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

Eric P. Xing[1], Qirong Ho[2], Wei Dai[1], Jin Kyu Kim[1], Jinliang Wei[1], Seunghak Lee[1], Xun Zheng[1], Pengtao Xie[1], Abhimanu Kumar[1], and Yaoliang Yu[1]

[1]School of Computer Science, Carnegie Mellon University

[2]Institute for Infocomm Research, A*STAR, Singapore

{*epxing,wdai,jinkyuk,jinlianw,seunghak,xunzheng,pengtaox,yaoliang*}*@cs.cmu.edu*,{*hoqirong,abhimanyu.kumar*}*@gmail.com*

ABSTRACT

How can one build a distributed framework that allows efficient deployment of a wide spectrum of modern advanced machine learning (ML) programs for industrial-scale problems using Big Models (100s of billions of parameters) on Big Data (terabytes or petabytes)? Contemporary parallelization strategies employ fine-grained operations and scheduling beyond the classic bulk-synchronous processing paradigm popularized by MapReduce, or even specialized operators relying on graphical 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 different ML programs at scale. We propose a general-purpose framework that systematically addresses data- and model-parallel challenges in large-scale ML, by leveraging several fundamental properties underlying ML programs that make them different from conventional operation-centric programs: error tolerance, dynamic structure, and nonuniform convergence; all stem from the optimization-centric nature shared in ML programs' mathematical definitions, and the iterative-convergent behavior of their algorithmic solutions. These properties present unique opportunities for an integrative system design, built on bounded-latency network synchronization and dynamic load-balancing scheduling, which is efficient, programmable, and enjoys provable correctness guarantees. We demonstrate how such a design in light of ML-first principles leads to significant performance improvements versus well-known implementations of several ML programs, allowing them to run in much less time and at considerably larger model sizes, on modestly-sized computer clusters.

Categories and Subject Descriptors

H.3.4 [**Information Storage and Retrieval**]: Systems and Software—*Distributed Systems*; G.3 [**Probability and Statistics**]: Probabilistic algorithms; G.4 [**Mathematical Software**]: Parallel and vector implementations

KDD 2015

DOI: http://dx.doi.org/10.1145/2783258.2783323.

General Terms

Design, Theory, Algorithms, Experimentation, Performance

Keywords

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 space and time bottlenecks. For example, the Clueweb 2012 web crawl[1] contains > 700m web pages as 27TB of text, 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 other hand, state-of-the-art image recognition systems have now embraced large-scale deep learning models with billions of parameters [17]; topic models with up to 10^6 topics can cover long-tail semantic word sets for substantially improved online advertising [26, 31]; and very-high-rank matrix factorization yields improved prediction on collaborative filtering problems [35]. Training such big models with a single machine can be prohibitively slow, if possible.

Despite the recent rapid development of many new ML models and algorithms aiming at scalable application [9, 28, 15, 36, 1, 5], 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. We believe that, from the scalable execution point of view, a main reason 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.

[1]`http://www.lemurproject.org/clueweb12.php`

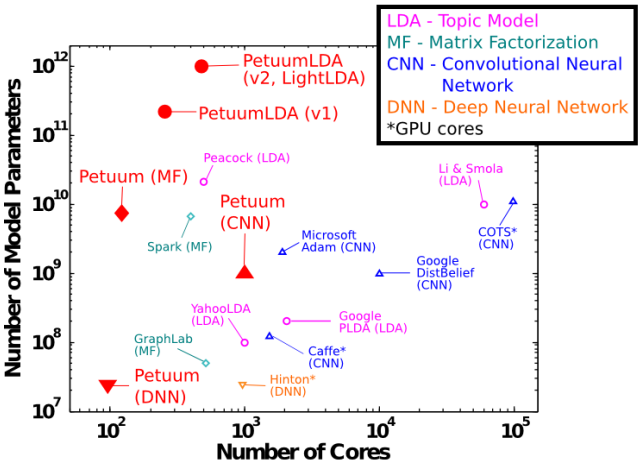

Figure 1: The scale of Big ML efforts in recent literature. A key goal of Petuum is to enable larger ML models to be run on fewer resources, even relative to highly-specialized implementations.

