Distributed Large-scale Natural Graph Factorization[*]

Amr Ahmed
Google
amra@google.com

Nino Shervashidze
INRIA, ENS
nino.shervashidze@inria.fr

Shravan Narayanamurthy
Microsoft Bangalore
shravanmn@gmail.com

Vanja Josifovski
Google
vanjaj@google.com

Alexander J. Smola
Carnegie Mellon University
and Google
alex@smola.org

ABSTRACT

Natural graphs, such as social networks, email graphs, or instant messaging patterns, have become pervasive through the internet. These graphs are massive, often containing hundreds of millions of nodes and billions of edges. While some theoretical models have been proposed to study such graphs, their analysis is still difficult due to the scale and nature of the data.

We propose a framework for large-scale graph decomposition and inference. To resolve the scale, our framework is distributed so that the data are partitioned over a shared-nothing set of machines. We propose a novel factorization technique that relies on partitioning a graph so as to minimize the number of neighboring *vertices* rather than *edges* across partitions. Our decomposition is based on a streaming algorithm. It is network-aware as it adapts to the network topology of the underlying computational hardware. We use local copies of the variables and an efficient asynchronous communication protocol to synchronize the replicated values in order to perform most of the computation without having to incur the cost of network communication. On a graph of *200 million* vertices and *10 billion* edges, derived from an email communication network, our algorithm retains convergence properties while allowing for almost linear scalability in the number of computers.

Categories and Subject Descriptors

[I.2.6] [**Artificial Intelligence**]: Learning – Parameter Learning; [E.1] [**Data Structures**]: Graphs and Networks

Keywords

Large-scale Machine Learning; Distributed Optimization; Graph Factorization; Matrix Factorization; Asynchronous Algorithms; Graph Algorithms; Graph Partitioning

1. INTRODUCTION

Very large natural graphs are common on the internet, particularly in the context of social networks. For instance, Facebook has at present in excess of 900M registered users. Similarly large graphs can be found among email and instant messaging graphs. Online recommendation for products and connections, as well as online advertising, clearly rely on the analysis of such networks, which makes inference on very large natural graphs a pressing problem. However, this problem is hard and cannot at present be solved on a single machine. For inference problems at the scale of billions of edges it is imperative to use a distributed setting where the graph is partitioned over a number of machines.

[*]The research was performed while all authors were at Yahoo Research.

1.1 Challenges

We examine the challenges in applying distributed latent variable models to graph factorization. Latent variable modeling is a promising technique for many analytics and predictive inference applications. However, parallelization of such models is difficult since many latent variable models require frequent synchronization of their state. The power-law nature of such graphs makes it difficult to use chromatic scheduling [18]. Furthermore, the bulk-synchronous processing paradigm of Map-Reduce [19] does not afford low-enough latency for fast convergence: this has been reported, e.g. in comparisons between bulk-synchronous convergence [20] and asynchronous convergence [23]. Consequently there is a considerable need for algorithms which address the following issues when performing inference on large natural graphs:

Graph Partitioning We need to find a communication-efficient partitioning of the graph in such a manner as to ensure that the number of neighboring *vertices* rather than the number of edges is minimized. This is relevant since latent variable models and their inference algorithms store and exchange parameters that are associated with vertices rather than edges [1].

Network Topology In many graph-based applications the cost of communication (and to some extent also computation) dwarfs the cost of storing data. Hence it is desirable to have an algorithm which is capable to layout data in a network-friendly fashion *on the fly* once we know the computational resources.

Variable Replication While the problem of variable synchronization for statistical inference with regular structure is by now well understood, the problem for graphs is more complex: The state space is much larger (each vertex holds parts of a state), rendering synchronization much more costly – unlike in aspect models [8] only few variables are global for all partitions.

Asynchronous Communication Finally there is the problem of eliminating the synchronization step in traditional bulk-synchronous systems [19, 21] on graphs. More specifically, uneven load distribution can lead to considerable inefficiencies in the bulk synchronous setting [24]. After all, it is the slowest machine that determines the runtime of each processing round (e.g. MapReduce). Asynchronous schemes, on the other hand, are nontrivial to implement as they often require elaborate locking and scheduling strategies.

Large-scale matrix factorization has received considerable attention in data mining and machine learning communities in recent years due to the proliferation of tasks such as recommendation (e.g., a matrix of movies vs. users), news personalization (users vs. news items), or keyword search (terms vs. documents). Several sequential and distributed algorithms have been proposed to tackle this problem over the last four years [11, 17, 25, 26, 16]. We use factorization as our test case and we relate to these approaches in Section 8.

1.2 Main Contributions

We address the issues raised in the previous section in a systematic fashion. To validate our strategy we apply our algorithm to a social graph with 200 million vertices and 10 billion edges. We show how this graph can be factorized in parallel on hundreds of computers performing distributed asynchronous optimization. We tackle the issues raised above as follows:

Graph Partitioning We propose a streaming algorithm for graph partitioning to ensure that the number of neighbors for any given partition does not exceed a machine-specific capacity limit. The algorithm proceeds by greedily adding vertices to partitions such as to minimize the amount of additional storage required per vertex.

Network Topology The data layout problem at runtime can be expressed as a quadratic assignment problem (QAP), which is NP-hard. We experimentally show that explicitly addressing the data layout problem by (approximately) solving the QAP significantly minimizes the cost of communication.

Variable Replication We resort to a global consensus formulation [6, 9]: we create local copies of global variables and optimize over the slightly relaxed optimization problem jointly.

Asynchronous Communication Unlike classical MM [6] or ADMM [9] methods for relaxation which alternate between explicit local and global steps for optimization, we perform integrated updates. This eliminates an explicit global update phase. Global constraints are updated asynchronously and independently.

Outline. Section 2 formally defines our optimization problem and its distributed variant. We propose a synchronous algorithm in Section 3 and an asynchronous algorithm in Section 4. Section 5 discusses large-scale graph partitioning and Section 6 details a novel approach to distribute parts of the graph over cluster nodes. Experiments on large natural graphs are presented in Section 7, followed by a discussion of related work in Section 8, and we conclude in Section 9.

2. GRAPH FACTORIZATION

Many social networks have an undirected graph as one of their core data structures. Graph factorization allows us to find latent (unobserved) factors associated with vertices (users) which can be used for further inference in the network. Furthermore, graph factorization is a test bed for considerably more sophisticated algorithms, such as nonparametric latent variable models. Hence it is highly desirable to solve the factorization problem efficiently.

We begin by specifying our notation: We are given an undirected graph $G = (V, E)$ with n nodes and m edges. In this paper, n is in the order of $10^8 - 10^9$ vertices and m is in the order of 10^{10} edges, hence typically $m = O(n)$. For a node $v \in V$, denote by $\mathcal{N}(v)$ the set of its neighbors, that is, $\{u \in V | (u, v) \in E\}$. The adjacency matrix of G, $Y \in \mathbb{R}^{n\times n}$ is symmetric, and our goal is to find a matrix $Z \in \mathbb{R}^{n\times r}$, with a given $r \ll n$, such that ZZ^T is close to Y in terms of observed (non-zero) entries.

