Petuum: A New Platform for Distributed Machine Learning on Big Data

Eric P. Xing, Qirong Ho, Wei Dai, Jin Kyu Kim, Jinliang Wei, Seunghak Lee, Xun Zheng, Pengtao Xie, Abhimanu Kumar, and Yaoliang Yu

Abstract—What is a systematic way to efficiently apply a wide spectrum of advanced ML programs to industrial scale problems, using Big Models (up to 100s of billions of parameters) on Big Data (up to terabytes or petabytes)? Modern parallelization strategies employ fine-grained operations and scheduling beyond the classic bulk-synchronous processing paradigm popularized by MapReduce, or even specialized graph-based execution that relies on graph representations of ML programs. The variety of approaches tends to pull systems and algorithms design in different directions, and it remains difficult to find a universal platform applicable to a wide range of ML programs at scale. We propose a general-purpose framework, Petuum, that systematically addresses data- and model-parallel challenges in large-scale ML, by observing that many ML programs are fundamentally optimization-centric and admit error-tolerant, iterative-convergent algorithmic solutions. This presents unique opportunities for an integrative system design, such as bounded-error network synchronization and dynamic scheduling based on ML program structure. We demonstrate the efficacy of these system designs versus well-known implementations of modern ML algorithms, showing that Petuum allows ML programs to run in much less time and at considerably larger model sizes, even on modestly-sized compute clusters.

Index Terms—Machine Learning, Big Data, Big Model, Distributed Systems, Theory, Data-Parallelism, Model-Parallelism

1 Introduction

Machine Learning (ML) is becoming a primary mechanism for extracting information from data. However, the surging volume of **Big Data** from Internet activities and sensory advancements, and the increasing needs for **Big Models** for ultra high-dimensional problems have put tremendous pressure on ML methods to scale beyond a single machine, due to both space and time bottlenecks. For example, on the Big Data front, the Clueweb 2012 web crawl[1] contains over 700 million web pages as 27TB of text data; while photo-sharing sites such as Flickr, Instagram and Facebook are anecdotally known to possess 10s of billions of images, again taking up TBs of storage. It is highly inefficient, if possible, to use such big data sequentially in a batch or scholastic fashion in a typical iterative ML algorithm. On the Big Model front, state-of-the-art image recognition systems have now embraced large-scale deep learning models with billions of parameters [1]; topic models with up to 10^6 topics can cover long-tail semantic word sets for substantially improved online advertising [2], [3]; and very-high-rank matrix factorization yields improved prediction on collaborative filtering problems [4]. Training such big models with a single machine can be prohibitively slow, if not impossible. While careful model design and feature engineering can certainly reduce the size of the model, they require domain-specific expertise and are fairly labor-intensive, hence the recent appeal (as seen in the above papers) of building high-capacity Big Models in order to substitute computation cost for labor cost.

- *Qirong Ho is with Institute of Infocomm Research, A*STAR Singapore.*
- *Eric P. Xing and all other authors are with Carnegie Mellon University's School of Computer Science.*

Manuscript received March 31, 2015.

1. http://www.lemurproject.org/clueweb12.php/

Despite the recent rapid development of many new ML models and algorithms aiming at scalable applications [5], [6], [7], [8], [9], [10], adoption of these technologies remains generally unseen in the wider data mining, NLP, vision, and other application communities for big problems, especially those built on advanced probabilistic or optimization programs. A likely reason for such a gap, at least from the scalable execution point of view, that prevents many state-of-the-art ML models and algorithms from being more widely applied at Big-Learning scales is the difficult migration from an academic implementation, often specialized for a small, well-controlled computer platform such as desktop PCs and small lab-clusters, to a big, less predictable platform such as a corporate cluster or the cloud, where correct execution of the original programs require careful control and mastery of low-level details of the distributed environment and resources through highly nontrivial distributed programming.

Many *programmable* platforms have provided partial solutions to bridge this research-to-production gap: while Hadoop [11] is a popular and easy to program platform, its implementation of MapReduce requires the program state to be written to disk every iteration, thus its performance on many ML programs has been surpassed by in-memory alternatives [12], [13]. One such alternative is Spark [12], which improves upon Hadoop by keeping ML program state in memory — resulting in large performance gains over Hadoop — whilst preserving the easy-to-use MapReduce programming interface. However, Spark ML implementations are often still slower than specially-designed ML implementations, in part because Spark does not offer flexible and fine-grained scheduling of computation and communication, which has been shown to be hugely advantageous, if not outright necessary, for fast and correct execution of advanced ML algorithms [14]. Graph-centric

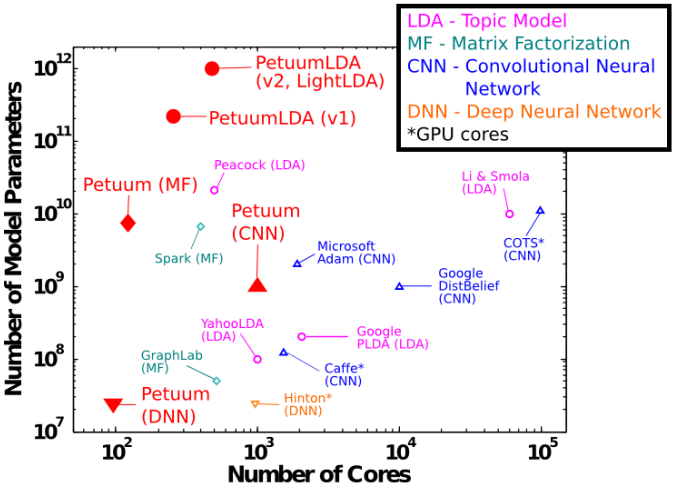


Fig. 1. The scale of Big ML efforts in recent literature. A key goal of Petuum is to enable larger ML models to be run on fewer resources, even relative to highly-specialized implementations.

platforms such as GraphLab [13] and Pregel [15] efficiently partition graph-based models with built-in scheduling and consistency mechanisms, but due to limited theoretical work, it is unclear whether asynchronous graph-based consistency models and scheduling will always yield correct execution of ML programs. Other systems provide low-level programming interfaces [16], [17], that, while powerful and versatile, do not yet offer higher-level general-purpose building blocks such as scheduling, model partitioning strategies, and managed communication that are key to simplifying the adoption of a wide range of ML methods. In summary, existing systems supporting distributed ML each manifest a unique tradeoff on efficiency, correctness, programmability, and generality.

In this paper, we explore the problem of building a distributed machine learning framework with a new angle toward the efficiency, correctness, programmability, and generality tradeoff. We observe that, a hallmark of most (if not all) ML programs is that they are defined by an explicit objective function over data (e.g., likelihood, error-loss, graph cut), and the goal is to attain optimality of this function, in the space defined by the model parameters and other intermediate variables. Moreover, these algorithms all bear a common style, in that they resort to an iterative-convergent procedure (see Eq. 1). It is noteworthy that iterative-convergent computing tasks are vastly different from conventional programmatic computing tasks (such as database queries and keyword extraction), which reach correct solutions only if every deterministic operation is correctly executed, and strong consistency is guaranteed on the intermediate program state — thus, operational objectives such as fault tolerance and strong consistency are absolutely necessary. However, an ML program's true goal is fast, efficient convergence to an optimal solution, and we argue that fine-grained fault tolerance and strong consistency are but one vehicle to achieve this goal, and might not even be the most efficient one.

We present a new distributed ML framework, *Petuum*, built on an ML-centric optimization-theoretic principle, as opposed to various operational objectives explored earlier. We begin by formalizing ML algorithms as *iterative-convergent* programs, which encompass a large space of modern ML, such as stochastic gradient descent [18] and coordinate descent [10] for convex optimization problems, proximal methods [19] for more complex constrained optimization, as well as MCMC [20] and variational inference [7] for inference in probabilistic models. To our knowledge, no existing programmable[2] platform has considered such a wide spectrum of ML algorithms, which exhibit diverse representation abstractions, model and data access patterns, and synchronization and scheduling requirements. So what are the shared properties across such a "zoo of ML algorithms"? We believe that the key lies in the recognition of a clear dichotomy between *data* (which is conditionally independent and persistent throughout the algorithm) and *model* (which is internally coupled, and is transient before converging to an optimum). This inspires a simple yet statistically-rooted bimodal approach to parallelism: *data parallel* and *model parallel* distribution and execution of a big ML program over a cluster of machines. This *data parallel*, *model parallel* approach keenly exploits the unique statistical nature of ML algorithms, particularly the following three properties: (1) Error tolerance — iterative-convergent algorithms are often robust against limited errors in intermediate calculations; (2) Dynamic structural dependency — during execution, the changing correlation strengths between model parameters are critical to efficient parallelization; (3) Non-uniform convergence — the number of steps required for a parameter to converge can be highly skewed across parameters. The core goal of Petuum is to execute these iterative updates in a manner that quickly converges to an optimum of the ML program's objective function, by exploiting these three statistical properties of ML, which we argue are fundamental to efficient large-scale ML in cluster environments.

This design principle contrasts that of several existing programmable frameworks discussed earlier. For example, central to the Spark framework [12] is the principle of perfect fault tolerance and recovery, supported by a persistent memory architecture (Resilient Distributed Datasets); whereas central to the GraphLab framework is the principle of local and global consistency, supported by a vertex programming model (the Gather-Apply-Scatter abstraction). While these design principles reflect important aspects of correct ML algorithm execution — e.g., atomic recoverability of each computing step (Spark), or consistency satisfaction for all subsets of model variables (GraphLab) — some other important aspects, such as the three statistical properties discussed above, or perhaps ones that could be more fundamental and general, and which could open more room for efficient system designs, remain unexplored.

To exploit these properties, Petuum introduces three novel system objectives grounded in the aforementioned key properties of ML programs, in order to accelerate their convergence at scale: (1) Petuum synchronizes the parameter states with bounded staleness guarantees, thereby achieves provably correct outcomes due to the error-tolerant

2. Our discussion is focused on platforms which provide libraries and tools for writing new ML algorithms. Because programmability is an important criteria for writing new ML algorithms, we exclude ML software that does not allow new algorithms to be implemented on top of them, such as AzureML and Mahout.

nature of ML, but at a much cheaper communication cost than conventional per-iteration bulk synchronization; (2) Petuum offers dynamic scheduling policies that take into account the changing structural dependencies between model parameters, so as to minimize parallelization error and synchronization costs; and (3) Since parameters in ML programs exhibit non-uniform convergence costs (i.e. different numbers of updates required), Petuum prioritizes computation towards non-converged model parameters, so as to achieve faster convergence.

To demonstrate this approach, we show how data-parallel and model-parallel algorithms can be implemented on Petuum, allowing them to scale to large data/model sizes with improved algorithm convergence times. The experiments section provides detailed benchmarks on a range of ML programs: topic modeling, matrix factorization, deep learning, Lasso regression, and distance metric learning. These algorithms are only a subset of the full open-source Petuum ML library[3]— the PMLlib, which we will briefly discuss in this paper. As illustrated in Figure 1, Petuum PMLlib covers a rich collection of advanced ML methods not usually seen in existing ML libraries; the Petuum platform enables PMLlib to solve a range of ML problems at large scales — scales that have only been previously attempted in a case-specific manner with corporate-scale efforts and resources — but using relatively modest clusters (10-100 machines) that are within reach of most ML practitioners.

2 Preliminaries: On Data Parallelism and Model Parallelism

We begin with a principled formulation of iterative-convergent ML programs, which exposes a dichotomy of data and model, that inspires the parallel system architecture (§3), algorithm design (§4), and theoretical analysis (§6) of Petuum. Consider the following programmatic view of ML as iterative-convergent programs, driven by an objective function.

Iterative-Convergent ML Algorithm: Given data D and a model objective function $\mathcal{L}$ (e.g. mean-squared loss, likelihood, margin), a typical ML problem can be grounded as executing the following update equation iteratively, until the model state (i.e., parameters and/or latent variables) A reaches some stopping criteria:

$$A^{(t)} = F(A^{(t-1)}, \Delta_{\mathcal{L}}(A^{(t-1)}, D)) \quad (1)$$

where superscript (t) denotes the iteration counter. The update function $\Delta_{\mathcal{L}}()$ (which improves the loss $\mathcal{L}$) performs computation on data D and model state A, and outputs intermediate results to be aggregated with the current estimate of A by $F()$ to produce the new estimate of A. For simplicity, in the rest of the paper we omit $\mathcal{L}$ in the subscript with the understanding that all ML programs of our interest here bear an explicit loss function that can be used to monitor the quality of convergence to a solution, as opposed to heuristics or procedures not associated such a loss function. Also for simplicity, we focus on iterative-convergent equations with an additive form:

$$A^{(t)} = A^{(t-1)} + \Delta(A^{(t-1)}, D), \quad (2)$$

3. Petuum is available as open source at http://petuum.org.

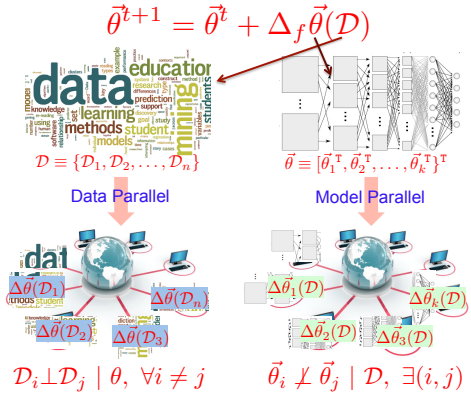


Fig. 2. The difference between data and model parallelism: data samples are always conditionally independent given the model, but some model parameters are not independent of each other.

