior of the distributed algorithm is presented. Second, the *weak scaling* of the solution is assessed with respect to increasing numbers of targeted latent communities. Next, the effects of *pipelining the computation* and varying the number of latent communities on the system's performance is analyzed. Further, we evaluate the efficiency and overhead associated with *network communication* between cluster nodes. Finally, we *contrast* the effectiveness of scaling the computation horizontally and vertically.

Empirical results were obtained by performing experiments on the VU and Leiden University DAS5 clusters which consist of 68 and 24 compute nodes respectively. Each compute node is equipped with a dual 8-core Intel Xeon E5-2630v3 CPU clocked at 2.40GHz, 64GB of memory and 8TB of storage. Moreover, the compute nodes of each site are interconnected by FDR InfiniBand. The MPI implementation used is MVAPICH2, which is optimally tuned for Infiniband [30]. All of the experiments reported in this section utilized publicly available graphs from the Stanford Large Network Dataset Collection (SNAP) [31]. Table II lists the collection of data sets used.

A. Strong Scaling

In order to evaluate the horizontal scalability of the distributed implementation we tested the system's performance across different cluster sizes while holding the problem size constant. For this study, we used the com-Friendster graph as it contains the largest number of vertices and edges. Figure 1-a presents the execution time of 2048 algorithm iterations across multiple cluster sizes. The x-axis starts from 8 worker nodes as the data set is too large to fit into the collective memory of a smaller cluster. As shown in the figure, the execution time steadily decreases by increasing the cluster size. A deeper analysis of the individual execution phases of the algorithm provides insights into the scalability of its building blocks. In addition to the total execution time, Figure 1-a presents the cumulative time spent in individual computational phases across iterations. Moreover, Figure 1-b presents the speedup achieved for the same experiments reported in Figure 1-a. As is clearly shown, the dominant phase of the execution is *update_phi_pi*. The reported total time for each cluster size is significantly less than the sum of the execution times of the individual phases. This is due to the overlapping execution of the two most expensive phases, namely, *update_phi_pi* and mini-batch deployment. Both of these phases initially gain significant speedup with the addition of compute nodes. However, the speedup curve gradually slows down for larger cluster sizes as the work granularity of each worker node decreases, limiting their

Name	#Vertices	#Edges	#Ground-truth communities	Description
com-LiveJournal	3,997,962	34,681,189	287,512	Online blogging social network
com-Friendster	65,608,366	1,806,067,135	957,154	Online gaming social network
com-Orkut	3,072,441	117,185,083	6,288,363	Online social network
com-Youtube	1,134,890	2,987,624	8,385	Video-sharing social network
com-DBLP	317,080	1,049,866	13,477	Computer science bibliography collaboration network
com-Amazon	334,863	925,872	75,149	Product co-purchasing network

Table II
SUMMARY OF SNAP GRAPH DATA SETS USED FOR EVALUATION.

resource utilization. The time spent in *update_beta_theta* remains relatively constant across cluster sizes as it performs an insignificant amount of work compared to the synchronization overhead of a collective MPI reduction operation contained within it.

B. Weak Scaling

A study of a system's weak scaling aids in the assessment of the communication and synchronization overheads associated with the management of large clusters. Similarly, it can expose complex issues that hinder performance at scale such as load imbalance. To fairly evaluate the algorithm's weak scaling behavior we conducted several executions varying the cluster size and the number of latent communities proportionally. This methodology ensures that each compute node performs a relatively constant amount of computational work across all configurations. However, the number and size of messages exchanged between the nodes would vary significantly. Figure 2-a presents the average execution time per algorithm iteration across different cluster sizes. On the other hand, Figure 2-b reports the number of communities for each cluster configuration. The relative change in the average execution time per iteration is insignificant even though the communication intensity increases for larger cluster sizes. This observation verifies that the system's overall overhead is minimal. Additionally, the experimental results show that the implementation is capable of achieving good speedups provided the input problem is large enough for the given cluster size.

