Engineering 2 (2016) 179–195

Contents lists available at ScienceDirect

Engineering

journal homepage: www.elsevier.com/locate/eng

Research
iCity & Big Data—Review

Strategies and Principles of Distributed Machine Learning on Big Data

Eric P. Xing *, Qirong Ho, Pengtao Xie, Dai Wei

School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, USA

ARTICLE INFO

Article history:
Received 29 December 2015
Revised 1 May 2016
Accepted 23 May 2016
Available online 30 June 2016

Keywords:
Machine learning
Artificial intelligence big data
Big model
Distributed systems
Principles
Theory
Data-parallelism
Model-parallelism

ABSTRACT

The rise of big data has led to new demands for machine learning (ML) systems to learn complex models, with millions to billions of parameters, that promise adequate capacity to digest massive datasets and offer powerful predictive analytics (such as high-dimensional latent features, intermediate representations, and decision functions) thereupon. In order to run ML algorithms at such scales, on a distributed cluster with tens to thousands of machines, it is often the case that significant engineering efforts are required—and one might fairly ask whether such engineering truly falls within the domain of ML research. Taking the view that "big" ML systems can benefit greatly from ML-rooted statistical and algorithmic insights—and that ML researchers should therefore not shy away from such systems design—we discuss a series of principles and strategies distilled from our recent efforts on industrial-scale ML solutions. These principles and strategies span a continuum from application, to engineering, and to theoretical research and development of big ML systems and architectures, with the goal of understanding how to make them efficient, generally applicable, and supported with convergence and scaling guarantees. They concern four key questions that traditionally receive little attention in ML research: How can an ML program be distributed over a cluster? How can ML computation be bridged with inter-machine communication? How can such communication be performed? What should be communicated between machines? By exposing underlying statistical and algorithmic characteristics unique to ML programs but not typically seen in traditional computer programs, and by dissecting successful cases to reveal how we have harnessed these principles to design and develop both high-performance distributed ML software as well as general-purpose ML frameworks, we present opportunities for ML researchers and practitioners to further shape and enlarge the area that lies between ML and systems. .

1. Introduction

Machine learning (ML) has become a primary mechanism for distilling structured information and knowledge from raw data, turning them into automatic predictions and actionable hypotheses for diverse applications, such as: analyzing social networks [1]; reasoning about customer behaviors [2]; interpreting texts, images, and videos [3]; identifying disease and treatment paths [4]; driving vehicles without the need for a human [5]; and tracking anomalous activity for cybersecurity [6], among others. The majority of ML applications are supported by a moderate number of families of well-developed ML approaches, each of which embodies a continuum of technical elements from model design, to algorithmic innovation, and even to perfection of the software implementation, and which attracts ever-growing novel contributions from the research and development community. Modern examples of such approaches include graphical models [7–9], regularized Bayesian models [10–12], nonparametric Bayesian models [13,14], sparse structured models [15,16], large-margin methods [17,18], deep learning [19,20], matrix factorization [21,22], sparse coding [23,24], and latent space modeling [1,25]. A common ML practice that ensures mathematical soundness and outcome reproducibility is for practitioners and

* Corresponding author.
E-mail address: epxing@cs.cmu.edu

http://dx.doi.org/10.1016/J.ENG.2016.02.008

researchers to write an ML program (using any generic high-level programming language) for an application-specific instance of a particular ML approach (e.g., semantic interpretation of images via a deep learning model such as a convolution neural network). Ideally, this program is expected to execute quickly and accurately on a variety of hardware and cloud infrastructure: laptops, server machines, graphics processing units (GPUs), cloud computing and virtual machines, distributed network storage, Ethernet and Infiniband networking, to name just a few. Thus, the program is hardware-agnostic but ML-explicit (i.e., following the same mathematical principle when trained on data and attaining the same result regardless of hardware choices).

With the advancements in sensory, digital storage, and Internet communication technologies, conventional ML research and development—which excel in model, algorithm, and theory innovations—are now challenged by the growing prevalence of big data collections, such as hundreds of hours of video uploaded to video-sharing sites every minute[†], or petabytes of social media on billion-plus-user social networks[‡]. The rise of big data is also being accompanied by an increasing appetite for higher-dimensional and more complex ML models with billions to trillions of parameters, in order to support the ever-increasing complexity of data, or to obtain still higher predictive accuracy (e.g., for better customer service and medical diagnosis) and support more intelligent tasks (e.g., driverless vehicles and semantic interpretation of video data) [26,27]. Training such big ML models over such big data is beyond the storage and computation capabilities of a single machine. This gap has inspired a growing body of recent work on distributed ML, where ML programs are executed across research clusters, data centers, and cloud providers with tens to thousands of machines. Given P machines instead of one machine, one would expect a nearly P-fold speedup in the time taken by a distributed ML program to complete, in the sense of attaining a mathematically equivalent or comparable solution to that produced by a single machine; yet, the reported speedup often falls far below this mark. For example, even recent state-of-the-art implementations of topic models [28] (a popular method for text analysis) cannot achieve 2× speedup with 4× machines, because of mathematical incorrectness in the implementation (as shown in Ref. [25]), while deep learning on MapReduce-like systems such as Spark has yet to achieve 5× speedup with 10× machines [29]. Solving this scalability challenge is therefore a major goal of distributed ML research, in order to reduce the capital and operational cost of running big ML applications.

Given the iterative-convergent nature of most—if not all—major ML algorithms powering contemporary large-scale applications, at a first glance one might naturally identify two possible avenues toward scalability: faster convergence as measured by iteration number (also known as convergence rate in the ML community), and faster per-iteration time as measured by the actual speed at which the system executes an iteration (also known as throughput in the system community). Indeed, a major current focus by many distributed ML researchers is on algorithmic correctness as well as faster convergence rates over a wide spectrum of ML approaches [30,31] However, it is difficult for many of the "accelerated" algorithms from this line of research to reach industry-grade implementations because of their idealized assumptions on the system—for example, the assumption that networks are infinitely fast (i.e., zero synchronization cost), or the assumption that all machines make the algorithm progress at the same rate (implying no background tasks and only a single user of the cluster, which are unrealistic expectations for real-world research and production clusters shared by many users). On the other hand, systems researchers focus on high iteration throughput (more iterations per second) and fault-recovery guarantees, but may choose to assume that the ML algorithm will work correctly under non-ideal execution models (such as fully asynchronous execution), or that it can be rewritten easily under a given abstraction (such as MapReduce or Vertex Programming) [32–34]. In both ML and systems research, issues from the other side can become oversimplified, which may in turn obscure new opportunities to reduce the capital cost of distributed ML. In this paper, we propose a strategy that combines ML-centric and system-centric thinking, and in which the nuances of both ML algorithms (mathematical properties) and systems hardware (physical properties) are brought together to allow insights and designs from both ends to work in concert and amplify each other.

Many of the existing general-purpose big data software platforms present a unique tradeoff among correctness, speed of execution, and ease-of-programmability for ML applications. For example, dataflow systems such as Hadoop and Spark [34] are built on a MapReduce-like abstraction [32] and provide an easy-to-use programming interface, but have paid less attention to ML properties such as error tolerance, fine-grained scheduling of computation, and communication to speed up ML programs. As a result, they offer correct ML program execution and easy programming, but are slower than ML-specialized platforms [35,36]. This (relative) lack of speed can be partly attributed to the bulk synchronous parallel (BSP) synchronization model used in Hadoop and Spark, in which machines assigned to a group of tasks must wait at a barrier for the slowest machine to finish, before proceeding with the next group of tasks (e.g., all Mappers must finish before the Reducers can start) [37]. Other examples include graph-centric platforms such as GraphLab and Pregel, which rely on a graph-based "vertex programming" abstraction that opens up new opportunities for ML program partitioning, computation scheduling, and flexible consistency control; hence, they are usually correct and fast for ML. However, ML programs are not usually conceived as vertex programs (instead, they are mathematically formulated as iterative-convergent fixed-point equations), and it requires non-trivial effort to rewrite them as such. In a few cases, the graph abstraction may lead to incorrect execution or suboptimal execution speed [38,39]. Of recent note is the parameter server paradigm [28,36,37,40,41], which provides a "design template" or philosophy for writing distributed ML programs from the ground up, but which is not a programmable platform or work-partitioning system in the same sense as Hadoop, Spark, GraphLab, and Pregel. Taking into account the common ML practice of writing ML programs for application-specific instances, a usable software platform for ML practitioners could instead offer two utilities: ① a ready-to-run set of ML workhorse implementations—such as stochastic proximal descent algorithms [42,43], coordinate descent algorithms [44], or Markov Chain Monte Carlo (MCMC) algorithms [45]—that can be re-used across different ML algorithm families; and ② an ML distributed cluster operating system supporting these workhorse implementations, which partitions and executes these workhorses across a wide variety of hardware. Such a software platform not only realizes the capital cost reductions obtained through distributed ML research, but even complements them by reducing the human cost (scientist- and engineer-hours) of big ML applications, through easier-to-use programming libraries and cluster management interfaces.

With the growing need to enable data-driven knowledge distil-

[†] https://www.youtube.com/yt/press/statistics.html
[‡] https://code.facebook.com/posts/229861827208629/scaling-the-facebook-data-warehouse-to-300-pb/

lation, decision making, and perpetual learning—which are representative hallmarks of the vision for machine intelligence—in the coming years, the major form of computing workloads on big data is likely to undergo a rapid shift from database-style operations for deterministic storage, indexing, and queries, to ML-style operations such as probabilistic inference, constrained optimization, and geometric transformation. To best fulfill these computing tasks, which must perform a large number of passes over the data and solve a high-dimensional mathematical program, there is a need to revisit the principles and strategies in traditional system architectures, and explore new designs that optimally balance correctness, speed, programmability, and deployability. A key insight necessary for guiding such explorations is an understanding that ML programs are optimization-centric, and frequently admit iterative-convergent algorithmic solutions rather than one-step or closed form solutions. Furthermore, ML programs are characterized by three properties: ① error tolerance, which makes ML programs robust against limited errors in intermediate calculations; ② dynamic structural dependencies, where the changing correlations between model parameters must be accounted for in order to achieve efficient, near-linear parallel speedup; and ③ non-uniform convergence, where each of the billions (or trillions) of ML parameters can converge at vastly different iteration numbers (typically, some parameters will converge in 2–3 iterations, while others take hundreds). These properties can be contrasted with traditional programs (such as sorting and database queries), which are transaction-centric and are only guaranteed to execute correctly if every step is performed with atomic correctness [32,34]. In this paper, we will derive unique design principles for distributed ML systems based on these properties; these design principles strike a more effective balance between ML correctness, speed, and programmability (while remaining generally applicable to almost all ML programs), and are organized into four upcoming sections: ① How to distribute ML programs; ② how to bridge ML computation and communication; ③ how to communicate; and ④ what to communicate. Before delving into the principles, let us first review some necessary background information about iterative-convergent ML algorithms.

2. Background: Iterative-convergent machine learning (ML) algorithms

With a few exceptions, almost all ML programs can be viewed as optimization-centric programs that adhere to a general mathematical form:

$$\max_A \mathcal{L}(\mathbf{x}, A) \text{ or } \min_A \mathcal{L}(\mathbf{x}, A),$$
$$\text{where } \mathcal{L}(\mathbf{x}, A) = f\left(\{x_i, y_i\}_{i=1}^N; A\right) + r(A) \tag{1}$$

In essence, an ML program tries to fit N data samples (which may be labeled or unlabeled, depending on the real-world application being considered), represented by $\mathbf{x} \equiv \{x_i, y_i\}_{i=1}^N$ (where y_i is present only for labeled data samples), to a model represented by A. This fitting is performed by optimizing (maximizing or minimizing) an overall objective function $\mathcal{L}$, composed of two parts: a loss function, f, that describes how data should fit the model, and a structure-inducing function, r, that incorporates domain-specific knowledge about the intended application, by placing constraints or penalties on the values that θ can take.

The apparent simplicity of Eq. (1) belies the potentially complex structure of the functions f and r, and the potentially massive size of the data **x** and model A. Furthermore, ML algorithm families are often identified by their unique characteristics on f, r, **x**, and A. For example, a typical deep learning model for image classification, such as Ref. [20], will contain tens of millions through billions of matrix-shaped model parameters in A, while the loss function f exhibits a deep recursive structure $f(\ ) = f_1\left(f_2\left(f_3(\ldots)+\ldots\right)+\ldots\right)$ that learns a hierarchical representation of images similar to the human visual cortex. Structured sparse regression models [4] for identifying genetic disease markers may use overlapping structure-inducing functions $r(\ ) = r_1(A_a) + r_2(A_b) + r_3(A_c) + \ldots$, where A_a, A_b, and A_c are overlapping subsets of A, in order to respect the intricate process of chromosomal recombination. Graphical models, particularly topic models, are routinely deployed on billions of documents **x**—that is, $N \geq 10^9$, a volume that is easily generated by social media such as Facebook and Twitter—and can involve up to trillions of parameters θ in order to capture rich semantic concepts over so much data [26].

Apart from specifying Eq. (1), one must also find the model parameters A that optimize $\mathcal{L}$. This is accomplished by selecting one out of a small set of algorithmic techniques, such as stochastic gradient descent [42], coordinate descent [44], MCMC[†] [45], and variational inference (to name just a few). The chosen algorithmic technique is applied to Eq. (1) to generate a set of iterative-convergent equations, which are implemented as program code by ML practitioners, and repeated until a convergence or stopping criterion is reached (or, just as often, until a fixed computational budget is exceeded). Iterative-convergent equations have the following general form:

$$A(t) = F\left(A(t-1), \Delta_{\mathcal{L}}\left(A(t-1), \mathbf{x}\right)\right) \tag{2}$$

where, the parentheses (t) denotes iteration number. This general form produces the next iteration's model parameters $A(t)$, from the previous iteration's $A(t-1)$ and the data **x**, using two functions: ① an update function $\Delta_{\mathcal{L}}$ (which increases the objective $\mathcal{L}$) that performs computation on data **x** and previous model state $A(t-1)$, and outputs intermediate results; and ② an aggregation function F that then combines these intermediate results to form $A(t)$. For simplicity of notation, we will henceforth omit $\mathcal{L}$ from the subscript of Δ—with the implicit understanding that all ML programs considered in this paper bear an explicit loss function $\mathcal{L}$ (as opposed to heuristics or procedures lacking such a loss function).

Let us now look at two concrete examples of Eqs. (1) and (2), which will prove useful for understanding the unique properties of ML programs. In particular, we will pay special attention to the four key components of any ML program: ① data **x** and model A; ② loss function $f(\mathbf{x}, A)$; ③ structure-inducing function $r(A)$; and ④ algorithmic techniques that can be used for the program.

Lasso regression. Lasso regression [46] is perhaps the simplest exemplar from the structured sparse regression ML algorithm family, and is used to predict a response variable y_i given vector-valued features x_i (i.e., regression, which uses labeled data)—but under the assumption that only a few dimensions or features in x_i are informative about y_i. As input, Lasso is given N training pairs **x** of the form $(x_i, y_i) \in \mathbb{R}^m \times \mathbb{R}$, $i = 1,\ldots, n$, where the features are m-dimensional vectors. The goal is to find a linear function, parametrized by the weight vector A, such that ① $A^{\mathrm{T}} x_i \approx y_i$, and ② the m-dimensional parameters A are sparse[‡] (most elements are zero):

$$\min_A \mathcal{L}_{\text{Lasso}}(\mathbf{x}, A), \text{ where } \mathcal{L}_{\text{Lasso}}(\mathbf{x}, A) = \underbrace{\frac{1}{2}\sum_{i=1}^{n}\left(A^{\mathrm{T}} x_i - y_i\right)^2}_{f\left(\{x_i, y_i\}_{i=1}^N; A\right)} + \underbrace{\lambda_n \sum_{j=1}^{m} |a_j|}_{r(A)} \tag{3}$$

[†] Strictly speaking, MCMC algorithms do not perform the optimization in Eq. (1) directly—rather, they generate samples from the function $\mathcal{L}$, and additional procedures are applied to these samples to find a optimizer A^*.

