Fugue: Slow-Worker-Agnostic Distributed Learning for Big Models on Big Data

Abhimanu Kumar **Alex Beutel** **Qirong Ho** **Eric P. Xing**
School of Computer Science, Carnegie Mellon University
5000 Forbes Avenue, Pittsburgh, PA 15213

Abstract

We present a scheme for fast, distributed learning on big (i.e. high-dimensional) models applied to big datasets. Unlike algorithms that focus on distributed learning in either the big data or big model setting (but not both), our scheme partitions both the data and model variables simultaneously. This not only leads to faster learning on distributed clusters, but also enables machine learning applications where both data and model are too large to fit within the memory of a single machine. Furthermore, our scheme allows worker machines to perform additional updates while waiting for slow workers to finish, which provides users with a tunable synchronization strategy that can be set based on learning needs and cluster conditions. We prove the correctness of such strategies, as well as provide bounds on the variance of the model variables under our scheme. Finally, we present empirical results for latent space models such as topic models, which demonstrate that our method scales well with large data and model sizes, while beating learning strategies that fail to take both data and model partitioning into account.

1 Introduction

Machine Learning applications continue to grow rapidly, in terms of both input data size (big data) as well as model complexity (big models). The big data challenge is already well-known — with some estimates putting the amount of data generated on the internet at 5 exabytes every two days [1] — and much effort has been devoted towards learning models on big datasets, particularly through stochastic optimization techniques that randomly partition the data over different machines. Such techniques have been the subject of much theoretical scrutiny [13, 19, 1, 10]. On the other hand, the big model issue, which is about learning models with an extremely large number of variables and/or parameters — such as the Google Brain deep network with over 1B parameters [6] — has just started to gain greater attention, and recent papers on this subject have primarily focused on intelligent partitioning of the model variables in order to minimize network synchronization costs [15, 6], albeit not always with theoretical backing.

Although big data and big models are both crucial research foci, there are few distributed machine learning efforts that explicitly consider both aspects in conjunction, with a notable example being the partitioned matrix factorization algorithm of Gemulla *et al.* [8], which divides the input matrix X into independent blocks that do not overlap on rows or columns — thus allowing the factors $AB = X$ to be updated correctly in parallel. More generally, in the big data, big model setting, the model variables may not all fit into a single machine's memory, which in turn imposes additional constraints on how the data is partitioned. Furthermore, careless partitioning of model variables across distributed machines imposes significant network synchronization costs [15], which are required whenever dependent variables or datapoints are placed on separate machines. If we are to effectively partition both data and model, it follows that we must carefully examine and exploit the interdependencies between data and model variables.

In this paper, we investigate the theoretical and algorithmic challenges for learning big latent space models on big data over a distributed cluster. We develop and analyze a stochastic optimization approach for such models expressible in a matrix form, such as topic modeling and dictionary learning. Our proposed algorithm, *Fugue*, exploits the structures of the model in question to partition the input data and model variables over the cluster, in a manner that automatically balances inter-machine network synchronization costs with performing useful computational work, even when the worker machines are not equally capable (e.g. because of different hardware or other concur-

[1] http://techcrunch.com/2010/08/04/schmidt-data/

Appearing in Proceedings of the 17^{th} International Conference on Artificial Intelligence and Statistics (AISTATS) 2014, Reykjavik, Iceland. JMLR: W&CP volume 33.

rently running programs), or the data cannot be evenly partitioned (e.g. due to the dependency structure of the model).

Fugue solves the "last reducer" or "straggler" issue in distributed systems, in which some worker machines can be much slower than others (because of cluster heterogeneity, or because other jobs are running on the same machine), causing faster workers to waste computational cycles waiting for them. Instead, our algorithm ensures that the faster workers will continue to perform useful work until the last reducer finishes — and if the last reducer is unable to finish, it can be restarted without affecting program correctness, while the faster workers continue doing work. Beacuse Fugue has modest synchronization and communication requirements, it is easy to implement on top of common distributed computing systems such as Hadoop — yet it can also be applied to specialized systems for big Machine Learning, such as parameter servers [9, 5, 2, 18], in order to further improve their performance. Finally, our theoretical and empirical analysis confirms that Fugue can provide faster convergence on a variety of latent space problems in ML; furthermore, careful control of inter-worker synchronization can lead to even faster convergence, which opens the door to intelligent exploitation of fine-grained synchronization schemes (such as the aforementioned parameter servers).

2 Related Work

Most existing literature is focused on learning under either big data or big model conditions, but rarely both together. Of the papers that focus on big data, most of them exploit data point indepedence to construct stochastic distributed optimization schemes with little need for inter-machine synchronization. For example, the PSGD algorithm [19] is completely data parallel and requires absolutely no inter-machine communication, therefore making it trivial to implement. However, in practice, one can almost always obtain faster convergence with some inter-machine communication, as our experiments will show. Another important class of methods are the fixed-delay algorithms [1, 13] in which machines communicate with a central server (or each other) in a fixed ordering. This fixed ordering is a serious practical limitation, because all machines will be held up by slowdowns or failures in just one machine. In contrast, our algorithm ensures that all machines continue to do useful work even under such conditions. Most importantly, unlike these big data algorithms, our algorithm can partition model variables (and not just datapoints) across machines, which is absolutely critical for massive models that cannot fit onto a single machine.

For learning on big models, the general strategy is to exploit the fact that each model variable usually depends on only a small number of other variables, and then partition model variables in a way that limits the number of dependencies that cross machines. The GraphLab system [15] is a good example of this concept, but it requires machine learning algorithms to be rewritten into "vertex programs", which can be awkward or even difficult for some algorithms. Furthermore, there has been little theoretical analysis on machine learning algorithms running under GraphLab. The Google Brain project [6] provides another example of the need for model partitioning, this time for a custom-built deep network meant for image and video feature extraction. However, the paper does not provide general partitioning strategies for arbitrary models. Finally, we note that the Hogwild paper [17] provides a theoretical analysis of certain issues related to model partitioning — specifically, the effect of errors produced when two worker threads update the same variable simultaneously. Aside from that, the paper does not provide or analyze model partitioning strategies, while their experiments only cover the shared-memory, single-machine setting.