C. Pipeline efficiency

As discussed in Section III, the collective memory of all worker nodes serves as the storage for the state of the computation. As such, the rows of π are equally and statically partitioned across all workers. Given that π accesses are random, a node in a cluster of size C must fetch $(C-1)/C$ of all read requests over the network. Therefore, large cluster configurations exhibit higher bandwidth demands and are more sensitive to network latency. To reduce the negative effects of network latency on the computation, a pipelining scheme was devised to prefetch data dependencies. Figure 3 presents the execution time of 1024 algorithm iterations on a 64-node cluster with double-buffering enabled and disabled. Naturally, increasing the number of communities causes a proportional increase in execution time. However, when double-buffering is enabled, some of the incurred network latency is hidden by overlapping it with computation. Moreover, since both computation time and network latency increase with larger K, the benefit of pipelining increases. This can be observed through the widening gap between both lines depicted in Figure 3.

Iteration stages	non-pipelined	pipelined
total	450	365
draw/deploy mini-batch	45.6	
update_phi	285	241
update_pi	3.8	4.6
update β/θ	25.9	33.6
Substages within update_phi		
draw/deploy mini-batch		26.2
load π	205	209
update ϕ	74	74

Table III
THE MOST IMPORTANT STAGES IN THE EXECUTION; COM-FRIENDSTER ON 65 COMPUTE NODES, WITH 12K COMMUNITIES. TIMES IN MS PER ITERATION. THE LOWER HALF OF THE TABLE SHOWS SUB-STAGES WITHIN *update_phi*. IN THE PIPELINING VERSION, THESE COMPONENTS ARE DONE IN PARALLEL.

Table III shows a breakdown of the most important time consumers; the left-hand column is without pipelining, the right-hand column has pipelining. *update_phi* is the dominant contribution, and within that, loading π from the DKV store takes by far the most time. The pipelining optimization overlaps loading π with generation and deployment of the mini-batch, and with the calculation of *update_phi*.

D. Horizontal vs. Vertical Scalability

One of the main drawbacks of designing a distributed solution for a given algorithm is the inherent complexity of communication and synchronization. A considerably simpler approach would be developing a multi-threaded version and running it on a machine with abundant memory and CPU cores. In such a context, access to all of the algorithm's state would be an order of magnitude faster than RDMA. Additionally, the overhead associated with synchronizing threads is negligible compared to using MPI primitives. To evaluate the efficacy of both approaches we utilized SURFsara's HPC Cloud system to instantiate a virtual machine with 40 Intel Xeon E7-4850 cores clocked at 2.00GHz cores and 1TB of memory. The physical machine underlying the HPC Cloud system contains 40 CPU cores and does not oversubscribe resources. Therefore, by provisioning all 40 cores we ensured that there is no resource contention from other users of the system. Figure 4 reports the execution time per algorithm iteration for two experimental setups. First, Figure 4-a shows the performance of executing the algorithm on the HPC Cloud system with 40 and 16 cores compared to a single DAS5 node with 16 cores. This test uses the com-DBLP data set to enable the use of a large number of communities without running out of memory. It is clear from the results that the performance can benefit from the additional cores provided by HPC Cloud system. Further, Figure 4-b tests the performance of the HPC Cloud system compared to 64 nodes of the DAS5 cluster using the com-Friendster data set. Clearly, the parallel and distributed implementation vastly outperforms the single-node multi-threaded solution. Moreover, the trajectory of both curves shows a widening gap between them suggesting that the relative performance difference will increase for larger K. In conclusion, the overhead of network communication in the distributed version is more than compensated by the increasing compute power compared to a single-node implementation.

E. Cluster Communication Efficiency

Since the distributed algorithm is data-intensive, a key aspect in improving its performance is to maximize the utility of the network resources. In figure 5, we provide maximum bandwidth numbers for read operations between one server and one client for a range of payload sizes, and compare these to the bandwidth achieved by *qperf*, which shows the best achievable performance for Infiniband. We present *qperf* bandwidth for both RDMA read and RDMA write operations; these are nearly identical, which corroborates the results from the Herd project for payloads upwards from 256B. The bandwidth achieved by our DKV store falls short of the *qperf* performance for packets less than 4KB; this is attributed to additional per-request overhead for the DKV store. For the largest packet size, the DKV store performance is hampered by the fact that its values are spread over a larger memory area, whereas *qperf* always reads from the same memory locations. For packets between 8KB and 512KB, our DKV achieves performance very close to *qperf*.