[‡] Sparsity has two benefits: It automatically controls the complexity of the model (i.e., if the data requires fewer parameters, then the ML algorithm will adjust as required), and improves human interpretation by focusing the ML practitioner's attention on just a few parameters.

or more succinctly in matrix notation:

$$\min_A \frac{1}{2}\left\|XA-y\right\|_2^2+\lambda_n\left\|A\right\|_1 \tag{4}$$

where, $X^{\mathrm{T}}=\left[x_1,\ldots,x_n\right]\in\mathbb{R}^{m\times n}$; $y=\left(y_1,\ldots,y_n\right)^{\mathrm{T}}\in\mathbb{R}^n$; $\left\|\cdot\right\|_2$ is the Euclidean norm on $\mathbb{R}^n$; $\left\|\cdot\right\|_1$ is the ℓ_1 norm on $\mathbb{R}^m$; and λ_n is some constant that balances model fit (the *f* term) and sparsity (the *g* term). Many algorithmic techniques can be applied to this problem, such as stochastic proximal gradient descent or coordinate descent. We will present the coordinate descent[†] iterative-convergent equation:

$$A_j\left(t\right)=\mathbb{S}\left(X_{\cdot j}^{\mathrm{T}}y-\sum_{k\neq j}X_{\cdot j}^{\mathrm{T}}X_{\cdot k}A_k\left(t-1\right),\lambda_n\right) \tag{5}$$

where, $\mathbb{S}\left(A_j,\lambda\right):=\mathrm{sign}\left(A_j\right)\left(\left|A_j\right|-\lambda\right)_+$ is the "soft-thresholding operator," and we assume the data is normalized so that for all *j*, $X_{\cdot j}^{\mathrm{T}}X_{\cdot j}=1$. Tying this back to the general iterative-convergent update form, we have the following explicit forms for Δ and *F*:

$$\begin{aligned}\Delta_{\mathrm{Lasso}}\left(A\left(t-1\right),\mathbf{x}\right)&=\begin{bmatrix}X_{\cdot 1}^{\mathrm{T}}y-\sum_{k\neq 1}X_{\cdot 1}^{\mathrm{T}}X_{\cdot k}A_k\left(t-1\right)\\ \vdots\\ X_{\cdot m}^{\mathrm{T}}y-\sum_{k\neq m}X_{\cdot m}^{\mathrm{T}}X_{\cdot k}A_k\left(t-1\right)\end{bmatrix}\\ F_{\mathrm{Lasso}}\left(A\left(t-1\right),u\right)&=\begin{bmatrix}\mathbb{S}\left(u_1,\lambda_n\right)\\ \vdots\\ \mathbb{S}\left(u_m,\lambda_n\right)\end{bmatrix}\end{aligned} \tag{6}$$

where, $u_j=\left[\Delta_{\mathrm{Lasso}}\left(A\left(t-1\right),\mathbf{x}\right)\right]_j$ is the *j*-th element of $\Delta_{\mathrm{Lasso}}\left(A\left(t-1\right),\mathbf{x}\right)$.

Latent Dirichlet allocation topic model. Latent Dirichlet allocation (LDA) [47] is a member of the graphical models ML algorithm family, and is also known as a "topic model" for its ability to identify commonly-recurring topics within a large corpus of text documents. As input, LDA is given *N* unlabeled documents $\mathbf{x}=\left\{x_i\right\}_{i=1}^N$, where each document x_i contains N_i words (referred to as "tokens" in the LDA literature) represented by $x_i=\left[x_{i1},\ldots,x_{ij},\ldots,x_{iN_i}\right]$. Each token $x_{ij}\in\left\{1,\ldots,V\right\}$ is an integer representing one word out of a vocabulary of *V* words—for example, the phrase "machine learning algorithm" might be represented as $x_i=\left[x_{i1},x_{i2},x_{i3}\right]=\left[25,60,13\right]$ (the correspondence between words and integers is arbitrary, and has no bearing on the accuracy of the LDA algorithm).

The goal is to find a set of parameters $A=\left\{\left\{z_{ij}\right\}_{i=1}^N,\left\{\delta_i\right\}_{i=1}^N,\left\{B_k\right\}_{k=1}^K\right\}$—"token topic indicators" $z_{ij}\in\left\{1,\ldots,K\right\}$ for each token in each document, "document-topic vectors" $\delta_i\in\mathrm{Simplex}\left(K\right)$ for each document, and *K* "word-topic vectors" (or simply, "topics") $B_k\in\mathrm{Simplex}\left(V\right)$—that maximizes the following log-likelihood[‡] equation:

$$\begin{aligned}&\max_A \mathcal{L}_{\mathrm{LDA}}\left(\mathbf{x},A\right),\\ &\text{where } \mathcal{L}_{\mathrm{LDA}}\left(\mathbf{x},A\right)=\underbrace{\sum_{i=1}^N\sum_{j=1}^{N_i}\left(\ln\mathbb{P}_{\mathrm{cate.}}\left(x_{ij}\middle|B_{z_{ij}}\right)+\ln\mathbb{P}_{\mathrm{cate.}}\left(z_{ij}\middle|\delta_i\right)\right)}_{f\left(\left\{x_i\right\}_{i=1}^N;A\right)}+\\ &\qquad\underbrace{\sum_{i=1}^N\ln\mathbb{P}_{\mathrm{Dirichlet}}\left(\delta_i\middle|\alpha\right)+\sum_{k=1}^K\ln\mathbb{P}_{\mathrm{Dirichlet}}\left(B_k\middle|\beta\right)}_{r\left(A\right)}\end{aligned} \tag{7}$$

where, $\mathbb{P}_{\mathrm{cate.}}\left(u\middle|v\right)\propto\prod_j v_j^{u_j}$ is the categorical (a.k.a., discrete) probability distribution; $\mathbb{P}_{\mathrm{Dirichlet}}\left(v\middle|\alpha\right)\propto\prod_j v_j^{\alpha-1}$ is the Dirichlet probability distribution; and α and β are constants that balance model fit (the *f* term) with the practitioner's prior domain knowledge about the document-topic vectors δ_i and the topics B_k (the *r* term). Similar to Lasso, many algorithmic techniques such as Gibbs sampling and variational inference (to name just two) can be used on the LDA model; we will consider the collapsed Gibbs sampling equations[††]:

$$\begin{aligned}&\forall\left(i,j\right),\ B_{k_{\mathrm{old}},w_{ij}}\left(t-1\right)-=1,\\ &\qquad B_{k_{\mathrm{new}},w_{ij}}\left(t-1\right)+=1,\\ &\qquad\delta_{i,k_{\mathrm{old}}}\left(t-1\right)-=1,\\ &\qquad\delta_{i,k_{\mathrm{new}}}\left(t-1\right)+=1,\\ &\text{where } k_{\mathrm{old}}=z_{ij}\left(t-1\right)\\ &\qquad k_{\mathrm{new}}=z_{ij}\left(t\right)\sim\mathbb{P}\left(z_{ij}\middle|x_{ij},\delta_i\left(t-1\right),B\left(t-1\right)\right)\end{aligned} \tag{8}$$

where, += and −= are the self-increment and self-decrement operators (i.e., δ, *B*, and *z* are being modified in-place); ~ $\mathbb{P}(\ )$ means "to sample from distribution $\mathbb{P}$," and $\mathbb{P}\left(z_{ij}\middle|x_{ij},\delta_i\left(t-1\right),B\left(t-1\right)\right)$ is the conditional probability[‡‡] of z_{ij} given the current values of $\delta_i\left(t-1\right)$ and $B\left(t-1\right)$. The update $\Delta_{\mathrm{LDA}}\left(A\left(t-1\right),\mathbf{x}\right)$ proceeds in two stages: ① execute Eq. (8) over all document tokens x_{ij}; and ② output $A\left(t\right)=\left\{\left\{z_{ij}\left(t-1\right)\right\}_{i=1}^N,\left\{\delta_i\left(t-1\right)\right\}_{i=1}^N,\left\{B_k\left(t-1\right)\right\}_{k=1}^K\right\}$. The aggregation $F_{\mathrm{LDA}}(A(t-1),\ldots)$ turns out to simply be the identity function.

2.1. Unique properties of ML programs

To speed up the execution of large-scale ML programs over a distributed cluster, we wish to understand their properties, with an eye toward how they can inform the design of distributed ML systems. It is helpful to first understand what an ML program is "not": Let us consider a traditional, non-ML program, such as sorting on MapReduce. This algorithm begins by distributing the elements to be sorted, $x_1,\ldots,x_N$, randomly across a pool of *M* mappers. The Mappers hash each element x_i into a key-value pair $(h(x_i), x_i)$, where *h* is an "order-preserving" hash function that satisfies $h(x) > h(y)$ if $x > y$. Next, for every unique key *a*, the MapReduce system sends all key-value pairs (a, x) to a Reducer labeled "*a*." Each Reducer then runs a sequential sorting algorithm on its received values *x* and, finally, the Reducers take turns (in ascending key order) to output their sorted values.

The first thing to note about MapReduce sort, is that it is single-pass and non-iterative—only a single Map and a single Reduce step are required. This stands in contrast to ML programs, which are iterative-convergent and repeat Eq. (2) many times. More importantly, MapReduce sort is operation-centric and deterministic, and does not tolerate errors in individual operations. For example, if some Mappers were to output a mis-hashed pair (a, x) where $a \neq h(x)$ (for the sake of argument, let us say this is due to improper recovery from a power failure), then the final output will be mis-sorted because *x* will be output in the wrong position. It is for this reason that Hadoop and Spark (which are systems that support MapReduce) provide strong operational correctness guarantees via robust fault-tolerant systems. These fault-tolerant systems certainly require additional engineering effort, and impose additional running time overheads in the form of hard-disk-based checkpoints and lineage trees [34,49]—yet they are necessary for operation-centric programs, which may fail to execute correctly in their absence.

This leads us to the first property of ML programs: **error tolerance**. Unlike the MapReduce sort example, ML programs are usually robust against minor errors in intermediate calculations. In Eq. (2), even if a limited number of updates $\Delta_\mathcal{L}$ are incorrectly computed or transmitted, the ML program is still mathematically guaranteed to converge to an optimal set of model parameters A^*—that is, the ML algorithm terminates with a correct output (even though it might take more iterations to do so) [37,40]. An good example is stochastic

[†] More specifically, we are presenting the form known as "block coordinate descent," which is one of many possible forms of coordinate descent.

[‡] A log-likelihood is the natural logarithm of a probability distribution. As a member of the graphical models ML algorithm family, LDA specifies a probability distribution, and hence has an associated log-likelihood.

[††] Note that collapsed Gibbs sampling re-represents $\boldsymbol{\delta}_i$ and $\boldsymbol{B}_k$ as integer-valued vectors instead of simplex vectors. Details can be found in Ref. [48].

[‡‡] There are a number of efficient ways to compute this probability. In the interest of keeping this article focused, we refer the reader to Ref. [48] for an appropriate introduction.

gradient descent (SGD), a frequently used algorithmic workhorse for many ML programs, ranging from deep learning to matrix factorization and logistic regression [50–52]. When executing an ML program that uses SGD, even if a small random vector ε is added to the model after every iteration, that is, $A(t) = A(t) + \varepsilon$, convergence is still assured; intuitively, this is because SGD always computes the correct direction of the optimum A^* for the update $\Delta_{\mathcal{L}}$, so moving $A(t)$ around simply results in the direction being re-computed to suit [37,40]. This property has important implications for distributed system design, as the system no longer needs to guarantee perfect execution, inter-machine communication, or recovery from failure (which requires substantial engineering and running time overheads). It is often cheaper to do these approximately, especially when resources are constrained or limited (e.g., limited inter-machine network bandwidth) [37,40].

In spite of error tolerance, ML programs can in fact be more difficult to execute than operation-centric programs, because of **dependency structure** that is not immediately obvious from a cursory look at the objective $\mathcal{L}$ or update functions $\Delta_{\mathcal{L}}$ and F. It is certainly the case that dependency structures occur in operation-centric programs: In MapReduce sort, the Reducers must wait for the Mappers to finish, or else the sort will be incorrect. In order to see what makes ML dependency structures unique, let us consider the Lasso regression example in Eq. (3). At first glance, the Δ_{Lasso} update Eq. (6) may look like they can be executed in parallel, but this is only partially true. A more careful inspection reveals that, for the j-th model parameter A_j, its update depends on $\sum_{k \neq j} X_j^{\mathrm{T}} X_k A_k\ (t-1)$. In other words, potentially every other parameter A_k is a possible dependency, and therefore the order in which the model parameters A are updated has an impact on the ML program's progress or even correctness [39]. Even more, there is an additional nuance not present in operation-centric programs: The Lasso parameter dependencies are not binary (i.e., are not only "on" or "off"), but can be soft-valued and influenced by both the ML program state and input data. Notice that if $X_j^{\mathrm{T}} X_k = 0$ (meaning that data column j is uncorrelated with column k), then A_j and A_k have zero dependency on each other, and can be updated safely in parallel [39]. Similarly, even if $X_j^{\mathrm{T}} X_k > 0$, as long as $A_k = 0$, then A_j does not depend on A_k. Such dependency structures are not limited to one ML program; careful inspection of the LDA topic model update Eq. (8) reveals that the Gibbs sampler update for x_{ij} (word token j in document i) depends on ① all other word tokens in document i, and ② all other word tokens b in other documents a that represent the exact same word, that is, $x_{ij} = x_{ab}$ [25]. If these ML program dependency structures are not respected, the result is either sub-ideal scaling with additional machines (e.g., < 2× speedup with 4× as many machines) [25] or even outright program failure that overwhelms the intrinsic error tolerance of ML programs [39].

A third property of ML programs is **non-uniform convergence**, the observation that not all model parameters A_j will converge to their optimal values A_j^* in the same number of iterations—a property that is absent from single-pass algorithms such as MapReduce sort. In the Lasso example in Eq. (3), the $r(A)$ term encourages model parameters A_j to be exactly zero, and it has been empirically observed that once a parameter reaches zero during algorithm execution, it is unlikely to revert to a non-zero value [39]. To put it another way, parameters that reach zero are already converged (with high, though not 100%, probability). This suggests that computation may be better prioritized toward parameters that are still non-zero, by executing Δ_{Lasso} more frequently on them—and such a strategy indeed reduces the time taken by the ML program to finish [39]. Similar non-uniform convergence has been observed and exploited in PageRank, another iterative-convergent algorithm [53].

Finally, it is worth noting that a subset of ML programs exhibit **compact updates**, in that the updates Δ_{Lasso} are, upon careful inspection, significantly smaller than the size of the matrix parameters, $|A|$. In both Lasso (Eq. (3)) and LDA topic models [47], the updates Δ_{Lasso} generally touch just a small number of model parameters, due to sparse structure in the data. Another salient example is that of "matrix-parametrized" models, where A is a matrix (such as in deep learning [54]), yet individual updates Δ_{Lasso} can be decomposed into a few small vectors (a so-called "low-rank" update). Such compactness can dramatically reduce storage, computation, and communication costs if the distributed ML system is designed with it in mind, resulting in order-of-magnitude speedups [55,56].

2.2. On data and model parallelism

For ML applications involving terabytes of data, using complex ML programs with up to trillions of model parameters, execution on a single desktop or laptop often takes days or weeks [20]. This computational bottleneck has spurred the development of many distributed systems for parallel execution of ML programs over a cluster [33–36]. ML programs are parallelized by subdividing the updates $\Delta_{\mathcal{L}}$ over either the data $\mathbf{x}$ or the model A—referred to respectively as data parallelism and model parallelism.