Finally, there are papers that tackle big data and big model issues together, such as the partitioned matrix factorization algorithm of Gemulla *et al.* [8], to which our work is most closely related. Unlike Gemulla *et al.*, our algorithm allows worker machines to perform useful variable updates continuously, without blocking or waiting for any other machine to complete its assigned task. This property is exceptionally beneficial on very large clusters, where machines often fail or slow down for any number of reasons. Thus, our algorithm is not bottlenecked by the slowest machine, unlike [8]. Furthermore, we provide substantially richer theoretical analysis, including convergence and variance bounds with constraints and analysis of the effect of non-blocking workers. In particular, work by Murata [16] lays the foundation for variance analysis of SGD algorithms, by providing variance bounds over datapoint selection.

A special class of big ML systems are the parameter servers, which provide a distributed shared-memory interface for large numbers of model parameters [2, 18], but perform no variable scheduling themselves. Recent work on parameter servers has led to new, theoretically-sound computational models such as Stale Synchronous Parallelism (SSP) [9, 5], which is fundamentally related to Fugue in that both allow faster workers to perform additional work, without being held up by slower workers. The main difference between SSP and Fugue is that SSP is employed to reduce inter-machine communication regardless of how updates are scheduled, whereas Fugue is used to intelligently schedule updates while assuming limited capacity for inter-machine communication. In future work, we intend to explore how Fugue and such computational models can be combined to tackle big data+model problems of even greater scale.

3 Fugue — Slow-Worker Agnostic Learning for Big Models on Big Data

The key principle behind Fugue is to exploit independent or weakly-dependent blocks of data, variables and parameters. For example, a probabilistic graphical model can con-

tain many latent variables and parameters that capture the modeler's generative assumptions about large datasets. In order to tackle problems of such scale, we need to exploit independence structures present in the data and model, so as to partition both over a distributed cluster.

Before describing our partitioning strategy, we motivate our method with examples of popular latent space models recently gaining much attention. Consider **Topic Modeling (TM)** [4]: given a *document by vocabulary* data matrix Y (with the rows normalized to sum to 1), we want to decompose it into two matrices: *docs by topics* π (which are model variables) and *topics by vocabulary* β (which are parameters). We formulate this task as an optimization problem with simplexial and non-negativity constraints:

$$\text{argmin}_{\pi,\beta} L(Y,\pi,\beta) = ||Y - \pi\beta||_P^p \quad (1)$$
$$\text{s.t. } \forall i,j,k \quad \sum_k \pi_{i,k} = 1, \sum_j \beta_{k,j} = 1, \quad \pi_{i,k} \geq 0, \beta_{k,j} \geq 0,$$

where $\|\cdot\|_P^P$ is an ℓ_p norm, typically ℓ_2. Fugue exploits structure in the form of *blocks* of document-word pairs $Y_{\mathcal{I},\mathcal{J}}$, in order to partition the topic model. We note that other matrix-decomposition-based algorithms for topic modeling also exist, such as the spectral decomposition method of Anandkumar *et al.* [3]. Another example is **Dictionary Learning (DL)** [11], in which the goal is to decompose a signal matrix Y into a dictionary D and a sparse reconstruction α:

$$\text{argmin}_{\alpha,D} L(Y,\alpha,D) = \frac{1}{2}||Y - D\alpha||_2^2 + \lambda||\alpha||_1 \quad (2)$$
$$\text{s.t. } \forall j, D_j^T D_j \leq 1.$$

Here, blocks of sample-feature pairs $Y_{\mathcal{I},\mathcal{J}}$ form the primary exploitable structure. A third example is multi-role or **Mixed-Membership Network Decomposition (MMND)**, where an $N \times N$ adjacency matrix Y is decomposed into an $N \times K$ matrix θ, whose i-th row is the normalized role-vector for node i, and a $K \times K$ role matrix B. Together, θ, B characterize the behavior of every node in the network, and the optimization problem is:

$$\text{argmin}_{\theta,B} L(Y,\theta,B) = \frac{1}{2}||Y - \theta B\theta^\top||_2^2 \quad (3)$$
$$\text{s.t. } \forall i, \quad \sum_j \theta_{ij} = 1, \quad \theta_{i,j} \geq 0.$$

The subgraphs $Y_{\mathcal{I},\mathcal{J}}$ between node sets $\mathcal{I}$ and $\mathcal{J}$ are the basic unit of partitioning used by Fugue.

The main difference between such latent space models and matrix factorization problems (unconstrained or non-negative) is that latent space models usually have more constraints: simplexial constraints in the case of topic modeling and mixed-membership network decomposition, and bounded inner product in the case of dictionary learning. To handle these, our distributed learning algorithm supports projection steps to ensure the final solution always satisfies the constraints. Some distributed algorithms have explicit theoretical guarantees under projection [13, 1], but involve complex synchronization schemes that are inefficient on large clusters and difficult to deploy on common systems like Hadoop. Others, such as Parallel Stochastic Gradient Descent (PSGD) [19], lack explicit projection theory but work well in practice; furthermore, they have simple synchronization requirements, making them suitable for large cluster deployments.

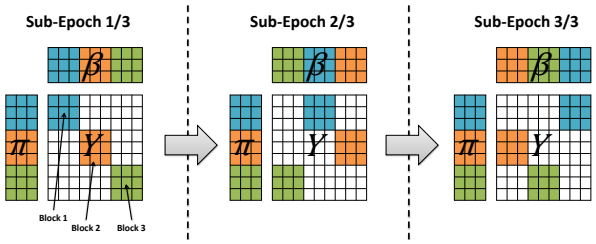

Figure 1: Partitioning strategy for data Y, model variables π, and parameters β. We show one epoch broken into multiple sub-epochs (3 in this case). Each sub-epoch is further divided into (colored) blocks, such that the data $Y_{i,j}$ (with its associated variables $\pi_{i,\cdot}$ and parameters $\beta_{\cdot,j}$) from one block do not share rows/columns with data $Y_{a,b}$ from another block. Taken together, all blocks from all sub-epochs cover every element of Y, π, β.

3.1 Partitioning and Scheduling Algorithm

We now explain how Fugue partitions a big data+model problem into indepedent blocks, followed by how it schedules these blocks for a flexible number of updates on each worker. We stress that Fugue is platform-agnostic: it can be implemented on top of Hadoop, MPI, or even parameter servers and distributed key-value stores — essentially, any platform with programmer control[2] over how data, variables and parameters are partitioned and scheduled for updates. Our experiments use Hadoop, in order to demonstrate that Fugue is highly efficient even without using specialized big ML platforms.