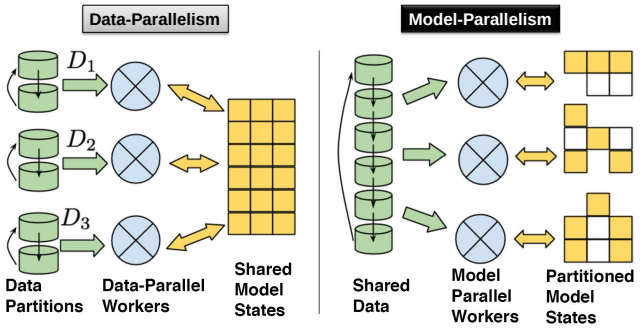


Fig. 3. Conceptual illustration of data and model parallelism. In data-parallelism, workers are responsible for generating updates $\Delta()$ on different data partitions, in order to updated the (shared) model state. In model-parallelism, workers generate Δ on different model partitions, possibly using all of the data.

i.e. the aggregation $F()$ is replaced with a simple addition. The approaches we propose can also be applied to this general $F()$.

In large-scale ML, both data D and model A can be very large. *Data-parallelism*, in which data is divided across machines, is a common strategy for solving Big Data problems, while *model-parallelism*, which divides the ML model, is common for Big Models. Both strategies are not exclusive, and can be combined to tackle challenging problems with large data D and large model A. Hence, every Petuum ML program is either data-parallel, model-parallel, or data-and-model-parallel, depending on problem needs. Below, we discuss the (different) mathematical implications of each parallelism (see Figure 2).

2.1 Data Parallelism

In *data-parallel* ML, the data D is partitioned and assigned to computational workers (indexed by $p = 1..P$); we denote the p-th data partition by D_p. The function $\Delta()$ can be applied to each data partition independently, and the results combined additively, yielding a data-parallel equation (left panel of Figure 2):

$$A^{(t)} = A^{(t-1)} + \sum_{p=1}^{P} \Delta(A^{(t-1)}, D_p). \quad (3)$$

This form is commonly seen in stochastic gradient descent or sampling-based algorithms. For example, in distance metric learning optimized via stochastic gradient descent (SGD), the data pairs are partitioned over different workers, and the intermediate results (subgradients) are computed on each partition, before being summed and applied to

the model parameters. A slightly modified form, $A^{(t)} = \sum_{p=1}^{P} \Delta(A^{(t-1)}, D_p)$, is used by some algorithms, such as variational EM.

Importantly, this *additive updates* property allows the updates $\Delta()$ to be computed at each local worker before transmission over the network, which is crucial because CPUs can produce updates $\Delta()$ much faster than they can be (individually) transmitted over the network. Additive updates are the foundation for a host of techniques to speed up data-parallel execution, such as minibatch, asynchronous and bounded-asynchronous execution, and parameter servers. Key to the validity of additivity of updates from different workers is the notion of *independent and identically distributed (iid)* data, which is assumed for many ML programs, and implies that each parallel worker contributes "equally" (in a statistical sense) to the ML algorithm's progress via $\Delta()$, no matter which data subset D_p it uses.

2.2 Model Parallelism

In *model-parallel* ML, the model A is partitioned and assigned to workers $p = 1..P$ and updated therein in parallel, running update functions $\Delta()$. Because the outputs from each $\Delta()$ affect different elements of A (hence denoted now by $\Delta_p()$ to make explicit the parameter subset affected at worker p), they are first *concatenated* into a full vector of updates (i.e., the full $\Delta()$), before aggregated with model parameter vector A (see right panel of Figure 2):

$$A^{(t)} = A^{(t-1)} + \text{Con}\left(\left\{\Delta_p(A^{(t-1)}, S_p^{(t-1)}(A^{(t-1)}))\right\}_{p=1}^{P}\right), \quad (4)$$

where we have omitted the data D for brevity and clarity. Coordinate descent algorithms for regression and matrix factorization are a canonical example of model-parallelism. Each update function $\Delta_p()$ also takes a scheduling function $S_p^{(t-1)}(A)$, which restricts $\Delta_p()$ to modify only a carefully-chosen subset of the model parameters A. $S_p^{(t-1)}(A)$ outputs a set of indices $\{j_1, j_2, \dots, \}$, so that $\Delta_p()$ only performs updates on $A_{j_1}, A_{j_2}, \dots$ — we refer to such selection of model parameters as *scheduling*. In some cases, the additive update formula above can be replaced by a *replace* operator that directly replaces corresponding elements in A with ones in the concatenated update vector.

Unlike data-parallelism which enjoys iid data properties, the model parameters A_j are not, in general, independent of each other (Figure 2), and it has been established that model-parallel algorithms can only be effective if the parallel updates are restricted to independent (or weakly-correlated) parameters [10], [13], [21], [22]. Hence, our definition of model-parallelism includes the *global scheduling mechanism* $S_p()$ that can select carefully-chosen parameters for parallel updating.

The scheduling function $S()$ opens up a large design space, such as fixed, randomized, or even dynamically-changing scheduling on the whole space, or a subset of, the model parameters. $S()$ not only can provide *safety and correctness* (e.g., by selecting independent parameters and thus minimize parallelization error), but can offer substantial *speed-up* (e.g., by prioritizing computation onto non-converged parameters). In the Lasso example, Petuum uses $S()$ to select coefficients that are weakly correlated (thus preventing divergence), while at the same time prioritizing coefficients far from zero (which are more likely to be non-converged).

2.3 Implementing Data- and Model-Parallel Programs

Data- and model-parallel programs exhibit a certain programming and systems desiderata: they are *stateful*, in that they continually update shared model parameters A. Thus, an ML platform needs to synchronize A across all running threads and processes, and this should be done via a high-performance, non-blocking asynchronous strategy that still guarantees convergence. If the program is model-parallel, it may require fine control over the order of parameter updates, in order to avoid non-convergence due to dependency violations — thus, the ML platform needs to provide fine-grained *scheduling* capability. We discuss some of the difficulties associated with achieving these desiderata.

Data- and model-parallel programs can certainly be written in a message-passing style, in which the programmer explicitly writes code to send and receive parameters over the network. However, we believe it is more desirable to provide a Distributed Shared Memory (DSM) abstraction, in which the programmer simply treats A like a global program variable, accessible from any thread/process in a manner similar to single-machine programming — no explict network code is required from the user, and all synchronization is done in the background. While DSM-like interfaces could be added to alternative ML platforms like Hadoop, Spark and GraphLab, these systems usually operate in either a bulk synchronous (prone to stragglers and blocking due to the high rate of update $\Delta()$ generation) or asynchronous (having no parameter consistency guarantee, and hence no convergence guarantee) fashion.

Model-parallel programs pose an additional challenge, in that they require fine-grained control over the parallel ordering of variable updates. Again, while it is completely possible to achieve such control via message-passing programming style, there is nevertheless an opportunity to provide a simpler abstraction, in which the user merely has to define the model scheduling function $S_p^{(t-1)}(A)$. In such an abstraction, networking and synchronization code is again hidden from the user. While Hadoop and Spark provide easy-to-use abstractions, their design does not give users fine-grained control over the ordering of updates — for example, MapReduce provides no control over the order in which mappers or reducers are executed. We note that GraphLab has a priority-based scheduler specialized for some model-parallel applications, but still does not expose a dedicated scheduling function $S_p^{(t-1)}(A)$. One could certainly modify Hadoop's or Spark's built-in schedulers to expose the required level of control, but we do not consider this reasonable for the average ML practitioner without strong systems expertise.

These considerations make effective data- and model-parallel programming challenging, and there is an opportunity to abstract them away via a platform that is specifically focused on data/model-parallel ML.

3 Petuum – a Platform for Distributed ML

A core goal of Petuum is to allow practitioners to easily implement data-parallel and model-parallel ML algorithms.

```
// Petuum Program Structure

schedule() {
  // This is the (optional) scheduling function
  // It is executed on the scheduler machines
  A_local = PS.get(A)  // Parameter server read
  PS.inc(A,change)  // Can write to PS here if needed
  // Choose variables for push() and return
  svars = my_scheduling(DATA,A_local)
  return svars
}

push(p = worker_id(), svars = schedule()) {
  // This is the parallel update function
  // It is executed on each of P worker machines
  A_local = PS.get(A)  // Parameter server read
  // Perform computation and send return values to pull()
  // Or just write directly to PS
  change1 = my_update1(DATA,p,A_local)
  change2 = my_update2(DATA,p,A_local)
  PS.inc(A,change1)  // Parameter server increment
  return change2
}

pull(svars=schedule(), updates=(push(1), ..., push(P)) ) {
  // This is the (optional) aggregation function
  // It is executed on the scheduler machines
  A_local = PS.get(A)  // Parameter server read
  // Aggregate updates from push(1..P) and write to PS
  my_aggregate(A_local,updates)
  PS.put(A,change)  // Parameter server overwrite
}
```

Fig. 4. Petuum Program Structure.

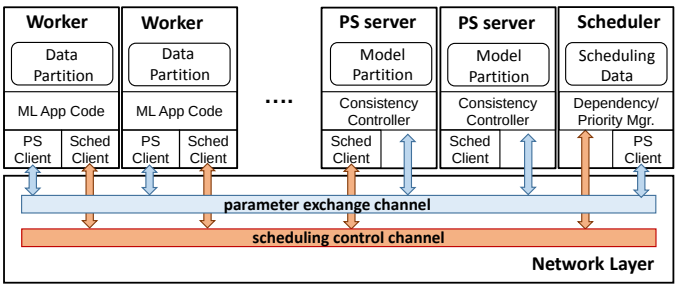


Fig. 5. Petuum system: scheduler, workers, parameter servers.

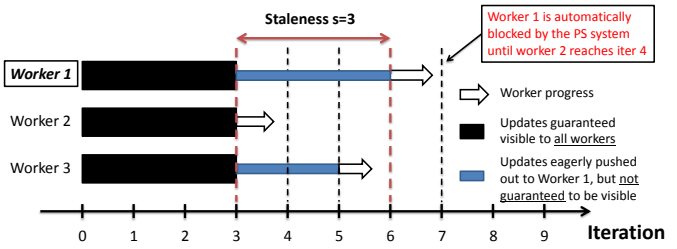


Fig. 6. **ESSP consistency model**, used by the Parameter Server. Workers are allowed to run at different speeds, but are prevented from being more than s iterations apart. Updates from the most recent s iterations are "eagerly" pushed out to workers, but are not guaranteed to be visible.

Petuum provides APIs to key systems that make data- and model-parallel programming easier: (1) a *parameter server* system, which allows programmers to access global model state A from any machine via a convenient *distributed shared-memory* interface that resembles single-machine programming, and adopts a bounded-asychronous consistency model that preserves data-parallel convergence guarantees, thus freeing users from explicit network synchronization; (2) a *scheduler*, which allows fine-grained control over the parallel ordering of model-parallel updates $\Delta()$ — in essence, the scheduler allows users to define their own ML application consistency rules.

3.1 Programming Interface

Figure 4 shows pseudocode for a generic Petuum program, consisting of three user-written functions (in either C++ or Java): a central scheduler function `schedule()`, a parallel update function `push()` (analogous to Map in MapReduce), and a central aggregation function `pull()` (analogous to Reduce). Data-parallel programs can be written with just `push()`, while model-parallel programs are written with all three functions `schedule()`, `push()` and `pull()`.

The model variables A are held in the parameter server, which can be accessed at any time from any function via the `PS` object. The `PS` object can be accessed from any function, and has 3 functions: `PS.get()` to read a parameter, `PS.inc()` to add to a parameter, and `PS.put()` to overwrite a parameter. With just these operations, the parameter server automatically ensures parameter consistency between all Petuum components; no additional user programming is necessary. In the example pseudocode, `DATA` is a placeholder for data D; this can be any 3rd-party data structure, database, or distributed file system.

3.2 Petuum System Design

ML algorithms exhibit several principles that can be exploited to speed up distributed ML programs: dependency structures between parameters, non-uniform convergence of parameters, and a limited degree of error tolerance [13], [14], [17], [21], [23], [24]. Through `schedule()`, `push()` and `pull()`, Petuum allows practitioners to write data-parallel and model-parallel ML programs that exploit these principles, and can be scaled to Big Data and Big Model applications. The Petuum system comprises three components (Fig. 5): parameter server, scheduler, and workers.

Parameter Server: The parameter server (PS) enables data-parallelism, by providing users with global read/write access to model parameters A, via a convenient distributed shared memory API that is similar to table-based or key-value stores. The PS API consists of three functions: `PS.get()`, `PS.inc()` and `PS.put()`. As the names suggest, the first function reads a part of the global A into local memory, while the latter two add or overwrite local changes into the global A.

To take advantage of ML error tolerance, the PS implements the Eager Stale Synchronous Parallel (ESSP) consistency model [14], [23], which reduces network synchronization and communication costs, while maintaining bounded-staleness convergence guarantees implied by ESSP. The ESSP consistency model ensures that, if a worker reads from parameter server at iteration c, it will definitely receive all updates from all workers computed at and before iteration $c - s - 1$, where s is a staleness threshold — see Figure 6 for an illustration. In Section 6, we will cover theoretical guarantees enjoyed by ESSP consistency.