Many platforms have provided partial solutions to bridge this research-to-production gap: while Hadoop [27] is a popular and easy to program platform, the simplicity of its MapReduce abstraction makes it difficult to exploit ML properties such as error tolerance (at least, not without considerable engineering effort to bypass MapReduce limitations), and its performance on many ML programs has been surpassed by alternatives [32, 20]. One such alternative is Spark [32], which generalizes MapReduce and scales well on data while offering an accessible programming interface; yet, Spark does not offer 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 [7]. Graph-centric platforms such as GraphLab [20] and Pregel [21] efficiently partition graph-based models with built-in scheduling and consistency mechanisms; but ML programs such as topic modeling and regression either do not admit obvious graph representations, or a graph representation may not be the most efficient choice; moreover, due to limited theoretical work, it is unclear whether asynchronous graph-based consistency models and scheduling will always yield correct execution of such ML programs. Other systems provide low-level programming interfaces [23, 19], 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 and coordinate descent for determining optimality or fixed-point in optimization programs [3, 12], MCMC and variational methods for graphical models [13, 15], proximal optimization and ADMM for structured sparsity problems [6, 4], among others. To our knowledge, no existing ML 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 recognizing 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 *dichotomous parallel* approach keenly exploits the unique statistical nature of ML algorithms, particularly 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 frameworks discussed earlier. For example, central to Spark [32] is the principle of perfect fault tolerance and recovery, supported by a persistent memory architecture (Resilient Distributed Datasets); whereas central to GraphLab 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 a bounded staleness guarantee, which achieves provably correct outcomes due to the error-tolerant 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 a data-parallel and a model-parallel algorithm can be implemented on Petuum, allowing them to scale to large data and model sizes with improved algorithm convergence times. Figure 1 offers a glimpse of the model scalability achievable on Petuum, where we show a range of Petuum ML programs at large model scales (up to a trillion parameters), on relatively modest clusters (10-100 machines) that are within reach of most ML practitioners. The experiments section provides more 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[2], which includes more algorithms not explored in this paper: random forests, K-means, sparse coding, MedLDA, SVM, multi-class logistic regression, with many others being actively developed for future releases.

2. PRELIMINARIES: ON DATA 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 (§5) 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 loss $\mathcal{L}$ (i.e., a fitness function such as RMS 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 iteration. 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 by $F()$. 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 and solution, as oppose to heuristics or procedures not associated such a loss function.

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,

[2]Petuum is available as open source at `http://petuum.org`.

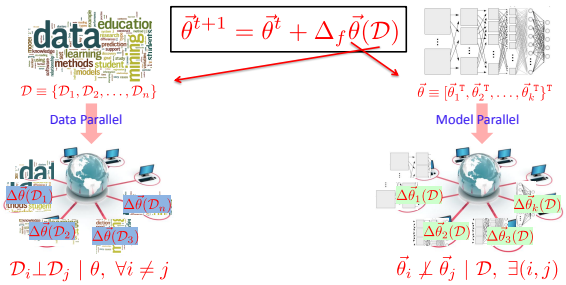

Figure 2: The difference between data and model parallelism: data samples are always conditionally independent given the model, but there are some model parameters that are not independent of each other.

whereas *model-parallelism*, which divides the ML model, is common for Big Models. Below, we discuss the (different) mathematical implications of each parallelism (see Fig. 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. We assume that the function $\Delta()$ can be applied to each of these data subsets independently, yielding a data-parallel update equation:

$$A^{(t)} = F(A^{(t-1)}, \textstyle\sum_{p=1}^{P} \Delta(A^{(t-1)}, D_p)). \quad (2)$$

In this definition, we assume that the $\Delta()$ outputs are aggregated via summation, which is commonly seen in stochastic gradient descent or sampling-based algorithms. For example, in distance metric learning problem which is optimized with stochastic gradient descent (SGD), the data pairs are partitioned over different workers, and the intermediate results (subgradients) are computed on each partition and are summed before applied to update the model parameters. Other algorithms can also be expressed in this form, such as variational EM algorithms $A^{(t)} = \sum_{p=1}^{P} \Delta(A^{(t-1)}, D_p)$. Importantly, this *additive updates* property allows the updates $\Delta()$ to be aggregated 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 $\Delta()$. Unlike data-parallelism, each update function $\Delta()$ also takes a scheduling function $S_p^{(t-1)}()$, which restricts $\Delta()$ to operate on a subset of the model parameters A:

$$A^{(t)} = F\left(A^{(t-1)}, \{\Delta(A^{(t-1)}, S_p^{(t-1)}(A^{(t-1)}))\}_{p=1}^{P}\right), \quad (3)$$

where we have omitted the data D for brevity and clarity. $S_p^{(t-1)}()$ outputs a set of indices $\{j_1, j_2, \ldots, \}$, so that $\Delta()$ only performs updates on $A_{j_1}, A_{j_2}, \ldots$ — we refer to such selection of model parameters as *scheduling*.