High-level overview. To learn latent space models effectively on a distributed cluster, we must exploit the interdependence of parameters and variables. As a running example, consider the topic modeling objective $L(Y,\pi,\beta)$: we can divide the data matrix Y into a sequence of sub-epochs, where each epoch consists of blocks that do not overlap on parameters β and variables π, and where the union over all epochs covers the entire matrix (Figure 1). This blocking strategy is attributed to Gemulla *et al.* [8], and it permits multiple machines to perform stochastic gradient descent (SGD) on different blocks in parallel, on a Hadoop cluster. However, it requires all workers to process a roughly equal number of data-points per block, which leads to problems with slow workers. In this paper, we intend to address this limitation. Our proposed algorithm allows faster workers to process extra data-points in their assigned block, which maximizes the cluster's computational efficiency.

At a high level, our algorithm proceeds one epoch at a time, performing SGD on all blocks within an epoch in parallel. In order to satisfy the problem constraints, we must interleave projection steps with the SGD algorithm. In this re-

[2]Systems that do not provide programmer control over partitioning/scheduling, e.g. GraphLab, are unsuitable for Fugue.

spect, the parameters β and variables π must be handled differently: while the simplexial projection for variables π can be performed by each worker independently of others, the simplexial projection for the parameters β requires workers to repeatedly synchronize with each other. This cannot be done through the standard MapReduce programming model, so our Hadoop implementation deviates from the MapReduce scheme and allows workers to write to the Hadoop Distributed File System (HDFS) in order to communicate projection information with each other. We find that this scheme works well in practice, while dedicated, memory-based synchronization systems such as parameter servers [5, 2, 18] have the potential to perform even better.

Partitioning strategy. More formally, let $\mathbf{\Psi}$ collectively refer to the variables and parameters π, β, and let ψ refer to individual elements of $\mathbf{\Psi}$. These definitions will make the subsequent analysis easier to understand. Thus, we rewrite the topic modeling objective L as:

$$\psi^{(t+1)} = \psi^{(t)} - \eta_t \nabla \mathcal{L}_{Y_{i,j}}(\psi^{(t)}), \tag{4}$$

and we apply parameter/variable projections each time we execute Eq. (4). Assuming the ℓ_2 norm, the differential of ψ with respect to π at a single data point $Y_{i,j}$ is

$$(\nabla L_{Y_{i,j}}(\psi))_\sigma = \begin{cases} -2(Y_{i,j} - \sum_k \pi_{i,k}\beta_{k,j})\beta_{\ell,j} & \text{if } \sigma = \pi_{i,\ell} \\ 0 & \text{if } \sigma = \pi_{i',\ell},\ i \neq i' \end{cases} \tag{5}$$

where σ is the element of π being differentiated, and $(\nabla L_{Y_{i,j}}(\psi))_\sigma = \frac{\partial L_{Y_{i,j}}}{\partial \sigma}$. The differentials with respect to $\sigma = \beta_{j,\ell}$ are similar. From these equations, we observe that the SGD update for $\pi_{i,\ell}$ at a particular datapoint $Y_{i,j}$ depends only on a small subset of the variables and parameters: specifically, $\pi_{i,k}, \beta_{k,j}$ where $k \in 1, \ldots, K$ and K is the number of topics we chose. Notice that the π all come from the same row i as $Y_{i,j}$, while the β all come from the same column j. Furthermore, the SGD updates for $\pi_{i,\ell}$ are zero for any datapoint $Y_{a,b}$ where $a \neq i$. A similar observation holds for the parameters: the SGD update for $\beta_{r,j}$ is zero for any datapoint $Y_{a,b}$ where $b \neq j$.

These observations lead to the following key insight: we can perform SGD on two datapoints $Y_{i,j}$ and $Y_{a,b}$ at the same time, provided $i \neq a$ and $j \neq b$ — in other words, as long as the datapoints do not overlap on their rows or columns. An intuitive proof goes like this: SGD updates on $Y_{i,j}$ only touch the variable row $\pi_{i,\cdot}$ and the parameter column $\beta_{\cdot,j}$, and both of them do not overlap with $Y_{a,b}$'s variable row $\pi_{a,\cdot}$ and parameter column $\beta_{\cdot,b}$. In other words, the SGD updates on $Y_{i,j}$ and $Y_{a,b}$ touch disjoint sets of variables π and parameters β. Furthermore, each $\pi_{i,\cdot}$ in row i only ever depends on other π in the same row i, and similarly for $\beta_{\cdot,j}$ and other β in column j. We therefore conclude that all quantities associated with row i and column j, namely $Y_{ij}, \pi_{i\cdot}, \beta_{\cdot j}$, are completely independent of the quantities from row a and column b, namely $Y_{ab}, \pi_{a\cdot}, \beta_{\cdot b}$. Hence, datapoints with disjoint rows and columns can be simultaneously used for SGD updates [8].

From the perspective of data, model and parameter partitioning, we have essentially partitioned datapoints Y, model variables π and parameters β into independent "collections" $\mathcal{C}_{ij}$, where i is a row index and j is a column index. In other words, the collection $\mathcal{C}_{ij}$ contains $Y_{ij}, \pi_{i\cdot}, \beta_{\cdot j}$, and can be "processed" (meaning that we run SGD on Y_{ij} to update $\pi_{i\cdot}$ and $\beta_{\cdot j}$) in parallel with any other collection $\mathcal{C}_{ab}$ where $a \neq i, b \neq j$.

We note that the above scheme applies to Dictionary Learning with only slight modification. For Mixed-Membership Network Decomposition, the presence of the symmetric term $\theta B \theta^T$ presents additional challenges. Instead, we replace $\theta B \theta^\top$ with θC where $C := B\theta^\top$, and recover B post-optimization via pseudoinversion: $B = C\theta(\theta^\top\theta)^{-1}$. The inversion cost is reasonable since $\theta^\top\theta$ is $K \times K$, while K is rarely ≥ 1000 in practice.