It is crucial to note that the two types of parallelism are complementary and asymmetric—complementary, in that simultaneous data and model parallelism is possible (and even necessary, in some cases), and asymmetric, in that data parallelism can be applied generically to any ML program with an independent and identically distributed (i.i.d.) assumption over the data samples $x_1,\ldots, x_N$. Such i.i.d. ML programs (from deep learning, to logistic regression, to topic modeling and many others) make up the bulk of practical ML usage, and are easily recognized by a summation over data indices i in the objective $\mathcal{L}$ (e.g., Lasso Eq. (3)). Consequently, when a workhorse algorithmic technique (e.g., SGD) is applied to $\mathcal{L}$, the derived update equations $\Delta_{\mathcal{L}}$ will also have a summation[†] over i, which can be easily parallelized over multiple machines, particularly when the number of data samples N is in the millions or billions. In contrast, model parallelism requires special care, because model parameters A_j do not always enjoy this convenient i.i.d. assumption (Fig. 1)—therefore, which parameters A_j are updated in parallel, as well as the order in which the updates $\Delta_{\mathcal{L}}$ happen, can lead to a variety of outcomes: from near-ideal P-fold speedup with P machines, to no additional speedups with additional machines, or even to complete program

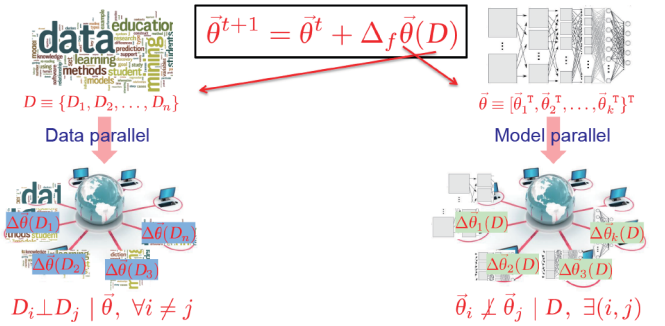

Fig. 1. The difference between data and model parallelism: Data samples are always conditionally independent given the model, but there are some model parameters that are not independent of each other.

[†] For Lasso coordinate descent Δ_{Lasso} (Eq. (5)), the summation over i is in the inner product $X_{\cdot j}^{\mathrm{T}} X_{\cdot k} = \sum_{i=1}^{N} X_{ij} X_{ik}$

failure. The dependency structures discussed for Lasso (Section 2.1) are a good example of the non-i.i.d. nature of model parameters. Let us now discuss the general mathematical forms of data and model parallelism, respectively.

Data parallelism. In data parallel ML execution, the data $\mathbf{x} = \{x_1,\ldots,x_N\}$ is partitioned and assigned to parallel computational workers or machines (indexed by $p = 1,\ldots, P$); we will denote the p-th data partition by $\mathbf{x}_p$. If the update function $\Delta_{\mathcal{L}}$ has an outermost summation over data samples i (as seen in ML programs with the commonplace i.i.d. assumption on data), we can split $\Delta_{\mathcal{L}}$ over data subsets and obtain a data parallel update equation, in which $\Delta_{\mathcal{L}}(A(t-1), \mathbf{x}_p)$ is executed on the p-th parallel worker:

$$A(t) = F\left(A(t-1), \sum_{p=1}^{P} \Delta_{\mathcal{L}}\left(A(t-1), \mathbf{x}_p\right)\right) \tag{9}$$

It is worth noting that the summation $\sum_{p=1}^{P}$ is the basis for a host of established techniques for speeding up data parallel execution, such as minibatches and bounded-asynchronous execution [37,40]. As a concrete example, we can write the Lasso block coordinate descent Eq. (6) in a data parallel form, by applying a bit of algebra:

$$\begin{aligned}
\Delta_{\text{Lasso}}\left(A(t-1), \mathbf{x}_p\right) &= \begin{bmatrix} \sum_{i\in\mathbf{x}_p}\left(X_{i1}y_i - \sum_{k\neq 1} X_{i1}X_{ik}A_k(t-1)\right) \\ \vdots \\ \sum_{i\in\mathbf{x}_p}\left(X_{im}y_i - \sum_{k\neq m} X_{im}X_{ik}A_k(t-1)\right) \end{bmatrix} \\
F_{\text{Lasso}}\left(A(t-1), u\right) &= \begin{bmatrix} \mathbb{S}\left(\left[\sum_{p=1}^{P}\Delta_{\text{Lasso}}\left(A(t-1), \mathbf{x}_p\right)\right]_1, \lambda_n\right) \\ \vdots \\ \mathbb{S}\left(\left[\sum_{p=1}^{P}\Delta_{\text{Lasso}}\left(A(t-1), \mathbf{x}_p\right)\right]_m, \lambda_n\right) \end{bmatrix}
\end{aligned} \tag{10}$$

where, $\sum_{i\in\mathbf{x}_p}$ means (with a bit of notation abuse) to sum over all data indices i included in $\mathbf{x}_p$.

Model parallelism. In model parallel ML execution, the model A is partitioned and assigned to workers/machines $p = 1,\ldots, P$, and updated therein by running parallel update functions $\Delta_{\mathcal{L}}$. Unlike data parallelism, each update function $\Delta_{\mathcal{L}}$ also takes a scheduling or selection function $S_{p,(t-1)}$, which restricts $\Delta_{\mathcal{L}}$ to operate on a subset of the model parameters A (one basic use is to prevent different workers from trying to update the same parameters):

$$A(t) = F\left(A(t-1), \left\{\Delta_{\mathcal{L}}\left(A(t-1), S_{p,(t-1)}\left(A(t-1)\right)\right)\right\}_{p=1}^{P}\right) \tag{11}$$

where, we have omitted the data $\mathbf{x}$ since it is not being partitioned over. $S_{p,(t-1)}$ outputs a set of indices $\{j_1, j_2,\ldots\}$, so that $\Delta_{\mathcal{L}}$ only performs updates on $A_{j_1}, A_{j_2},\ldots$; we refer to such selection of model parameters as scheduling. The model parameters A_j are not, in general, independent of each other, and it has been established that model parallel algorithms are effective only when each iteration of parallel updates is restricted to a subset of mutually independent (or weakly correlated) parameters [39,57–59], which can be performed by $S_{p,(t-1)}$.

The Lasso block coordinate descent updates (Eq. (6)) can be easily written in a simple model parallel form. Here, $S_{p,(t-1)}$ chooses the same fixed set of parameters for worker p on every iteration, which we refer to by $j_{p1},\ldots,j_{pm_p}$:

$$\begin{aligned}
\Delta_{\text{Lasso}}\left(A(t-1), S_{p,(t-1)}\left(A(t-1)\right)\right) &= \begin{bmatrix} X_{\cdot j_{p1}}^{\mathrm{T}} y - \sum_{k\neq j_{p1}} X_{\cdot j_{p1}}^{\mathrm{T}} X_{\cdot k} A_k(t-1) \\ \vdots \\ X_{\cdot j_{pm_p}}^{\mathrm{T}} y - \sum_{k\neq j_{pm_p}} X_{\cdot j_{pm_p}}^{\mathrm{T}} X_{\cdot k} A_k(t-1) \end{bmatrix} \\
F_{\text{Lasso}}\left(A(t-1), \ldots\right) &= \begin{bmatrix} \mathbb{S}\left(\left[\Delta_{\text{Lasso}}\left(A(t-1), S_{1,(t-1)}\left(A(t-1)\right)\right)\right]_1, \lambda_n\right) \\ \vdots \\ \mathbb{S}\left(\left[\Delta_{\text{Lasso}}\left(A(t-1), S_{1,(t-1)}\left(A(t-1)\right)\right)\right]_{m_1}, \lambda_n\right) \\ \vdots \\ \mathbb{S}\left(\left[\Delta_{\text{Lasso}}\left(A(t-1), S_{P,(t-1)}\left(A(t-1)\right)\right)\right]_1, \lambda_n\right) \\ \vdots \\ \mathbb{S}\left(\left[\Delta_{\text{Lasso}}\left(A(t-1), S_{P,(t-1)}\left(A(t-1)\right)\right)\right]_{m_P}, \lambda_n\right) \end{bmatrix}
\end{aligned} \tag{12}$$

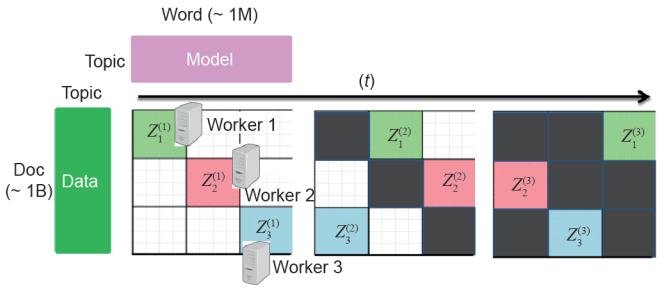

Fig. 2. High-level illustration of simultaneous data and model parallelism in LDA topic modeling. In this example, the three parallel workers operate on data/model blocks $Z_1^{(1)}$, $Z_2^{(1)}$, and $Z_3^{(1)}$ during iteration 1, then move on to blocks $Z_1^{(2)}$, $Z_2^{(2)}$, and $Z_3^{(2)}$ during iteration 2, and so forth.

On a closing note, simultaneous data and model parallelism is also possible, by partitioning the space of data samples and model parameters (x_i, A_j) into disjoint blocks. The LDA topic model Gibbs sampling equations (Eq. (8)) can be partitioned in such a block-wise manner (Fig. 2), in order to achieve near-perfect speedup with P machines [25].

3. Principles of ML system design

The unique properties of ML programs, when coupled with the complementary strategies of data and model parallelism, interact to produce a complex space of design considerations that goes beyond the ideal mathematical view suggested by the general iterative-convergent update equation, Eq. (2). In this ideal view, one hopes that the Δ and F functions simply need to be implemented equation-by-equation (e.g., following the Lasso regression data and model parallel equations given earlier), and then executed by a general-purpose distributed system—for example, if we chose a MapReduce abstraction, one could write Δ as Map and F as Reduce, and then use a system such as Hadoop or Spark to execute them. The reality, however, is that the highest-performing ML implementations are not built in such a naive manner; and, furthermore, they tend to be found in ML-specialized systems rather than on general-purpose MapReduce systems [26,31,35,36]. The reason is that high-performance ML goes far beyond an idealized MapReduce-like view, and involves numerous considerations that are not immediately obvious from the mathematical equations: considerations such as what data batch size to use for data parallelism, how to partition the model for model parallelism, when to synchronize model views between workers, step size selection for gradient based algorithms, and even the order in which to perform Δ updates.

The space of ML performance considerations can be intimidating even to veteran practitioners, and it is our view that a systems interface for parallel ML is needed, both to ① facilitate the organized, scientific study of ML considerations, and also to ② organize these considerations into a series of high-level principles for developing new distributed ML systems. As a first step toward organizing these principles, we will divide them according to four high-level questions: If an ML program's equations (Eq. (2)) tell the system "what to compute," then the system must consider: ① How to distribute the computation; ② How to bridge computation with inter-machine communication; ③ How to communicate between machines; and ④ What to communicate. By systematically addressing the ML considerations that fall under each question, we show that it is possible to build sub-systems whose benefits complement and accrue with each other, and which can be assembled into a full distributed ML system that enjoys orders-of-magnitude speedups in ML program execution time.

3.1. How to distribute: Scheduling and balancing workloads

In order to parallelize an ML program, we must first determine how best to partition it into multiple tasks—that is, we must partition the monolithic Δ in Eq. (2) into a set of parallel tasks, following the data parallel form (Eq. (9)) or the model parallel form (Eq. (11))—or even a more sophisticated hybrid of both forms. Then, we must schedule and balance those tasks for execution on a limited pool of P workers or machines: That is, we ① decide which tasks go together in parallel (and just as importantly, which tasks should not be executed in parallel); ② decide the order in which tasks will be executed; and ③ simultaneously ensure that each machine's share of the workload is well-balanced.

These three decisions have been carefully studied in the context of operation-centric programs (such as the MapReduce sort example), giving rise (for example) to the scheduler system used in Hadoop and Spark [34]. Such operation-centric scheduler systems may come up with a different execution plan—the combination of decisions ① to ③ —depending on the cluster configuration, existing workload, or even machine failure; yet, crucially, they ensure that the outcome of the operation-centric program is perfectly consistent and reproducible every time. However, for ML iterative-convergent programs, the goal is not perfectly reproducible execution, but rather convergence of the model parameters A to an optimum of the objective function $\mathcal{L}$ (i.e., A approaches to within some small distance ε of an optimum A^*). Accordingly, we would like to develop a scheduling strategy whose execution plans allow ML programs to provably terminate with the same quality of convergence every time—we will refer to this as "correct execution" for ML programs. Such a strategy can then be implemented as a scheduling system, which creates ML program execution plans that are distinct from operation-centric ones.

Dependency structures in ML programs. In order to generate a correct execution plan for ML programs, it is necessary to understand how ML programs have internal dependencies, and how breaking or violating these dependencies through naive parallelization will slow down convergence. Unlike operation-centric programs such as sorting, ML programs are error-tolerant, and can automatically recover from a limited number of dependency violations—but too many violations will increase the number of iterations required for convergence, and cause the parallel ML program to experience suboptimal, less-than-P-fold speedup with P machines.

Let us understand these dependencies through the Lasso and LDA topic model example programs. In the model parallel version of Lasso (Eq. (12)), each parallel worker $p \in \{1,\ldots, P\}$ performs one or more Δ_{Lasso} calculations of the form $X_j^{\mathrm{T}} y - \sum_{k \neq j} X_j^{\mathrm{T}} X_k A_k\,(t-1)$, which will then be used to update A_j. Observe that this calculation depends on all other parameters A_k, $k \neq j$ through the term $X_j^{\mathrm{T}} X_k A_k\,(t-1)$, with the magnitude of the dependency being proportional to ① the correlation between the j-th and k-th data dimensions, $X_j^{\mathrm{T}} X_k$; and ② the current value of parameter $A_k\,(t-1)$. In the worst case, both the correlation $X_j^{\mathrm{T}} X_k$ and $A_k\,(t-1)$ could be large, and therefore updating A_j, A_k sequentially (i.e., over two different iterations t, $t+1$) will lead to a different result from updating them in parallel (i.e., at the same time in iteration t). Ref. [57] noted that, if the correlation is large, then the parallel update will take more iterations to converge than the sequential update. It intuitively follows that we should not "waste" computation trying to update highly correlated parameters in parallel; rather, we should seek to schedule uncorrelated groups of parameters for parallel updates, while performing updates for correlated parameters sequentially [39].

For LDA topic modeling, let us recall the Δ_{LDA} updates (Eq. (8)): For every word token w_{ij} (in position j in document i), the LDA Gibbs sampler updates four elements of the model parameters B, δ (which are part of A): $B_{k_{\text{old}},w_{ij}}\,(t-1) -= 1$, $B_{k_{\text{new}},w_{ij}}\,(t-1) += 1$, $\delta_{i,k_{\text{old}}}\,(t-1) -= 1$, and $\delta_{i,k_{\text{new}}}\,(t-1) += 1$, where $k_{\text{old}} = z_{ij}\,(t-1)$ and $k_{\text{new}} = z_{ij}\,(t-1) \sim \mathbb{P}(z_{ij}\,|x_{ij}, \delta_i(t-1), B\,(t-1))$. These equations give rise to many dependencies between different word tokens w_{ij} and w_{uv}. One obvious dependency occurs when $w_{ij} = w_{uv}$, leading to a chance that they will update the same elements of B (which happens when k_{old} or k_{new} are the same for both tokens). Furthermore, there are more complex dependencies inside the conditional probability $\mathbb{P}(z_{ij}\,|x_{ij}, \delta_i(t-1), B\,(t-1))$; in the interest of keeping this article at a suitably high level, we will summarize by noting that elements in the columns of, that is, $B_{,v}$, are mutually dependent, while elements in the rows of δ, that is, $\delta_{i,}$, are also mutually dependent. Due to these intricate dependencies, high-performance parallelism of LDA topic modeling requires a simultaneous data and model parallel strategy (Fig. 2), where word tokens w_{ij} must be carefully grouped by both their value $v = w_{ij}$ and their document i, which avoids violating the column/row dependencies in B and δ [25].