Scheduler: The scheduler system enables model-parallelism, by allowing users to control which model parameters are updated by worker machines. This is performed through a user-defined scheduling function `schedule()` (corresponding to $S_p^{(t-1)}()$), which outputs a set of parameters for each worker. The scheduler sends the *identities* of these parameters to workers via the scheduling control channel (Fig. 5), while the actual parameter values are delivered through the parameter server system. In Sec-

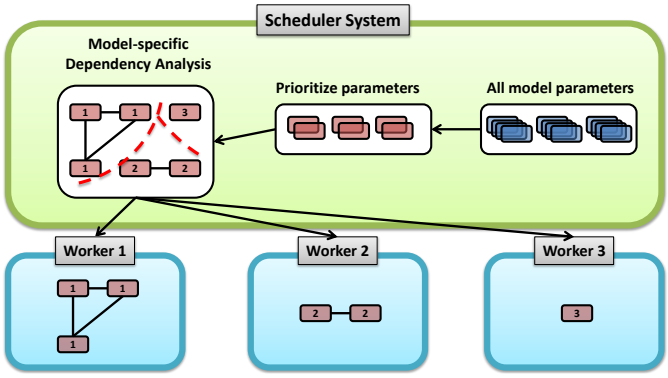


Fig. 7. **Scheduler system.** Using algorithm or model-specific criteria, the Petuum scheduler prioritizes a small subset of parameters from the full model A. This is followed by a dependency analysis on the prioritized subset: parameters are further subdivided into groups, where a parameter A_i in group g_u is must be independent of all other parameters A_j in all other groups g_v. This is illustrated as a graph partitioning, although the implementation need not actually construct a graph.

tion 6, we will discuss the theoretical guarantees enjoyed by model-parallel schedules.

Several common patterns for schedule design are worth highlighting: *fixed-scheduling* (`schedule_fix()`) dispatches model parameters A in a pre-determined order; static, round-robin schedules (e.g. repeatedly loop over all parameters) fit the `schedule_fix()` model. *Dependency-aware* (`schedule_dep()`) scheduling allows re-ordering of variable/parameter updates to accelerate model-parallel ML algorithms, e.g. in Lasso regression, by analyzing the dependency structure over model parameters A. Finally, *prioritized scheduling* (`schedule_pri()`) exploits uneven convergence in ML, by prioritizing subsets of variables $U^{sub} \subset A$ according to algorithm-specific criteria, such as the magnitude of each parameter, or boundary conditions such as KKT. These common schedules are provided as pre-implemented software libraries, or users can opt to write their own `schedule()`.

Workers: Each worker p receives parameters to be updated from `schedule()`, and then runs parallel update functions `push()` (corresponding to $\Delta()$) on data D. While `push()` is being executed, the model state A can be easily synchronized with the parameter server, using the `PS.get()` and `PS.inc()` API. After the workers finish `push()`, the scheduler may use the new model state to generate future scheduling decisions.

Petuum intentionally does not enforce a data abstraction, so that any data storage system may be used — workers may read from data loaded into memory, or from disk, or over a distributed file system or database such as HDFS. Furthermore, workers may touch the data in any order desired by the programmer: in data-parallel stochastic algorithms, workers might sample one data point at a time, while in batch algorithms, workers might instead pass through all data points in one iteration.

4 Petuum Parallel Algorithms

Now we turn to development of parallel algorithms for large-scale distributed ML problems, in light of the data and model parallel principles underlying Petuum. As examples, we explain how to use Petuum's programming interface to implement novel or state-of-the-art versions of the following 4 algorithms: (1) data-parallel Distance Metric Learning, (2) model-parallel Lasso regression, (3) model-parallel topic modeling (LDA), and (4) model-parallel Matrix Factorization. These algorithms all enjoy significant performance advantages over the previous state-of-the-art and existing open-source software, as we will show.

Through pseudocode, it can be seen that Petuum allows these algorithms to be easily realized on distributed clusters, without dwelling on low level system programming, or non-trivial recasting of our ML problems into representations such as RDDs or vertex programs. Instead our ML problems can be coded at a high level, more akin to Matlab or R. We round off with a brief description of how we used Petuum implement a couple of other ML algorithms.

4.1 Data-Parallel Distance Metric Learning

Let us first consider a large-scale Distance Metric Learning (DML) problem. DML improves the performance of other ML programs such as clustering, by allowing domain experts to incorporate prior knowledge of the form "data points x, y are similar (or dissimilar)" [25] — for example, we could enforce that "books about science are different from books about art". The output is a distance function $d(x, y)$ that captures the aforementioned prior knowledge. Learning a proper distance metric [25], [26] is essential for many distance based data mining and machine learning algorithms, such as retrieval, k-means clustering and k-nearest neighbor (k-NN) classification. DML has not received much attention in the Big Data setting, and we are not aware of any distributed implementations of DML.

The most popular version of DML tries to learn a Mahalanobis distance matrix M (symmetric and positive-semidefinite), which can then be used to measure the distance between two samples $D(x, y) = (x - y)^\mathsf{T} M(x - y)$. Given a set of "similar" sample pairs $\mathcal{S} = \{(x_i, y_i)\}_{i=1}^{|\mathcal{S}|}$, and a set of "dissimilar" pairs $\mathcal{D} = \{(x_i, y_i)\}_{i=1}^{|\mathcal{D}|}$, DML learns the Mahalanobis distance by optimizing

$$\begin{array}{ll} \min_M & \sum_{(x,y)\in\mathcal{S}} (x-y)^\mathsf{T} M(x-y) \\ s.t. & (x-y)^\mathsf{T} M(x-y) \geq 1, \forall (x,y) \in \mathcal{D} \\ & M \succeq 0 \end{array} \quad (5)$$

where $M \succeq 0$ denotes that M is required to be positive semidefinite. This optimization problem minimizes the Mahalanobis distances between all pairs labeled as similar, while separating dissimilar pairs with a margin of 1.

In its original form, this optimization problem is difficult to parallelize due to the constraint set. To create a data-parallel optimization algorithm and implement it on Petuum, we shall relax the constraints via slack variables (similar to SVMs). First, we replace M with $L^\mathsf{T} L$, and introduce slack variables ξ to relax the hard constraint in Eq.(5), yielding

$$\begin{array}{ll} \min_L & \sum_{(x,y)\in\mathcal{S}} \|L(x-y)\|^2 + \lambda \sum_{(x,y)\in\mathcal{D}} \xi_{x,y} \\ s.t. & \|L(x-y)\|^2 \geq 1 - \xi_{x,y}, \xi_{x,y} \geq 0, \forall (x,y) \in \mathcal{D} \end{array} \quad (6)$$

```
// Data-Parallel Distance Metric Learning

schedule() { // Empty, do nothing }

push() {
  L_local = PS.get(L) // Bounded-async read from PS
  change = 0
  for c=1..C    // Minibatch size C
    (x,y) = draw_similar_pair(DATA)
    (a,b) = draw_dissimilar_pair(DATA)
    change += DeltaL(L_local,x,y,a,b)  // SGD using Eq (8)
  PS.inc(L,change/C)  // Add gradient to param server
}

pull() { // Empty, do nothing }
```

Fig. 8. Petuum DML data-parallel pseudocode.

Using hinge loss, the constraint in Eq.(6) can be eliminated, yielding an unconstrained optimization problem:

$$\begin{aligned}\min_L \quad & \sum_{(x,y)\in\mathcal{S}} \|L(x-y)\|^2 \\ & +\lambda \sum_{(x,y)\in\mathcal{D}} \max(0, 1-\|L(x-y)\|^2)\end{aligned} \tag{7}$$

Unlike the original constrained DML problem, this relaxation is fully data-parallel, because it now treats the dissimilar pairs as iid data to the loss function (just like the similar pairs); hence, it can be solved via data-parallel Stochastic Gradient Descent (SGD). SGD can be naturally parallelized over data, and we partition the data pairs onto P machines. Every iteration, each machine p randomly samples a mini-batch of similar pairs $\mathcal{S}_p$ and dissimilar pairs $\mathcal{D}_p$ from its data shard, and computes the following update to L:

$$\begin{aligned}\triangle L_p \quad &= \quad \textstyle\sum_{(x,y)\in\mathcal{S}_p} 2L(x-y)(x-y)^\mathsf{T} \\ &- \quad \textstyle\sum_{(a,b)\in\mathcal{D}_p} 2L(a-b)(a-b)^\mathsf{T} \cdot \mathbb{I}(\|L(a-b)\|^2 \leq 1)\end{aligned} \tag{8}$$

where $\mathbb{I}(\cdot)$ is the indicator function.

Figure 8 shows pseudocode for Petuum DML, which is simple to implement because the parameter server system `PS` abstracts away complex networking code under a simple `get()/read()` API. Moreover, the `PS` automatically ensures high-throughput execution, via a bounded-asynchronous consistency model (Stale Synchronous Parallel) that can provide workers with stale local copies of the parameters L, instead of forcing workers to wait for network communication. Later, we will review the strong consistency and convergence guarantees provided by the SSP model.

Since DML is a data-parallel algorithm, only the parallel update `push()` needs to be implemented (Figure 8). The scheduling function `schedule()` is empty (because every worker touches every model parameter L), and we do not need aggregation `push()` for this SGD algorithm. In our next example, we will show how `schedule()` and `push()` can be used to implement model-parallel execution.

4.2 Model-Parallel Lasso

Lasso is a widely used model to select features in high-dimensional problems, such as gene-disease association studies, or in online advertising via ℓ_1-penalized regression [27]. Lasso takes the form of an optimization problem:

$$\min_{\boldsymbol{\beta}} \ell(\mathbf{X}, \mathbf{y}, \boldsymbol{\beta}) + \lambda \textstyle\sum_j |\beta_j|, \tag{9}$$

where λ denotes a regularization parameter that determines the sparsity of $\boldsymbol{\beta}$, and $\ell(\cdot)$ is a non-negative convex loss function such as squared-loss or logistic-loss; we assume that $\mathbf{X}$ and $\mathbf{y}$ are standardized and consider (9) without an intercept. For simplicity but without loss of generality, we let $\ell(\mathbf{X}, \mathbf{y}, \boldsymbol{\beta}) = \frac{1}{2}\|\mathbf{y} - \mathbf{X}\boldsymbol{\beta}\|_2^2$; other loss functions (e.g. logistic) are straightforward and can be solved using the same approach [10]. We shall solve this via a coordinate descent (CD) model-parallel approach, similar but not identical to [10], [22].

```
// Model-Parallel Lasso

schedule() {
  for j=1..J     // Update priorities for all params beta_j
    c_j = square(beta_j) + eta // Magnitude prioritization
  (s_1, ..., s_L') = random_draw(distrib(c_1, ..., c_J))
  // Choose L<L' pairwise-independent beta_j
  (j_1, ..., j_L) = correlation_check(s_1, ..., s_L')
  return (j_1, ..., j_L)
}

correlation_check(s_1, ..., s_L') {
  selection = (s_1)
  remaining = (s_2, ..., s_L')
  for i=2..L // Select L parameters which are independent
    s = remaining.front()
    remaining.delete(s)
    correlated = false
    for each element j in selection
      // In Lasso, parameters s, j are independent if
      // their data columns x_s, x_j have dot product
      // below a small threshold tau
      if dotprod(x[:,s],x[:,j]) > tau
      correlated = true
    if correlated == false
      selection.append(s)
  return selection
}

push(p = worker_id(), (j_1, ..., j_L) = schedule() ) {
  // DATA[p] are row indices of data x,y stored at worker p
  (z_p[j_1], ..., z_p[j_L])=partial(DATA[p], j_1, ..., j_L)
  return z_p
}

partial(DATA[p], j_1, ..., j_L) {
  // Partial comp. for all L beta_j; calls PS.get(beta)
  result = ()
  for a=1..L
    temp = 0
    for i in DATA[p]
      temp += delta_{i,j_a}^{(t)} // Compute via Eq (10)
    result.append(temp)
  return result
}

pull((j_1, ..., j_L) = schedule(),
     (z_1, ..., z_P) = (push(1), ..., push(P)) ) {
  for a=1..L    // Aggregate partial comp from P workers
    newval = sum_threshold(z_1[j_a], ..., z_P[j_a])
    PS.put(beta[j_a],newval)  // Overwrite to PS
}
```

Fig. 9. Petuum Lasso model-parallel pseudocode.

The simplest parallel CD Lasso , shotgun [10], selects a random subset of parameters to be updated in parallel. We now present a scheduled model-parallel Lasso that improves upon shotgun: the Petuum scheduler chooses parameters that are nearly independent with each other[4], thus guaranteeing convergence of the Lasso objective. In addition, it prioritizes these parameters based on their distance to convergence, thus speeding up optimization.

Why is it important to choose independent parameters via scheduling? Parameter dependencies affect the CD update equation in the following manner: by taking the gradient of (9), we obtain the CD update for β_j:

$$\delta_{ij}^{(t)} \leftarrow \mathbf{x}_{ij}\mathbf{y}_i - \textstyle\sum_{k\neq j} \mathbf{x}_{ij}\mathbf{x}_{ik}\beta_k^{(t-1)}, \tag{10}$$

4. In the context of Lasso, this means the data columns $\mathbf{x}_{\cdot j}$ corresponding to the chosen parameters j have very small pair-wise dot product, below a threshold τ.

```
// Model-Parallel Topic Model (LDA)

schedule() {
  for p=1..P:   // "cyclic" schedule
    x=(p-1+c) mod P    // c is a persistent local variable
    j_p = (xM/P,(x+1)M/P)   // p's word range
  c=c+1
  return (j_1, ..., j_P)
}

push(p = worker_id(), (j_1, ..., j_L) = schedule() ) {
  V_local = PS.get(V)
  [lower,upper] = j_p   // Only touch w_ij in word range
  for each token z_ij at this worker p:
    if w_ij in range(lower,upper):
      old = z_ij
      new = LDAGibbsSample(U_i, V_local, w_ij, z_ij)
      Update U_i with (old, new)
      Record old, new values of z_ij in R_p
  return R_p
}

LDAGibbsSample(U_i, V_local, w_ij, z_ij) {
  // probs is a K-dimensional categorial distribution.
  // It can be computed using SparseLDA
  // or other collapsed Gibbs samplers for LDA
  probs = P(z_{ij} | U, V_local) // Compute using Eq (12)
  choice = random_draw(probs)
  return choice
}

pull((j_1, ..., j_L) = schedule(),
     (R_1, ..., R_P) = (push(1), ..., push(P)) ) {
  Update V with token changes in (R_1, ... ,R_P)
}
```

Fig. 10. Petuum Topic Model (LDA) model-parallel pseudocode.

$$\beta_j^{(t)} \leftarrow S(\textstyle\sum_{i=1}^{N} \delta_{ij}^{(t)}, \lambda), \tag{11}$$

where $S(\cdot, \lambda)$ is a soft-thresholding operator, defined by $S(\beta_j, \lambda) \equiv \text{sign}(\beta)\,(|\beta| - \lambda)$. In (11), if $\mathbf{x}_j^T \mathbf{x}_k \neq 0$ (i.e., nonzero correlation) and $\beta_j^{(t-1)} \neq 0$ and $\beta_k^{(t-1)} \neq 0$, then a coupling effect is created between the two features β_j and β_k. Hence, they are no longer conditionally independent given the data: $\beta_j \not\perp \beta_k | \mathbf{X}, \mathbf{y}$. If the j-th and the k-th coefficients are updated concurrently, parallelization error may occur, causing the Lasso problem to converge slowly (or even diverge outright).