F. Convergence of Large Datasets

The previous sections focused on the computational performance of the distributed implementation. However, the algorithm's throughput does not necessarily indicate how fast it can converge to a solution. More specifically, it remains unclear how many iterations are needed for the algorithm to reach a stable state and terminate. Therefore,

Figure 1. (a) Execution time of 2048 algorithm iterations for the same problem size (com-Friendster, K=1024, M=16384 n=32) across different cluster sizes. (b) Speedup achieved for same experiments in (a) with respect to 8 nodes

Figure 2. (a) Average execution time per algorithm iteration varying the number of latent communities proportionally to the number of compute nodes. (b) The exact number of communities used for each data point in (a).

we now shift our focus to the convergence time to assess the system's utility. Figure 6 presents the convergence time of the data sets listed in Table II. The results in sub-figures a, b and c were obtained by using 65 compute nodes of DAS5. In Figure 6-a, the number of communities was set to 12K which fully occupied the aggregate memory resources of all 64 worker nodes since com-Friendster has roughly 65 million vertices. In this case, the algorithm reached a stable state after 3-4 hours. Next, Figures 6-b and 6-c present the convergence of com-LiveJournal and com-Orkut respectively. As the number of vertices in these data sets is an order of magnitude smaller than com-Friendster, we could use a larger number of communities to fill up the collective memory. Naturally, the convergence time was extended as the complexity of the algorithm increases dramatically with larger K. However, the system was capable of reporting results for both in around 40 hours. Figures 6-d, 6-e and 6-f were all configured to use the number of ground-truth communities associated with their respective data sets, com-Youtube, com-DBLP and com-Amazon. Since this yields a much smaller storage requirement, these experiments were conducted on 14, 24 and 24 cluster nodes respectively.

The results reported in Figure 6 verify that the distributed implementation of the algorithm is capable of detecting overlapping communities within real graphs in reasonable time given the available compute resources. The time required to reach convergence may vary depending on a graph's properties and the targeted number of communities. To the best of our knowledge, the data sets used in this study are the largest publicly available organic graphs.

V. Conclusion

The recent advancements of machine learning algorithms make them ideal candidates to solve complex problems. However, using these solutions in order to process problems at scale is still a daunting task. In particular, knowledge of parallel and distributed computing techniques is necessary to facilitate the development of such systems and streamline their execution time.

In this paper, we shared our experience in developing a highly scalable and efficient solution for a stochastic gradient Markov chain Monte Carlo algorithm that detects overlapping communities in graphs. The system design had to overcome several obstacles in order to achieve high performance. Specifically, we discussed how the algorithm was structured to facilitate its parallelization. Moreover, we evaluated the efficacy of overlapping computation with communication to hide latency. Further, we demonstrated the use of a mixture of MPI and RDMA primitives to speedup the communication between cluster nodes.

We conducted a thorough empirical evaluation of the system to study its strong and weak scalability on 65 cluster nodes using large data sets. Additionally, we assessed the efficiency of the algorithm's resource utilization. Finally, a demonstration of the implementation's utility was provided by processing 6 different organic data sets.

References

[1] L. Bottou, "Large-scale machine learning with stochastic gradient descent," in *Proceedings of COMPSTAT'2010*. Springer, 2010, pp. 177–186.

[2] 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, pp. 1–122, 2011.

[3] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

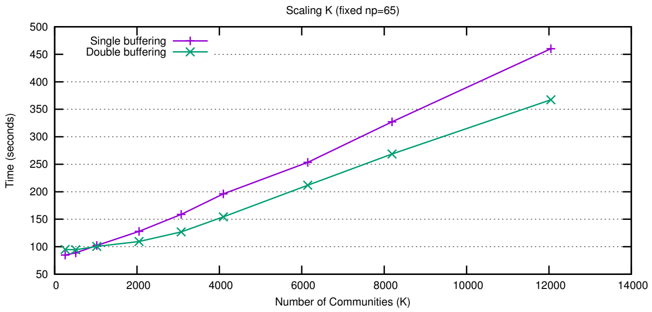

Figure 3. Performance effect of varying the number of communities on the algorithm's execution time on 64 nodes when using single- or double-buffering.

Figure 4. Performance comparison between the distributed implementation running on DAS5 and the multi-threaded solution on machine with 40 cores and 1TB of RAM.

[4] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data," *ACM Transactions on Computer Systems (TOCS)*, vol. 26, no. 2, p. 4, 2008.

[5] J. Yang and J. Leskovec, "Defining and evaluating network communities based on ground-truth," *Knowledge and Information Systems*, vol. 42, no. 1, pp. 181–213, 2015.