Scheduling in ML programs. In light of these dependencies, how can we schedule the updates Δ in a manner that avoids violating as many dependency structures as possible (noting that we do not have to avoid all dependencies thanks to ML error tolerance)—yet, at the same time, does not leave any of the P worker machines idle due to lack of tasks or poor load balance? These two considerations have distinct yet complementary effects on ML program execution time: Avoiding dependency violations prevents the progress per iteration of the ML program from degrading compared to sequential execution (i.e., the program will not need more iterations to converge), while keeping worker machines fully occupied with useful computation ensures that the iteration throughput (iterations executed per second) from P machines is as close to P times that of a single machine. In short, near-perfect P-fold ML speedup results from combining near-ideal progress per iteration (equal to sequential execution) with near-ideal iteration throughput (P times sequential execution). Thus, we would like to have an ideal ML scheduling strategy that attains these two goals.

To explain how ideal scheduling can be realized, we return to our running Lasso and LDA examples. In Lasso, the degree to which two parameters A_j and A_k are interdependent is influenced by the data correlation $X_j^{\mathrm{T}} X_k$ between the j-th and k-th feature dimensions—we refer to this and other similar operations as a dependency check. If $X_j^{\mathrm{T}} X_k < \kappa$ for a small threshold κ, then A_j and A_k will have little influence on each other. Hence, the ideal scheduling strategy is to find all pairs (j, k) such that $X_j^{\mathrm{T}} X_k < \kappa$, and then partition the parameter indices $j \in \{1,\ldots, m\}$ into independent subsets $\mathbf{A}_1$, $\mathbf{A}_2$,…—where two subsets $\mathbf{A}_a$ and $\mathbf{A}_b$ are said to be independent if for any $j \in \mathbf{A}_a$ and any $k \in \mathbf{A}_b$, we have $X_j^{\mathrm{T}} X_k < \kappa$. These subsets $\mathbf{A}$ can then be safely assigned to parallel worker machines (Fig. 3), and each machine will update the parameters $j \in \mathbf{A}$ sequentially (thus preventing dependency violations) [39].

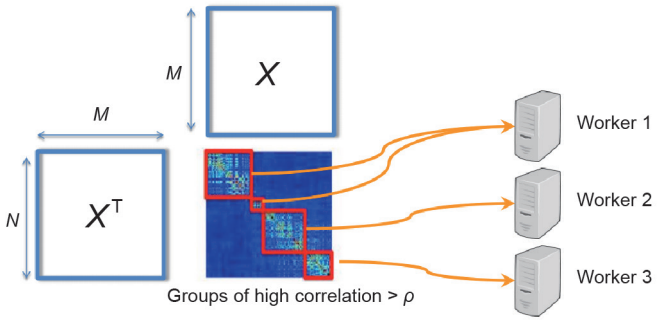

Fig. 3. Illustration of ideal Lasso scheduling, in which parameter pairs (j, k) are grouped into subsets (red blocks) with low correlation between parameters in different subsets. Multiple subsets can be updated in parallel by multiple worker machines; this avoids violating dependency structures because workers update the parameters in each subset sequentially.

As for LDA, careful inspection reveals that the update equations Δ_{LDA} for word token w_{ij} (Eq. (8)) may ① touch any element of column $B_{\cdot,w_{ij}}$, and ② touch any element of row $\delta_{i,\cdot}$. In order to prevent parallel worker machines from operating on the same columns/rows of B and δ, we must partition the space of words $\{1,\ldots, V\}$ (corresponding to columns of B) into P subsets $\mathbf{V}_1,\ldots, \mathbf{V}_P$, as well as partition the space of documents $\{1,\ldots, N\}$ (corresponding to rows of δ) into P subsets $\mathbf{D}_1,\ldots, \mathbf{D}_P$. We may now perform ideal data and model parallelization as follows: First, we assign document subset $\mathbf{D}_p$ to machine p out of P; then, each machine p will only Gibbs sample word tokens w_{ij} such that $i \in \mathbf{D}_p$ and $w_{ij} \in \mathbf{V}_p$. Once all machines have finished, they rotate word subsets $\mathbf{V}_p$ among each other, so that machine p will now Gibbs sample w_{ij} such that $i \in \mathbf{D}_p$ and $w_{ij} \in \mathbf{V}_{p+1}$ (or for machine P, $w_{ij} \in \mathbf{V}_1$). This process continues until P rotations have completed, at which point the iteration is complete (every word token has been sampled) [25]. Fig. 2 illustrates this process.

In practice, ideal schedules like the ones described above may not be practical to use. For example, in Lasso, computing $X_j^{\mathrm{T}} X_k$ for all $O(m^2)$ pairs (j, k) is intractable for high-dimensional problems with large m (millions to billions). We will return to this issue shortly, when we introduce structure aware parallelization (SAP), a provably near-ideal scheduling strategy that can be computed quickly.

Compute prioritization in ML programs. Because ML programs exhibit non-uniform parameter convergence, an ML scheduler has an opportunity to prioritize slower-to-converge parameters A_j, thus improving the progress per iteration of the ML algorithm (i.e., because it requires fewer iterations to converge). For example, in Lasso, it has been empirically observed that the sparsity-inducing ℓ_1 norm (Eq. (4)) causes most parameters A_j to ① become exactly zero after a few iterations, after which ② they are unlikely to become non-zero again. The remaining parameters, which are typically a small minority, take much longer to converge (e.g., 10 times more iterations) [39].

A general yet effective prioritization strategy is to select parameters A_j with probability proportional to their squared rate of change, $(A_j(t-1) - A_j(t-2))^2 + \varepsilon$, where ε is a small constant that ensures that stationary parameters still have a small chance to be selected. Depending on the ratio of fast- to slow-converging parameters, this prioritization strategy can result in an order-of-magnitude reduction in the number of iterations required to converge by Lasso regression [39]. Similar strategies have been applied to PageRank, another iterative-convergent algorithm [53].

Balancing workloads in ML programs. When executing ML programs over a distributed cluster, they may have to stop in order to exchange parameter updates, that is, synchronize—for example, at the end of Map or Reduce phases in Hadoop and Spark. In order to reduce the time spent waiting, it is desirable to load-balance the work on each machine, so that they proceed at close to the same rate. This is especially important for ML programs, which may exhibit skewed data distributions; for example, in LDA topic models, the word tokens w_{ij} are distributed in a power-law fashion, where a few words occur across many documents, while most other words appear rarely. A typical ML load-balancing strategy might apply the classic bin packing algorithm from computer science (where each worker machine is one of the "bins" to be packed), or any other strategy that works for operation-centric distributed systems such as Hadoop and Spark.

However, a second, less-appreciated challenge is that machine performance may fluctuate in real-world clusters, due to subtle reasons such as changing datacenter temperature, machine failures, background jobs, or other users. Thus, load-balancing strategies that are predetermined at the start of an iteration will often suffer from stragglers, machines that randomly become slower than the rest of the cluster, and which all other machines must wait for when performing parameter synchronization at the end of an iteration [37,40,60]. An elegant solution to this problem is to apply slow-worker agnosticism [38], in which the system takes direct advantage of the iterative-convergent nature of ML algorithms, and allows the faster workers to repeat their updates Δ while waiting for the stragglers to catch up. This not only solves the straggler problem, but can even correct for imperfectly-balanced workloads. We note that another solution to the straggler problem is to use bounded-asynchronous execution (as opposed to synchronous MapReduce-style execution), and we will discuss this solution in more detail in Section 3.2.

Structure aware parallelization. Scheduling, prioritization, and load balancing are complementary yet intertwined; the choice of parameters A_j to prioritize will influence which dependency checks the scheduler needs to perform, and in turn, the "independent subsets" produced by the scheduler can make the load-balancing problem more or less difficult. These three functionalities can be combined into a single programmable abstraction, to be implemented as part of a distributed system for ML. We call this abstraction structure aware parallelization (SAP), in which ML programmers can specify how to ① prioritize parameters to speed up convergence; ② perform dependency checks on the prioritized parameters, and schedule them into independent subsets; and ③ load-balance the independent subsets across the worker machines. SAP exposes a simple, MapReduce-like programming interface, where ML programmers implement three functions: ① "schedule()," in which a small number of parameters are prioritized, and then exposed to dependency checks; ② "push()," which performs $\Delta_{\mathcal{L}}$ in parallel on worker machines; and ③ "pull()," which performs F. Load balancing is automatically handled by the SAP implementation, through a combination of classic bin packing and slow-worker agnosticism.

Importantly, the SAP schedule() does not naively perform $O(m^2)$ dependency checks; instead, a few parameters $\mathbf{A}$ are first selected via prioritization (where $|\mathbf{A}| \ll m$). The dependency checks are then performed on $\mathbf{A}$, and the resulting independent subsets are updated via push() and pull(). Thus, SAP only updates a few parameters A_j per iteration of schedule(), push(), and pull(), rather than the full model A. This strategy is provably near-ideal for a broad class of model parallel ML programs:

Theorem 1 (adapted from Ref. [35])**: SAP is close to ideal execution.** *Consider objective functions of the form $\mathcal{L} = f(A) + r(A)$, where $r(A) = \sum_j r(A_j)$ is separable, $A \in \mathbb{R}^d$, and f has β-Lipschitz continuous gradient in the following sense:*

$$f(A+z) \le f(A) + z^{\mathrm{T}} \nabla f(A) + \frac{\beta}{2} A^{\mathrm{T}} X^{\mathrm{T}} X z \tag{13}$$

Let $X = [x_1, \ldots, x_d]$ be the data samples re-represented as d feature vectors. W.l.o.g., we assume that each feature vector x_i is normalized, that is, $\|x_i\|_2 = 1$, $i = 1, \ldots, d$. Therefore, $|x_i^{\mathrm{T}} x_j| \le 1$ for all i and j.

Suppose we want to minimize $\mathcal{L}$ via model parallel coordinate descent. Let $S_{\text{ideal}}()$ be an oracle (i.e., ideal) schedule that always proposes P random features with zero correlation. Let $A^{(t)}_{\text{ideal}}$ be its parameter trajectory, and let $A^{(t)}_{\text{SAP}}$ be the parameter trajectory of SAP scheduling. Then,

$$\mathbb{E}\left[\left|A^{(t)}_{\text{ideal}} - A^{(t)}_{\text{SAP}}\right|\right] \le \frac{2dPm}{(t+1)^2 \hat{P}} L^2 X^{\mathrm{T}} X C \tag{14}$$

for constants C, m, L, and $\hat{P}$.

This theorem says that the difference between the $S_{\text{SAP}}()$ parameter estimate A_{SAP} and the ideal oracle estimate A_{ideal} rapidly vanishes, at a fast $1/(t+1)^2 = O(t^{-2})$ rate. In other words, one cannot do much better than $S_{\text{SAP}}()$ scheduling—it is near-optimal.

SAP's slow-worker agnostic load balancing also comes with a theoretical performance guarantee—it not only preserves correct ML convergence, but also improves convergence per iteration over naive scheduling:

Theorem 2 (adapted from Ref. [38])**: SAP slow-worker agnosti-**

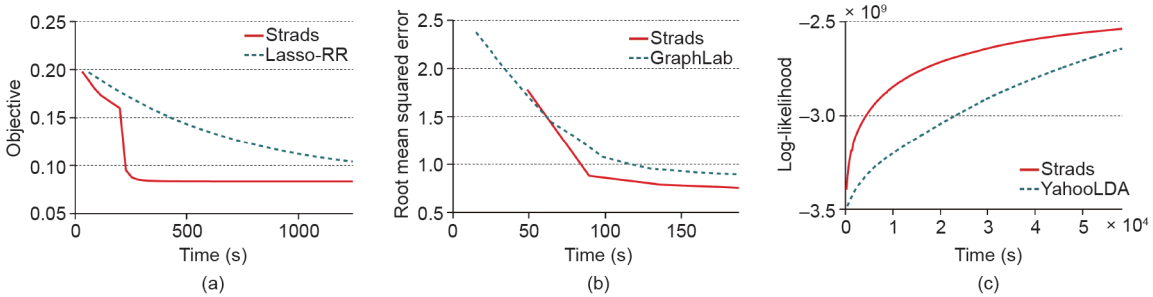

Fig. 4. Objective function $\mathcal{L}$ progress versus time plots for three ML programs—(a) Lasso regression (100M features, 9 machines), (b) matrix factorization (MF) (80 ranks, 9 machines), (c) latent Dirichlet allocation (LDA) topic modeling (2.5M vocab, 5K topics, 32 machines)—executed under Strads, a system that realizes the structure aware parallelization (SAP) abstraction. By using SAP to improve progress per iteration of ML algorithms, Strads achieves faster time to convergence (steeper curves) than other general- and special-purpose implementations—Lasso-RR (a.k.a., Shotgun algorithm), GraphLab, and YahooLDA. Adapted from Ref. [39].

cism improves convergence progress per iteration. *Let the current variance (intuitively, the uncertainty) in the model be* $\mathrm{Var}(A)$*, and let* $n_p > 0$ *be the number of updates performed by worker p (including additional updates due to slow-worker agnosticism). After* n_p *updates,* $\mathrm{Var}(A)$ *is reduced to*

$$\mathrm{Var}\left(A^{+n_p}\right) = \mathrm{Var}(A) - c_1\eta_t n_p \mathrm{Var}(A) - c_2\eta_t n_p \mathrm{CoVar}(A, \nabla\mathcal{L}) + c_3\eta_t^2 n_p + O(\mathrm{cubic}) \quad (15)$$

where, $\eta_t > 0$ *is a step-size parameter that approaches zero as* $t \to \infty$*;* c_1*,* c_2*,* $c_3 > 0$ *are problem-specific constants;* $\nabla\mathcal{L}$ *is the stochastic gradient of the ML objective function* $\mathcal{L}$*;* $\mathrm{CoVar}(a, b)$ *is the covariance between* a *and* b*, and* $O(\mathrm{cubic})$ *represents third-order and higher terms that shrink rapidly toward zero.*

A low variance $\mathrm{Var}(A)$ indicates that the ML program is close to convergence (because the parameters A have stopped changing quickly). The above theorem shows that additional updates n_p do indeed lower the variance—therefore, the convergence of the ML program is accelerated. To see why this is the case, we note that the second and third terms are always negative; furthermore, they are $O(\eta_t)$, so they dominate the fourth positive term (which is $O(\eta_t^2)$ and therefore shrinks toward zero faster) as well as the fifth positive term (which is third-order and shrinks even faster than the fourth term).

Empirically, SAP systems achieve order-of-magnitude speedups over non-scheduled and non-balanced distributed ML systems. One example is the Strads system [39], which implements SAP schedules for several algorithms, such as Lasso regression, matrix factorization, and LDA topic modeling, and achieves superior convergence times compared to other systems (Fig. 4).