Petuum's `schedule()`, `push()` and `pull()` interface is readily suited to implementing scheduled model-parallel Lasso. We use `schedule()` to choose parameters with low dependency, and to prioritize non-converged parameters. Petuum pipelines `schedule()` and `push()`; thus `schedule()` does not slow down workers running `push()`. Furthermore, by separating the scheduling code `schedule()` from the core optimization code `push()` and `pull()`, Petuum makes it easy to experiment with complex scheduling policies that involve prioritization and dependency checking, thus facilitating the implementation of new model-parallel algorithms — for example, one could use `schedule()` to prioritize according to KKT conditions in a constrained optimization problem, or to perform graph-based dependency checking like in Graphlab [13]. Later, we will show that the above Lasso schedule `schedule()` is guaranteed to converge, and gives us near optimal solutions by controlling errors from parallel execution. The pseudocode for scheduled model parallel Lasso under Petuum is shown in Figure 9.

4.3 Topic Model (LDA):

Topic Modeling uncovers semantically-coherent topics from unstructured document corpora, and is widely used in industry — e.g. Yahoo's YahooLDA [28], and Google's Rephil [29]. The most well-known member of the topic modeling family is Latent Dirichlet Allocation (LDA): given a corpus of N documents and a pre-specified K for number of topics, the objective of LDA inference is to output K "topics" (discrete distributions over V unique words in the corpus), as well as the topic distribution of each document (a discrete distribution over topics).

One popular LDA inference technique is collapsed Gibbs sampling, a Markov Chain Monte Carlo algorithm that samples the topic assignments for each "token" (word position) in each document until convergence. This is an iterative-convergent algorithm that repeatedly updates three types of model state parameters: an M-by-K "word-topic table" V, an N-by-K "doc-topic" table U, and the token topic assignments z_{ij}. The LDA Gibbs sampler update is

$$P(z_{ij} = k \mid U, V) \propto \frac{\alpha + U_{ik}}{K\alpha + \sum_{\ell=1}^{K} U_{i\ell}} + \frac{\beta + V_{w_{ij},k}}{M\beta + \sum_{m=1}^{M} V_{mk}}, \tag{12}$$

where z_{ij} are topic assignments to each word "token" w_{ij} in document i. The document word tokens w_{ij}, topic assignments z_{ij} and doc-topic table rows U_i are partitioned across worker machines and kept fixed, as is common practice with Big Data. Although there are multiple parameters, the only one that is read and updated by all parallel worker (and hence needs to be globally-stored) is the word-topic table V.

We adopt a model-parallel approach to LDA, and use a `schedule()` (Algorithm 10) that cycles rows of the word-topic table (rows correspond to different words, e.g. "machine" or "learning") across machines, to be updated via `push()` and `pull()`; data is kept fixed at each machine. This `schedule()` ensures that no two workers will ever touch the same rows of V in the same iteration[5], unlike previous LDA implementations [28] which allow workers to update any parameter, resulting in dependency violations.

Note that the function `LDAGibbsSample()` in `push()` can be replaced with any recent state-of-the art Gibbs sampling algorithm, such as the fast Metropolis-Hastings algorithm in LightLDA [3]. Our experiments use the SparseLDA algorithm [30], to ensure a fair comparison with YahooLDA [28] (which also uses SparseLDA).

4.4 Matrix Factorization (MF):

MF is used in recommendation, where users' item preferences are predicted based on other users' partially observed ratings. The MF algorithm decomposes an incomplete observation matrix $X^{N \times M}$ into two smaller matrices $W \in \mathbb{R}^{K \times N}$ and $H \in \mathbb{R}^{K \times M}$ such that $W^T H \approx X$, where $K \ll \min\{M, N\}$ is a user-specified rank:

$$\min_{W,H} \textstyle\sum_{(i,j)\in\Omega} ||X_{ij} - \boldsymbol{w}_i^T \boldsymbol{h}_j||^2 + Reg(W, H), \tag{13}$$

where $Reg(W, H)$ is a regularizer such as the Frobenius norm, and Ω indexes the observed entries in X. High-rank decompositions of large matrices provide improved accuracy [4], and can be solved by a model-parallel stochastic gradient approach that ensures workers never touch the

5. Petuum LDA's "cyclic" schedule differs from the *model streaming* in [3]; the latter has workers touch the *same* set of parameters, one set at a time. Model streaming can easily be implemented in Petuum, by changing `schedule()` to output the same word range for every j_p.

```
// Model-Parallel Matrix Factorization (MF)

schedule() {
  // Fugue-style cyclic scheduling
  for p=1..P:   // workers cycle cols, keep rows fixed
    a=(p-1+c) mod P    // c is a persistent local variable
    j_p = (aM/P,(a+1)M/P)   // p's column range
  c=c+1
  return (j_1, ..., j_P)
}

push(p = worker_id(), (j_1, ..., j_L) = schedule() ) {
  [clow,cup] = j_p   // Only touch X_ij in these cols
  [rlow,rup] = ((p-1)N/P,pN/P)   // X_ij in these rows
  data = getData(X,rlow,rup,clow,cup)  // Submatrix of X
  [idx,count] = 0 // Used to iterate over data
  while count < limit: // Use exactly "limit" data samples
    [i,j,val] = data[idx] // Fetch the next X_ij = val
    W_i_local = PS.get(W_i)
    H_j_local = PS.get(H_j)
    // Upd. W row i
    localW[i] += MFRowSGD(i,j,val,W_i_local,H_j_local)
    // Upd. H col j
    localH[j] += MFColSGD(i,j,val,W_i_local,H_j_local)
    idx = (idx+1) mod length(data) // Repeat over data
    count += 1
  return R_p = (localW,localH)
}

MFRowSGD(i,j,X_ij,W_i,H_j) {
  update = [0,...,0] // Length K
  for k=1..K
    update[k] = deltaW_{ik} // Compute from Eq (14)
  return update
}

MFColSGD(i,j,X_ij,W_i,H_j) {
  update = [0,...,0] // Length K
  for k=1..K
    update[k] = deltaH_{kj} // Compute from Eq (15)
  return update
}

pull((j_1, ..., j_L) = schedule(),
     (R_1, ..., R_P) = (push(1), ..., push(P)) ) {
  for p=1..P:
    [localW,localH] = R_p
    PS.inc(W,localW)
    PS.inc(H,localH)
}
```

Fig. 11. Petuum MF model-parallel pseudocode.

same elements of W, H in the same iteration. There are two update equations, for W, H respectively:

$$\delta W_{ik} = \sum_{(a,b)\in\Omega} \mathbb{I}(a=i)\left[-2X_{ab}H_{kb} + 2W_{a\cdot}H_{\cdot b}H_{kb}\right], \quad (14)$$

$$\delta H_{kj} = \sum_{(a,b)\in\Omega} \mathbb{I}(b=j)\left[-2X_{ab}W_{ak} + 2W_{a\cdot}H_{\cdot b}W_{ak}\right], \quad (15)$$

where $\mathbb{I}()$ is the indicator function.

Previous systems using this approach [18] exhibited a *load-balancing* issue, because the rows of X exhibit a power-law distribution of non-zero entries; this was theoretically solved by the Fugue algorithm implemented on Hadoop [31], which essentially repeats the available data x_{ij} at each worker to restore load balance. Petuum can implement Fugue SGD MF as Algorithm 11; we also provide an Alternating Least Squares implementation for comparison against other ALS-using systems like Spark and GraphLab.

4.5 Other Algorithms

We have implemented other data- and model-parallel algorithms on Petuum as well. Here, we briefly mention a few algorithms whose data/model-parallel implementation on Petuum substantially differs from existing software. Many other ML algorithms are included in the Petuum open-source code.

Deep Learning (DL): We implemented two types on Petuum: a general-purpose fully-connected Deep Neural Network (DNN) using the cross-entropy loss, and a Convolutional Neural Network (CNN) for image classification based off the open-source Caffe project. We adopt a data-parallel strategy `schedule_fix()`, where each worker uses its data subset to perform updates `push()` to the full model A. While this data-parallel strategy could be amenable to MapReduce, Spark and GraphLab, we are not aware of DL implementations on those platforms.

Logstic Regression (LR) and Support Vector Machines (SVM): Petuum implements LR and SVM using the same dependency-checking, prioritized model-parallel strategy as the Lasso Algorithm 9. Dependency checking and prioritization are not easily implemented on MapReduce and Spark. While GraphLab has support for these features; the key difference with Petuum is that Petuum's scheduler performs dependency checking on small subsets of parameters at a time, whereas GraphLab performs graph partitioning on all parameters at once (which can be costly).

Maximum Entropy Discrimination LDA (MedLDA): MedLDA [32] is a constrained variant of LDA, that uses side information to constrain and improve the recovered topics. Petuum implements MedLDA using a data-parallel strategy `schedule_fix()`, where each worker uses `push()` to alternate between Gibbs sampling (like regular LDA) and solving for Lagrange multiplers associated with the constraints.

5 Big Data Ecosystem Support

To support ML at scale in production, academic, or cloud-compute clusters, Petuum provides a ready-to-run ML library, called **PMLlib**; Table 1 shows the current list of ML applications, with more applications are actively being developed for future releases. Petuum also integrates with Hadoop ecosystem modules, thus reducing the engineering effort required to deploy Petuum in academic and real-world settings.

Many industrial and academic clusters run Hadoop, which provides, in addition to the MapReduce programming interface, a job scheduler that allows multiple programs to run on the same cluster (YARN) and a distributed filesystem for storing Big Data (HDFS). However, programs that are written for stand-alone clusters are not compatible with YARN/HDFS, and vice versa, applications written for YARN/HDFS are not compatible with stand alone clusters.

Petuum solves this issue by providing common libraries that work on either Hadoop or non-Hadoop clusters. Hence, all Petuum PMLlib applications (and new user-written applications) can be run in stand-alone mode or deployed as YARN jobs to be scheduled alongside other MapReduce jobs, and PMLlib applications can also read/write input data and output results from both the local machine filesystem as well as HDFS. More specifically, Petuum provides a YARN launcher that will deploy any Petuum application (including user-written ones) onto a Hadoop cluster; the YARN launcher will automatically restart failed Petuum jobs and ensure that they always complete. Petuum also provides a data access library with C++ iostreams (or Java file streams) for HDFS access, which allows users to write

ML Application	Problem scale achieved on given cluster size
Topic Model (LDA)	220b params (22m unique words, 10k topics) on 256 cores and 1TB memory
Constrained Topic Model (MedLDA)	610m params (61k unique words, 10k topics, 20 classes) on 16 cores and 128GB memory
Convolutional Neural Networks (CNN)	1b params on 1024 *CPU* cores and 2TB memory
Fully-connected Deep Neural Networks (DNN)	24m params on 96 *CPU* cores and 768GB memory
Matrix Factorization (MF)	20m-by-20k input matrix, rank 400 (8b params) on 128 cores and 1TB memory
Non-negative Matrix Factorization (NMF)	20m-by-20k input matrix, rank 50 (1b params) on 128 cores and 1TB memory
Sparse Coding (SC)	1b params on 128 cores and 1TB memory
Logistic Regression (LR)	100m params (50k samples, 2b nonzeros) on 512 cores and 1TB memory
Multi-class Logistic Regression (MLR)	10m params (10 classes, 1m features) on 512 cores and 1TB memory
Lasso Regression	100m params (50k samples, 2b nonzeros) on 512 cores and 1TB memory
Support Vector Machines (SVM)	100m params (50k samples, 2b nonzeros) on 512 cores and 1TB memory
Distance Metric Learning (DML)	441m params (63k samples, 21k feature dimension) on 64 cores and 512GB memory
K-means clustering	1m params (10m samples, 1k feature dimension, 1k clusters) on 32 cores and 256GB memory
Random Forest	8000 trees (2m samples) on 80 cores and 640 GB memory

TABLE 1

Petuum ML Library (PMLlib): ML applications and achievable problem scale for a given cluster size. Petuum's goal is to solve large model and data problems using medium-sized clusters with only 10s of machines (100-1000 cores, 1TB+ memory). Running time varies between 10s of minutes to several days, depending on the application.