Distributed update scheduling. Since collections $\mathcal{C}_{ij}$ with disjoint rows/columns can be processed simultaneously, let us consider grouping them into multiple blocks $S_b \subseteq Y$, such that the blocks have disjoint rows/columns (Figure 1). While we cannot process collections $\mathcal{C}_{ij}$ within the same block in parallel (because they might share rows/columns), we can process collections from *different* blocks in parallel, as they are guaranteed to be non-overlapping. Thus, if we managed to construct P non-overlapping blocks, we can spawn P workers to perform SGD in parallel. Although it is impossible to find a set of non-overlapping blocks that covers all of Y, we can find multiple disjoint sets of non-overlapping blocks that, taken together, cover Y completely (Figure 1). We call these sets *sub-epochs*, which are processed sequentially (while the blocks within a sub-epoch are processed in parallel). An *epoch* is a sequence of sub-epochs that fully covers Y.

Fugue differs from Gemulla *et al.* in that within a sub-epoch, we allow different worker-blocks to perform varying numbers of SGD updates per collection $\mathcal{C}_{ij}$ (whereas Gemulla *et al.* require equal numbers of updates per worker). This makes our algorithm much more efficient whenever there are slow worker machines (which are common in large clusters), since faster workers can keep running until synchronization, rather than wasting computational time waiting for slower workers to catch up. The full algorithm is shown in Algorithm 1, and we shall prove that it converges, along with several other important properties.

4 Theoretical Analysis

We now analyse Fugue (Algorithm 1), and prove that our strategy of allowing multiple SGD iterations on each data/variable/parameter collection $\mathcal{C}_{ij} = \{Y_{ij}, \pi_{i\cdot}, \beta_{\cdot j}\}$ leads to faster convergence (under reasonable conditions). Furthermore, we analyze the variance of the model state Ψ under Fugue, and show that it remains bounded under certain assumptions. Briefly, the variance can be attributed to two aspects of our partitioning strategy: (1) running mul-

```
Input : Y, β, π, sub-epoch size d
π ← π_0, β ← β_0
Block Y, π, β into corresponding w blocks
while not converged do
    Pick step size η_S
    Pick w blocks(S_1, ..., S_w) to form sub-epoch S
    for b = 0, ..., w − 1 in parallel do
        Run SGD on the training points Y_ij ∈ S_b
        // (until every block is ready to synchronize)
        // Workers can use datapoints in S_b multiple times while waiting
        Apply appropriate projections
        // (e.g. on variables π in topic modeling)
    end
    Apply appropriate projections
    // (e.g. on parameters β in topic modeling)
end
```

Algorithm 1: Fugue, our slow-worker agnostic learning algorithm, applied to topic modeling.

tiple blocks within a sub-epoch in parallel, and (2) splitting the data matrix into a sequence of sub-epochs. These variance bounds distinguish our analysis from Gemulla *et al.* [8], who did not provide such bounds for their blocking strategy. We will show that allowing additional iterations on fast workers is better than waiting for slow workers, by proving that the model state Ψ's variance converges to zero under an appropriate step-size sequence.

Assume we have w worker processors, and that in each sub-epoch, every processor is assigned to a distinct block i — henceforth, we shall use index i to refer interchangeably to processors or blocks. We now define the following terms:

Definition 1 : **Key states and parameters**

- n_i, κ_i and N_w: Let n_i be the number of datapoints that worker i touches (with repetition) in its assigned block, before transitioning to the next sub-epoch. In other words, if worker i was assigned n datapoints, and touches each point $\kappa_i \geq 1$ times on average, then
$$n_i = \kappa_i n \qquad \text{and} \qquad N_w = \sum_{i=1}^{w} n_i \tag{6}$$
- η_t: SGD step size at iteration t. An iteration is defined as one SGD update on one datapoint.
- $\nabla\mathcal{L}(\psi^{(t)})$: Exact gradient at iteration t.
- $\delta L^{(t)}(V^{(t)}, \psi^{(t)})$: Stochastic gradient at iteration t, i.e. $\nabla\mathcal{L}_{Y_{i,j}}(\psi^{(t)})$ for some i, j.
- ε_t: Error due to stochastic update at iteration t, $\left[\nabla\mathcal{L}(\psi^{(t)}) - \delta L^{(t)}(V^{(t)}, \psi^{(t)})\right]$.
- $\psi^{(t)}$: Model state ψ (see Eq. 4) at iteration t.

We now introduce $V^{(t)}(\psi^{(t+1)}, \psi^{(t)})$, a state potential function defined over a previous state $\psi^{(t)}$, a future state $\psi^{(t+1)}$, and the data points $y^{(t)}$ picked at iteration t.

Definition 2 : **State Potential Function (V)**

$V^{(t)}$ encodes the probability that $\psi^{(t)}$ will be updated to $\psi^{(t+1)}$ when the algorithm performs the stochastic update over datapoint $y^{(t)}$. We also define an n_i-dimensional state potential $V = \left(V^{(t+1)}, V^{(t+2)} \ldots V^{(t+n_i)}\right)$, which encodes the probability distribution of updates caused by all n_i iterations in block i of a sub-epoch (assuming that block i starts at iteration $t+1$).

Next, assumptions on the error terms ε_t and step sizes η_i:

Assumption 1 : **Errors and Step-sizes**

- **Martingale difference error** ε_t: The error terms ε_t form a martingale difference sequence.
- **Variance bound on** ε_t: For all t, we have $E[\varepsilon_t^2] < D$.
- **Step size** η_t **assumption**: $\sum \eta_t^2 < \infty$.

The condition that error terms are a martingale difference sequence is weaker (easier to satisfy) than assuming error terms ε_t are independent of each other. The martingale difference assumption means that the stochastic gradient $\delta L^{(t)}(V^{(t)}, \psi^{(t)})$, when conditioned on the initial model state $\psi^{(0)}$ and previous gradients $\delta L^{(i)}(V^{(i)}, \psi^{(i)})$ for all $i < t$, depends only on the current model stae $\psi^{(t)}$. Fugue satisfies this martingale assumption because our blocking strategy ensures that parallel parameter updates (from different blocks) never overlap on the same elements of ψ. For models with more complex dependency structures (e.g. arbitrary graphical models), the martingale assumption may allow for parallelization opportunities even when such non-overlapping structure is absent.