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 [18, 5, 25, 20]. Hence, our definition of model-parallelism includes a *global scheduling mechanism* 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 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 in a high-performance non-blocking manner that still guarantees convergence. Ideally, the platform should also offer easy, global-variable-like access to A (as opposed to cumbersome message-passing, or non-stateful MapReduce-like functional interfaces). If the program is model-parallel, it may require fine control over parameter scheduling to avoid non-convergence; such capability is not available in Hadoop, Spark nor GraphLab without code modification. Hence, there is an opportunity to address these considerations via a platform tailored to data- and model-parallel ML.

3. THE PETUUM FRAMEWORK

A core goal of Petuum is to allow easy implementation of data- and model-parallel ML algorithms. Petuum provides APIs to key systems that make this task 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 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 [14, 7, 18, 33, 19, 20]. 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. 3): scheduler, workers, and parameter server, and Petuum ML programs are written in C++ (with Java support coming in the near future).

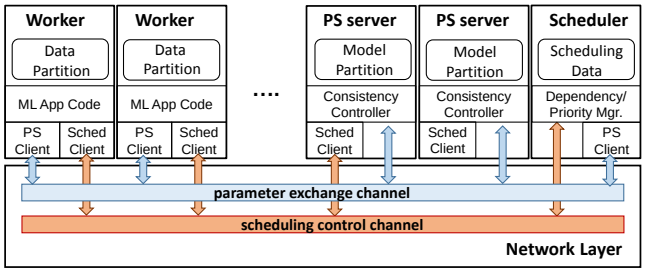

Figure 3: Petuum scheduler, workers, parameter servers.

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 — for example, a simple schedule might pick a random parameter for every worker, while a more complex scheduler (as we will show) may pick parameters according to multiple criteria, such as pair-wise independence or distance from convergence. The scheduler sends the *identities* of these parameters to workers via the scheduling control channel (Fig. 3), while the actual parameter values are delivered through a parameter server system that we will soon explain; the scheduler is responsible only for deciding *which* parameters to update. In Section 5, we will discuss theoretical guarantees enjoyed by model-parallel schedules.

Several common patterns for schedule design are worth highlighting: the simplest is a **fixed-schedule** (`schedule_fix()`), which dispatches parameters A in a pre-determined order (as is common in existing ML implementations). Static, round-robin schedules (e.g. repeatedly loop over all parameters) fit the `schedule_fix()` model. Another type of schedule is **dependency-aware** (`schedule_dep()`) scheduling, which allows re-ordering of variable/parameter updates to accelerate model-parallel ML algorithms such as Lasso regression. This type of schedule analyzes the dependency structure over model parameters A, in order to determine their best parallel execution order. 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.

Because scheduling functions `schedule()` may be compute-intensive, Petuum uses pipelining to overlap scheduling computations `schedule()` with worker execution, so workers are always doing useful computation. The scheduler is also responsible for central aggregation via the `pull()` function (corresponding to $F()$), if it is needed.

Workers: Each worker p receives parameters to be updated from `schedule()`, and then runs parallel update functions `push()` (corresponding to $\Delta()$) on data D. Petuum intentionally does not specify 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. While `push()` is being executed, the model state A is automatically synchronized with the parameter server via the parameter exchange channel, using a distributed shared memory programming interface that conveniently re-

```
// 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
}
```

Figure 4: Petuum Program Structure.

sembles single-machine programming. After the workers finish `push()`, the scheduler may use the new model state to generate future scheduling decisions.

Parameter Server: The parameter servers (PS) provide global access to model parameters A (distributed over many machines), via a convenient distributed shared memory API that is similar to table-based or key-value stores. To take advantage of ML-algorithmic principles, the PS implements Stale Synchronous Parallel (SSP) consistency [14, 7], which reduces network synchronization costs, while maintaining bounded-staleness convergence guarantees implied by SSP. We will discuss these guarantees in Section 5. Unlike PS-only systems that only support data-parallelism [19], Petuum's combined scheduler-and-PS design allows for both data- and model-parallel algorithms, which run asynchronously and enjoy provable speedup guarantees with more machines.

Fault tolerance is handled by checkpoint-and-restart, which is suitable for up to 100s of machines; a more sophisticated strategy for 1000s of machines is part of future work. To further improve network performance, Petuum can be configured to obey bandwidth limits and a logical network topology (e.g. ring, grid or fat-tree).

3.2 Programming Interface

Figure 4 shows a basic Petuum program, consisting of a central scheduler function `schedule()`, a parallel update function `push()`, and a central aggregation function `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 SSP consistency model automatically ensures parameter consistency between all Petuum components; no additional user programming is necessary. Finally, we use `DATA` to represent the data D; as noted earlier, this can be any 3rd-party data structure, database, or distributed file system.