3.2. How to bridge computation and communication: Bridging models and bounded asynchrony

Many parallel programs require worker machines to exchange program states between each other—for example, MapReduce systems such as Hadoop take the key-value pairs (a, b) created by all Map workers, and transmit all pairs with key a to the same Reduce worker. For operation-centric programs, this step must be executed perfectly without error; recall the MapReduce sort example (Section 2), where sending keys to two different Reducers results in a sorting error. This notion of operational correctness in parallel programming is underpinned by the BSP model [61,62], a bridging model that provides an abstract view of how parallel program computations are interleaved with inter-worker communication. Programs that follow the BSP bridging model alternate between a computation phase and a communication phase or synchronization barrier (Fig. 5), and the effects of each computation phase are not visible to worker machines until the next synchronization barrier has completed.

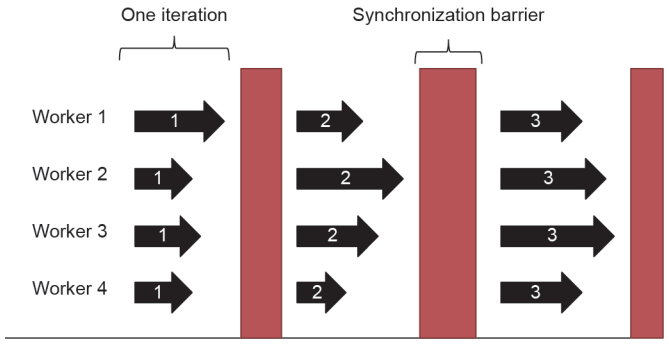

Fig. 5. Bulk synchronous parallel (BSP) bridging model. For ML programs, the worker machines wait at the end of every iteration for each other, and then exchange information about parameters A_j during the synchronization barrier.

Because BSP creates a clean separation between computation and communication phases, many parallel ML programs running under BSP can be shown to be serializable—that is to say, they are equivalent to a sequential ML program. Serializable BSP ML programs enjoy all the correctness guarantees of their sequential counterparts, and these strong guarantees have made BSP a popular bridging model for both operation-centric programs and ML programs [32,34,63]. One disadvantage of BSP is that workers must wait for each other to reach the next synchronization barrier, meaning that load balancing is critical for efficient BSP execution. Yet, even well-balanced workloads can fall prey to stragglers, machines that become randomly and unpredictably slower than the rest of the cluster [60], due to real-world conditions such as temperature fluctuations in the datacenter, network congestion, and other users' programs or background tasks. When this happens, the program's efficiency drops to match that of the slowest machine (Fig. 5)—and in a cluster with thousands of machines, there may even be multiple stragglers. A second disadvantage is that communication between workers is not instantaneous, so the synchronization barrier itself can take a non-trivial amount of time. For example, in LDA topic modeling running on 32 machines under BSP, the synchronization barriers can be up to six times longer than the iterations [37]. Due to these two disadvantages, BSP ML programs may suffer from low iteration throughput, that is, P machines do not produce a P-fold increase in throughput.

As an alternative to running ML programs on BSP, asynchronous parallel execution (Fig. 6) has been explored [28,33,52], in which worker machines never wait for each other, and always communicate model information throughout the course of each iteration. Asynchronous execution usually obtains a near-ideal P-fold increase in iteration throughput, but unlike BSP (which ensures serializability and hence ML program correctness), it often suffers from decreased

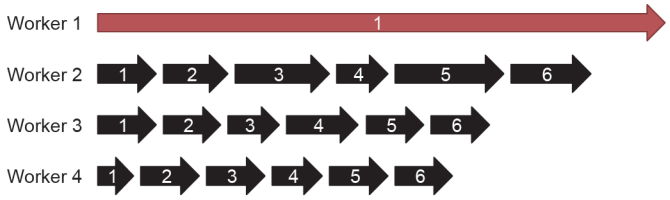

Fig. 6. Asynchronous parallel execution. Worker machines running ML programs do not have to wait for each other, and information about model parameters A_j is exchanged asynchronously and continuously between workers. Because workers do not wait, there is a risk that one machine could end up many iterations slower than the others, which can lead to unrecoverable errors in ML programs. Under a BSP system, this would not happen because of the synchronization barrier.

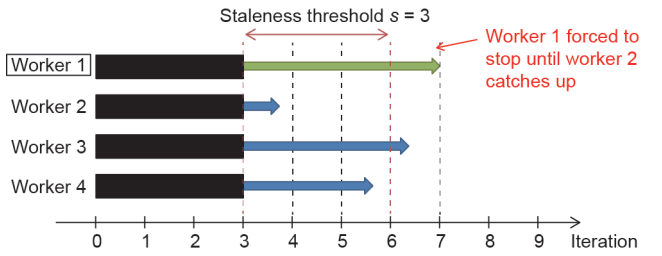

Fig. 7. Stale synchronous parallel (SSP) bridging model. Compared to BSP, worker machines running ML programs may advance ahead of each other, up to s iterations apart (where s is called the staleness threshold). Workers that get too far ahead are forced to stop, until slower workers catch up. Like asynchronous parallel execution, information about model parameters A_j is exchanged asynchronously and continuously between workers (with a few additional conditions so as to ensure correct ML convergence), without the need for synchronization barriers. The advantage of SSP is that it behaves like asynchronous parallel execution most of the time, yet SSP can also stop workers as needed to ensure correct ML execution.

convergence progress per iteration. The reason is that asynchronous communication causes model information to become delayed or stale (because machines do not wait for each other), and this in turn causes errors in the computation of Δ and F. The magnitude of these errors grows with the delays, and if the delays are not carefully bounded, the result is extremely slow or even incorrect convergence [37,40]. In a sense, there is "no free lunch"—model information must be communicated in a timely fashion between workers.

BSP and asynchronous execution face different challenges in achieving ideal P-fold ML program speedups—empirically, BSP ML programs have difficulty reaching the ideal P-fold increase in iteration throughput [37], while asynchronous ML programs have difficulty maintaining the ideal progress per iteration observed in sequential ML programs [25,37,40,]. A promising solution is bounded-asynchronous execution, in which asychronous execution is permitted up to a limit. To explore this idea further, we present a bridging model called stale synchronous parallel (SSP) [37,64], which generalizes and improves upon BSP.

Stale synchronous parallel. Stale synchronous parallel (SSP) is a bounded-asynchronous bridging model, which enjoys a similar programming interface to the popular BSP bridging model. An intuitive, high-level explanation goes as follows: We have P parallel workers or machines that perform ML computations Δ and F in an iterative fashion. At the end of each iteration t, SSP workers signal that they have completed their iterations. At this point, if the workers were instead running under BSP, a synchronization barrier would be enacted for inter-machine communication. However, SSP does not enact a synchronization barrier. Instead, workers may be stopped or allowed to proceed as SSP sees fit; more specifically, SSP will stop a worker if it is more than s iterations ahead of any other worker, where s is called the staleness threshold (Fig. 7).

More formally, under SSP, every worker machine keeps an iteration counter t, and a local view of the model parameters A. SSP worker machines "commit" their updates Δ, and then invoke a "clock()" function that ① signals that their iteration has ended, ② increments their iteration counter t, and ③ informs the SSP system to start communicating Δ to other machines, so they can update their local views of A. This clock() is analogous to BSP's synchronization barrier, but is different in that updates from one worker do not need to be immediately communicated to other workers—as a consequence, workers may proceed even if they have only received a partial subset of the updates. This means that the local views of A can become stale, if some updates have not been received yet. Given a user-chosen staleness threshold $s \geq 0$, an SSP implementation or system enforces at least the following bounded staleness conditions:

- **Bounded clock difference:** The iteration counters on the slowest and fastest workers must be $\leq s$ apart—otherwise, SSP forces the fastest worker to wait for the slowest worker to catch up.
- **Timestamped updates:** At the end of each iteration t (right before calling clock()), each worker commits an update Δ, which is timestamped with time t.
- **Model state guarantees:** When a worker with clock t computes Δ, its local view of A is guaranteed to include all updates Δ with timestamp $\leq t - s - 1$. The local view may or may not contain updates Δ from other workers with timestamp $> t - s - 1$.
- **Read-my-writes:** Each worker will always include its own updates Δ, in its own local view of A.

Since the fastest and slowest workers are $\leq s$ clocks apart, a worker's local view of A at iteration t will include all updates Δ from all workers with timestamps in $[0, t - s - 1]$, plus some (or possibly none) of the updates whose timestamps fall in the range $[t - s, t + s - 1]$. Note that SSP is a strict generalization of BSP for ML programs: When $s = 0$, the first range becomes $[0, t - 1]$ while the second range becomes empty, which corresponds exactly to BSP execution of an ML program.

Because SSP always limits the maximum staleness between any pair of workers to s, it enjoys strong theoretical convergence guarantees for both data parallel and model parallel execution. We state two complementary theorems to this effect:

Theorem 3 (adapted from Ref. [40])**: SSP data parallel convergence theorem.** *Consider convex objective functions of the form* $\mathcal{L} = f(A) = \sum_{t=1}^{T} f_t(A)$*, where the individual components* f_t *are also convex. We search for a minimizer* A^* *via data parallel SGD on each component* ∇f_t *under SSP, with staleness parameter s and P workers. Let the data parallel updates be* $\Delta_t := -\eta_t \nabla_t f_t(A_t)$ *with* $\eta_t = \eta/\sqrt{t}$*. Under suitable conditions (*f_t *are L-Lipschitz and bounded divergence* $D(A\|A') \leq F^2$*), we have the following convergence rate guarantee:*

$$P\left[\frac{R[A]}{T} - \frac{1}{\sqrt{T}}\left(\eta L^2 + \frac{F^2}{\eta} + 2\eta L^2 \mu_\gamma\right) \geq \tau\right]$$

$$\leq \exp\left\{\frac{-T\tau^2}{2\bar{\eta}_T \sigma_\gamma + \frac{2}{3}\eta L^2 (2s+1) P\tau}\right\}$$

where, $R[A] := \sum_{t=1}^{T} f_t(\tilde{A}_t) - f(A^*)$; $\bar{\eta}_T = \frac{\eta^2 L^4 \ln(T+1)}{T} = o(1)$ *as* $T \to \infty$*; in particular, s is the maximum staleness under SSP;* μ_γ *is the average staleness experienced by the distributed system, and* σ_γ *is the variance of the staleness.*

This data parallel SSP theorem has two implications: First, data parallel execution under SSP is correct (just like BSP) because $R[A]/T$ (the difference between the SSP parameter estimate and the true optimum) converges to $O(T^{-1/2})$ in probability with an exponential

tail-bound. Second, it is important to keep the actual staleness and asynchrony as low as possible; the convergence bound becomes tighter with lower maximum staleness s, and lower average μ_γ and variance σ_γ of the staleness experienced by the workers. For this reason, naive asynchronous systems (e.g., Hogwild! [31] and YahooLDA [28]) may experience poor convergence in complex production environments, where machines may temporarily slow down due to other tasks or users—in turn causing the maximum staleness s and staleness variance σ_γ to become arbitrarily large, leading to poor convergence rates.

Theorem 4: SSP model parallel asymptotic consistency. *We consider minimizing objective functions of the form $\mathcal{L} = f(A, D) + g(A)$ where $A \in \mathbb{R}^d$, using a model parallel proximal gradient descent procedure that keeps a centralized "global view," A, (e.g., on a key-value store) and stale local worker views A^p on each worker machine. If the descent step size satisfies $\eta < 1/(L_f + 2Ls)$, then the global view A and local worker views A^p will satisfy:*

(1) $\sum_{t=0}^{\infty} \left\| A(t+1) - A(t) \right\|^2 < \infty$;

(2) $\lim_{t\to\infty} \left\| A(t+1) - A(t) \right\| = 0$, *and for all p,* $\lim_{t\to\infty} \left\| A(t) - A^p(t) \right\| = 0$;

(3) The limit points of $\{A(t)\}$ coincide with those of $\{A^p(t)\}$, and both are critical points of $\mathcal{L}$.

Items 1 and 2 imply that the global view A will eventually stop changing (i.e., will converge), and the stale local worker views A^p will converge to the global view A; in other words, SSP model parallel execution will terminate to a stable answer. Item 3 further guarantees that the local and global views $A^p(t)$ and $A(t)$ will reach an optimal solution to $\mathcal{L}$; in other words, SSP model parallel execution outputs the correct solution. Given additional technical conditions, we can further establish that SSP model parallel execution converges at rate $O(t^{-1})$.

The above two theorems show that both data parallel and model parallel ML programs running under SSP enjoy near-ideal convergence progress per iteration (which approaches close to BSP and sequential execution). For example, the Bösen system [37,40,41] uses SSP to achieve up to ten-fold shorter convergence times, compared to the BSP bridging model—and SSP with properly selected staleness values will not exhibit non-convergence, unlike asynchronous execution (Fig. 8). In summary, when SSP is effectively implemented and tuned, it can come close to providing the best of both worlds: near-ideal progress per iteration close to BSP, and near-ideal P-fold iteration througput similar to asynchronous execution—and hence, a near-ideal P-fold speedup in ML program execution time.

3.3. How to communicate: Managed communication and topologies

The bridging models (BSP and SSP) just discussed place constraints on when ML computation should occur relative to the communication of updates Δ to model parameters A, in order to guarantee correct ML program execution. However, within the constraints set by a bridging model, there is still room to prescribe how, or in what order, the updates Δ should be communicated over the network. Consider the MapReduce sort example, under the BSP bridging model: The Mappers need to send key-value pairs (a, b) with the same key a to the same Reducer. While this can be performed via a bipartite topology (every Mapper communicates with every Reducer), one could instead use a star topology, in which a third set of machines first aggregates all key-value pairs from the Mappers, and then sends them to the Reducers.

ML algorithms under the SSP bridging model open up an even wider design space: Because SSP only requires updates Δ to "arrive no later than s iterations," we could choose to send more important updates first, following the intuition that this should naturally improve algorithm progress per iteration. These considerations are important because every cluster or datacenter has its own physical switch topology and available bandwidth along each link. We will discuss these considerations with the view that choosing the correct communication management strategy will lead to a noticable improvement in both ML algorithm progress per iteration and iteration throughput. We now discuss several ways in which communication management can be applied to distributed ML systems.

Continuous communication. In the first implementations of the SSP bridging model, all inter-machine communication occurred right after the end of each iteration (i.e., right after the SSP clock() command) [37], while leaving the network idle at most other times (Fig. 9). The resulting burst of communication (gigabytes to terabytes) may cause synchronization delays (where updates take longer than expected to reach their destination), and these can be optimized away by adopting a continuous style of communication, where the system waits for existing updates to finish transmission before starting new ones [41].