G	the given graph
$n = \|V\|$	number of nodes
$m = \|E\|$	number of edges
$\mathcal{N}(v)$	set of neighbors of node v
$Y \in \mathbb{R}^{n\times n}$	Y_{ij} is the weight on edge between i and j
$Z \in \mathbb{R}^{n\times r}$	factor matrix with vector Z_i for node i
$X_i^{(k)} \in \mathbb{R}^r$	idem, local to machine k
λ, μ	regularization parameters
$\{O_1, \ldots, O_K\}$	pairwise disjoint partitions of V
B_k	$\{v \in V \| v \notin O_k, \exists u \in O_k, (u, v) \in E\}$
V_k	$O_k \cup B_k$

Objective

There exists a wide range of objective functions for graph reconstruction and factorization [13]. Many of them share the property that we have vertex attributes Z_i which can be used to determine whether an edge (i, j) should be established. This is also a consequence of the Aldous-Hoover [3] theorem. A discussion of this rich variety of models is beyond the scope of the current paper.

For the purpose of demonstrating the feasibility of our model we restrict ourselves to a simple inner product model where we assume that information regarding the presence of an edge Y_{ij} can be captured efficiently via $\langle Z_i, Z_j\rangle$. Moreover, we impose a small amount of regularization to ensure that the problem remains well-posed even in the absence sufficient data. Note that the algorithms we describe *apply* to a much larger family of models. However, for the benefit of a focused and streamlined presentation we limit ourselves to a simple regularized Gaussian matrix factorization. More advanced models are subject to research and will be reported elsewhere. We use the following objective to recover Z.

$$f(Y, Z, \lambda) = \frac{1}{2} \sum_{(i,j)\in E} \big(Y_{ij} - \langle Z_i, Z_j\rangle\big)^2 + \frac{\lambda}{2} \sum_i \|Z_i\|^2 \quad (1)$$

The gradient of f with respect to the row i of Z is given by

$$\frac{\partial f}{\partial Z_i} = - \sum_{j\in\mathcal{N}(i)} \big(Y_{ij} - \langle Z_i, Z_j\rangle\big)Z_j + \lambda Z_i \quad (2)$$

For a pair $(i, j) \in E$ this amounts to

$$- \big(Y_{ij} - \langle Z_i, Z_j\rangle\big)Z_j - \lambda Z_i. \quad (3)$$

It is straightforward to include other variables, such as per-node and common additive biases. Stochastic gradient descent is a common way of solving this nonconvex problem. Algorithm 1 gives the pseudocode for this procedure for a

Algorithm 1 Sequential stochastic gradient descent

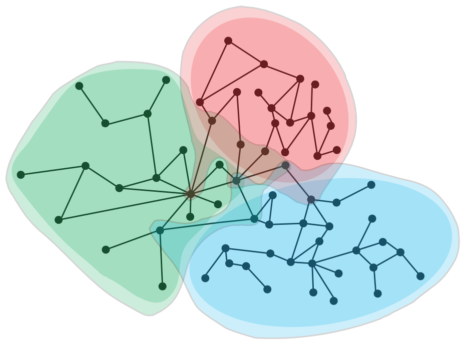
```
Require: Matrix $Y \in \mathbb{R}^{n\times n}$, rank $r$, accuracy $\epsilon$
Ensure: Find a local minimum of (1)
 1: Initialize $Z' \in \mathbb{R}^{n\times r} \in$ at random
 2: $t \leftarrow 1$
 3: repeat
 4:     $Z' \leftarrow Z$
 5:     for all edges $(i,j) \in E$ do
 6:         $\eta \leftarrow \frac{1}{\sqrt{t}}$
 7:         $t \leftarrow t+1$
 8:         $Z_i \leftarrow Z_i + \eta[(Y_{ij} - \langle Z_i, Z_j\rangle)Z_j + \lambda Z_i]$
 9:     end for
10: until $\|Z - Z'\|^2_{\text{Frob}} \leq \epsilon$
11: return $Z$
```

single machine. By construction its runtime complexity is linear in the number of iterations and edges m in G.

Distributing the Optimization

Since the algorithm requires access to a matrix, the two-dimensional pattern of access will result in non-sequential I/O if the data were to be stored on disk. Thus, in our solution we aim to keep the data in main memory. Whenever both the number of vertices is large (e.g. 10^8 users) and the attribute space is nontrivial (e.g. hundreds of bytes), it is impossible to store the full graph. It would take 10GB to 1TB for efficient representation. Hence, we need to distribute data on a cluster of workstations where each one retains parts of the global state. This raises some issues:

- We need to select subsets of vertices on each machine, such that the partial views of the graph fit on it.
- We need to keep solutions on subsets from diverging.
- We need an efficient communications layer for speed.



We achieve this by partitioning data such as to minimize the number of neighboring vertices rather than minimizing the number of edge cuts. For now we assume that we have a partitioning of the graph into non-overlapping clusters of nodes $O_k \subseteq V$ such that the union V_k of each cluster with the set of its direct neighbors, $V_k = O_k \cup B_k$ where $B_k = \{v \in V | v \notin O_k, \exists u \in O_k, (u,v) \in E\}$, fits on a single machine.

We describe a partitioning algorithm in Section 5. We call nodes in O_k *owned* and nodes in B_k *borrowed* by machine k. In the diagram above darker shapes represent non-overlapping partitions O_k, and lighter shapes encompassing them depict V_k, the overlapping extensions of O_k obtained by including their neighboring nodes.

Distributed Objective

Each machine can optimize the objective restricted to the cluster of nodes assigned to it, V_k, but the solutions obtained by different machines have to be encouraged to match on nodes which are shared between clusters, B_k. We borrow intuition from the theory of dual multipliers [9, 6]. That is, we use variables $X_i^{(k)}$ on machine k to act as a proxy for Z_i. If we were to enforce

$$\sum_{k=1}^{K} \sum_{i \in V_k} \left\| Z_i - X_i^{(k)} \right\|^2 = 0 \tag{4}$$

the decomposed problem would be equivalent to the original problem of solving (1). Since this is clearly impossible, we relax the above constraints and add them with a suitable Lagrange multiplier μ to the objective. This yields the Lagrange function (and distributed objective)

$$\begin{aligned} & L(Y, Z, X^{(1)}, \ldots, X^{(K)}, \lambda, \mu) \qquad (5)\\ =& \sum_{k=1}^{K} f_k(Y, X^{(k)}, \lambda) + \frac{1}{2}\sum_{k=1}^{K}\left[\mu \sum_{i\in V_k} \|Z_i - X_i^{(k)}\|^2\right] \end{aligned}$$

$$\begin{aligned} \text{where } & f_k(Y, X^{(k)}, \lambda) \qquad (6)\\ =& \frac{1}{2} \sum_{\substack{(i,j)\in E,\\ i,j \in V_k}} \left(Y_{ij} - \langle X_i^{(k)}, X_j^{(k)}\rangle\right)^2 + \frac{\lambda}{2}\sum_{i\in V_k} \|X_i^{(k)}\|^2. \end{aligned}$$

Dual ascent in the Lagrange function L with respect to μ means that we end up increasing μ commensurate to the extent to which the equality constraint is violated. That said, if we were to start with a large value of μ the problem would not be numerically well-conditioned in X and Z, hence this only makes sense once we are close to optimality. Note that here the goal is to learn Z and the copies $X^{(k)}$ serve as auxiliary parameters. Updates in μ can be carried out, e.g. by dual gradient ascent. In the following sections we will describe two approaches of optimizing this objective.