4.1 Convergence of Fugue

First, using the definition of V^t, we obtain the relation
$$p(\psi^{(t+1)}|\psi^{(t)})d\psi^{(t)} = p(V^{(t)}(\psi^{(t+1)}, \psi^{(t)}))dV^{(t)}(\psi^{(t+1)}, \psi^{(t)}). \tag{7}$$
We can interpret this equation as follows: fix an particular update event $\psi^{(t)} \to \psi^{(t+1)}$, then $V^{(t)}(\psi^{(t+1)}, \psi^{(t)})$ represents the event that some datapoint $y^{(t)}$ gets chosen, while $dV^{(t)}(\psi^{(t+1)}, \psi^{(t)})$ is the probability that said choice leads to the update event $\psi^{(t)} \to \psi^{(t+1)}$. The intuition here is that $V^{(t)} = V^{(t)}(\psi^{(t+1)}, \psi^{(t)})$ is a function that keeps track of the state of $\psi^{(t)}$ and $\psi^{(t+1)}$, and that depends on the datapoint ${Y_{i,j}}^{(t)}$ chosen by the SGD update.

Next, by definition of a martingale difference sequence:
$$\begin{aligned} E\Big[\nabla\mathcal{L}(\psi^{(t)}) - \delta L^{(t)}(V^{(t)}, \psi^{(t)}) \mid \\ \delta L^{(i)}(V^{(i)}, \psi^{(i)}), \psi^{(i)}, i < t, \psi^{(t)}\Big] &= 0, \\ E\left[\varepsilon_t | \varepsilon_i, i < t\right] &= 0. \end{aligned} \tag{8}$$
In other words, the error term ε_t has zero expectation when conditioned on previous errors. We now have the necessary tools to provide a convergence guarantee:

Theorem 1 *The stochastic updates $\psi^{(t+1)} = \psi^{(t)} - \eta_t \nabla\mathcal{L}_{Y_{i,j}}(\psi^{(t)})$ from algorithm 1 and the exact gradient descent updates $\psi^{(t+1)} = \psi^{(t)} - \eta_t \nabla\mathcal{L}(\psi^{(t)})$ converge to the same set of limit points asymptotically, given that the error terms ε_t are a martingale difference sequence, and $E[\varepsilon_i^2] < D$ (bounded variance), and $\sum \eta_i^2 < \infty$.*

We defer the proof to the appendix. This theorem says that, asymptotically, the error terms cancel each other out, and

therefore our stochastic algorithm will find the same set of optima as an exact gradient aglrotihm. The theorem also easily extends to cover projections, by applying the Arzela-Ascoli theorem and considering the limits of converging sub-sequences of our algorithm's SGD updates [12].

4.2 Variance of ψ within a sub-epoch

We now bound the variance of the model state ψ, when it is updated inside block i in a sub-epoch.

Assumption 2 Assume for simplicity that the parameter being updated in block i is univariate. The analysis can be easily extended to multivariate updates.

Theorem 2 *Within block i, suppose we update the model state ψ using n_i datapoints (Eq. 6). Then the variance of ψ after those n_i updates is*

$$\begin{aligned} Var(\psi^{t+n_i}) =& Var(\psi^t) - 2\eta_t n_i \Omega_0 (Var(\psi^t)) \\ & - 2\eta_t n_i \Omega_0 CoVar(\psi_t, \bar{\delta}_t) + \eta_t^2 n_i \Omega_1 \\ & + \underbrace{\mathcal{O}(\eta_t^2 \rho_t) + \mathcal{O}(\eta_t \rho_t^2) + \mathcal{O}(\eta_t^3) + \mathcal{O}(\eta_t^2 \rho_t^2)}_{\Delta_t} \end{aligned}$$

Constants Ω_0, Ω_1 are defined in Theorems 1, 2 in the Appendix.

The proof is left to the Appendix. Note that the constants Ω_0 and Ω_1 are independent of the datapoints picked or the iteration number; they only depend on the global optima of the problem. The above theorem consists of 4 important terms on the first line, plus another 4 cubic (or higher order) terms Δ_t that quickly converge to zero and can be ignored. The basic intuition is as follows: when the algorithm is far from an optimum, $Var(\psi^{(t)})$ and $CoVar(\psi^{(t)}, \bar{\delta}_t)$ are large, so the 2nd and 3rd terms dominate and the variance of ψ decreases quickly. When close to an optimum, the constant 4th term with Ω_1 dominates, causing the algorithm to oscillate unless the step size η_t^2 is small — which we have ensured via shrinking step sizes $\sum \eta_t^2 < \infty$. Hence, the 4th term also shrinks to zero as $t \to \infty$, and therefore the variance of ψ also decreases when close to an optimum.

4.3 Variance of ψ between sub-epochs

Let us consider the variance of ψ across entire sub-epochs. First, note that within a sub-epoch, two blocks Z_i and $Z_{i'}$ are independent if for each datapoint $y \in Z_i$ and $y' \in Z_{i'}$, the following holds:

$$\begin{aligned} \nabla L_y(\psi) &= \nabla L_y(\psi - \eta \nabla L_{y'}(\psi)) \\ \text{and} \quad \nabla L_{y'}(\psi) &= \nabla L_{y'}(\psi - \eta \nabla L_y(\psi)) \end{aligned} \tag{9}$$

In other words, even when the model state ψ is perturbed by the stochastic gradient on y', the stochastic gradient on y must not change (and vice versa). By our earlier argument on collections $\mathcal{C}_{ij} = \{Y_{ij}, \pi_{i\cdot}, \beta_{\cdot j}\}$, this condition holds true for any pair of points from distinct blocks. Thus, within a sub-epoch, distinct blocks Z_i and $Z_{i'}$ operate on disjoint subsets of Ψ, hence their update equations are independent of each other. At the end of a sub-epoch S_{n+1}, the algorithm synchronizes the model state $\Psi_{S_{n+1}}$ by aggregating the non-overlapping updates $\delta\psi^i_{S_{n+1}}$ from all blocks S^i_{n+1}. Therefore, we can write the variance $Var(\Psi_{S_{n+1}})$ at the end of sub-epoch S_{n+1} as