Continuous communication can be achieved by a rate limiter in the SSP implementation, which queues up outgoing communications, and waits for previous communications to finish before sending out the next in line. Importantly, regardless of whether the ML algorithm is data parallel or model parallel, continuous communication still preserves the SSP bounded staleness conditions—and therefore, it continues to enjoy the same worst-case convergence progress per iteration guarantees as SSP. Furthermore, because managed communication reduces synchronization delays, it also provides a small (two- to three-fold) speedup to overall convergence time [41], which is partly due to improved iteration throughput (because of fewer synchronization delays), and partly due to improved progress per iteration (fewer delays also means lower average stale-

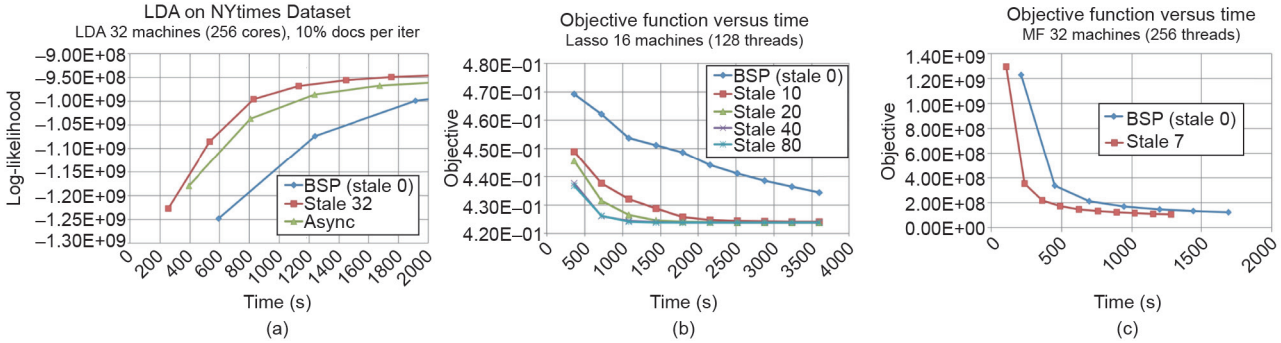

Fig. 8. Objective function $\mathcal{L}$ progress versus time plots for three ML programs—(a) LDA topic modeling, (b) Lasso regression, and (c) matrix factorization (MF)—executed under Bösen, a system that realizes the SSP bridging model. By using SSP (with a range of different staleness values) to improve the iteration throughput of ML algorithms, Bösen achieves faster time to convergence (steeper curves) than both the BSP bridging model (used in Hadoop and Spark) and fully asynchronous modes of execution. In particular, fully asynchronous execution did not successfully converge for Lasso and matrix factorization, and hence the curves are omitted. Adapted from Ref. [37].

ness in local parameter views A; hence, SSP's progress per iteration is improved, according to Theorem 3).

Wait-free back-propagation. The deep learning family of ML models [20,52] presents a special opportunity for continuous communication, due to their highly layered structure. Two observations stand out in particular: ① the "back-propagation" gradient descent algorithm—used to train deep learning models such as convolutional neural networks (CNNs)—proceeds in a layer-wise fashion; and ② the layers of a typical CNN (such as "AlexNet" [20]) are highly asymmetric in terms of model size $|A|$ and require computation for the back-propagation—usually, the top, fully connected layers have approximately 90% of the parameters, while the bottom convolutional layers account for 90% of the back-propagation computation [56]. This allows for a specialized type of continuous communication, which we call wait-free back-propagation: After performing back-propagation on the top layers, the system will communicate their parameters while performing back-propagation on the bottom layers. This spreads the computation and communication out in an optimal fashion, in essence "overlapping 90% computation with 90% communication."

Update prioritization. Another communication management strategy is to prioritize available bandwidth, by focusing on communicating updates (or parts of) Δ that contribute most to convergence. This idea has a close relationship with SAP, discussed in Section 3.1. While SAP prioritizes computation toward more important parameters, update prioritization ensures that the changes to these important parameters are quickly propagated to other worker machines, so that their effects are immediately felt. As a concrete example, in ML algorithms that use SGD (e.g., logistic regression and Lasso regression), the objective function $\mathcal{L}$ changes proportionally to the parameters A_j, and hence the fastest-changing parameters A_j are often the largest contributors to solution quality.

Thus, the SSP implementation can be further augmented by a prioritizer, which rearranges the updates in the rate limiter's outgoing queue, so that more important updates will be sent out first. The prioritizer can support strategies such as the following:

(1) Absolute magnitude prioritization: Updates to parameters A_j are re-ordered according to their recent accumulated change $|\delta_j|$, which works well for ML algorithms that use SGD.

(2) Relative magnitude prioritization: This is the same as absolute magnitude, but the sorting criteria is δ_j/A_j, that is, the accumulated change normalized by the current parameter value A_j. Empirically, these prioritization strategies already yield another 25% speedup, on top of SSP and continuous communication [41], and there is potential to explore strategies tailored to a specific ML program (similar to the SAP prioritization criteria for Lasso).

Parameter storage and communication topologies. A third communication management strategy is to consider the placement of model parameters A across the network (parameter storage), as well as the network routes along which parameter updates Δ should be communicated (communication topologies). The choice of parameter storage strongly influences the communication topologies that can be used, which in turn impacts the speed at which parameter updates Δ can be delivered over the network (as well as their staleness). Hence, we begin by discussing two commonly used paradigms for storing model parameters (Fig. 10):

(1) Centralized storage: A "master view" of the parameters A is partitioned across a set of server machines, while worker machines maintain local views of the parameters. Communication is asymmetric in the following sense: Updates Δ are sent from workers to servers, and workers receive the most up-to-date version of the parameters A from the server.

(2) Decentralized storage: Every worker maintains its own local view of the parameters, without a centralized server. Communication is symmetric: Workers send updates Δ to each other, in order to bring their local views of A up to date.

The centralized storage paradigm can be supported by a master-slave network topology (Fig. 11), where machines are organized into a bipartite graph with servers on one side, and workers on the other; whereas the decentralized storage paradigm can be supported by a peer-to-peer (P2P) topology (Fig. 12), where each worker machine broadcasts to all other workers. An advantage of the master-slave network topology is that it reduces the number of messages that need to be sent over the network: Workers only need to send updates Δ to the servers, which aggregate them using F, and update

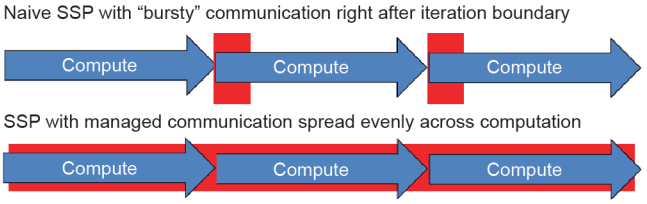

Fig. 9. Managed communication in SSP spreads network communication evenly across the duration of computation, instead of sending all updates at once right after the iteration boundary.

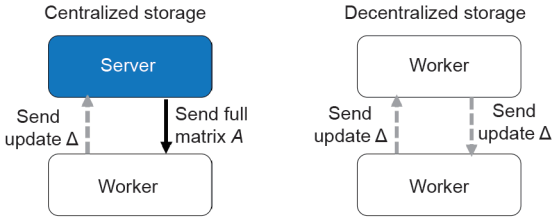

Fig. 10. Two paradigms for parameter storage: centralized and decentralized. Note that both paradigms have different communication styles: Centralized storage communicates updates Δ from workers to servers, and actual parameters A from servers to workers; decentralized storage only communicates updates Δ between workers.

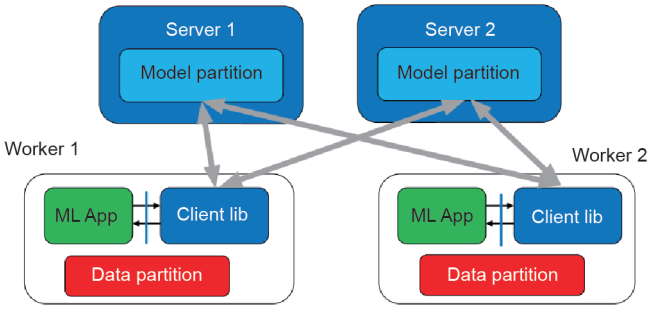

Fig. 11. Master-slave (bipartite) network topology for centralized parameter storage. Servers only communicate with workers, and vice versa. There is no server-server or worker-worker communciation.

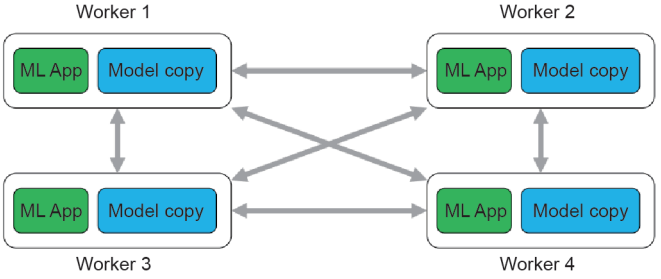

Fig. 12. Peer-to-peer (P2P) network topology for decentralized parameter storage. All workers may communicate with any other worker.

the master view of the parameters A. The updated parameters can then be broadcast to the workers as a single message, rather than as a collection of individual updates Δ. In total, only $O(P)$ messages need to be sent. In contrast, P2P topologies must send $O(P^2)$ messages every iteration, because each worker must broadcast Δ to every other worker.

However, when δ has a compact or compressible structure—such as low-rank-ness in matrix-parameterized ML programs such as deep learning, or sparsity in Lasso regression—the P2P topology can achieve considerable communication savings over the master-slave topology. By compressing or re-representing Δ in a more compact low-rank or sparse form, each of the $O(P^2)$ P2P messages can be made much smaller than the $O(P)$ master-to-slave messages, which may not admit compression (because the messages consist of the actual parameters A, not the compressible updates Δ). Furthermore, even the $O(P^2)$ P2P messages can be reduced, by switching from a full P2P to a partially connected Halton sequence topology (Fig. 13) [65], where each worker only communicates with a subset of workers. Workers can reach any other worker by routing messages through intermediate nodes. For example, the routing path 1→2→5→6 is one way to send a message from worker 1 to worker 6. The intermediate nodes can combine messages meant for the same destination, thus reducing the number of messages per iteration (and further reducing network load). However, one drawback to the Halton sequence topology is that routing increases the time taken for messages to reach their destination, which raises the average staleness of parameters under the SSP bridging model. For example, the message from worker 1 to worker 6 would be three iterations stale. The Halton sequence topology is nevertheless a good option for very large cluster networks, which have limited P2P bandwidth.

By combining the various aspects of "how to communicate"—continuous communication, update prioritization, and a suitable combination of parameter storage and communication topology—we can design a distributed ML system that enjoys multiplicative speed benefits from each aspect, resulting in an almost order-of-magnitude speed improvement on top of what SAP (how to distribute) and SSP (bridging models) can offer. For example, the Bösen SSP system enjoys up to an additional four-fold speedup from continuous communication and update prioritization, as shown in Figs. 14 and 15 [41].

3.4. What to communicate

Going beyond how to store and communicate updates Δ between worker machines, we may also ask "what" needs to be communicated in each update Δ. In particular, is there any way to reduce the number of bytes required to transmit Δ, and thus further alleviate the comunication bottleneck in distributed ML programs [55]? This question is related to the idea of lossless compression in operation-centric programs; for example, Hadoop MapReduce is able to compresses key-value pairs (a, b) to reduce their transmission cost from Mappers to Reducers. For data parallel ML programs, a commonly used strategy for reducing the size of Δ messages is to aggregate (i.e., sum) them before transmission over the network, taking advantage of the additive structure within F (such as in the Lasso data parallel example, Eq. (10)). Such early aggregation is preferred for centralized parameter storage paradigms that communicate full parameters A from servers to workers [37,40], and it is natural to ask if there are other strategies that may perhaps be better suited to different storage paradigms.

To answer this question, we may inspect the mathematical structure of ML parameters A, and the nature of their updates Δ. A number of popular ML programs have matrix-structured parameters $\mathbf{A}$ (we use boldface to distinguish from the generic A). Examples include multiclass logistic regression (MLR), neural networks (NN) [60], distance metric learning (DML) [66], and sparse coding [23]. We refer to these as matrix-parameterized models (MPMs), and note that $\mathbf{A}$ can be very large in modern applications: In one application of MLR to Wikipedia [67], $\mathbf{A}$ is a 325K-by-10K matrix containing several billion entries (tens of gigabytes). It is also worth pointing out that typical computer cluster networks can at most transmit

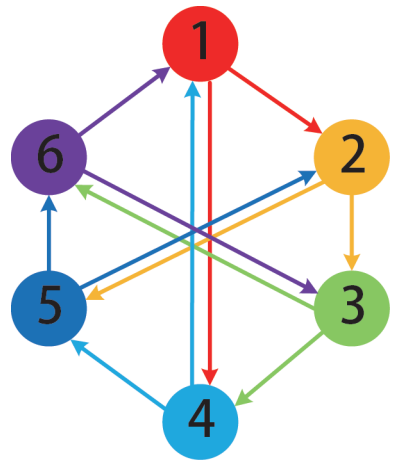

Fig. 13. Halton sequence topology for decentralized parameter storage. Workers may communicate with other workers through an intermediate machine; for example, worker 1 can reach worker 5 by relaying updates Δ through worker 2.

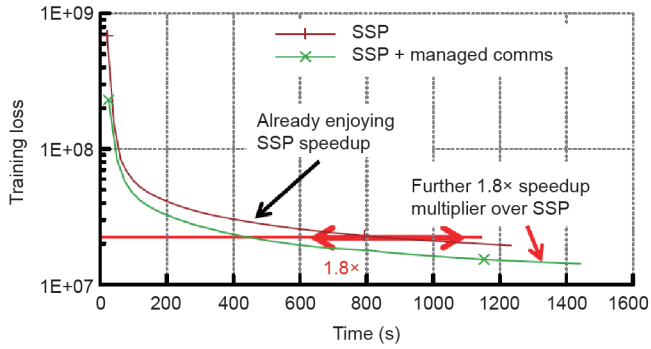

Fig. 14. Matrix factorization: Continuous communication with SSP achieves a further 1.8-times improvement in convergence time over SSP alone. Experiment settings: Netflix dataset with rank 400, on eight machines (16 cores each) and gigabit ethernet (GbE). Adapted from Ref. [41].

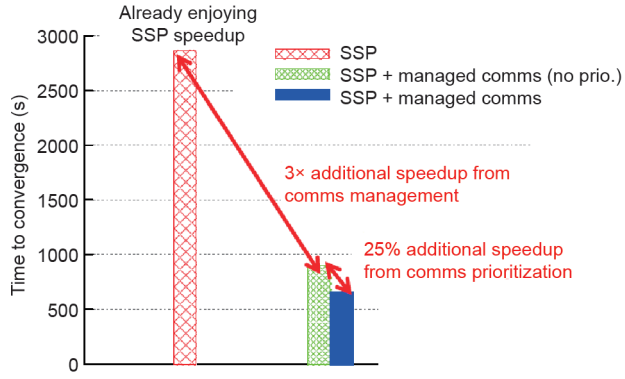

Fig. 15. Latent Dirichlet allocation topic modeling: Continuous communication with SSP achieves a further three-times improvement in convergence time over SSP alone. Moreover, if update prioritization is also enabled, the convergence time improves by another 25%. Experiment settings: NYTimes dataset with 1000 topics, on 16 machines (16 cores each) and GbE. Adapted from Ref. [41].

a few gigabytes per second between two machines; hence, naive synchronization of such matrices $\mathbf{A}$ and their updates Δ is not instantaneous. Because parameter synchronization occurs many times across the lifetime of an iterative-convergent ML program, the time required for synchronization can become a substantial bottleneck.

More formally, an MPM is an ML objective function with the following specialized form:

$$\mathcal{L}(\mathbf{x},\mathbf{A}) = \min_{\mathbf{A}}\left[\frac{1}{N}\sum_{i=1}^{N} f_i(\mathbf{A}\mathbf{u}_i,\mathbf{v}_i)\right] + r(\mathbf{A}) \quad (16)$$

where, the model parameters are a K-by-D matrix $\mathbf{A} \in \mathbb{R}^{K\times D}$; each loss function f_i is defined over $\mathbf{A}$ and the data samples $\mathbf{x} = \{(\mathbf{u}_i,\mathbf{v}_i)\}_{i=1}^{N}$. Specifically, f_i must depend on the product $\mathbf{A}\mathbf{u}_i$ (and not on $\mathbf{A}$ or $\mathbf{u}_i$ individually). $r(\mathbf{A})$ is a structure-inducing function such as a regularizer. A well-known example of Eq. (16) is MLR, which is used in classification problems involving tens of thousands of classes K (e.g., web data collections such as Wikipedia). In MLR, $\mathbf{A}$ is the weight coefficient matrix, $\mathbf{u}_i$ is the D-dimensional feature vector of data sample i, $\mathbf{v}_i$ is a K-dimensional indicator vector representing the class label of data sample i, and the loss function f_i is composed of a cross-entropy error function and a softmax mapping of $\mathbf{A}\mathbf{u}_i$. A key property of MPMs is that each update Δ is a low-rank matrix and can be factored into small vectors, called sufficient factors, that are cheap to transmit over the network.