3. BASELINE: SYNCHRONOUS OPTIMIZATION

As a baseline to solve the above optimization problem, we follow [7, 9] and use the following synchronous block-minimization strategy for minimizing L with respect to X, Z: We randomly initialize Z and set $X = Z$ on all machines; Subsequently we alternate until convergence:

Latent Parameter Update: Optimize X for fixed Z. This step decomposes over all machines and can be done in parallel. Each machine minimizes (7). This modifies only the local copies of $X_i^{(k)}$ for all $i \in V_k$

$$\underset{X^{(k)}}{\text{minimize}}\, f_k(Y, X^{(k)}, \lambda) + \frac{1}{2}\mu \sum_{i\in V_k} \|Z_i - X_i^{(k)}\|^2 \tag{7}$$

These updates are analogous to standard matrix factorization and can be obtained using stochastic gradient descent (SGD) with the following gradient for $X_i^{(k)}$:

$$\begin{aligned} \frac{\partial f}{\partial X_i^{(k)}} =& -\sum_{j\in \mathcal{N}(i)} \left(Y_{ij} - \langle X_i^{(k)}, X_j^{(k)}\rangle\right) X_j^{(k)} \\ & + \lambda X_i^{(k)} + \mu(X_i^{(k)} - Z_i). \end{aligned} \tag{8}$$

Algorithm 2 Synchronous Optimization

Require: Matrix $Y \in \mathbb{R}^{n \times n}$, rank r, accuracy ϵ
Ensure: Find a local minimum of (5) over X and Z

```
1: Global Server
2: Initialize $Z \in \mathbb{R}^{n \times r}$ at random
3: repeat
4:     $Z' \leftarrow Z$
5:     for all $k$ do send $Z_{v_x}$ to machine $k$
6:     Wait until all machines finish local optimization
7:     for all $k$ do receive $X^{(x)}$ from machine $k$
8:     Update $Z$ by solving (9)
9: until $\sum_i \|Z'_i - Z_i\|^2 \leq \epsilon$
10: Send termination signal to all machines
11: Machine $k$
12: repeat
13:     Receive global factors $Z_{v_k}$ and match $X^{(k)}$ to it
14:     repeat
15:         $X^{(k)'} \leftarrow X^{(k)}$
16:         for all nodes $i \in v_k$ do update $X_i^{(x)}$ using (8)
17:     until $\sum_i \|X_i^{(k)'} - X_i^{(k)}\|^2 \leq \epsilon$
18:     Send $X^{(k)}$ to the global server
19: until Global server signals termination
```

Latent Synchronizer Update: A second phase minimizes Z given X. This is a global update step:

$$\underset{Z}{\text{minimize}} \frac{1}{2} \sum_{k=1}^{K} \left[\mu \sum_{i \in V_k} \|Z_i - X_i^{(k)}\|^2 \right] \tag{9}$$

This term has a closed form solution, simply by setting Z_i to the average of each machine's local copy. In other words, the different copies of each latent factor are averaged to produce a new Z which is again distributed to all machines.

Algorithm 2 describes a client-server architecture which can be easily mapped to MapReduce. The drawback of the synchronous optimization procedure is that at each step, all machines must wait until every machine has finished computation. Consequently the slowest machine will determine the execution speed of the algorithm — a problem known as the curse of the last reducer [24].

4. ASYNCHRONOUS OPTIMIZATION

While the synchronous variant has merit in terms of being easy to implement, it suffers from rather middling speed of convergence. This is due to both the systems-specific issues of having to wait for the slowest machine *and* the fact that aggressive synchronization makes for slow convergence. The key idea is to carry out stochastic gradient descent on $X^{(k)}$ and Z jointly. That is, we repeat the following steps in machine k: pick a node i; update its factors $X_i^{(k)}$, and its corresponding global versions Z_i and repeat until convergence. The part of the gradient for node Z_i restricted to its local version at machine k, $X_i^{(k)}$, is given by

$$\frac{\partial f}{\partial Z_i}\left[X_i^{(k)}\right] = \mu(Z_i - X_i^{(k)}). \tag{10}$$

There are several difficulties with the above approach. First, the global factors Z_i are not stored in machine k and might be stored in a rather distant machine. Thus a naive implementation of this idea would proceed as follows: First pick node i in machine k, obtain its global version Z_i from the global server and lock it, update $X_i^{(k)}$ and Z_i using a gradient step with the gradient from (10), send the updated Z_i to the global server and unlock it. Clearly, this approach is rather inefficient and would be slower than the synchronous version described in Section 3. A possible solution is for machine k to *cache* the value of the global variable locally and use it while refreshing its cache from the global server periodically. If we let $Z_i^{(k)}$ denote the cached value of the global variable Z_i at machine k, then the asynchronous method proceeds as follows: For each node i in machine k, first update $X_i^{(k)}$ using

$$\begin{aligned}\frac{\partial f}{\partial X_i^{(k)}} = &- \sum_{j \in \mathcal{N}(i)} \left(Y_{ij} - \langle X_i^{(k)}, X_j^{(k)} \rangle\right) X_j^{(k)} \\ &+ \lambda X_i^{(k)} + \mu(X_i^{(k)} - Z_i^{(k)}).\end{aligned} \tag{11}$$

Now we still need to update the value of the global variable and refresh the local copy of the global variable. Before specifying these details, we first describe the architecture and how the global variables are distributed across machines, and then give the full algorithm in Section 4.3.

4.1 Client Architecture

Recall that we assume to have a suitable partitioning of the graph into K partitions which are mapped to K machines. Each machine $k \in \{1 \ldots K\}$ runs a client processing the nodes assigned to it. Edge information via Y_{ij} is ingested by optimizer threads in parallel. An updater thread incorporates the changes using (11) to shared memory. This decoupling is useful since this way the optimizer threads, which are computationally more expensive, do not need to acquire write locks to incorporate changes. Instead, all writes are delegated to a separate thread. Occasionally checkpoints are committed to file. This ensures fault tolerance.

Each client holds both the local copy $X_i^{(k)}$ and the cached values of the global variables, $Z_i^{(k)}$. The synchronizer thread takes care of refreshing the cached copies of the global variables periodically. It executes a gradient descent step on Z using (10). The overall client architecture resembles that of [1, 23], however the modules here execute different functions as described in Section 4.3.

4.2 Global Variable Distribution

To distribute the global variables among a set of servers we use the ICE asynchronous RPC server by www.zeroc.com. Each machine k executes a client and a server program. We distribute the global variables among the servers such that the server at machine k is assigned the global variables corresponding to the nodes owned by the partition k, O_k. This enables efficient communication since, among all clients, client k has to be the one requesting global copies of nodes in O_k most frequently when updating its local copies.