[6] J. Xie, S. Kelley, and B. K. Szymanski, "Overlapping community detection in networks: The state-of-the-art and comparative study," *ACM Computing Surveys (csur)*, vol. 45, no. 4, p. 43, 2013.

[7] E. M. Airoldi, D. M. Blei, S. E. Fienberg, and E. P. Xing, "Mixed membership stochastic blockmodels," in *Advances in Neural Information Processing Systems*, 2009, pp. 33–40.

[8] P. K. Gopalan, S. Gerrish, M. Freedman, D. M. Blei, and D. M. Mimno, "Scalable inference of overlapping communities," in *Advances in Neural Information Processing Systems*, 2012, pp. 2249–2257.

[9] D. Koller and N. Friedman, *Probabilistic graphical models: principles and techniques*. MIT press, 2009.

[10] M. D. Hoffman, D. M. Blei, C. Wang, and J. Paisley, "Stochastic variational inference," *The Journal of Machine Learning Research*, vol. 14, no. 1, pp. 1303–1347, 2013.

[11] P. K. Gopalan and D. M. Blei, "Efficient discovery of overlapping communities in massive networks," *Proceedings of the National Academy of Sciences*, vol. 110, no. 36, pp. 14 534–14 539, 2013.

[12] M. Welling and Y. W. Teh, "Bayesian learning via stochastic gradient Langevin dynamics," in *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, 2011, pp. 681–688.

[13] S. Patterson and Y. W. Teh, "Stochastic gradient Riemannian Langevin dynamics on the probability simplex," in *Advances in Neural Information Processing Systems*, 2013, pp. 3102–3110.

[14] S. Ahn, B. Shahbaba, and M. Welling, "Distributed stochastic gradient MCMC," in *Proceedings of the 31st International Conference on Machine Learning (ICML-14)*, 2014, pp. 1044–1052.

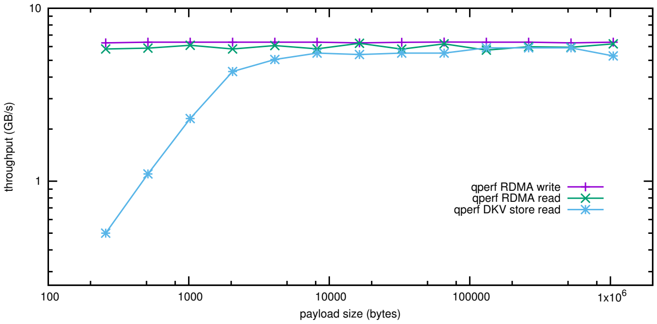

Figure 5. Performance of DKV store compared to qperf.

[15] S. Ahn, A. Korattikara, and M. Welling, "Bayesian posterior sampling via stochastic gradient Fisher scoring," *arXiv preprint arXiv:1206.6380*, 2012.

[16] W. Li, S. Ahn, and M. Welling, "Scalable MCMC for mixed membership stochastic blockmodels," *CoRR*, vol. abs/1510.04815, 2015. [Online]. Available: http://arxiv.org/abs/1510.04815

[17] M. Girolami and B. Calderhead, "Riemann Manifold Langevin and Hamiltonian Monte Carlo Methods," *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, vol. 73, no. 2, pp. 123–214, 2011.

[18] D. M. Blei, A. Y. Ng, and M. I. Jordan, "Latent Dirichlet Allocation," *the Journal of machine Learning research*, vol. 3, pp. 993–1022, 2003.

[19] F. Huang, U. N. Niranjan, M. U. Hakeem, P. Verma, and A. Anandkumar, "Fast Detection of Overlapping Communities via Online Tensor Methods on GPUs," *CoRR*, 2013. [Online]. Available: http://arxiv.org/abs/1309.0787

[20] Z. Bu, C. Zhang, Z. Xia, and J. Wang, "A fast parallel modularity optimization algorithm (FPMQA) for community detection in online social network," *Knowledge-Based Systems*, vol. 50, pp. 246 – 259, 2013. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0950705113001901

[21] V. Lyzinski, M. Tang, A. Athreya, Y. Park, and C. E. Priebe, "Community Detection and Classification in Hierarchical Stochastic Blockmodels," *ArXiv e-prints*, Mar. 2015.