Sufficient factor broadcasting (SFB). In order to exploit the sufficient factor property in MPMs, let us look closely at the updates Δ. The ML objective function Eq. (16) can be solved by either the stochastic proximal gradient descent (SPGD) [37,52,60,65] or stochastic dual coordinate ascent (SDCA) [68–72] algorithmic techniques, among others. For example, in SPGD, the update function Δ can be decomposed into a sum over vectors $\mathbf{b}_i\mathbf{c}_i^{\mathrm{T}}$, where $\mathbf{b}_i = \frac{\partial f(\mathbf{A}\mathbf{u}_i,\mathbf{v}_i)}{\partial f(\mathbf{A}\mathbf{u}_i)}$ and $\mathbf{c}_i = \mathbf{u}_i$; SDCA updates Δ also admit a similar decomposition[†] [55]. Instead of transmitting $\Delta = \sum_i \mathbf{b}_i\mathbf{c}_i^{\mathrm{T}}$ (total size KD) between workers, we can instead transmit the individual vectors $\mathbf{b}_i$ and $\mathbf{c}_i$ (total size $S(K+D)$, where S is the number of data samples processed in the current iteration), and reconstruct Δ at the destination machine.

This sufficient factor broadcasting (SFB) strategy is well-suited to decentralized storage paradigms, where only updates Δ are transmitted between workers. It may also be applied to centralized storage paradigms, though only for transmissions from workers to servers; the server-to-worker direction sends full matrices $\mathbf{A}$ that no longer have the sufficient factor property [60]. At this point, it is natural to ask how the combination of decentralized storage and SFB interacts with the SSP bridging model: Will the ML algorithm still output the correct answer under such a P2P setting? The following theorem provides an affirmative answer.

Theorem 5 (adapted from Ref. [55])**: SFB under SSP, convergence theorem.** *Let $\mathbf{A}_p(t)$, $p = 1,\ldots,P$, and $\mathbf{A}(t)$ be the local worker views and a "reference" view respectively, for the ML objective function $\mathcal{L}$ in Eq.* (16) *(assuming $r \equiv 0$) being solved by SFB under the SSP bridging model with staleness s. Under mild assumptions, we have*

(1) $\lim_{t\to\infty}\max_p \|\mathbf{A}(t) - \mathbf{A}_p(t)\| = 0$, that is, the local worker views converge to the reference view, implying that all worker views will be the same after sufficient iterations t.

(2) There exists a common subsequence of $\mathbf{A}_p(t)$ and $\mathbf{A}(t)$ that converges almost surely to a stationary point of $\mathcal{L}$, with rate $O\left(\frac{Ps\log(t)}{\sqrt{t}}\right)$.

Intuitively, Theorem 5 says that all local worker views $\mathbf{A}_p(t)$ eventually converge to stationary points (local minima) of the objective function $\mathcal{L}$, even though updates Δ can be stale by up to s iterations. Thus, SFB under decentralized storage is robust under the SSP bridging model—which is especially useful for topologies such as the Halton sequence that increase the staleness of updates, in exchange for lower bandwidth usage.

Empirically, SFB can greatly reduce the communication costs for MPMs: For a variety of MPMs, Fig. 16 shows the time taken to reach a fixed objective value using the BSP bridging model. MPMs running under SFB converge faster than when running under a centralized storage paradigm that transmits full updates Δ (referred to as "full matrix synchronization" or FMS). We also compare MPMs running under SFB to baseline implementations included with Spark v1.3.1

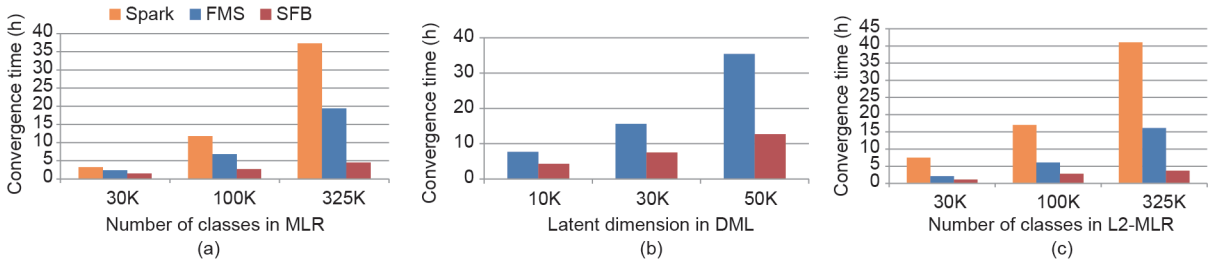

Fig. 16. Convergence time versus model size for (a) multiclass logistic regression (MLR), (b) distance metric learning (DML), and (c) L2-MLR.

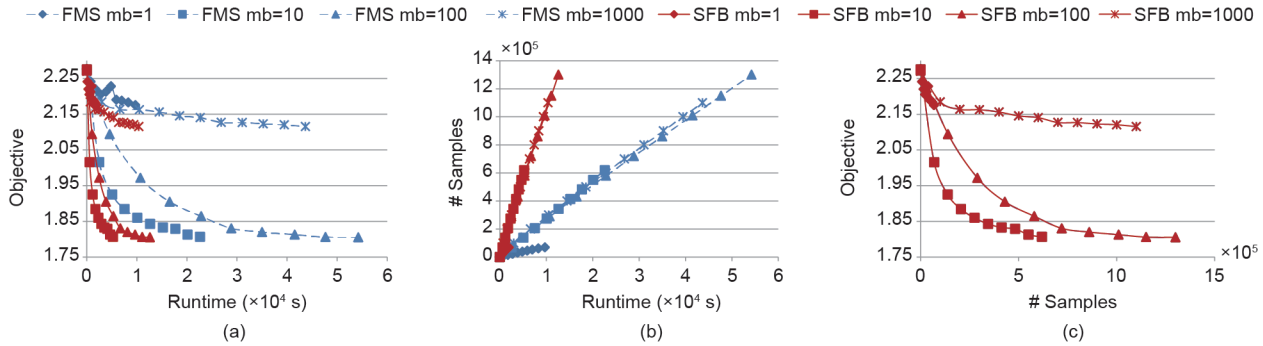

Fig. 17. (a) MLR objective versus runtime; (b) samples versus runtime; (c) objective versus samples.

[†] More generally, $\mathbf{b}_i$ and $\mathbf{c}_i$ may be "thin matrices" instead of vectors. SFB works as long as $\mathbf{b}_i$ and $\mathbf{c}_i$ are much smaller than $\mathbf{A}$.

(not all MPMs being evaluated are available on Spark). This is because SFB has lower communication costs, so a greater proportion of algorithm running time is spent on computation instead of on network waiting; we show this in Fig. 17, which plots data samples processed per second (i.e., iteration throughput) and algorithm progress per sample (i.e., progress per iteration) for MLR, under BSP consistency and varying minibatch sizes. Fig. 17(b) shows that SFB processes far more samples per second than FMS, while Fig. 17(c) shows that SFB and FMS yield exactly the same algorithm progress per sample under BSP.

To understand the impact of SFB on Δ communication costs, let us examine Fig. 18, which shows the total computation time as well as the network communication time required by SFB and FMS to converge, across a range of SSP staleness values. In general, higher Δ communication costs and lower staleness will increase the time the ML program spends waiting for network communication. For all staleness values, SFB requires far less network waiting (because SFBs are much smaller than full matrices in FMS). Computation time for SFB is slightly longer than for FMS because ① update matrices Δ must be reconstructed on each SFB worker, and ② SFB requires a few more iterations for convergence than FMS, due to slightly higher average parameter staleness compared with FMS. Overall, SFB's reduction in network waiting time far surpasses the added computation time, and hence SFB outperforms FMS.

As a final note, there are situations that naturally call for a hybrid of SFB and full Δ transmission. A good example is deep learning using convolutional neural networks (previously discussed under the topic of wait-free back-propagation in Section 3.3): The top layers of a typical CNN are fully connected and use matrix parameters containing millions of elements, whereas the bottom layers are convolutional and involve tiny matrices with at most a few hundred elements. It follows that it is more efficient to ① apply SFB to the top layers' updates (transmission cost is $S(K + D) \ll KD$ because K and D are large relative to S); and ② aggregate (sum) the bottom layers' updates before transmission (cost is $KD \ll S(K + D)$ because S is large relative to K and D) [56].

4. Petuum: A realization of the ML system design principles

We conclude this paper by noting that the four principles of ML system design have been partially realized by systems that are highly specialized for one or a few ML programs [28,31,36,58,60]. This presents ML practitioners with a choice between the aforementioned monolithic yet high-performance "towers" (specialized systems that require substantial engineering to maintain and upgrade), or the more general-purpose yet slower "platforms" such as Hadoop and Spark (which are relatively easy to deploy and maintain). In order to address this dichotomy, we have realized the principles of ML system design in the Petuum distributed ML framework [35], whose architecture is outlined in Fig. 19. The intent behind Petuum is to provide a generic distributed system for ML algorithms running on big data, by abstracting system implementation details and the four design principles away from the ML programmer—who is then freed to focus on programming the key ML functions $\mathcal{L}$, Δ, and F.

Compared to general-purpose distributed programming platforms for operation-centric programs (such as Hadoop and Spark), Petuum takes advantage of the unique properties of iterative-convergence ML programs—error tolerance, dependency structures, non-uniform convergence, and compact updates—in order to improve both the convergence rate and per-iteration time for ML algorithms, and thus achieve close-to-ideal P-fold speedup with P machines. Petuum runs on compute clusters and cloud computing, supporting from tens to thousands of machines, and provides programming interfaces for C++ and Java, while also supporting Yet Another Resource Negotiator (YARN) and Hadoop Distributed File System (HDFS) to allow execution on existing Hadoop clusters. Two major systems underlie Petuum (Fig. 19):

(1) Bösen, a bounded-asynchronous distributed key-value store for data parallel ML programming: Bösen uses the SSP consistency model, which allows asynchronous-like performance that outperforms MapReduce and bulk synchronous execution, yet does not sacrifice ML algorithm correctness.

(2) Strads, a dynamic scheduler for model parallel ML programming: Strads performs fine-grained scheduling of ML update operations, prioritizing computation on the parts of the ML program that need it most, while avoiding unsafe parallel operations that could lead to non-convergence in ML programs.

Currently, Petuum features an ML library with over 10 ready-to-run algorithms (implemented on top of Bösen and Strads), including classic algorithms such as logistic regression, K-means, and random forest, and newer algorithms such as supervised topic models (MedLDA), deep learning, distance metric learning, and sparse coding. In particular, the Petuum deep learning system, Poseidon [56], fully exemplifies the "platform" nature of Petuum: Poseidon takes the well-established but single-machine Caffe project[†], and turns it into a distributed GPU system by replacing the memory access routines within Caffe with the Bösen distributed key-value store's distributed shared memory programming interfaces. The biggest advantage of this platform approach is familiarity and usability—existing Caffe users do not have to learn a new tool in order to take advantage of GPUs distributed across a cluster.

Looking toward the future, we envision that Petuum might become the foundation of an ML distributed cluster operating system that provides a single-machine or laptop-like experience for ML application users and programmers, while making full use of the computational capacity provided by datacenter-scale clusters with thou-

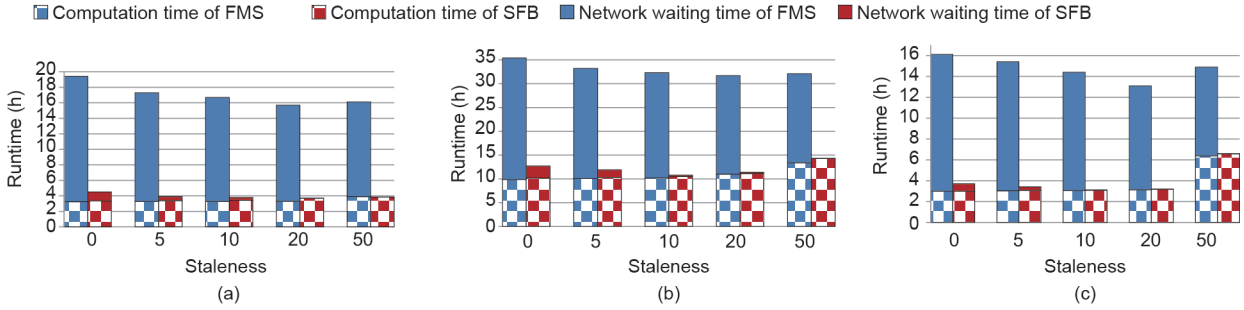

Fig. 18. Computation time versus network waiting time for (a) MLR, (b) DML, and (c) L2-MLR.

[†] http://caffe.berkeleyvision.org/

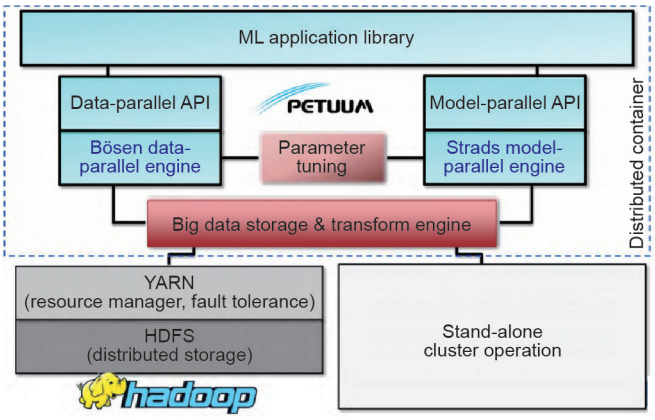

Fig. 19. Architecture of Petuum, a distributed ML system for big data and big models. API: application programming interface; YARN: Yet Another Resource Negotiator; HDFS: Hadoop Distributed File System.

sands of machines. Achieving this vision will certainly require new systems such as containerization, cluster resource management and scheduling, and user interfaces to be developed, which are necessary steps to reduce the substantial human or operational cost of deploying massive-scale ML applications in a datacenter environment. By building such systems into the ML-centric Petuum platform—which reduces the capital cost of ML applications by enabling them to run faster on fewer machines—we can thus prepare for the eventual big data computational shift from database-style operations to ML-style operations.

Compliance with ethics guidelines

Eric P. Xing, Qirong Ho, Pengtao Xie, and Dai Wei declare that they have no conflict of interest or financial conflicts to disclose.