Note that other clients will also require access to nodes in O_k if these nodes appear in their *borrowed* sets. It is important here to examine the amount of replication in our solutions: As $\{O_1, \ldots, O_K\}$ are assumed to be non-overlapping partitions of G, we have $n = |V| = \sum_k |O_k|$. Denote by $N := \sum_k |B_k|$ the total number of *borrowed* nodes. For each client k, we need to keep $2(|O_k| + |B_k|)$ copies of factor

Algorithm 3 Asynchronous Distributed Optimization

```
Require: Matrix $Y \in \mathbb{R}^{n \times n}$, rank $r$, accuracy $\epsilon$
Ensure: Find a local minimum of (5) over $X$ and $Z$
 1: Initialization
 2: for all $i$ do Initialize $Z_i$ randomly
 3: for all $k, i \in O_k$ do $X_i^{(k)} \leftarrow Z_i$
 4: for all $k, i \in B_k$ do $X_i^{(k)} \leftarrow Z_i^{(k)} \leftarrow Z_i$
 5: Client $k$ Optimizer Thread
 6: repeat
 7:   for all nodes $i \in V_k$ do
 8:     Update $X_i^{(x)}$ using the gradient from (11)
 9:   end for
10: until Termination
11: Client $k$ Send Thread
12: repeat
13:   for all nodes $i \in V_k$ do
14:     Send $X_i^{(k)}$ to the Global server
15:   end for
16: until Termination
17: Client $k$ receive $(Z_i)$
18: $Z_i^{(k)} \leftarrow Z_i$
19: Global Server Receive $(X_i^{(k)})$
20: Update $Z_i$ using the gradient form (10)
21: Send $Z_i$ to machine $k$
```

vectors: one for the local copies $X_i^{(k)}$ and one for the cached copies of the global variables, $Z_i^{(k)}$, for $i \in O_k \cup B_k$. This amounts to $2n + 2N$ variables in $\mathbb{R}^r$. We then add to them n global variables, which makes the total memory consumption $(3n+2N)r$. This is the same order of magnitude (albeit slightly more) as for the synchronous optimizer in Section 3.

4.3 Communication Protocol and Convergence

The details of the optimization strategy are given in Algorithm 3. The optimizer thread in the client is rather straightforward. Hence we focus here on the synchronization threads: Their goal is to execute the gradient step over the global variable Z_i and to refresh the local copies of the global variables, $Z_i^{(k)}$. We use asynchronous communication as follows: Each client executes a thread that loops through the client local memory storing $X^{(k)})_i$. For each node i, it sends this local copy to the server holding the global variable Z_i. When the server receives this values, it performs a gradient step using (10) and returns the new value of Z_i. Upon reception of this value, the client synchronization thread updates the cached copy $Z_i^{(k)}$ with this new value which is then used by the optimizer thread. All these steps are done in parallel. This is analogous to the lock-free SGD [22], albeit in a distributed setting.

We denote by the *synchronization time* the time needed to move the local copies in machine k to the global servers and receive a response from the global sever. This is the time needed to perform a synchronization pass. Niu et al. [22] showed that the **convergence rate** of the lock-free algorithm is inversely proportional to the synchronization time, and the number of overlapped nodes. *Our goal is thus to accelerate convergence as much as possible*, and we tackle this problem by partitioning the graph in order to minimize the number of overlapped nodes (Section 5) and by allocating partitions to machines taking into consideration the inter-partition overlap which dictates the communication bandwidth needed between any two partitions (Section 6) and thus *minimize synchronization time.*

5. GRAPH PARTITIONING

5.1 Motivation

As discussed above, we need an efficient algorithm to partition very large graphs into portions that are small enough to fit onto individual machines. For our purpose it matters that we perform *vertex* partitioning of the graph, that is, we need to minimize the number of neighboring vertices rather than the number of edges that are cut: The bulk of the computational cost is incurred by updating the local factors $X_i^{(k)}$ and updating the global factors Z_i. Both updates require access to the up-to-date value of the global factor, thereby inducing cross-node communication. This increases the cost of the computation. For all nodes $\{u \in O_k | \forall v \in \mathcal{N}(u), v \in O_k\}$ that reside on the same machine with all their neighbors, the changes affect only local factors - no synchronization with the global factors outside the same machine is required[1].

In summary, we would like to find a partitioning that reduces the total number of *borrowed* nodes, that is, the sum of $|B_k|$ over all partitions k. Moreover, since the nodes in B_k are replicated across machines and as the number of nodes in each partition is constrained by the size of the main memory, reducing their number increases the throughput of the whole architecture and requires fewer partitions.

Ideally, each machine would hold one or several connected components of the graph, thereby avoiding communication with other machines. This would require the largest connected component and all the variables associated with it to easily fit on one machine. However, this is unlikely to happen in natural graphs, in which the largest connected component usually contains a very large fraction of nodes: In the email graph used in our experiments the largest connected component contains more than 90% of all nodes.

More formally, we can postulate the partitioning of the graph nodes V into $\{O_1, \ldots, O_K\}$ with all O_k pairwise disjoint and $\bigcup_k O_k = V$, as a constrained optimization problem:

$$\underset{O_1,\ldots,O_K}{\text{minimize}} \sum_{k=1}^{K} |B_k| \tag{12}$$
$$\text{subject to } O_i \cap O_j = \emptyset \text{ and } |O_k \cup B_k| \leq \text{capacity}$$

Such balanced graph partitioning is an NP-hard problem [5]. There exist efficient heuristics, such as METIS [15], and its parallel version, ParMETIS, that can be configured to return partitions with size not exceeding a given threshold. Unfortunately ParMETIS *did not scale to 10 Billion* edges. To address this we propose two algorithms: a flat greedy partitioning and a faster hierarchical greedy strategy.

5.2 A Greedy Single-Pass Algorithm

Assume that the graph is stored on a disk as an adjacency list with each line containing a vertex and a list of its

[1]While an update to every node i requires access to the global factor of node i, if i is owned by a given machine, say m, and not borrowed by any other machine, then the global factor will reside on machine m and be only updated by machine m, which speeds up convergence.

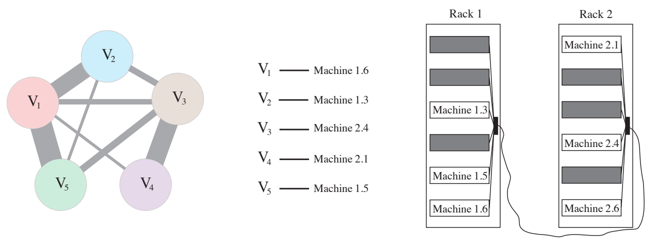


Figure 1: Left: nodes correspond to overlapping partitions $V_k = O_k \cup B_k$ of the graph. The thickness of the edges is commensurate with how much is shared between nodes $|V_i \cap V_j|$. Right: two racks with 3 busy (shaded) and 3 available machines each. Communication time between racks is slower than within a rack. Middle: assignment of the partitions to machines to minimize communication cost.

neighbors. Our algorithm consumes this list in a streaming fashion, requiring only a single pass. It proceeds as follows:

- We keep a set of open partitions in memory, corresponding to O_k and B_k for $k \in \{1, \ldots K\}$. These are dynamically updated as we receive data.
- A partition is considered open if $|B_k| + |O_k| \leq$ capacity. That is, if it can still accommodate additional nodes. Note that once a partition k is closed, the sets B_k and O_k remain automatically fixed since B_k already contains all neighbors of O_k.
- Once a vertex i is read from file we have to assign it to an open partition that will not overflow if we add i to it. We first compute the overhead of adding node i to each open partition, say partition k, as follows: Node i would increase the size of O_k by one. However, B_k would change based on how many nodes from the neighborhood of i, $\mathcal{N}(i)$, are already in k. We count the extra nodes added to B_k as a result of adding i to k for all open partition k and we choose to add node i to partition k with the minimum increase in B_k. In case of a tie, we favor the partition with the smallest number of total borrowed nodes, $|B_k|$.
- If no open partition can accommodate node i, for instance, if adding i would make any of the open partitions overflow, we increase the total number of partitions in memory by δK.