$$\begin{aligned} & Var(\Psi_{S_{n+1}}) = \sum_{i=1}^{w} Var(\psi^i_{S_{n+1}}) \\ =& \sum_{i=1}^{w} \Big[Var(\psi^i_{S_n}) - 2\eta_{S_n} n_i \Omega_0^i (Var(\psi^i_{S_n})) \\ & - 2\eta_{S_n} n_i \Omega_0^i (CoVar(\psi^i_{S_n}, \bar{\delta}^i_{S_n})) + \eta^2_{S_n} n_i \Omega_1^i + \Delta_{S^i_n} \Big] \\ =& Var(\Psi_{S_n}) - 2\eta_{S_n} \sum_{i=1}^{w} n_i \Omega_0^i Var(\psi^i_{S_n}) \\ & - 2\eta_{S_n} \sum_{i=1}^{w} n_i \Omega_0^i CoVar(\psi^i_{S_n}, \bar{\delta}^i_{S_n}) + \eta^2_{S_n} \sum_{i=1}^{w} n_i \Omega_1^i + \mathcal{O}(\Delta_{S_n}), \end{aligned} \tag{10}$$

where the 2nd line is proven in the Appendix. The interpretation is similar to Theorem 2: when far from an optimum, the negative terms dominate and the variance shrinks; when close to an optimum, the positive Ω_1^i term dominates but is also shrinking because of the step sizes $\eta^2_{S_n}$. Again, we can ignore the higher-order terms $\mathcal{O}(\Delta_{S_n})$.

4.4 Slow-worker agnosticism

We now explain why allowing fast processors to do extra updates is beneficial. In Eq. 10, we saw that the variance after each sub-epoch S depends on the number of datapoints touched n_i and the step size η_{S_n}. Let us choose η_{S_n} small enough so that the variance-decreasing terms dominate, i.e.

$$\begin{aligned} & 2\eta_{S_n} \sum_{i=1}^{w} n_i \Omega_0^i Var(\psi^i_{S_n}) + 2\eta_{S_n} \sum_{i=1}^{w} n_i \Omega_0^i CoVar(\psi^i_{S_n}, \bar{\delta}^i_{S_n}) \\ & > \eta^2_{S_n} \sum_{i=1}^{w} n_i \Omega_1^i. \end{aligned} \tag{11}$$

This implies $Var(\Psi_{S_{n+1}}) < Var(\Psi_{S_n})$. Hence, using more datapoints n_i decreases the variance of the model state ψ, provided that we choose η_{S_n} so that Eq. 11 holds. This is easy to satisfy: the RHS of Eq. 11 is $\mathcal{O}(\eta^2_{S_n})$ while the LHS is $\mathcal{O}(\eta_{S_n})$, so we just set η_{S_n} small enough.

Let us now take stock: Eq. 11 tells us that using additional datapoints in any block can decrease the variance of Ψ, while Theorem 1 tells us that the algorithm converges asymptotically, regardless of the number of processors and the number of datapoints assigned to each processor. Because the algorithm converges, decreasing the variance will only move Ψ towards an optimum, and therefore it makes sense for faster processors to perform more updates (with appropriate choice of step size η_{S_n}) and decrease the variance of Ψ, rather than wait for slow processors to finish. However, if the step size η_{S_n} is set incorrectly, then using too many updates n_i could increase the variance of Ψ.

It is important to note that Fugue is *not* equivalent to fully asynchronous computation (e.g. Hogwild [17]), in which every worker can proceed to arbitrary data points, variables and parameters, without regard to what other workers are doing. Fugue requires that all workers advance to the

next sub-epoch at the same time, in order to preserve independence between parallel blocks (and thus the martingale condition). Bounded-synchronization schemes like Stale Synchronous Parallel [9, 5] may allow Fugue to advance workers to different sub-epochs at different times, thus increasing the effectiveness of fast workers without losing of convergence guarantees.

5 Experiments

We compare Fugue implemented on Hadoop to three self-implemented baselines: (a) BarrieredFugue (an extension of Distributed SGD [8] with convergence guarantees for projection and non-negativity constraints) on Hadoop, (b) Parallel SGD [19] on Hadoop, and (c) constrained Matrix Factorization on distributed GraphLab [15], a modification of the default GraphLab MF toolkit that regularly projects variables to maintain constraints (without altering the input graph). We test all methods on our 3 latent space models: topic modeling, dictionary learning, and mixed-membership network decomposition. All methods were tuned to their optimum parameters.

Compared to the baselines, our method has several theoretical and practical advantages: unlike BarrieredFugue, our algorithm allows fast workers to continue doing work while waiting for slow workers to synchronize, and unlike PSGD, we explicitly partition the data/variables/parameters into collections $\mathcal{C}_{ij} = \{Y_{ij}, \pi_{i\cdot}, \beta_{\cdot j}\}$ instead of averaging updates over all data points. Finally, we note that GraphLab is poorly-suited for implementing the simplex and inner-product constraints required by our topic modeling, dictionary learning and mixed-membership network decomposition models. This is because the constraints are over entire matrix rows, creating dependencies over all row variables — which is especially problematic for topic modeling, because the vocabulary matrix β has V columns, and V can be $\geq 100K$ words in practice. As we shall show, such long-range dependencies hurt GraphLab's performance, because GraphLab picks sets of variables for updating without regard to the constraints — whereas a better strategy is to schedule as many dependent elements together as possible.

Cluster Hardware The Hadoop algorithms (ours, BarrieredFugue, PSGD) were run on a Hadoop cluster with the following machine specifications: 2x Intel Xeon E5440 @ 2.83GHz (8 cores per machine), 16GB RAM, 10Gbit Ethernet. Because the Hadoop cluster did not support MPI programs, we ran GraphLab on a newer cluster with the following machine specifications: 2x Intel Xeon E5-2450 @ 2.1-2.9GHz (16 cores per machine), 128GB RAM, 10Gbit Ethernet. Thus, the GraphLab experiments have significantly more memory, but slightly slower processors.

Datasets Statistics for all datasets are summarized in Table 1. For topic modeling, we simulated datasets of various sizes, using the 300K-document NY Times dataset as a building block. The resulting β matrices contain 102,660 columns (words), and between 1.2M to 76.8M rows (documents); zero word counts are treated as missing entries. For dictionary learning, we used a 630,716-image (rows) sample from ImageNet [7], with 1000 features per image (columns). The resulting matrix is fully dense (no missing entries). For mixed-membership network decomposition, we used the Stanford web graph [14], with 281,903 vertices (rows and columns). The resulting adjacency matrix contains all edges, as well as 0.5% of the non-edges (all other non-edges are treated as missing entries).