References

[1] Airoldi EM, Blei DM, Fienberg SE, Xing EP. Mixed membership stochastic blockmodels. J Mach Learn Res 2008;9:1981–2014.
[2] Ahmed A, Ho Q, Eisenstein J, Xing EP, Smola AJ, Teo CH. Unified analysis of streaming news. In: Proceedings of the 20th International Conference on World Wide Web; 2011 Mar 28–Apr 1; Hyderabad, India; 2011. p. 267–76.
[3] Zhao B, Xing EP. Quasi real-time summarization for consumer videos. In: Proceedings of the 2014 IEEE Conference on Computer Vision and Pattern Recognition (CVPR); 2014 Jun 23–28; Columbus, OH, USA; 2014. p. 2513–20.
[4] Lee S, Xing EP. Leveraging input and output structures for joint mapping of epistatic and marginal eQTLs. Bioinformatics 2012;28(12):i137–46.
[5] Thrun S, Montemerlo M, Dahlkamp H, Stavens D, Aron A, Diebel J, et al. Stanley: the robot that won the DARPA Grand Challenge. J Field Robot 2006;23(9):661–92.
[6] Chandola V, Banerjee A, Kumar V. Anomaly detection: a survey. ACM Comput Surv 2009;41(3):15:1–15:58.
[7] Wainwright MJ, Jordan MI. Graphical models, exponential families, and variational inference. Hanover: Now Publishers Inc.; 2008.
[8] Koller D, Friedman N. Probabilistic graphical models: principles and techniques. Cambridge: MIT Press; 2009.
[9] Xing EP. Probabilistic graphical models [Internet]. [cited 2016 Jan 1]. Available from: https://www.cs.cmu.edu/~epxing/Class/10708/lecture.html.
[10] Zhu J, Xing EP. Maximum entropy discrimination markov networks. J Mach Learn Res 2009;10:2531–69.
[11] Zhu J, Ahmed A, Xing EP. MedLDA: maximum margin supervised topic models for regression and classication. In: Proceedings of the 26th Annual International Conference on Machine Learning; 2009 Jun 14–18; Montreal, Canada; 2009. p. 1257–64.
[12] Zhu J, Chen N, Xing EP. Bayesian inference with posterior regularization and applications to innite latent SVMs. J Mach Learn Res 2014;15(1):1799–847.
[13] Griffiths TL, Ghahramani Z. Infinite latent feature models and the Indian buffet process. In: Weiss Y, Schölkopf B, Platt JC, editors Proceedings of the Neural Information Processing Systems 2005; 2005 Dec 5–8; Vancouver, Canada; 2005. p. 475–82.
[14] Teh YW, Jordan MI, Beal MJ, Blei DM. Hierarchical dirichlet processes. J Am Stat Assoc 2006;101(476):1566–81.
[15] Yuan M, Lin Y. Model selection and estimation in regression with grouped variables. J R Stat Soc B 2006;68(1):49–67.
[16] Kim S, Xing EP. Tree-guided group lasso for multi-response regression with structured sparsity, with applications to eQTL mapping. Ann Appl Stat 2012;6(3):1095–117.
[17] Burges CJC. A tutorial on support vector machines for pattern recognition. Wires Data Min Knowl 1998;2(2):121–67.
[18] Taskar B, Guestrin C, Koller D. Max-margin Markov networks. In: Thrun S, Saul LK, Schölkopf B, editors Proceedings of the Neural Information Processing Systems 2003; 2003 Dec 8–13; Vancouver and Whistler, Canada; 2003. p. 25–32.
[19] Hinton G, Deng L, Yu D, Dahl GE, Mohamed A, Jaitly N, et al. Deep neural networks for acoustic modeling in speech recognition: the shared views of four research groups. IEEE Signal Proc Mag 2012;29(6):82–97.
[20] Krizhevsky A, Sutskever I, Hinton GE. ImageNet classification with deep convolutional neural networks. In: Pereira F, Burges CJC, Bottou L, Weinberger KQ, editors Proceedings of the Neural Information Processing Systems 2012; 2012 Dec 3–8, Lake Tahoe, USA; 2012. p. 1097–105.
[21] Lee DD, Seung HS. Learning the parts of objects by non-negative matrix factorization. Nature 1999;401(6755):788–91.
[22] Salakhutdinov R, Mnih A. Probabilistic matrix factorization. In: Platt JC, Koller D, Singer Y, Roweis ST, editors Proceedings of the Neural Information Processing Systems 2007; 2007 Dec 3–6; Vancouver, Canada; 2007. p.1257–64.
[23] Olshausen BA, Field DJ. Sparse coding with an overcomplete basis set: a strategy employed by V1? Vision Res 1997;37(23):3311–25.
[24] Lee H, Battle A, Raina R, Ng AY. Efficient sparse coding algorithms. In: Schölkopf B, Platt JC, Hoffman T, editors Proceedings of the Neural Information Processing Systems 2006; 2006 Dec 4–7; Vancouver, Canada; 2006. p. 801–8.
[25] Zheng X, Kim JK, Ho Q, Xing EP. Model-parallel inference for big topic models. 2014. Eprint arXiv:1411.2305.
[26] Yuan J, Gao F, Ho Q, Dai W, Wei J, Zheng X, et al. LightLDA: big topic models on modest compute clusters. 2014. Eprint arXiv:1412.1576.
[27] Coates A, Huval B, Wang T, Wu DJ, Ng AY, Catanzaro B. Deep learning with COTS HPC systems. In: Proceedings of the 30th International Conference on Machine Learning; 2013 Jun 16–21; Atlanta, GA, USA; 2013. p. 1337–45.
[28] Ahmed A, Aly M, Gonzalez J, Narayanamurthy S, Smola AJ. Scalable inference in latent variable models. In: Proceedings of the 5th International Conference on Web Search and Data Mining; 2012 Feb 8–12; Seattle, WA, USA; 2012. p. 123–32.
[29] Moritz P, Nishihara R, Stoica I, Jordan MI. SparkNet: training deep networks in spark. 2015. Eprint arXiv:1511.06051.
[30] Agarwal A, Duchi JC. Distributed delayed stochastic optimization. In: Shawe-Taylor J, Zemel RS, Bartlett PL, Pereira F, Weinberger KQ, editors Proceedings of the Neural Information Processing Systems 2011; 2011 Dec 12–17; Granada, Spain; 2011. p. 873–81.
[31] Niu F, Recht B, Re C, Wright SJ. HOGWILD!: a lock-free approach to parallelizing stochastic gradient descent. In: Shawe-Taylor J, Zemel RS, Bartlett PL, Pereira F, Weinberger KQ, editors Proceedings of the Neural Information Processing Systems 2011; 2011 Dec 12–17; Granada, Spain; 2011. p. 693–701.
[32] Dean J, Ghemawat S. MapReduce: simplified data processing on large clusters. Commun ACM 2008;51(1):107–13.
[33] Gonzalez JE, Low Y, Gu H, Bickson D, Guestrin C. PowerGraph: distributed graph-parallel computation on natural graphs. In: Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation; 2012 Oct 8–10; Hollywood, CA, USA; 2012. p. 17–30.
[34] Zaharia M, Chowdhury M, Das T, Dave A, Ma J, McCauley M, et al. Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In: Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation; 2012 Apr 25–27; San Jose, CA, USA; 2012. p. 2:1–2:14.
[35] Xing EP, Ho Q, Dai W, Kim JK, Wei J, Lee S, et al. Petuum: a new platform for distributed machine learning on big data. IEEE Trans Big Data 2015;1(2):49–67.
[36] Li M, Andersen DG, Park JW, Smola AJ, Ahmed A, Josifovski V, et al. Scaling distributed machine learning with the parameter server. In: Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation; 2014 Oct 6–8; Broomfield, CO, USA; 2014. p. 583–98.
[37] Ho Q, Cipar J, Cui H, Kim JK, Lee S, Gibbons PB, et al. More effective distributed ML via a stale synchronous parallel parameter server. In: Burges CJC, Bottou L, Welling M, Ghahramani Z, Weinberger KQ, editors Proceedings of the Neural Information Processing Systems 2013; 2013 Dec 5–10; Lake Tahoe, USA; 2013. p. 1223–31.
[38] Kumar A, Beutel A, Ho Q, Xing EP. Fugue: slow-worker-agnostic distributed learning for big models on big data. In: Kaski S, Corander J, editors Proceedings of the 17th International Conference on Artificial Intelligence and Statistics (AISTATS) 2014; 2014 Apr 22–25; Reykjavik, Iceland; 2014. p. 531–9.
[39] Lee S, Kim JK, Zheng X, Ho Q, Gibson GA, Xing EP. On model parallelization and scheduling strategies for distributed machine learning. In: Ghahramani Z, Welling M, Cortes C, Lawrence ND, Weinberger KQ, editors Proceedings of the Neural Information Processing Systems 2014; 2014 Dec 8–13; Montreal, Canada; 2014. p. 2834–42.
[40] Dai W, Kumar A, Wei J, Ho Q, Gibson G, Xing EP. High-performance distributed ML at scale through parameter server consistency models. In: Proceedings of the 29th AAAI Conference on Artificial Intelligence; 2015 Jan 25–30; Austin, TX, USA; 2015. p. 79–87.
[41] Wei J, Dai W, Qiao A, Ho Q, Cui H, Ganger GR, et al. Managed communication and consistency for fast data-parallel iterative analytics. In: Proceedings of the 6th ACM Symposium on Cloud Computing; 2015 Aug 27–29; Kohala Coast, HI, USA, 2015. p. 381–94.
[42] Bottou L. Large-scale machine learning with stochastic gradient descent. In: Lechevallier Y, Saporta G, editors Proceedings of COMPSTAT'2010; 2010 Aug

22–27; Paris France. New York: Springer; 2010. p. 177–86.
[43] Zhou Y, Yu Y, Dai W, Liang Y, Xing EP. On convergence of model parallel proximal gradient algorithm for stale synchronous parallel system. In: Gretton A, Robert CC, editors Proceedings of the 19th International Conference on Artificial Intelligence and Statistics (AISTATS) 2016; 2016 May 7–11; Cadiz, Spain; 2016. p. 713–22.
[44] Fercoq O, Richtárik P. Accelerated, parallel and proximal coordinate descent. SIAM J Optim 2013;25(4):1997–2023.
[45] Gilks WR. Markov Chain Monte Carlo. In: Encyclopedia of biostatistics. 2nd ed. New York: John Wiley and Sons, Inc., 2005.
[46] Tibshirani R. Regression shrinkage and selection via the lasso. J R Statist Soc B 1996;58(1):267–88.
[47] Blei DM, Ng AY, Jordan MI. Latent dirichlet allocation. J Mach Learn Res 2003;3:993–1022.
[48] Yao L, Mimno DM, McCallum A. Effcient methods for topic model inference on streaming document collections. In: Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining; 2009 Jun 28–Jul 1; Paris, France; 2009. p. 937–46.
[49] Dean J, Ghemawat S. MapReduce: simplified data processing on large clusters. In: Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation—Volume 6; 2004 Dec 6–8; San Francisco, CA, USA; 2004. p. 137–50.
[50] Zhang T. Solving large scale linear prediction problems using stochastic gradient descent algorithms. In: Proceedings of the 21st International Conference on Machine Learning; 2004 Jul 4–8; Banff, Canada; 2004. p. 116.
[51] Gemulla R., Nijkamp E, Haas PJ, Sismanis Y. Large-scale matrix factorization with distributed stochastic gradient descent. In Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining; 2011 Aug 21–24; San Diego, CA, USA; 2011. p. 69–77.
[52] Dean J, Corrado G, Monga R, Chen K, Devin M, Mao M, et al. Large scale distributed deep networks. In: Pereira F, Burges CJC, Bottou L, Weinberger KQ, editors Proceedings of the Neural Information Processing Systems 2012; 2012 Dec 3–8, Lake Tahoe, USA; 2012. p. 1232–40.
[53] Low Y, Gonzalez J, Kyrola A, Bickson D, Guestrin C, Hellerstein JM. GraphLab: a new framework for parallel machine learning. In: Proceedings of the 26th Conference on Uncertainty in Artificial Intelligence (UAI 2010); 2010 Jul 8–11, Catalina Island, CA, USA; 2010.
[54] Hinton GE, Salakhutdinov RR. Reducing the dimensionality of data with neural networks. Science 2006;313(5786):504–7.
[55] Xie P, Kim JK, Zhou Y, Ho Q, Kumar A, Yu Y, et al. Distributed machine learning via sufficient factor broadcasting. 2015. Eprint arXiv:1409.5705.
[56] Zhang H, Hu Z, Wei J, Xie P, Kim G, Ho Q, et al. Poseidon: a system architecture for effcient GPU-based deep learning on multiple machines. 2015. Eprint arXiv:1512.06216.
[57] Bradley JK, Kyrola A, Bickson D, Guestrin C. Parallel coordinate descent for L_1-regularized loss minimization. In: Proceedings of the 28th International Conference on Machine Learning; 2011 Jun 28–Jul 2; Bellevue, WA, USA; 2011.
[58] Scherrer C, Tewari A, Halappanavar M, Haglin D. Feature clustering for accelerating parallel coordinate descent. In: Pereira F, Burges CJC, Bottou L, Weinberger KQ, editors Proceedings of the Neural Information Processing Systems 2012; 2012 Dec 3–8, Lake Tahoe, USA; 2012. p. 28–36.
[59] Low Y, Gonzalez J, Kyrola A, Bickson D, Guestrin C, Hellerstein JM. Distributed GraphLab: a framework for machine learning and data mining in the cloud. In: Proceedings of the VLDB Endowment; 2012 Aug 27–31; Istanbul, Turkey; 2012;5(8): 716–27.
[60] Chilimbi T, Suzue Y, Apacible J, Kalyanaraman K. Project Adam: building an efficient and scalable deep learning training system. In: Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation; 2014 Oct 6–8; Broomfield, CO, USA; 2014. p. 571–82.
[61] Valiant LG. A bridging model for parallel computation. Commun ACM 1990;33(8):103–11.
[62] McColl WF. Bulk synchronous parallel computing. In: Davy JR, Dew PM, editors Abstract machine models for highly parallel computers. Oxford: Oxford University Press; 1995. p. 41–63.
[63] Malewicz G, Austern MH, Bik AJC, Dehnert JC, Horn I, Leiser N, et al. Pregel: a system for large-scale graph processing. In: Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data; 2010 Jun 6–11; Indianapolis, IN, USA; 2010. p. 135–46.
[64] Terry D. Replicated data consistency explained through baseball. Commun ACM 2013;56(12):82–9.
[65] Li H, Kadav A, Kruus E, Ungureanu C. MALT: distributed data-parallelism for existing ML applications. In: Proceedings of the 10th European Conference on Computer Systems; 2015 Apr 21–25; Bordeaux, France; 2015. Article No.: 3.
[66] Xing EP, Jordan MI, Russell SJ, Ng AY. Distance metric learning with application to clustering with side-information. In: Becker S, Thrun S, Obermayer K, editors Proceedings of the Neural Information Processing Systems 2002; 2002 Dec 9–14; Vancouver, Canada; 2002. p. 505–12.
[67] Partalas I, Kosmopoulos A, Baskiotis N, Artieres T, Paliouras G, Gaussier E, et al. LSHTC: A benchmark for large-scale text classification. 2015. Eprint arXiv:1503.08581.
[68] Hsieh CJ, Chang KW, Lin CJ, Sathiya Keerthi S, Sundararajan S. A dual coordinate descent method for large-scale linear SVM. In: Proceedings of the 25th International Conference on Machine Learning; 2008 Jul 5–9; Helsinki, Finland; 2008. p. 408–15.
[69] Shalev-Shwartz S, Zhang T. Stochastic dual coordinate ascent methods for regularized loss. J Mach Learn Res 2013;14(1):567–99.
[70] Yang T. Trading computation for communication: distributed stochastic dual coordinate ascent. In: Burges CJC, Bottou L, Welling M, Ghahramani Z, Weinberger KQ, editors Proceedings of the Neural Information Processing Systems 2013; 2013 Dec 5–10; Lake Tahoe, USA; 2013. p. 629–37.
[71] Jaggi M, Smith V, Takac M, Terhorst J, Krishnan S, Hofmann T, et al. Communication-efficient distributed dual coordinate ascent. In: Ghahramani Z, Welling M, Cortes C, Lawrence ND, Weinberger KQ, editors Proceedings of the Neural Information Processing Systems 2014; 2014 Dec 8–13; Montreal, Canada; 2014. p. 3068–76.
[72] Hsieh CJ, Yu HF, Dhillon IS. PASSCoDe: parallel asynchronous stochastic dual co-ordinate descent. In: Bach F, Blei D, editors Proceedings of the 32nd International Conference on Machine Learning; 2015 Jul 6–11; Lille, France; 2015. p. 2370–9.