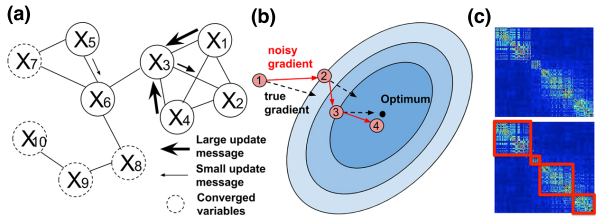


Fig. 12. Key properties of ML algorithms: (a) Non-uniform convergence; (b) Error-tolerant convergence; (c) Dependency structures amongst variables.

generic file stream code that works on both HDFS files the local filesystem. The data access library also provides pre-implemented routines to load common data formats, such as CSV, libSVM, and sparse matrix.

While Petuum PMLlib applications are written in C++ for maximum performance, new Petuum applications can be written in either Java or C++; Java has the advantages of easier deployment and a wider user base.

6 Principles and Theory

Our iterative-convergent formulation of ML programs, and the explicit notion of data and model parallelism, make it convenient to explore three key properties of ML programs — error-tolerant convergence, non-uniform convergence, dependency structures (Fig. 12) — and to analyze how Petuum exploits these properties in a theoretically-sound manner to speed up ML program completion at Big Learning scales.

Some of these properties have previously been successfully exploited by a number of bespoke, large-scale implementations of popular ML algorithms: e.g. topic models [3], [17], matrix factorization [33], [34], and deep learning [1]. It is notable that MapReduce-style systems (such as Hadoop [11] and Spark [12]) often do not fare competitively against these custom-built ML implementations, and one of the reasons is that these key ML properties are difficult to exploit under a MapReduce-like abstraction. Other abstractions may offer a limited degree of opportunity — for example, vertex programming [13] permits graph dependencies to influence model-parallel execution.

6.1 Error tolerant convergence

Data-parallel ML algorithms are often robust against minor errors in intermediate calculations; as a consequence, they still execute correctly even when their model parameters

A experience synchronization delays (i.e. the P workers only see old or stale parameters), provided those delays are strictly bounded [8], [9], [14], [23], [31], [35]. Petuum exploits this error-tolerance to reduce network communication/synchronization overheads substantially, by implementing the Stale Synchronous Parallel (SSP) consistency model [14], [23] on top of the parameter server system, which provides all machines with access to parameters A.

The SSP consistency model guarantees that if a worker reads from parameter server at iteration c, it is guaranteed to receive all updates from all workers computed at and before iteration $c-s-1$, where s is the staleness threshold. If this is impossible because some straggling worker is more than s iterations behind, the reader will stop until the straggler catches up and sends its updates. For stochastic gradient descent algorithms (such as the DML program), SSP has very attractive theoretical properties [14], which we partially re-state here:

Theorem 1 (adapted from [14]). **SGD under SSP, convergence in probability:** Let $f(\mathbf{x}) = \sum_{t=1}^{T} f_t(\mathbf{x})$ be a convex function, where the f_t are also convex. We search for a minimizer $\mathbf{x}^*$ via stochastic gradient descent on each component ∇f_t under SSP, with staleness parameter s and P workers. Let $\mathbf{u}_t := -\eta_t \nabla_t f_t(\tilde{\mathbf{x}}_t)$ with $\eta_t = \frac{\eta}{\sqrt{t}}$. Under suitable conditions (f_t are L-Lipschitz and bounded divergence $D(x||x') \leq F^2$), we have

$$P\left[\frac{R[X]}{T} - \frac{1}{\sqrt{T}}\left(\eta L^2 + \frac{F^2}{\eta} + 2\eta L^2 \mu_\gamma\right) \geq \tau\right] \leq \exp\left\{\frac{-T\tau^2}{2\bar{\eta}_T \sigma_\gamma + \frac{2}{3}\eta L^2 (2s+1)P\tau}\right\}$$

where $R[X] := \sum_{t=1}^{T} f_t(\tilde{x}_t) - f(x^*)$, and $\bar{\eta}_T = \frac{\eta^2 L^4 (\ln T + 1)}{T} = o(1)$ as $T \to \infty$.

This theorem has two implications: (1) learning under the SSP model is *correct* (like Bulk Synchronous Parallel learning), because $\frac{R[X]}{T}$ — which is the difference between the SSP parameter estimate and the true optimum — converges to $O(T^{-1/2})$ in probability with an exponential tail-bound; (2) keeping staleness (i.e. asynchrony) as low as possible improves per-iteration convergence — specifically, the bound becomes tighter with lower maximum staleness s, and lower average μ_γ and variance σ_γ of the staleness experienced

by the workers. Conversely, naive asynchronous systems (e.g. Hogwild! [35] and YahooLDA [28]) may experience poor convergence, particularly in production environments where machines may slow down due to other tasks or users. Such slowdown can cause the maximum staleness s and staleness variance σ_γ to become arbitrarily large, leading to poor convergence rates. In addition to the above theorem (which bounds the distribution of **x**), Dai *et al.* also showed that the variance of **x** can be bounded, ensuring reliability and stability near an optimum [14].

6.2 Dependency structures

Naive parallelization of model-parallel algorithms (e.g. coordinate descent) may lead to uncontrolled parallelization error and non-convergence, caused by inter-parameter dependencies in the model. The mathematical definition of dependency differs between algorithms and models; examples include the Markov Blanket structure of graphical models (explored in GraphLab [13]) and deep neural networks (partially considered in [5]), or the correlation between data dimensions in regression problems (explored in [22]).

Such dependencies have been thoroughly analyzed under fixed execution schedules (where each worker updates the same set of parameters every iteration) [10], [22], [36], but there has been little research on *dynamic schedules* that can react to changing model dependencies or model state A. Petuum's scheduler allows users to write dynamic scheduling functions $S_p^{(t)}(A^{(t)})$ — whose output is a set of model indices $\{j_1, j_2 0, \dots\}$, telling worker p to update $A_{j_1}, A_{j_2}, \dots$ — as per their application's needs. This enables ML programs to analyze dependencies at run time (implemented via `schedule()`), and select subsets of independent (or nearly-independent) parameters for parallel updates.

To motivate this, we consider a generic optimization problem, which many regularized regression problems — including the Petuum Lasso example — fit into:

Definition: Regularized Regression Problem (RRP)

$$\min_{\mathbf{w}\in\mathbb{R}^d} f(\mathbf{w}) + r(\mathbf{w}), \tag{16}$$

where $\mathbf{w}$ is the parameter vector, $r(\mathbf{w}) = \sum_i r(w_i)$ is separable and f has β-Lipschitz continuous gradient in the following sense:

$$f(\mathbf{w}+\mathbf{z}) \leq f(\mathbf{w}) + \mathbf{z}^\top \nabla f(\mathbf{w}) + \tfrac{\beta}{2}\mathbf{z}^\top X^\top X \mathbf{z}, \tag{17}$$

where $X = [\mathbf{x}_1, \dots, \mathbf{x}_d]$ are d feature vectors. W.l.o.g., we assume that each feature vector $\mathbf{x}_i$ is normalized, i.e., $\|\mathbf{x}_i\|_2 = 1, i = 1, \dots, d$. Therefore $|\mathbf{x}_i^\top \mathbf{x}_j| \leq 1$ for all i, j.

In the regression setting, $f(\mathbf{w})$ represents a least-squares loss, $r(\mathbf{w})$ represents a separable regularizer (e.g. ℓ_1 penalty), and $\mathbf{x}_i$ represents the i-th feature column of the design (data) matrix, each element in $\mathbf{x}_i$ is a separate data sample. In particular, $|\mathbf{x}_i^\top \mathbf{x}_j|$ is the correlation between the i-th and j-th feature columns. The parameters $\mathbf{w}$ are simply the regression coefficients.

In the context of the model-parallel equation (4), we can map the model $A = \mathbf{w}$, the data $D = X$, and the update equation $\Delta(A, S_p(A))$ to

$$w_{j_p}^+ \leftarrow \arg\min_{z\in\mathbb{R}} \tfrac{\beta}{2}[z - (w_{j_p} - \tfrac{1}{\beta} g_{j_p})]^2 + r(z), \tag{18}$$

where $S_p^{(t)}(A)$ has selected a single coordinate j_p to be updated by worker p — thus, P coordinates are updated in every iteration. The aggregation function $F()$ simply allows each update w_{j_p} to pass through without change.

The effectiveness of parallel coordinate descent depends on how the schedule $S_p^{(t)}()$ selects the coordinates j_p. In particular, naive random selection can lead to poor convergence rate or even divergence, with error proportional to the correlation $|\mathbf{x}_{j_a}^\top \mathbf{x}_{j_b}|$ between the randomly-selected coordinates j_a, j_b [10], [22]. An effective and cheaply-computable schedule $S_{RRP,p}^{(t)}()$ involves randomly proposing a small set of $Q > P$ features $\{j_1, \dots, j_Q\}$, and then finding P features in this set such that $|\mathbf{x}_{j_a}^\top \mathbf{x}_{j_b}| \leq \theta$ for some threshold θ, where j_a, j_b are any two features in the set of P. This requires at most $\mathcal{O}(B^2)$ evaluations of $|\mathbf{x}_{j_a}^\top \mathbf{x}_{j_b}| \leq \theta$ (if we cannot find P features that meet the criteria, we simply reduce the degree of parallelism). We have the following convergence theorem:

Theorem 2. **$S_{RRP}()$ convergence:** Let $\epsilon := \frac{d(\mathbb{E}[P^2]/\mathbb{E}[P-1])(\rho-1)}{d^2} \approx \frac{(\mathbb{E}[P-1])(\rho-1)}{d} < 1$, where ρ is a constant that depends on the input data $\mathbf{x}$ and the scheduler $S_{RRP}()$. After t steps, we have

$$\mathbb{E}[F(\mathbf{w}^{(t)}) - F(\mathbf{w}^\star)] \leq \frac{Cd\beta}{\mathbb{E}[P(1-\epsilon)]}\frac{1}{t}, \tag{19}$$

where $F(\mathbf{w}) := f(\mathbf{w}) + r(\mathbf{w})$ and $\mathbf{w}^\star$ is a minimizer of F. $\mathbb{E}[P]$ is the average degree of parallelization over all iterations — we say "average" to account for situations where the scheduler cannot select P nearly-independent parameters (due to high correlation in the data). The proof for this theorem can be found in the Appendix. For most real-world data sets, this is not a problem, and $\mathbb{E}[P]$ is equal to the number of workers.

This theorem states that $S_{RRP}()$-scheduling (which is used by Petuum Lasso) achieves close to P-fold (linear) improvement in per-iteration convergence (where P is the number of workers). This comes from the $1/\mathbb{E}[P(1-\epsilon)]$ factor on the RHS of Eq. (19); for input data $\mathbf{x}$ that is sparse and high-dimensional, the $S_{RRP}()$ scheduler will cause $\rho - 1$ to become close to zero, and therefore ϵ will also be close to zero — thus the per-iteration convergence rate is improved by nearly P-fold. We contrast this against a naive system that selects coordinates at random; such a system will have far larger $\rho - 1$, thus degrading per-iteration convergence.

In addition to asymptotic convergence, we show that S_{RRP}'s trajectory is close to ideal parallel execution:

Theorem 3. **$S_{RRP}()$ is close to ideal execution:** Let $S_{ideal}()$ be an oracle schedule that always proposes P random features with zero correlation. Let $\mathbf{w}_{ideal}^{(t)}$ be its parameter trajectory, and let $\mathbf{w}_{RRP}^{(t)}$ be the parameter trajectory of $S_{RRP}()$. Then,

$$\mathbb{E}[|\mathbf{w}_{ideal}^{(t)} - \mathbf{w}_{RRP}^{(t)}|] \leq \frac{2dPm}{(t+1)^2\hat{P}} L^2 X^\top X C, \tag{20}$$

for constants $C, m, L, \hat{P}$. The proof for this theorem can be found in the Appendix.

This theorem says that the difference between the $S_{RRP}()$ parameter estimate $\mathbf{w}_{RRP}$ and the ideal oracle estimate $\mathbf{w}_{ideal}$ rapidly vanishes, at a fast $1/(t+1)^2$ rate. In other

words, one cannot do much better than $S_{RRP}()$ scheduling — it is near-optimal.

We close this section by noting that $S_{RRP}()$ is different from Scherrer *et al.* [22], who pre-cluster all M features before starting coordinate descent, in order to find "blocks" of nearly-independent parameters. In the Big Data and especially Big Model setting, feature clustering can be prohibitive — fundamentally, it requires $\mathcal{O}(M^2)$ evaluations of $|\mathbf{x}_i^\top \mathbf{x}_j|$ for all M^2 feature combinations (i, j), and although greedy clustering algorithms can mitigate this to some extent, feature clustering is still impractical when M is very large, as seen in some regression problems [27]. The proposed $S_{RRP}()$ only needs to evaluate a small number of $|\mathbf{x}_i^\top \mathbf{x}_j|$ every iteration. Furthermore, the random selection in $S_{RRP}()$ can be replaced with *prioritization* to exploit non-uniform convergence in ML problems, as explained next.

6.3 Non-uniform convergence

In model-parallel ML programs, it has been empirically observed that some parameters A_j can converge in much fewer/more updates than other parameters [21]. For instance, this happens in Lasso regression because the model enforces sparsity, so most parameters remain at zero throughout the algorithm, with low probability of becoming non-zero again. Prioritizing Lasso parameters according to their magnitude greatly improves convergence per iteration, by avoiding frequent (and wasteful) updates to zero parameters [21].

We call this *non-uniform ML convergence*, which can be exploited via a dynamic scheduling function $S_p^{(t)}(A^{(t)})$ whose output changes according to the iteration t — for instance, we can write a scheduler $S_{mag}()$ that proposes parameters with probability proportional to their current magnitude $(A_j^{(t)})^2$. This $S_{mag}()$ can be combined with the earlier dependency structure checking, leading to a *dependency-aware, prioritizing scheduler*. Unlike the dependency structure issue, prioritization has not received as much attention in the ML literature, though it has been used to speed up the PageRank algorithm, which is iterative-convergent [37].