[22] A. Prat-Pérez, D. Dominguez-Sal, and J.-L. Larriba-Pey, "High quality, scalable and parallel community detection for large real graphs," in *Proceedings of the 23rd International Conference on World Wide Web*, ser. WWW '14. New York, NY, USA: ACM, 2014, pp. 225–236. [Online]. Available: http://doi.acm.org/10.1145/2566486.2568010

[23] OpenMP Architecture Review Board, "OpenMP application program interface," Specification, 2011. [Online]. Available: http://www.openmp.org/mp-documents/OpenMP3.1.pdf

[24] Message Passing Forum, "MPI: A message-passing interface standard," Knoxville, TN, USA, Tech. Rep., 1994.

[25] T. Hamada and N. Nakasato, "InfiniBand Trade Association, InfiniBand Architecture Specification, Volume 1, Release 1.0, http://www.infinibandta.com," in *International Conference on Field Programmable Logic and Applications, 2005*, pp. 366–373.

[26] M. Beck and M. Kagan, "Performance Evaluation of the RDMA over Ethernet (RoCE) Standard in Enterprise Data Centers Infrastructure," in *Proceedings of the 3rd Workshop on Data Center - Converged and Virtual Ethernet Switching*, ser. DC-CaVES '11. International Teletraffic Congress, 2011, pp. 9–15. [Online]. Available: http://dl.acm.org/citation.cfm?id=2043535.2043537

[27] J. Ousterhout, A. Gopalan, A. Gupta, A. Kejriwal, C. Lee, B. Montazeri, D. Ongaro, S. J. Park, H. Qin, M. Rosenblum, S. Rumble, R. Stutsman, and S. Yang, "The RAMCloud storage system," *ACM Trans. Comput. Syst.*, vol. 33, no. 3, pp. 7:1–7:55, Aug. 2015. [Online]. Available: http://doi.acm.org/10.1145/2806887

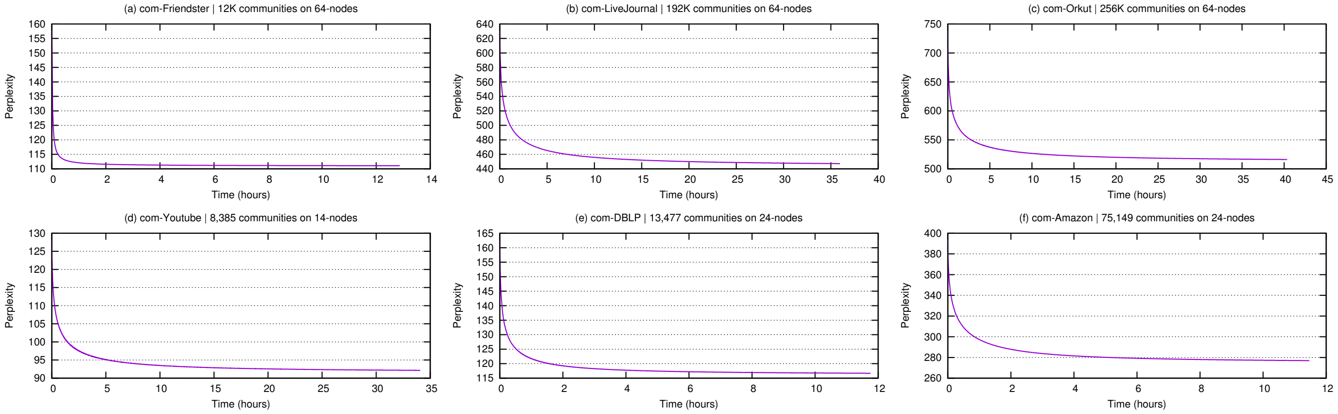

Figure 6. Convergence time of 6 different data sets.

[28] A. Dragojević, D. Narayanan, M. Castro, and O. Hodson, "Farm: Fast remote memory," in *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. Seattle, WA: USENIX Association, Apr. 2014, pp. 401–414. [Online]. Available: https://www.usenix.org/conference/nsdi14/technical-sessions/dragojević

[29] A. Kalia, M. Kaminsky, and D. G. Andersen, "Using RDMA efficiently for key-value services," *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 4, pp. 295–306, Aug. 2014. [Online]. Available: http://doi.acm.org/10.1145/2740070.2626299

[30] D. Panda, K. Tomko, K. Schulz, and A. Majumdar, "The MVAPICH project: Evolution and sustainability of an open source production quality MPI library for HPC," November 2013.

[31] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection," http://snap.stanford.edu/data, Jun. 2014.