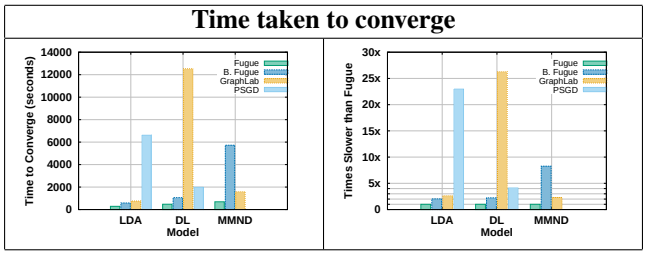

Figure 3: Time taken by all methods to converge on the three ML models, on an absolute scale (left) as well as a relative scale (right). The methods plateau at these values in the respective plots shown in figure 2. The bar for PSGD is absent in the figure as it never reaches 0.059 and stops around objective value 0.092.

Dataset	Dimensions	Nonzeros	Size (GB)
NyTimes	$0.3 * 10^6 \times 102{,}660$	$0.1 * 10^9$	1.49
ImageNet	$0.63 * 10^6 \times 1{,}000$	$0.63 * 10^9$	7.99
WebGraph	$0.28 * 10^6 \times 0.28 * 10^6$	$0.31 * 10^9$	4.46
NyTimes4	$1.2 * 10^6 \times 102{,}660$	$0.4 * 10^9$	6.08
NyTimes16	$4.8 * 10^6 \times 102{,}660$	$1.6 * 10^9$	25.12
NyTimes64	$19.2 * 10^6 \times 102{,}660$	$6.4 * 10^9$	103.4
NyTimes256	$76.8 * 10^6 \times 102{,}660$	$25.6 * 10^9$	421.42

Table 1: Dimension, filesize and nonzero statistics for our datasets. The biggest dataset (NyTimes256) is approximately 0.5 terabytes. Note that the ImageNet dataset is 100% dense.

Stopping Criteria and Parameter Tuning We stop each method when its objective value reaches 0.0065 (TM), 0.059 (MMND), or 0.49 (DL). These are the values at which all methods were observed to plateau (Figure 2). For parameter tuning details, please refer to the Appendix.

Results and Discussion Our results are shown and explained in Figure 2. The convergence plots reveal that our method converges faster and to a better solution than all three baselines, on all three ML models. The scalability plots also reveal that our method converges faster (on topic modeling) for any number of topics and documents, and generally requires fewer processors to converge quickly. We note that GraphLab takes more than twice as long as Fugue to converge (noting that the GraphLab machines had a slightly slower average clockspeed). Moreover, GraphLab sometimes oscillates because its local vertex synchronization prevents it from applying the global projections frequently enough.

The bottom left plot shows the minimum number of machines Fugue and GraphLab require for topic modeling on fixed a number of documents. The primary reason for needing more machines is memory, and we see that GraphLab requires additional machines at a faster rate (despite having 128GB per 16-core machine), whereas Fugue scales much

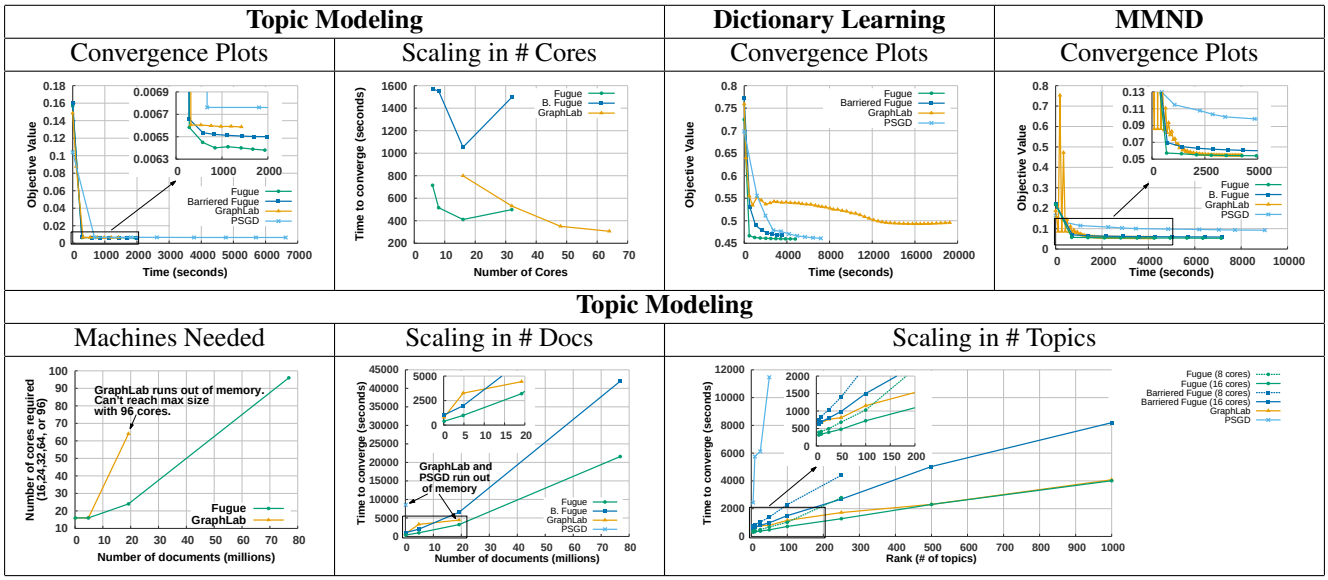

Figure 2: Convergence and scalability plots for the three models (TM, DL, MMND), under our Fugue and baselines (BarrieredFugue, PSGD, GraphLab). Unless otherwise stated in the plot, all methods were run with 16 cores and rank $K = 25$. For all topic modeling plots except "# of Docs" and "Machines Required", we used the NyTimes4 dataset (Table 1). The convergence plots reveal the objective trajectory and final value of each method, while the scalability plots show how each method fares (on topic modeling) as we increase the problem rank, number of processor cores, and data size. In the bottom left, we also show the minimum number of machines required for a given topic modeling dataset size, for Fugue and GraphLab.