The prioritizing schedule $S_{mag}()$ can be analyzed in the context of the Lasso problem. First, we rewrite it by duplicating the original J features with opposite sign, as in [10]: $F(\boldsymbol{\beta}) := \min_{\boldsymbol{\beta}} \frac{1}{2} \|\mathbf{y} - \mathbf{X}\boldsymbol{\beta}\|_2^2 + \lambda \sum_{j=1}^{2J} \beta_j$. Here, $\mathbf{X}$ contains $2J$ features and $\beta_j \geq 0$, for all $j = 1, \ldots, 2J$.

Theorem 4. [Adapted from [21]] **Optimality of Lasso priority scheduler:** Suppose $\mathcal{B}$ is the set of indices of coefficients updated in parallel at the t-th iteration, and ρ is sufficiently small constant such that $\rho \delta\beta_j^{(t)} \delta\beta_k^{(t)} \approx 0$, for all $j \neq k \in \mathcal{B}$. Then, the sampling distribution $p(j) \propto (\delta\beta_j^{(t)})^2$ approximately maximizes a lower bound on $\mathbb{E}_{\mathcal{B}}[F(\boldsymbol{\beta}^{(t)}) - F(\boldsymbol{\beta}^{(t)} + \delta\boldsymbol{\beta}^{(t)})]$.

This theorem shows that a prioritizing scheduler will speed up Lasso convergence, by decreasing the objective as much as is theoretically possible every iteration.

In practice, the Petuum scheduler system approximates $p(j) \propto (\delta\beta_j^{(t)})^2$ with $p'(j) \propto (\beta_j^{(t-1)})^2 + \eta$, in order to allow *pipelining* of multiple iterations for faster real-time convergence[6]. The constant η ensures that all β_j's have a non-zero probability of being updated.

7 Performance

Petuum's ML-centric system design supports a variety of ML programs, and improves their performance on Big Data in the following senses: (1) Petuum ML implementations achieve significantly faster convergence rate than well-optimized single-machine baselines (i.e., DML implemented on single machine, and Shotgun [10]); (2) Petuum ML implementations can run faster than other programmable platforms (e.g. Spark, GraphLab[7]), because Petuum can exploit model dependencies, uneven convergence and error tolerance; (3) Petuum ML implementations can reach larger model sizes than other programmable platforms, because Petuum stores ML program variables in a lightweight fashion (on the parameter server and scheduler); (4) for ML programs without distributed implementations, we can implement them on Petuum and show good scaling with an increasing number of machines. We emphasize that Petuum is, for the moment, primarily about allowing ML practitioners to implement and experiment with new data/model-parallel ML algorithms on small-to-medium clusters. Our experiments are therefore focused on clusters with 10-100 machines, in accordance with our target users.

7.1 Hardware Configuration

To demonstrate that Petuum is adaptable to different hardware generations, our experiments used 3 clusters with varying specifications: **Cluster-1** has up to 128 machines with 2 AMD cores, 8GB RAM, 1Gbps Ethernet; **Cluster-2** has up to 16 machines with 64 AMD cores, 128GB RAM, 40Gbps Infiniband; **Cluster-3** has up to 64 machines with 16 Intel cores, 128GB RAM, 10Gbps Ethernet.

7.2 Parameter Server and Scheduler Performance

Petuum's Parameter Server (PS) and Scheduler speed up existing ML algorithms by improving *iteration throughput* and *iteration quality* respectively. We measure iteration throughput as "iterations executed per second", and we quantify iteration quality by plotting the ML objective function $\mathcal{L}$ against iteration number — "objective progress per iteration". In either case, the goal is to improve the ML algorithm's real-time convergence rate, quantified by plotting the objective function $\mathcal{L}$ against real time ("objective progress per second").

Parameter Server (PS): We consider how the PS improves iteration throughput (through the Eager SSP consistency model), evaluated using PLMlib's Matrix Factorization with the `schedule()` function disabled (in order to remove the beneficial effect of scheduling, so we may focus on the PS). This experiment was conducted using 64 **Cluster-3** machines on a 332GB sparse matrix (7.7m by 288k entries, 26b nonzeros, created by duplicating the Netflix dataset 16 times horizontally and 16 times vertically). We compare

6. Without this approximation, pipelining is impossible because $\delta\beta_j^{(t)}$ is unavailable until all computations on $\beta_j^{(t)}$ have finished.

7. We omit Hadoop and Mahout, as it is already well-established that Spark and GraphLab significantly outperform it [12], [13].

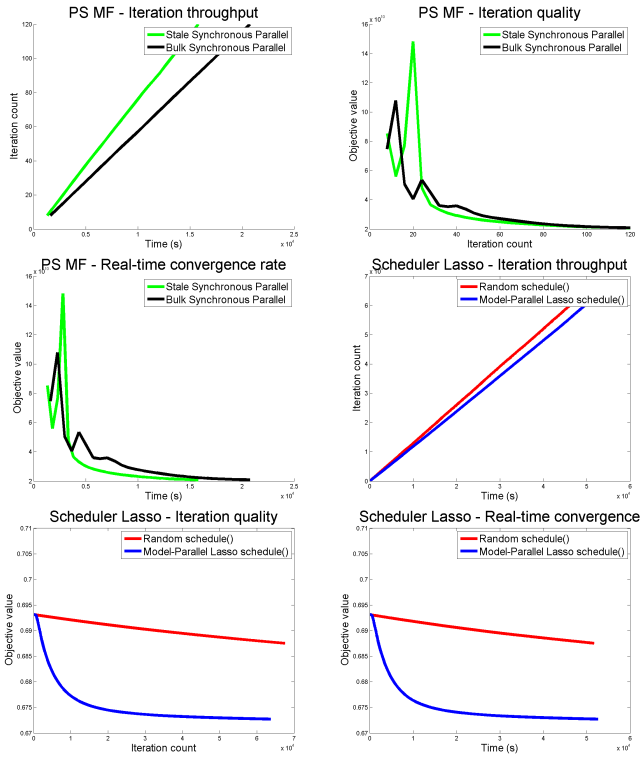


Fig. 13. Performance increase in ML applications due to the Petuum Parameter Server (PS) and Scheduler. The Eager Stale Synchronous Parallel (ESSP) consistency model (on the PS) improves the number of iterations executed per second (throughput) while maintaining per-iteration quality. Prioritized, dependency-aware scheduling allows the Scheduler to improve the quality of each iteration, while maintaining iteration throughput. In both cases, overall real-time convergence rate is improved — 30% improvement for the PS Matrix Factorization example, and several orders of magnitude for the Scheduler Lasso example.

the performance of MF running under Petuum PS's Eager SSP mode (using staleness $s = 2$; higher staleness values did not produce additional benefit), versus running under MapReduce-style Bulk Synchronous Parallel (BSP) mode. Figure 13 shows that ESSP provides a 30% improvement to iteration throughput (top left), without a significantly affecting iteration quality (top right). Consequently, the MF application converges 30% faster in real time (middle left).

The iteration throughput improvement occurs because ESSP allows both gradient computation and inter-worker communication to occur at the same time, whereas classic BSP execution requires computation and communication to alternate in a mutually exclusive manner. Because the maximum staleness $s = 2$ is small, and because ESSP eagerly pushes parameter updates as soon as they are available, there is almost no penalty to iteration quality despite allowing staleness.

Scheduler: We examine how the Scheduler improves iteration quality (through a well-designed `schedule()` function), evaluated using PMLlib's Lasso application. This experiment was conducted using 16 **Cluster-2** on a simulated 150GB sparse dataset (50m features); adjacent features in the dataset are highly correlated in order to simulate the effects of realistic feature engineering. We compare the original PMLlib Lasso (whose `schedule()` performs prioritization and dependency checking) to a simpler version whose `schedule()` selects parameters at random (the shotgun algorithm [10]). Figure 13 shows that PMLlib Lasso's `schedule()` slightly decreases iteration throughput (middle right), but greatly improves iteration quality (bottom left), resulting in several orders of magnitude improvement to real-time convergence (bottom right).

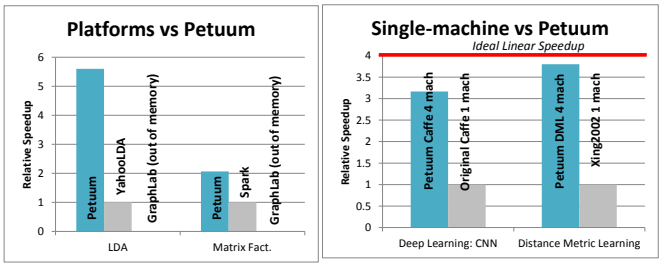


Fig. 14. **Left:** Petuum relative speedup vs popular platforms (larger is better). Across ML programs, Petuum is at least 2-10 times faster than popular implementations. **Right:** Petuum allows single-machine algorithms to be transformed into cluster versions, while still achieving near-linear speedup with increasing number of machines (Caffe CNN and DML).

The iteration quality improvement is mostly due to prioritization; we note that without prioritization, 85% of the parameters would converge within 5 iterations, but the remaining 15% would take over 100 iterations. Moreover, prioritization alone is not enough to achieve fast convergence speed — when we repeated the experiment with a prioritization-only `schedule()` (not shown), the parameters became unstable, which caused the objective function to diverged. This is because dependency checking is necessary to avoid correlation effects in Lasso (discussed in the proof to Theorem 2), which we observed were greatly amplified under the prioritization-only `schedule()`.

7.3 Comparison to Programmable Platforms

Figure 14 (left) compares Petuum to popular platforms for writing new ML programs (Spark v1.2 and GraphLab), as well as a well-known cluster implementation of LDA (YahooLDA). We compared Petuum to Spark, GraphLab and YahooLDA on two applications: LDA and MF. We ran LDA on 128 **Cluster-1** machines, using 3.9m English Wikipedia abstracts with unigram ($V = 2.5$m) and bigram ($V = 21.8$m) vocabularies; the bigram vocabulary is an example of feature engineering to improve performance at the cost of additional computation. The MF comparison was performed on 10 **Cluster-2** machines using the original Netflix dataset.

Speed: For MF and LDA, Petuum is between 2-6 times faster than other platforms (Figures 14, 15). For MF, Petuum uses the same model-parallel approach as Spark and GraphLab, but it performs twice as fast as Spark, while GraphLab ran out of memory (due to the need to construct an explicit graph representation, which consumes significant memory). On the other hand, Petuum LDA is nearly 6 times faster than YahooLDA; the speedup mostly comes from the Petuum LDA `schedule()` (Figure 10), which performs correct model-parallel execution by only allowing each worker to operate on disjoint parts of the vocabulary. This is similar to GraphLab's implementation,

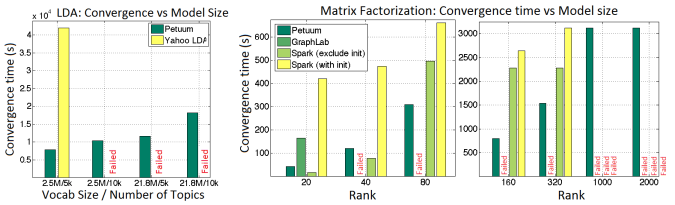


Fig. 15. **Left:** LDA convergence time: Petuum vs YahooLDA (lower is better). Petuum's data-and-model-parallel LDA converges faster than YahooLDA's data-parallel-only implementation, and scales to more LDA parameters (larger vocab size, number of topics). **Right panels:** Matrix Factorization convergence time: Petuum vs GraphLab vs Spark. Petuum is fastest and the most memory-efficient, and is the only platform that could handle Big MF models with rank $K \geq 1000$ on the given hardware budget.

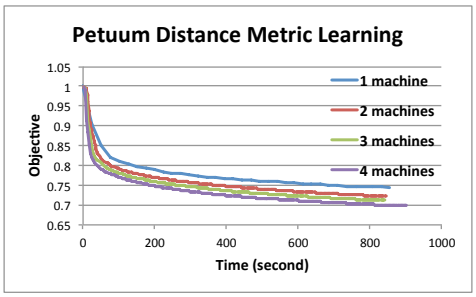


Fig. 16. Petuum DML objective vs time convergence curves, from 1 to 4 machines.

but is far more memory-efficient because Petuum does not need to construct a full graph representation of the problem.

Model Size: We show that Petuum supports larger ML models for the same amount of cluster memory. Figure 15 shows ML program running time versus model size, given a fixed number of machines — the left panel compares Petuum LDA and YahooLDA; PetuumLDA converges faster and supports LDA models that are > 10 times larger[8], allowing long-tail topics to be captured. The right panels compare Petuum MF versus Spark and GraphLab; again Petuum is faster and supports much larger MF models (higher rank) than either baseline. Petuum's model scalability comes from two factors: (1) model-parallelism, which divides the model across machines; (2) a lightweight parameter server system with minimal storage overhead. In contrast, Spark and GraphLab have additional overheads that may not be necessary in an ML context — Spark needs to construct a "lineage graph" in order to preserve its strict fault recovery guarantees, while GraphLab needs to represent the ML problem in graph form. Because ML applications are error-tolerant, fault recovery can be performed with lower overhead through periodic checkpointing.

7.4 Fast Cluster Implementations of New ML Programs

Petuum facilitates the development of new ML programs without existing cluster implementations; here we present two case studies. The first is a cluster version of the open-source Caffe CNN toolkit, created by adding ~ 600 lines of Petuum code. The basic data-parallel strategy in Caffe was left unchanged, so the Petuum port directly tests Petuum's efficiency. We tested on 4 **Cluster-3** machines, using a 250k subset of Imagenet with 200 classes, and 1.3m model parameters. Compared to the original single-machine Caffe (which does not have the overhead of network communication), Petuum approaches linear speedup (3.1-times speedup on 4 machines, Figure 14 right plot) due to the parameter server's ESSP consistency for managing network communication.