For efficiency, we maintain O_k and B_k as bit vectors of length n where n is the total number of nodes in the graph. We implemented a multi-threaded version of the algorithm on an 8-core machine. In our experiments, it took 166 minutes to partition a graph with 200 million nodes and 10 billion edges. It results in 150 partitions, each of them holding at most 3 million nodes. Note that any advances in the area of vertex partitioning graphs can be utilized in our framework.

5.3 A Hierarchical Approximation

The dominant cost in generating the partitioning is to compute the set of candidate improvements for each vertex. This requires search over all O_k and B_k for all neighbors of a given vertex. Hence, if the total number of partitions is small, should be able to perform this step faster. This suggests a hierarchical extension of the above *flat* algorithm: First partition into a set of larger supernodes and then repartition the latter again. We considered the following variants:

Hierarchical (Hier) We run partitioning to obtain a set of H splits. Then we re-partition each of the H splits.

Random (HierRandom) We randomly split the nodes of the graph into H parts and repartition the result.

Locality Sensitive Hashing (HierLSH) We use LSH [10] to group nodes into H partitions. We consider each row of the connectivity matrix as a vector and hash it into a b-dimensional bitvector with $b \ll n$ using the inner product representation of [10]. Subsequently we partition the hash range in H buckets and partition based on proximity to the buckets. This operation is very fast as it only involves bit-wise comparisons.

6. ONLINE DATA LAYOUT

So far we discussed offline performance optimization, that is, optimization prior to requesting computers from a resource pool. Many server centers have nonuniform network layout, that is, the bandwidth within a rack is considerably higher than between racks, due to oversubscription of top-of-rack routers. Hence it pays to ensure that partitions with the largest overlap share the same rack whereas more loosely connected partitions are best relegated to different racks, as described in Figure 1.

As we shall see, this leads to a quadratic assignment problem. While such problems are NP-hard, it pays to make at least *some* progress towards optimality rather than picking an entirely random configuration. More formally, the communication required between two machines holding V_k and V_l respectively is given by the amount of updates on nodes belonging to one partition and borrowed by the other. That is, it amounts to

$$C_{kl} = |O_k \cap B_l| + |O_l \cap B_k| \text{ for } k \neq l \text{ and } C_{kk} = |O_k| + |B_k|.$$

It follows by Gerschgorin's theorem [12] that $C \succeq 0$ since it is diagonally dominant.[2] The second matrix of interest in our case is the communication time matrix. The latter is given by the inverse point-to-point bandwidth.

$$D_{kl} = \frac{1}{\mathrm{bandwidth}(k,l)} \text{ for } k \neq l \text{ and } D_{kk} = 0 \text{ otherwise.}$$

Hence the aggregate time required to synchronize all vertex partitions is given by

$$\sum_{kl} D_{kl} C_{kl} = \mathrm{tr}\, DC \tag{13}$$

Now denote by π a permutation matrix encoding an assignment of partitions V_k to machines l, i.e., $\pi_{kl} = 1$. With slight abuse of notation we also denote this by $\pi(k) = l$. In this case the aggregate time required to synchronize data becomes machines is given by

$$T(\pi) = \mathrm{tr}\, C\pi D\pi^\top. \tag{14}$$

Maximizing $T(\pi)$ amounts to a quadratic assignment problem and is NP-hard in general, even for approximations. Nonetheless, optimizing for π is always better than using the random configuration that the system assigns. We use an off-the-self heuristic QAP solver as a preprocessing step.

[2]The actual value of C_{kk} is immaterial for the purpose of optimization. However, it just serves to make the optimization problem more tractable since it will allow us to lower-bound the arising quadratic assignment problem.

To summarize, our architecture proceeds as follows: We first partition the graph into K partitions. Then we obtain K machines from the cloud. We query the bandwidth between each pair of machines and then let a single machine solve the QAP above to determine the optimal partition layout. This layout is then propagated to all machines to let each machine know it assigned partition. Afterwards, Algorithm 3 is executed until convergence. The time needed to solve the QAP is negligible and in the order of a few seconds.

7. EXPERIMENTS

7.1 Data

To demonstrate our ideas we recorded the volume of email exchanges between all pairs of nodes in a subset of users of Yahoo! Mail during a nondisclosed period of time. The graph contains only the users who sent at least one message/email during the period under consideration. Edges $Y_{ij} = Y_{ji} = 1$ and between users i and j were added if user i contacted user j. The weight of the edge is commensurate with the amount of communications between the users. The graph has *200 million* nodes and *10 billion* edges. The largest connected component contains more than 90% of all nodes. The task we consider in our experiments is to predict the volume of email exchange between held-out test pairs of users selected uniformly at random from the whole graph.

It is important to stress that we use this task purely as a test bed for our framework. A plethora of methods for solving this specific task exist and will be studied in subsequent work. The point of our experiments is simply to show that the distributed factorization framework is effective.

7.2 Experimental Setup

To evaluate convergence we compare three systems: a single machine setup, a synchronous parallel implementation and an asynchronous implementation.

Single Machine is used as a baseline on a subset of the graph since the entire graph would not fit into a single computer. It employs the standard matrix factorization algorithm described in Section 2.

Baseline: Parallel Synchronous implements the parallel baseline synchronous algorithm of Section 3. This baseline is similar to ADMM of [9]. For fairness, to avoid the wasteful overhead of Hadoop's MapReduce framework we used the same architecture as described in Section 4. This has the advantage of keeping graph data local between iterations. Moreover it eliminates the need to allocate mappers at each iteration, which is an expensive operation.

Parallel Asynchronous implements the asynchronous algorithm described in Section 4. Unless otherwise stated, we use the flat streaming partitioning algorithm from Section 5.2 and solve a quadratic assignment problem (QAP) for improved task layout over machines.

Unless otherwise stated, we use $r = 50$ as the number of factors, and tune the regularization parameters and learning rate on a small subset of the graph using the single machine baseline. The experiments aim to answer the following:

- How does the quality of the solution of the distributed algorithm compare to that of the single-machine algorithm?