Fugue Model	Wait/Sync Time	Epoch Time
Dictionary Learning	32.6	471
Topic Modeling	110.6	391
MM Network Decomp.	182	692

Table 2: Comparison of synchronization time vs epoch time, for each ML model under Fugue. Dictionary learning has the smallest wait:epoch ratio, because the input matrix is fully dense (hence every worker has the same workload). In contrast, Topic Modeling has the highest wait:epoch ratio, because the normalization constraints on the topic-word matrix impose additional synchronization costs. More waiting means faster workers perform more updates, so TM converges in fewer iterations than DL.

more gently (even with only 16GB per 8-core machine). This is because Fugue uses Hadoop and HDFS, so it never loads the entire model into memory (unlike GraphLab). In fact, GraphLab runs out of memory so quickly, that it cannot scale past 20 million documents on a 128GB machine (see "# of Docs" plot).

Compared to BarrieredFugue and PSGD, Fugue succeeds beacuse it (1) it partitions both data and model variables to account for dependencies (which PSGD lacks), while (2) it allows faster workers to do more work before synchronization (which BarrieredFugue lacks). In particular, PSGD must hold the whole model on each machine due to non-partitioning (a memory bottleneck for rank $K \geq 100$). Moreover, PSGD needs one machine to average all models (a computational bottleneck), hence it does not scale well with additional cores (and does not even finish on MMND). Figure 3 shows convergence times for Fugue, BarrieredFugue, PSGD, and GraphLab over the three models: TM, MMND and DL. Fugue is faster by anywhere between $2.6\times$ (vs GraphLab on TM) to $26.2\times$ (vs GraphLab on Dictionary Learning).

Table 2 provides insights about slow and fast workers in Fugue. For example, dense input matrices (e.g. our dictionary learning dataset) result in balanced workloads, thus every worker is equally balanced. On the other hand, the topic-word matrix constraints in topic modeling increase inter-epoch synchronization times, providng an opportunity for workers to conduct extra updates.

Although Fugue exhibits good performance, there remain areas for improvement — for example, Fugue exhibits diminishing returns going from 16 to 32 cores on topic modeling. This is due to increased synchronization costs, which dominate the computational benefits from using additional cores. We expect that moving from Hadoop to a parameter server system [9] will alleviate the synchronization bottleneck, thus allowing Fugue to harness more machines. Furthermore, while Fugue's row/column-wise partitioning strategy for data/variables/parameters works well for the latent space models we have presented, it does not apply to all possible ML models: for example, graphical models and deep networks can have arbitrary structure between parameters and variables, while problems on time-series data will have sequential or autoregressive dependencies between datapoints. In such cases, a row/column-wise partitioning will not work. Nevertheless, the *idea* and basic theoretical analysis of grouping data, variables and parameters into independent collections still holds; only the partitioning strategy needs to be changed. This opens up rich possibilities for future work on general-purpose partitioning algorithms under our framework.

References

[1] Alekh Agarwal and John C Duchi. Distributed delayed stochastic optimization. In *Decision and Control (CDC), 2012 IEEE 51st Annual Conference on*, pages 5451–5452. IEEE, 2012.

[2] Amr Ahmed, Moahmed Aly, Joseph Gonzalez, Shravan Narayanamurthy, and Alexander J. Smola. Scalable inference in latent variable models. In *WSDM*, pages 123–132, 2012.

[3] Animashree Anandkumar, Dean P Foster, Daniel Hsu, Sham M Kakade, and Yi-Kai Liu. Two svds suffice: Spectral decompositions for probabilistic topic modeling and latent dirichlet allocation. *arXiv preprint arXiv:1204.6703*, 2012.

[4] David M Blei and J Lafferty. Topic models. *Text mining: classification, clustering, and applications*, 10:71, 2009.

[5] James Cipar, Qirong Ho, Jin Kyu Kim, Seunghak Lee, Gregory R Ganger, Garth Gibson, Kimberly Keeton, and Eric Xing. Solving the straggler problem with bounded staleness. In *To appear in HotOS 14*, 2013.

[6] J Dean, G Corrado, R Monga, K Chen, M Devin, Q Le, M Mao, M Ranzato, A Senior, P Tucker, K Yang, and A Ng. Large scale distributed deep networks. In *Advances in Neural Information Processing Systems 25*, 2012.

[7] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. ImageNet: A Large-Scale Hierarchical Image Database. In *CVPR09*, 2009.

[8] Rainer Gemulla, Erik Nijkamp, Peter J. Haas, and Yannis Sismanis. Large-scale matrix factorization with distributed stochastic gradient descent. In *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '11, pages 69–77, New York, NY, USA, 2011. ACM.

[9] Q. Ho, J. Cipar, H. Cui, J.-K. Kim, S. Lee, P. B. Gibbons, G. Gibson, G. R. Ganger, and E. P. Xing. More effective distributed ml via a stale synchronous parallel parameter server. In *Advances in Neural Information Processing Systems 26*, 2013.

[10] Matt Hoffman, David M Blei, Chong Wang, and John Paisley. Stochastic variational inference. *arXiv preprint arXiv:1206.7051*, 2012.

[11] Kenneth Kreutz-Delgado, Joseph F. Murray, Bhaskar D. Rao, Kjersti Engan, Te-Won Lee, and Terrence J. Sejnowski. Dictionary learning algorithms for sparse representation. *Neural Comput.*, 15(2):349–396, February 2003.

[12] H. Kushner and G. Yin. *Stochastic Approximation and Recursive Algorithms and Applications*. Springer, 2003.

[13] John Langford, Alex J Smola, and Martin Zinkevich. Slow learners are fast. In *Advances in Neural Information Processing Systems*, pages 2331–2339, 2009.

[14] Jure Leskovec, Kevin J. Lang, Anirban Dasgupta, and Michael W. Mahoney. Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters, 2008.

[15] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M. Hellerstein. Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud. *PVLDB*, 2012.

[16] Noboru Murata. A statistical study on on-line learning. In *Online Learning and Neural Networks*. Cambridge University Press, 1998.

[17] Feng Niu, Benjamin Recht, Christopher Ré, and Stephen J Wright. Hogwild!: A lock-free approach to parallelizing stochastic gradient descent. In *NIPS*, 2011.

[18] Russell Power and Jinyang Li. Piccolo: building fast, distributed programs with partitioned tables. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, pages 1–14. USENIX Association, 2010.

[19] Martin Zinkevich, Markus Weimer, Alex Smola, and Lihong Li. Parallelized stochastic gradient descent. *Advances in Neural Information Processing Systems*, 23(23):1–9, 2010.