Second, we compare the Petuum DML program against the original DML algorithm proposed in [25] (denoted by Xing2002), implemented using SGD on a single machine (with parallelization over matrix operations). The intent is to show that, even for ML algorithms that have received less research attention towards scalability (such as DML), one can still implement a reasonably simple data-parallel SGD algorithm on Petuum, and enjoy the benefits of parallelization over a cluster. The DML experiment was run on 4 **Cluster-2** machines, using the 1-million-sample Imagenet [38] dataset with 1000 classes (21.5k-by-21.5k matrix with 220m model parameters), and 200m similar/dissimilar statements. The Petuum DML implementation converges 3.8 times faster than Xing2002 on 4 machines (Figure 14, right plot). We also evaluated Petuum DML's convergence speed on 1-4 machines (Figure 16) — compared to using 1 machine, Petuum DML achieves 3.8 times speedup with 4 machines and 1.9 times speedup with 2 machines.

8 Summary and Future Work

Petuum provides ML practitioners with an ML library and ML programming platform, capable of handling Big Data and Big ML Models with performance that is competitive with specialized implementations, while running on reasonable cluster sizes (10-100 machines). This is made possible by systematically exploiting the unique properties of iterative-convergent ML algorithms — error tolerance, dependency structures and uneven convergence; these properties have yet to be thoroughly explored in general-purpose Big Data platforms such as Hadoop and Spark.

In terms of feature set, Petuum is still relatively immature compared to Hadoop and Spark, and lacks the following: fault recovery from partial program state (critical for scaling to 1000+ machines), ability to adjust resource usage on-the-fly in running jobs, scheduling jobs for multiple users (multi-tenancy), a unified data interface that closely integrates with databases and distributed file systems, and support for interactive scripting languages such as Python and R. The lack of these features imposes a barrier to entry for new users, and future work on Petuum will address these issues — but in a manner consistent with Petuum's focus on iterative-convergent ML properties. For example, fault recovery in ML does not require perfect, synchronous checkpoints (used in Hadoop and Spark); instead, checkpoints with ESSP-style bounded error consistency can be used. This in turn opens up new ways to achieve on-the-fly resource adjustment and multi-tenancy.

Acknowledgments

This work is supported in part by DARPA FA87501220324, and NSF IIS1447676 grants to Eric P. Xing.

8. LDA model size is equal to vocab size times number of topics.

APPENDIX

PROOF OF THEOREM 2

We prove that the Petuum $S_{RRP}()$ scheduler makes the Regularized Regression Problem converge. We note that $S_{RRP}()$ has the following properties: (1) the scheduler *uniformly* randomly selects Q out of d coordinates (where d is the number of features); (2) the scheduler performs *dependency checking* and retains P out of Q coordinates; (3) in parallel, each of the P workers is assigned one coordinate, and performs coordinate descent on it:

$$w_{j_p}^+ \leftarrow \arg\min_{z\in\mathbb{R}} \tfrac{\beta}{2}[z-(w_{j_p}-\tfrac{1}{\beta}g_{j_p})]^2 + r(z), \qquad (21)$$

where $g_j = \nabla_j f(\mathbf{w})$ is the j-th partial derivative, and the coordinate j_p is assigned to the p-th worker. Note that (21) is simply the gradient update: $w \leftarrow w - \frac{1}{\beta}g$, followed by applying the proximity operator of r.

As we just noted, $S_{RRP}()$ scheduling selects P coordinates out of Q by performing *dependency checking*: effectively, the scheduler will put coordinates i and j into the same "block" iff $|\mathbf{x}_i^\top \mathbf{x}_j| \leq \theta$ for some "correlation threshold" $\theta \in (0,1)$. The idea is that coordinates in the same block will never be updated in parallel; the algorithm must choose the P coordinates from P distinct blocks. In order to analyze the effectiveness of this procedure, we will consider the following matrix:

$$\forall i, A_{ii} = 1, \quad \forall i \neq j, A_{ij} = \begin{cases} \mathbf{x}_i^\top \mathbf{x}_j, & \text{if } |\mathbf{x}_i^\top \mathbf{x}_j| \leq \theta \\ 0, & \text{otherwise} \end{cases}. \qquad (22)$$

This matrix A captures the impact of grouping coordinates into blocks, and its spectral radius $\rho = \rho(A)$ will be used to show that scheduling entails a nearly P-fold improvement in convergence with P processors. A simple bound for the spectral radius $\rho(A)$ is:

$$|\rho - 1| \leq \sum_{j\neq i} |A_{ij}| \leq (d-1)\theta. \qquad (23)$$

$S_{RRP}()$ scheduling sets the correlation threshold θ to a small constant, causing the spectral radius ρ to also be small (which will lead to a nearly P-fold improvement in per-iteration convergence rate). We contrast $S_{RRP}()$ with random shotgun-style [10] scheduling, which is equivalent to setting $\theta = 1$; this causes ρ to become large, which will degrade the per-iteration convergence rate.

Finally, let N denote the number of pairs (i,j) that pass the dependency check $|\mathbf{x}_i^\top \mathbf{x}_j| \leq \theta$. In high-dimensional problems with over 100 million dimensions, it is often the case that $N \approx d^2$, because each coordinate i is unlikely to be correlated with more than a few other coordinates j. We therefore assume $N \approx d^2$ for our analysis. We note that P, the number of coordinates selected for parallel update by the scheduler, is a random variable (because it may not always be possible to select P independent coordinates). Our analysis therefore considers the expected value $\mathbb{E}[P]$. We are now ready to prove Theorem 2:

Theorem 2: Let $\epsilon := \frac{d(\mathbb{E}[P^2]/\mathbb{E}[P-1])(\rho-1)}{N} \approx \frac{(\mathbb{E}[P-1])(\rho-1)}{d} < 1$, then after t steps, we have

$$\mathbb{E}[F(\mathbf{w}^{(t)}) - F(\mathbf{w}^\star)] \leq \frac{Cd\beta}{\mathbb{E}[P(1-\epsilon)]}\frac{1}{t}, \qquad (24)$$

where $F(\mathbf{w}) := f(\mathbf{w}) + r(\mathbf{w})$ and $\mathbf{w}^\star$ denotes a (global) minimizer of F (whose existence is assumed for simplicity).

Proof of Theorem 2: We first bound the algorithm's progress at step t. To avoid cumbersome double indices, let $\mathbf{w} = \mathbf{w}_t$ and $\mathbf{z} = \mathbf{w}_{t+1}$. Then, by applying (17), we have

$$\begin{aligned}
&\mathbb{E}[F(\mathbf{z}) - F(\mathbf{w})] \\
&\leq \mathbb{E}\Big[\sum_{p=1}^{P} g_{j_p}(w_{j_p}^+ - w_{j_p}) + r(w_{j_p}^+) - r(w_{j_p}) \\
&+ \frac{\beta}{2}(w_{j_p}^+ - w_{j_p})^2 + \frac{\beta}{2}\sum_{p\neq q}(w_{j_p}^+ - w_{j_p})(w_{j_q}^+ - w_{j_q})\mathbf{x}_{j_p}^\top \mathbf{x}_{j_q}\Big] \\
&= \frac{\mathbb{E}[P]}{d}\Big[\mathbf{g}^\top(\mathbf{w}^+ - \mathbf{w}) + r(\mathbf{w}^+) - r(\mathbf{w}) + \frac{\beta}{2}\|\mathbf{w}^+ - \mathbf{w}\|_2^2\Big] \\
&\qquad + \frac{\beta\mathbb{E}[P(P-1)]}{2N}(\mathbf{w}^+ - \mathbf{w})^\top(A - I)(\mathbf{w}^+ - \mathbf{w}) \\
&\leq -\frac{\beta\mathbb{E}[P]}{2d}\|\mathbf{w}^+ - \mathbf{w}\|_2^2 + \frac{\beta\mathbb{E}[P(P-1)](\rho-1)}{2N}\|\mathbf{w}^+ - \mathbf{w}\|_2^2 \\
&\leq -\frac{\beta\mathbb{E}[P(1-\epsilon)]}{2d}\|\mathbf{w}^+ - \mathbf{w}\|_2^2,
\end{aligned}$$

where we define $\epsilon = \frac{d(\mathbb{E}[P^2]/\mathbb{E}[P-1])(\rho-1)}{N}$, and the second inequality follows from the optimality of $\mathbf{w}^+$ as defined in (21). Therefore as long as $\epsilon < 1$, the algorithm is decreasing the objective. This in turn puts a limit on the maximum number of parallel workers, which is inversely proportional to the spectral radius ρ.

The rest of the proof follows the same line as the shotgun paper [10]. Briefly, consider the case where $0 \in \partial r(\mathbf{w}_t)$, then

$$F(\mathbf{w}_{t+1}) - F(\mathbf{w}^\star) \leq (\mathbf{w}_{t+1} - \mathbf{w}^\star)^\top \mathbf{g} \leq \|\mathbf{w}_{t+1} - \mathbf{w}^\star\|_2 \cdot \|\mathbf{g}\|_2,$$

and $\|\mathbf{w}_{t+1} - \mathbf{w}_t\|_2^2 = \|\mathbf{g}\|_2^2/\beta^2$. Thus, defining $\delta_t = F(\mathbf{w}_t) - F(\mathbf{w}^\star)$, we have

$$\begin{aligned}
\mathbb{E}(\delta_{t+1} - \delta_t) &\leq -\frac{\mathbb{E}[P(1-\epsilon)]}{2d\beta\|\mathbf{w}_{t+1} - \mathbf{w}^\star\|_2^2}\mathbb{E}(\delta_{t+1}^2) \qquad (25) \\
&\leq -\frac{\mathbb{E}[P(1-\epsilon)]}{2d\beta\|\mathbf{w}_{t+1} - \mathbf{w}^\star\|_2^2}[\mathbb{E}(\delta_{t+1})]^2. \qquad (26)
\end{aligned}$$

Using induction it follows that $E(\delta_t) \leq \frac{Cd\beta}{\mathbb{E}[P(1-\epsilon)]}\frac{1}{t}$ for some universal constant C. □

The theorem confirms two intuitions: The larger the number of selected coordinates $\mathbb{E}[P]$ (i.e. more parallel workers), the faster the algorithm converges per-iteration; however, this also increases ϵ, demonstrating a tradeoff between parallelization and correctness. Also, the smaller the variance $\mathbb{E}[P^2]$, the faster the algorithm converges (since ϵ is proportional to it).

Remark: We compare Theorem 2 with Shotgun [10] and the Block greedy algorithm in [22]. The convergence rate we get is similar to shotgun, but with a significant difference: Our spectral radius $\rho = \rho(A)$ is potentially much smaller than shotgun's $\rho(X^\top X)$, since by partitioning we zero out all entries in the correlation matrix $X^\top X$ that are bigger than the threshold θ. In other words, we get to control the spectral radius while shotgun is totally passive.

The convergence rate in [22] is $\frac{CB}{P(1-\epsilon')}\frac{1}{t}$, where $\epsilon' = \frac{(P-1)(\rho'-1)}{B-1}$. Compared with ours, we have a bigger (hence worse) numerator (d vs. B) but the denominator (ϵ' vs. ϵ) are not directly comparable: we have a bigger spectral radius

ρ and bigger d while [22] has a smaller spectral radius ρ' (essentially taking a submatrix of our A) and smaller $B-1$. Nevertheless, we note that [22] may have a higher per-step complexity: each worker needs to check all of its assigned τ coordinates just to update one "optimal" coordinate. In contrast, we simply pick a random coordinate, and hence can be much cheaper per-step.

PROOF OF THEOREM 3

For the Regularized Regression Problem, we prove that the Petuum $S_{RRP}()$ scheduler produces a solution trajectory $\mathbf{w}_{RRP}^{(t)}$ that is close to ideal execution:

Theorem 3: ($S_{RRP}()$ is close to ideal execution) Let $S_{ideal}()$ be an oracle schedule that always proposes P random features with zero correlation. Let $\mathbf{w}_{ideal}^{(t)}$ be its parameter trajectory, and let $\mathbf{w}_{RRP}^{(t)}$ be the parameter trajectory of $S_{RRP}()$. Then,

$$E[|\mathbf{w}_{ideal}^{(t)} - \mathbf{w}_{RRP}^{(t)}|] \leq \frac{2JPm}{(T+1)^2\hat{P}} L^2 X^T X C, \quad (27)$$

C is a data dependent constant, m is the strong convexity constant, L is the domain width of A_j, and $\hat{P}$ is the expected number of indexes that $S_{RRP}()$ can actually parallelize in each iteration (since it may not be possible to find P nearly-independent parameters).

We assume that the objective function $F(\mathbf{w}) = f(\mathbf{w}) + r(\mathbf{w})$ is strongly convex — for certain problems, this can be achieved through parameter replication, e.g. $\min_{\mathbf{w}} \frac{1}{2}||y - X\mathbf{w}||_2^2 + \lambda \sum_{j=1}^{2M} \mathbf{w}_j$ is the replicated form of Lasso regression seen in Shotgun [10].