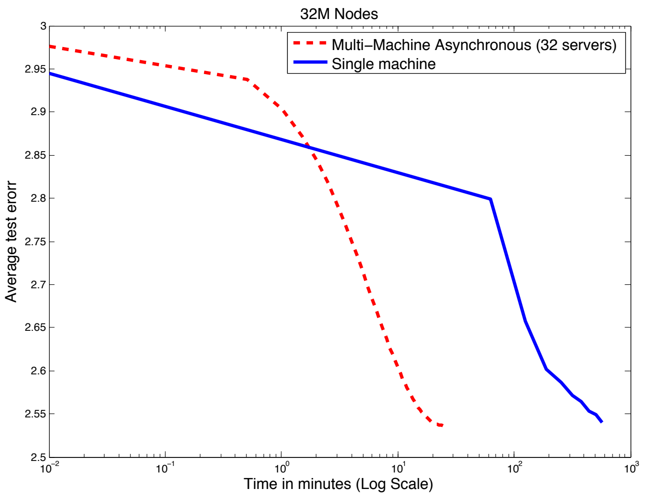


Figure 2: Convergence results on a subgraph of 32 Million nodes using 20 factors. Solid line: solution quality obtained by a single machine. Dashed line: solution quality obtained by asynchronous optimization using 32 machines.

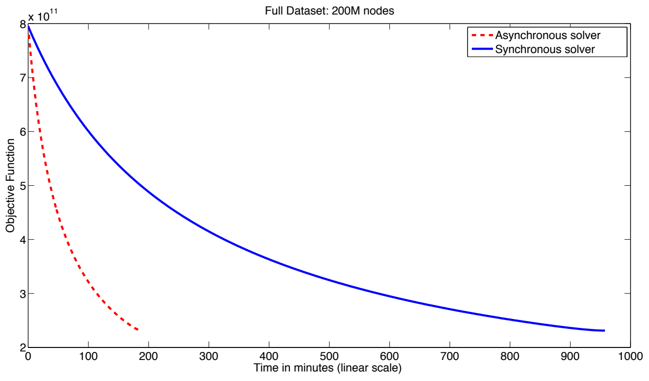


Figure 3: Convergence for synchronous (baseline) vs. asynchronous optimization using the full graph.

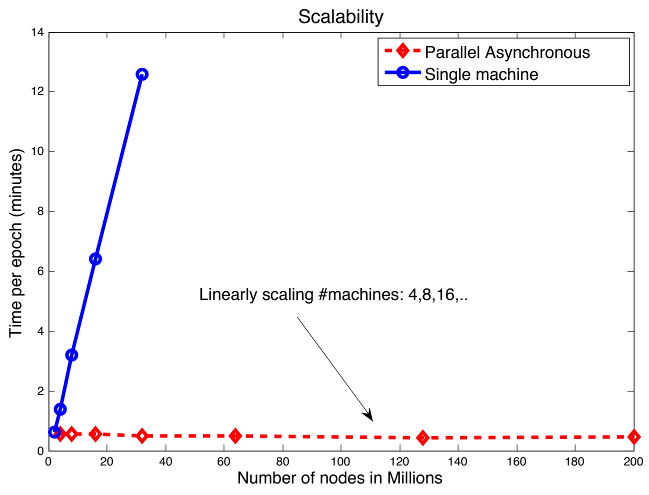


Figure 4: Scaling behavior as a function of the amount of data. For a fixed number of machines the time per epoch increases linearly whereas for fixed amount of data per machine time remains constant.

- How does asynchronous optimization affect convergence?
- How does the algorithm scale in terms of computation and communication with increasing graph size?
- What is the added value of the data layout algorithm?
- What is the effect of the quality of graph partitioning on the performance of the system?

7.3 Solution Quality

To assess the quality of the solution obtained from the asynchronous multi-machine algorithm, we compare its test error over the held-out dataset to the error obtained using the single-machine algorithm. To be able to perform optimization in a single machine, we randomly chose a subgraph of 32 million nodes and used 20 factors to be able to fit it in the main memory of the machine.

Figure 2 presents the evolution of the test error as training proceeds in both systems. While the convergence of the test error in the first minutes seems to be faster in a single machine, later this changes dramatically and the multi-machine asynchronous setting achieves convergence more than an order of magnitude faster than the single-machine scheme. The reason for this initial behavior is due to the time needed to initially synchronize the factors across machines. The final test errors achieved by both methods are identical.

To compare the synchronous and asynchronous (both multi-machine) alternatives we use the full 200M graph and set the number of factors to 50. Figure 3 shows the convergence curve of the objective function (i.e., the training error) of the two approaches[3]. As we can see from this figure, the asynchronous algorithm is almost an order of magnitude faster than the synchronous version. As noted earlier, the reason for this happening is that the synchronous algorithm is as fast as the slowest machine due to the barrier effect enforced before updating the global factors in each iteration. The asynchronous algorithm does not suffer from this effect. Finally, we note that the quality of the solution obtained by the two systems is again almost indistinguishable.

7.4 Communication and Scalability

We first assess how the runtime changes as a function of the graph size. Linear scaling is nontrivial — many algorithms do not slow down even if the amount of resources scales with the problem size. We show that our algorithm indeed scales linearly up to graphs with 200 million nodes. Figure 4 presents the time required for a single iteration of stochastic gradient descent (i.e., traverse all edges and update parameters of their endpoints). For each point in Figure 4 we run flat streaming partitioning algorithm. The number of machines scales linearly with the number of nodes in the graph.

As can be seen from Figure 4, the runtime for a single machine is linear in the amount of data. This is expected, as the complexity of the stochastic gradient descent is linear in the number of edges in the graph G, and natural graphs are often sparse. Note that we cannot handle more than 32 million vertices on a single machine. On multiple machines, we observe a constant line, indicating that our architecture can handle hundreds of millions of vertices without any problem. For all graph sizes up to 32 millions of nodes, the test

[3]It is valid to compare the two algorithms using the training error since it is computed using the global factors and the comparison is done using the same number of factors, so there is no danger of overfitting.

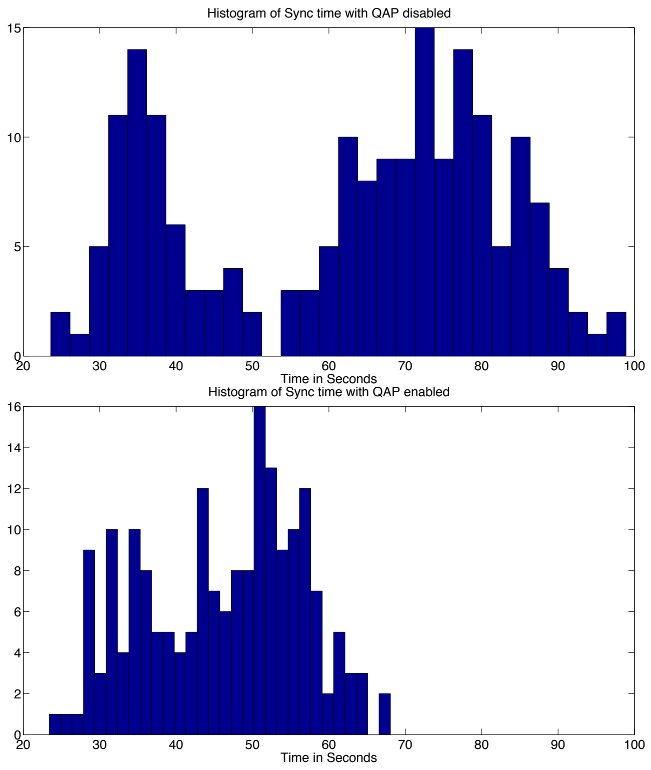


Figure 5: The effect of data layout optimization on the time to perform a synchronization pass over all nodes in a given machine, shown as a histogram over all machines. Top: without optimization. Bottom: with data layout optimization.

error obtained using single and multi-machine settings was indistinguishable, as can be seen in Figure 2.