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. We focus on a new data-parallel Distance Metric Learning algorithm, and a new model-parallel Lasso algorithm, but our strategies apply to a broad spectrum of other ML problems as briefly discussed at the end of this section. We show that with the Petuum system framework, we can easily realize these algorithms 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.

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)" [29] — 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 [8, 29] 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.

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

$$\begin{aligned} &\min_M \sum_{(x,y)\in\mathcal{S}} (x-y)^\mathsf{T} M(x-y) \\ &s.t.\ \ (x-y)^\mathsf{T} M(x-y) \geq 1, \forall (x,y) \in \mathcal{D}, \text{ and } M \succeq 0 \end{aligned} \tag{4}$$

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

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.(4), yielding

$$\begin{aligned} \min_L \quad & \sum_{(x,y)\in\mathcal{S}} \|L(x-y)\|^2 + \lambda \sum_{(x,y)\in\mathcal{D}} \xi_{x,y} \\ s.t. \quad & \|L(x-y)\|^2 \geq 1 - \xi_{x,y}, \xi_{x,y} \geq 0, \forall (x,y) \in \mathcal{D} \end{aligned} \tag{5}$$

Using hinge loss, the constraint in Eq.(5) can be eliminated, yielding an unconstrained optimization problem:

$$\min_L \sum_{(x,y)\in\mathcal{S}} \|L(x-y)\|^2 + \lambda \sum_{(x,y)\in\mathcal{D}} \max(0, 1 - \|L(x-y)\|^2) \tag{6}$$

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

```
// Data-Parallel Distance Metric Learning

schedule() { // Empty, do nothing }

push() {
  L_local = PS.get(L) // Bounded-async read from param server
  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 from Eq 7
  PS.inc(L,change/C)  // Add gradient to param server
}

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

Figure 5: Petuum DML data-parallel pseudocode.

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:

$$\begin{aligned} \triangle L_p &= \textstyle\sum_{(x,y)\in\mathcal{S}_p} 2L(x-y)(x-y)^\mathsf{T} \\ &- \textstyle\sum_{(a,b)\in\mathcal{D}_p} 2L(a-b)(a-b)^\mathsf{T} \cdot \mathbb{I}(\|L(a-b)\|^2 \leq 1) \end{aligned} \quad (7)$$

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

Figure 5 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. In Section 5, 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 5). 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 [11]. Lasso takes the form of an optimization problem:

$$\min_{\boldsymbol{\beta}} \ell(\mathbf{X}, \mathbf{y}, \boldsymbol{\beta}) + \lambda \textstyle\sum_j |\beta_j|, \quad (8)$$

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 (8) 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 [5]. We shall solve this via a coordinate descent (CD) model-parallel approach, similar but not identical to [5, 25].

The simplest parallel CD Lasso , shotgun [5], 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" (to be elaborated shortly), thus guaranteeing convergence of the Lasso objective. In addition, it prioritizes these parameters based on their distance to convergence, thus speeding up optimization.

```
// Model-Parallel Lasso

schedule() {
  for j=1..J     // Update priorities for all coeffs beta_j
    c_j = square(beta_j) + eta // Magnitude prioritization
  (s_1, ..., s_L') = random_draw(distribution(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)
}

push(p = worker_id(), (j_1, ..., j_L) = schedule() ) {
  // Partial computation for L chosen beta_j; calls PS.get(beta)
  (z_p[j_1], ..., z_p[j_L]) = partial(DATA[p], j_1, ..., j_L)
  return z_p
}

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

Figure 6: Petuum Lasso model-parallel pseudocode.

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 (8), we obtain the CD update for β_j:

$$\beta_j^{(t)} \leftarrow S(\mathbf{x}_j^T\mathbf{y} - \textstyle\sum_{k\neq j} \mathbf{x}_j^T\mathbf{x}_k\beta_k^{(t-1)}, \lambda), \quad (9)$$

where $S(\cdot, \lambda)$ is a soft-thresholding operator, defined by $S(\beta_j, \lambda) \equiv \text{sign}(\beta)(|\beta| - \lambda)$. In (9), if $\mathbf{x}_j^T\mathbf{x}_k \neq 0$ (nonzero correlation) and $\beta_j^{(t-1)} \neq 0$ and $\beta_k^{(t-1)} \neq 0$, then a coupling effect is created between 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 [20]. In Section 5, we 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 6.

4.3 Other Algorithms

We have implemented other data/model-parallel algorithms on Petuum — we briefly mention a few, while noting that many others are included in the Petuum open-source library.