Lemma 1: The difference between successive updates is:

$$F(\mathbf{w} + \Delta\mathbf{w}) - F(\mathbf{w}) \leq -(\Delta\mathbf{w})^T \Delta\mathbf{w} + \frac{1}{2}(\Delta\mathbf{w})^T X^T X \Delta\mathbf{w} \quad (28)$$

Proof of Lemma 1: The Taylor expansion of $F(\mathbf{w} + \Delta\mathbf{w})$ around $\mathbf{w}$ coupled with the fact that $F(\mathbf{w})'''$ (3rd-order) and higher order derivatives are zero leads to the above result. □

Proof of Theorem 3: By using Lemma 1, and telescoping sum:

$$F(\mathbf{w}_{ideal}^{(T)}) - F(\mathbf{w}_{ideal}^{(0)}) = \sum_{t=1}^{T} -(\Delta\mathbf{w}_{ideal}^{(t)})^\top \Delta\mathbf{w}_{ideal}^{(t)} + \frac{1}{2}(\Delta\mathbf{w}_{ideal}^{(t)})^\top X^\top X \Delta\mathbf{w}_{ideal}^{(t)} \quad (29)$$

Since S_{ideal} chooses P features with 0 correlation,

$$F(\mathbf{w}_{ideal}^{(T)}) - F(\mathbf{w}_{ideal}^{(0)}) = \sum_{t=1}^{T} -(\Delta\mathbf{w}_{ideal}^{(t)})^\top \Delta\mathbf{w}_{ideal}^{(t)}$$

Again using Lemma 1, and telescoping sum:

$$F(\mathbf{w}_{RRP}^{(T)}) - F(\mathbf{w}_{RRP}^{(0)}) = \sum_{t=1}^{T} -(\Delta\mathbf{w}_{RRP}^{(t)})^\top \Delta\mathbf{w}_{RRP}^{(t)} + \frac{1}{2}(\Delta\mathbf{w}_{RRP}^{(t)})^\top X^\top X \Delta\mathbf{w}_{RRP}^{(t)} \quad (30)$$

Taking the difference of the two sequences, we have:

$$F(\mathbf{w}_{ideal}^{(T)}) - F(\mathbf{w}_{RRP}^{(T)}) = \left(\sum_{t=1}^{T} -(\Delta\mathbf{w}_{ideal}^{(t)})^\top \Delta\mathbf{w}_{ideal}^{(t)}\right) - \left(\sum_{t=1}^{T} -(\Delta\mathbf{w}_{RRP}^{(t)})^\top \Delta\mathbf{w}_{RRP}^{(t)} + \frac{1}{2}(\Delta\mathbf{w}_{RRP}^{(t)})^\top X^\top X \Delta\mathbf{w}_{RRP}^{(t)}\right) \quad (31)$$

Taking expectations w.r.t. the randomness in iteration, indices chosen at each iteration, and the inherent randomness in the two sequences, we have:

$$\begin{aligned} &E[|F(\mathbf{w}_{ideal}^{(T)}) - F(\mathbf{w}_{RRP}^{(T)})|] = \\ &E[|(\sum_{t=1}^{T} -(\Delta\mathbf{w}_{ideal}^{(t)})^T \Delta\mathbf{w}_{ideal}^{(t)}) \\ &- (\sum_{t=1}^{T} -(\Delta\mathbf{w}_{RRP}^{(t)})^T \Delta\mathbf{w}_{RRP}^{(t)} + \frac{1}{2}(\Delta\mathbf{w}_{RRP}^{(t)})^\top X^\top X \Delta\mathbf{w}_{RRP}^{(t)})|] \\ &= (C_{data} + \frac{1}{2}) E[|\sum_{t=1}^{T} (\Delta\mathbf{w}_{RRP}^{(t)})^\top X^\top X \Delta\mathbf{w}_{RRP}^{(t)})|], \end{aligned} \quad (32)$$

where C_{data} is a data dependent constant. Here, the difference between $(\Delta\mathbf{w}_{ideal}^{(t)})^\top \Delta\mathbf{w}_{ideal}^{(t)}$ and $(\Delta\mathbf{w}_{RRP}^{(t)})^\top \Delta\mathbf{w}_{RRP}^{(t)}$ can only be possible due to $(\Delta\mathbf{w}_{RRP}^{(t)})^\top X^\top X \Delta\mathbf{w}_{RRP}^{(t)}$.

Following the proof in the shotgun paper [10], we get

$$E[|F(\mathbf{w}_{ideal}^{(t)}) - F(\mathbf{w}_{RRP}^{(t)})|] \leq \frac{2dP}{(t+1)^2\hat{P}} L^2 X^\top X C, \quad (33)$$

where d is the length of $\mathbf{w}$ (number of features), C is a data dependent constant, L is the domain width of $\mathbf{w}_j$ (i.e. the difference between its maximum and minimum possible values), and $\hat{P}$ is the expected number of indexes that $S_{RRP}()$ can actually parallelize in each iteration.

Finally, we apply the strong convexity assumption to get

$$E[|\mathbf{w}_{ideal}^{(t)} - \mathbf{w}_{RRP}^{(t)}|] \leq \frac{2dPm}{(t+1)^2\hat{P}} L^2 X^\top X C, \quad (34)$$

where m is the strong convexity constant. □

REFERENCES

[1] Q. Le, M. Ranzato, R. Monga, M. Devin, K. Chen, G. Corrado, J. Dean, and A. Ng, "Building high-level features using large scale unsupervised learning," in *ICML*, 2012.

[2] Y. Wang, X. Zhao, Z. Sun, H. Yan, L. Wang, Z. Jin, L. Wang, Y. Gao, J. Zeng, Q. Yang *et al.*, "Towards topic modeling for big data," *arXiv preprint arXiv:1405.4402*, 2014.

[3] J. Yuan, F. Gao, Q. Ho, W. Dai, J. Wei, X. Zheng, E. P. Xing, T.-Y. Liu, and W.-Y. Ma, "Lightlda: Big topic models on modest compute clusters," in *Accepted to International World Wide Web Conference*, 2015.

[4] Y. Zhou, D. Wilkinson, R. Schreiber, and R. Pan, "Large-scale parallel collaborative filtering for the netflix prize," in *Algorithmic Aspects in Information and Management*, 2008.

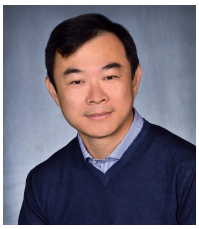
[5] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, Q. Le, M. Mao, M. Ranzato, A. Senior, P. Tucker, K. Yang, and A. Ng, "Large scale distributed deep networks," in *NIPS 2012*, 2012.

[6] S. A. Williamson, A. Dubey, and E. P. Xing, "Parallel markov chain monte carlo for nonparametric mixture models," in *ICML*, 2013.

[7] M. D. Hoffman, D. M. Blei, C. Wang, and J. Paisley, "Stochastic variational inference," *JMLR*, vol. 14, 2013.

[8] M. Zinkevich, J. Langford, and A. J. Smola, "Slow learners are fast," in *NIPS*, 2009.

[9] A. Agarwal and J. C. Duchi, "Distributed delayed stochastic optimization," in *NIPS*, 2011.
[10] J. K. Bradley, A. Kyrola, D. Bickson, and C. Guestrin, "Parallel coordinate descent for l1-regularized loss minimization," in *ICML*, 2011.
[11] T. White, *Hadoop: The definitive guide*. O'Reilly Media, Inc., 2012.
[12] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: cluster computing with working sets," in *HotCloud*, 2010.
[13] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein, "Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud," *PVLDB*, 2012.
[14] W. Dai, A. Kumar, J. Wei, Q. Ho, G. Gibson, and E. P. Xing, "High-performance distributed ml at scale through parameter server consistency models," in *AAAI*, 2015.
[15] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *ACM SIGMOD International Conference on Management of data*. ACM, 2010.
[16] R. Power and J. Li, "Piccolo: building fast, distributed programs with partitioned tables," in *OSDI*. USENIX Association, 2010.
[17] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su, "Scaling distributed machine learning with the parameter server," in *OSDI*, 2014.
[18] R. Gemulla, E. Nijkamp, P. J. Haas, and Y. Sismanis, "Large-scale matrix factorization with distributed stochastic gradient descent," in *KDD*, 2011. [Online]. Available: http://doi.acm.org/10.1145/2020408.2020426
[19] X. Chen, Q. Lin, S. Kim, J. Carbonell, and E. Xing, "Smoothing proximal gradient method for general structured sparse learning," in *UAI*, 2011.
[20] T. L. Griffiths and M. Steyvers, "Finding scientific topics," *PNAS*, vol. 101, no. Suppl 1, pp. 5228–5235, 2004.
[21] S. Lee, J. K. Kim, X. Zheng, Q. Ho, G. Gibson, and E. P. Xing, "On model parallelism and scheduling strategies for distributed machine learning," in *NIPS*, 2014.
[22] C. Scherrer, A. Tewari, M. Halappanavar, and D. Haglin, "Feature clustering for accelerating parallel coordinate descent," *NIPS*, 2012.
[23] Q. Ho, J. Cipar, H. Cui, J.-K. Kim, S. Lee, P. B. Gibbons, G. Gibson, G. R. Ganger, and E. P. Xing, "More effective distributed ml via a stale synchronous parallel parameter server," in *NIPS*, 2013.
[24] Y. Zhang, Q. Gao, L. Gao, and C. Wang, "Priter: A distributed framework for prioritized iterative computations," in *SOCC*, 2011.
[25] E. P. Xing, M. I. Jordan, S. Russell, and A. Y. Ng, "Distance metric learning with application to clustering with side-information," in *Advances in neural information processing systems*, 2002, pp. 505–512.
[26] J. V. Davis, B. Kulis, P. Jain, S. Sra, and I. S. Dhillon, "Information-theoretic metric learning," in *Proceedings of the 24th international conference on Machine learning*. ACM, 2007, pp. 209–216.
[27] H. B. M. et. al., "Ad click prediction: a view from the trenches," in *KDD*, 2013.
[28] A. Ahmed, M. Aly, J. Gonzalez, S. Narayanamurthy, and A. J. Smola, "Scalable inference in latent variable models," in *WSDM*, 2012.
[29] K. P. Murphy, *Machine learning: a probabilistic perspective*, Cambridge, MA, 2012.
[30] L. Yao, D. Mimno, and A. McCallum, "Efficient methods for topic model inference on streaming document collections," in *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*, ser. KDD '09. New York, NY, USA: ACM, 2009, pp. 937–946.
[31] A. Kumar, A. Beutel, Q. Ho, and E. P. Xing, "Fugue: Slow-worker-agnostic distributed learning for big models on big data," in *AISTATS*.
[32] J. Zhu, X. Zheng, L. Zhou, and B. Zhang, "Scalable inference in max-margin topic models," in *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2013, pp. 964–972.
[33] H.-F. Yu, C.-J. Hsieh, S. Si, and I. Dhillon, "Scalable coordinate descent approaches to parallel matrix factorization for recommender systems," in *Data Mining (ICDM), 2012 IEEE 12th International Conference on*. IEEE, 2012, pp. 765–774.
[34] A. Kumar, A. Beutel, Q. Ho, and E. P. Xing, "Fugue: Slow-worker-agnostic distributed learning for big models on big data," in *Proceedings of the Seventeenth International Conference on Artificial Intelligence and Statistics*, 2014, pp. 531–539.
[35] F. Niu, B. Recht, C. Ré, and S. J. Wright, "Hogwild!: A lock-free approach to parallelizing stochastic gradient descent," in *NIPS*, 2011.
[36] P. Richtárik and M. Takáč, "Parallel coordinate descent methods for big data optimization," *arXiv preprint arXiv:1212.0873*, 2012.
[37] Y. Zhang, Q. Gao, L. Gao, and C. Wang, "Priter: A distributed framework for prioritizing iterative computations," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 24, no. 9, pp. 1884–1893, 2013.
[38] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*. IEEE, 2009, pp. 248–255.



Eric P. Xing Dr. Eric Xing is a Professor of Machine Learning in the School of Computer Science at Carnegie Mellon University, and the director of the CMU Center for Machine Learning and Health. His principal research interests lie in the development of machine learning and statistical methodology; especially for solving problems involving automated learning, reasoning, and decision-making in high-dimensional, multimodal, and dynamic possible worlds in social and biological systems. Professor Xing received a Ph.D. in Molecular Biology from Rutgers University, and another Ph.D. in Computer Science from UC Berkeley. His current work involves, 1) foundations of statistical learning, including theory and algorithms for estimating time/space varying-coefficient models, sparse structured input/output models, and nonparametric Bayesian models; 2) framework for parallel machine learning on big data with big model in distributed systems or in the cloud; 3) computational and statistical analysis of gene regulation, genetic variation, and disease associations; and 4) application of machine learning in social networks, natural language processing, and computer vision. He is an associate editor of the Annals of Applied Statistics (AOAS), the Journal of American Statistical Association (JASA), the IEEE Transaction of Pattern Analysis and Machine Intelligence (PAMI), the PLoS Journal of Computational Biology, and an Action Editor of the Machine Learning Journal (MLJ), the Journal of Machine Learning Research (JMLR). He is a member of the DARPA Information Science and Technology (ISAT) Advisory Group, and a Program Chair of ICML 2014.

Qirong Ho Dr. Qirong Ho is a scientist at the Institute for Infocomm Research, A*STAR, Singapore, and an adjunct assistant professor at the Singapore Management University School of Information Systems. His primary research focus is distributed cluster software systems for Machine Learning at Big Data scales, with a view towards correctness and performance guarantees. In addition, Dr. Ho has performed research on statistical models for large-scale network analysis — particularly latent space models for visualization, community detection, user personalization and interest prediction — as well as social media analysis on hyperlinked documents with text and network data. Dr. Ho received his PhD in 2014, under Eric P. Xing at Carnegie Mellon University's Machine Learning Department. He is a recipient of the Singapore A*STAR National Science Search Undergraduate and PhD fellowships.