Second, we study the effect of the data layout scheme. Figure 5 shows the distribution of the synchronization time across clients with and without solving the quadratic assignment problem to minimize the synchronization time. We see that without optimization the synchronization time is bimodal. There are a few clients that are assigned partitions with relatively few nodes to be synchronized, and therefore no matter where these partitions are allocated, their synchronization time is small while other closely connected ones end up in different racks. By solving the QAP, we move clients with demanding communication constraints in close proximity with each other. This smoothes out the total synchronization time, thus accelerating convergence.

Finally, we study the effect of graph size and number of factors on the synchronization time. From Figure 6, we see that as the number of factors increases, synchronization rates do not get significantly affected, but computation time per iteration increases almost linearly (due to large dimensionality). Thus, we can synchronize more often per iteration, which will again bring variables up to date faster. As we simultaneously linearly increase the number of machines (4, 8, 16 etc) and the graph size (4×10^6, 8×10^6, 16×10^6, etc), the average time to synchronize a single machine does not increase too rapidly, thanks to solving the QAP: this is the case since by exploiting local graph structure the aggregate network load is sublinear in the graph size. We conclude again that our architecture shows good scalability.

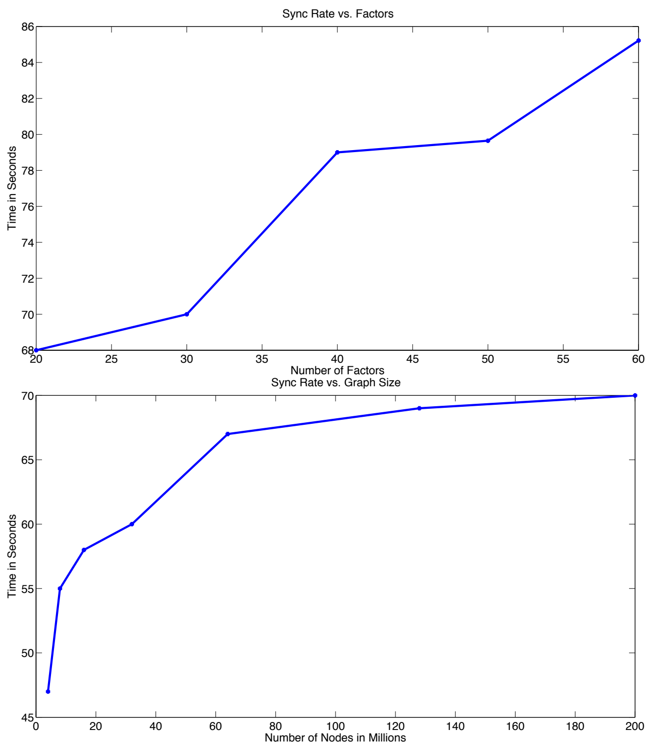


Figure 6: The time it takes to make a single synchronization pass over the nodes in a given machine as a function of the number of factors, and as a function of the graph size. Note the essentially linear dependency in terms of the payload. Furthermore note that the increase is sublinear in terms of the size of the graph. This follows from locality effects.

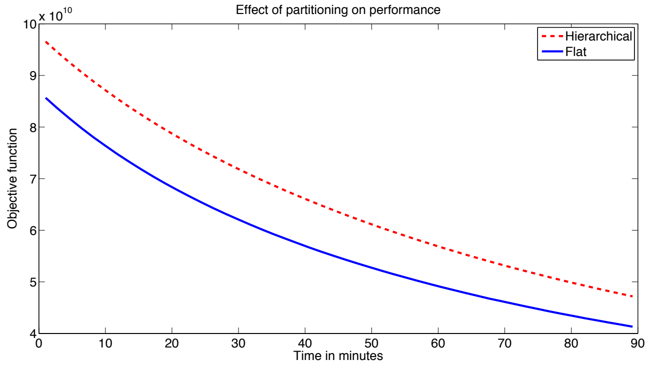


Figure 7: Convergence for flat and hierarchical partitioning, showing the benefit of preprocessing.

7.5 Graph Partitioning

We conclude our experimental analysis by studying the effect that the quality of graph partitioning has on the performance of our system. Intuitively, a good partitioning algorithm should minimize the number of borrowed nodes. This is important since the larger the number of borrowed nodes, the harder it is to bring them to agreement and the longer it takes to converge to a feasible solution on which the local factors agree with the global factors. On the other hand, as we noted in Section 5, minimizing the total number of borrowed nodes is an NP-hard problem, thus there is a trade-off between the time we spend to get a good partitioning of the nodes and its quality.

Table 1: Effect of the partitioning algorithm on variable replication and synchronization time.

Method	Borrowed nodes (millions)	Partitioning (minutes)	Sync time (seconds)
Flat	252.31	166.0	71.5
Hierarchical	392.33	48.7	85.9
HierLSH	640.67	17.8	136.1
HierRandom	720.88	11.6	145.2

We first study the performance of the streaming algorithm discussed in Section 5. Table 1 provides the time needed for partitioning, the quality of the partitioning and the time it takes to perform a single synchronization pass using this partitioning in the asynchronous setting.

- For the hierarchical algorithms we use two levels with 10 splits at the first level.
- For the LSH variant, we used a hash of 128 bits.

This table shows that random and LSH schemes are very fast but result in poor performance and slow synchronizing rates. The hierarchical algorithm achieves almost 4-fold speed-up over the flat algorithm but yields a partitioning with more borrowed nodes and slower synchronization rates. To help better understand this trade-off between flat and hierarchical partitioning, we show in Figure 7 the convergence curve of the training error of the asynchronous algorithm when using each of them. As we can see from this figure, the flat algorithm results in a better partitioning with fewer borrowed nodes and, as such, enables faster convergence and better quality of the final model. On the test data we found that the solution obtained by using the output form the hierarchical algorithm is 6.4% worse than the solution obtained when using the output of the flat partitioning algorithm.

Finally while we were not able to run METIS or other partitioning algorithms on our large scale graphs, we plan in the future to combine METIS [15] or any alternative partitioning algorithm such as [4] with our hierarchical algorithm so that we start to use these algorithms once the graph reaches a manageable size.

8. RELATED WORK

There exist a number of approaches related to the algorithms proposed in the present paper. They fall into three categories: collaborative filtering, distributed latent variable inference algorithms, and graph partitioning algorithms.

Collaborative Filtering Several recent papers addressed the rather related but not identical problem of distributed matrix factorization mostly in collaborative filtering settings [11, 17, 25, 26, 16]. Note that most of these approaches rely on synchronous approaches while, as we showed in this paper, asynchronous approaches are faster. Second, several of the algorithms proposed for distributed collaborative filtering would not be as effective for graph factorization as the rows and columns share the same factors which limits the admissible permutations for block-based optimization algorithms such as [11]. As a rough comparison of throughput with [11], using their published numbers we note that their algorithm on average took 72 seconds per a single iteration over the Netflix data that has 100 Million non-zero entries using 64 machines with 50 factors (a throughput of 21.7K entries per machine per second). However, our asynchronous algorithm took 40 seconds per a single iteration over 10 Billion non-zero entries using 200 machines with 50 factors (a throughput of 1.25 Million entries per machine per second). Moreover, we also note that the famous alternating least squares (ALS) method for matrix factorization [26] can not be applied in our setting efficiently out of the box without modification as the rows and columns of the adjacency matrix of the graph represent the same object.

Graph Partitioning [4] addressed the problem of overlapping clustering for distributed computation which shares some of the flavor of our graph partitioning algorithm. However, the goal in [4] was to facilitate random walk models which results in a different algorithm than the one we gave in Section 5. Moreover, while the authors in [4] only provided simulation results as a proof of concept, here we give a full implementation on a large-scale natural graph orders of magnitude larger.

Distributed Latent Variable Models Closest in spirit to the present work are [1, 2] and [23]. Effectively, they are precursors to the present approach in describing a scalable generic architecture for latent variable models. While our client architecture slightly overlaps with [1] in terms of the module structure inside each client, the semantics of each module is not the same and the semantics of the communication protocol is totally different: here the messages represent gradients while in [1] the messages perform a δ-aggregation on an Abelian group. In fact, experiments using Abelian group updates showed that the algorithm diverges on natural graphs and that a dual decomposition as described in this paper is vital. Finally, we address many practical issues such as graph partitioning and data layout.

Finally, `graphlab.org` provides a general framework for analyzing large graphical models. The version described in [18] allows for efficient computation and scheduling algorithms whenever the degree of vertices is not too high. Very recently (October 2012) a paper describing a new version of GraphLab [14] which provides functionality for dealing with high degree vertices was published. This makes it an interesting platform for designing large scale graph factorization algorithms. In other words, this provides an attractive tool to *implement* the algorithms described in the present paper. Due to the recency of this relevant work we did not compare our native implementation with one relying on GraphLab generic primitives. [4]

[4]We hypothesize that our task-based native implementation is faster than an implementation using GraphLab generic primitives. As an anecdotal evidence, our native asynchronous LDA implementation [1] is 33% faster than an implementation based on GraphLab generic primitives – personal communication with the GraphLab team. While clearly the task here is different, we plan though to compare our native implementation with a GraphLab-based implementation for the graph factorization task in the future.

9. DISCUSSION AND CONCLUSIONS

In this paper we addressed the problem of factorizing natural graphs. We gave an augmented representation of the factorization problem which is amenable to distributed computation and we described two algorithms for optimizing the resulting objective function. Our contributions are the following:

- We provide an efficient algorithm for *vertex partitioning* the graph and demonstrate that it is important.
- We describe automatic task layout at runtime and show that it is efficient in practice.
- Asynchronous optimization is highly beneficial for scalable inference, providing an order of magnitude speedup.
- We perform factorization on one of the largest natural user graphs currently available.

In summary, we describe efficient algorithms and experiments at a scale previously not achievable for distributed inference and factorization. Future work will see improvements in the graph partitioning algorithm. Moreover, we plan to address many further graph-based latent variable models by integrating the techniques in this paper with the generic latent variable architecture in [1].

10. REFERENCES

[1] A. Ahmed, M. Aly, J. Gonzalez, S. Narayanamurthy, and A. Smola. Scalable inference in latent variable models. In *WSDM*, 2012.

[2] A. Ahmed, Y Low, M. Aly, V. Josifovski, and A. Smola. Scalable Distributed Inference of Dynamic User Interests for Behavioral Targeting. In *KDD*, 2011.

[3] D. Aldous. Representations for partially exchangeable arrays of random variables. *Journal of Multivariate Analysis*, 11(4):581–598, 1981.

[4] R. Andersen, D. Gleich, and V. Mirrokni. Overlapping clusters for distributed computation. In *WSDM*, 2012.

[5] K. Andreev and H. Räcke. Balanced graph partitioning. In *Parallelism in algorithms and architectures*, pages 120–124, 2004.

[6] D. P. Bertsekas. *Nonlinear Programming*. Athena Scientific, Belmont, MA, 1995.

[7] D. P. Bertsekas. *Nonlinear Programming*. Athena Scientific, Belmont, MA, second edition, 1999.

[8] D. Blei, A. Ng, and M. Jordan. Latent Dirichlet allocation. *JMLR*, 3:993–1022, 2003.

[9] 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*, 3(1):1–123, 2010.

[10] M. Charikar. Similarity estimation techniques from rounding algorithms. In *ACM Tymposium on Theory of Computing*, pages 380–388, 2002.

[11] R. Gemulla, E. Nijkamp, P. J. Haas, and Y. Sismanis. Large-scale matrix factorization with distributed stochastic gradient descent. In *KDD*, 69–77, 2011.

[12] G. H. Golub and C. F. Van Loan. *Matrix Computations*. John Hopkins Press, 1996.

[13] T. Griffiths and Z. Ghahramani. Infinite latent feature models and the Indian Buffet Process. *NIPS 18*, 475–482, 2006.

[14] J. Gonzalez, Y. Low, H. Gu, D. Bickson and C. Guestrin. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs. *OSDI* , October, 2012.

[15] G. Karypis and V. Kumar. *MeTis: Unstrctured Graph Partitioning and Sparse Matrix Ordering System, Version 2.0*, 1995.

[16] Y. Koren, R. Bell, and C. Volinsky. Matrix factorization techniques for recommender systems. *IEEE Computer*, 42(8):30–37, 2009.

[17] C. Liu, H.-C. Yang, J. Fan, L.-W. He, and Y.-M. Wang. Distributed nonnegative matrix factorization for web-scale dyadic data analysis on mapreduce. In *WWW*, 681–690, 2010.

[18] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. GraphLab: A new parallel framework for machine learning. In *Conference on Uncertainty in Artificial Intelligence*, 2010.

[19] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *ACM ICDM*, 135–146. 2010.

[20] D. Newman, A. Asuncion, P. Smyth, and M. Welling. Distributed algorithms for topic models. *Journal of Machine Learning Research*, 10:1801–1828, 2009.

[21] C. Olston, E. Bortnikov, K. Elmeleegy, F. Junqueira, and B. Reed. Interactive analysis of web-scale data. In *CIDR*, 2009.

[22] B. Recht, C. Re, S. Wright, and F. Niu. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *NIPS*, pages 693–701, 2011.

[23] A. J. Smola and S. Narayanamurthy. An architecture for parallel topic models. In *VLDB*, 2010.

[24] S. Suri and S. Vassilvitskii. Counting triangles and the curse of the last reducer. In *WWW*, 607–614. 2011.

[25] K. Yu, S. Zhu, J. Lafferty, and Y. Gong. Fast nonparametric matrix factorization for large-scale collaborative filtering. In *SIGIR*, pages 211–218, 2009.

[26] Y. Zhou, D. Wilkinson, R. Schreiber, and R. Pan. Large-scale parallel collaborative filtering for the Netflix prize. In *Algorithmic Aspects in Information and Management*, pages 337–348, 2008.