Topic Model (LDA): For LDA, the key parameter is the "word-topic" table, that needs to be updated by all worker machines. We adopt a simultaneous data-and-model-parallel approach to LDA, and use a fixed schedule function `schedule_fix()` to cycle disjoint subsets of the word-topic table and data across machines for updating (via `push()` and `pull()`), without violating structural dependencies in LDA.

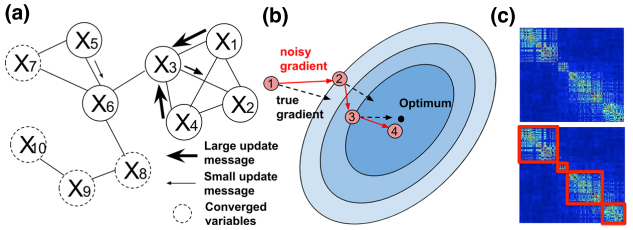

Figure 7: Key properties of ML algorithms: (a) Non-uniform convergence; (b) Error-tolerant convergence; (c) Dependency structures amongst variables.

Matrix Factorization (MF): High-rank decompositions of large matrices for improved accuracy [35] can be solved by a model-parallel approach, and we implement it via a fixed schedule function `schedule_fix()`, where each worker machine only performs the model update `push()` on a disjoint, unchanging subset of factor matrix rows.

Deep Learning (DL): We implemented two types on Petuum: a fully-connected Deep Neural Network (DNN) using cross-entropy loss, and a Convolutional NN (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. This data-parallel strategy may be amenable to MapReduce, Spark and GraphLab, though we are not aware of DL implementations on them.

5. PRINCIPLES AND THEORY

Our iterative-convergent formulation of ML programs, and the explicit notions of data and model parallelism, make it convenient to explore three key ML program properties — error-tolerant convergence, non-uniform convergence, dependency structures (Fig. 7) — 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 been successfully used in bespoke, large-scale implementations of popular ML algorithms: topic models [31, 19], matrix factorization [30, 16], and deep learning [17]. It is notable that MapReduce-style systems (such as Hadoop [27] and Spark [32]) often do not fare competitively against these custom-built ML implementations, and one of the reasons is that the key ML properties are difficult to harness under a MapReduce-like abstraction. Other abstractions may offer a limited degree of opportunity — for example, vertex programming [20] permits graph dependencies to influence model-parallel execution.

5.1 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 [22, 14, 7, 36, 1, 16]. Petuum exploits this error-tolerance to reduce network communication/synchronization overheads substantially, by implementing the Stale Synchronous Parallel (SSP) consistency model [14, 7] 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 in the DML program), SSP has very attractive theoretical properties [7], which we partially re-state here:

THEOREM 1 (ADAPTED FROM [7]). **SGD under SSP, convergence in probability:** *Let $f(\boldsymbol{x}) = \sum_{t=1}^{T} f_t(\boldsymbol{x})$ be a convex function, where the f_t are also convex. We search for a minimizer $\boldsymbol{x}^*$ via stochastic gradient descent on each component ∇f_t under SSP, with staleness parameter s and P workers. Let $\boldsymbol{u}_t := -\eta_t \nabla_t f_t(\tilde{\boldsymbol{x}}_t)$ 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 means that $\frac{R[X]}{T}$ converges to $O(T^{-1/2})$ in probability with an exponential tail-bound; convergence is faster when the observed staleness average μ_γ and variance σ_γ are smaller (SSP ensures $\mu_\gamma, \sigma_\gamma$ are as small as possible). Dai *et al.* also showed that the variance of $\mathbf{x}$ can be bounded, ensuring reliability and stability near an optimum [7].

5.2 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. Such dependencies have been thoroughly analyzed under fixed execution schedules (where each worker updates the same set of parameters every iteration) [25, 5, 24], 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 $\{j_1, j_2, \ldots\}$, telling worker p to update $A_{j_1}, A_{j_2}, \ldots$ — 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)

$$\min_{\mathbf{w} \in \mathbb{R}^d} f(\mathbf{w}) + r(\mathbf{w}), \tag{10}$$

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

$$f(\mathbf{w} + \mathbf{z}) \leq f(\mathbf{w}) + \mathbf{z}^\top \nabla f(\mathbf{w}) + \frac{\beta}{2} \mathbf{z}^\top X^\top X \mathbf{z}, \tag{11}$$

where $X = [\mathbf{x}_1, \ldots, \mathbf{x}_d]$ are d feature vectors. W.l.o.g., we assume that each feature vector $\mathbf{x}_i$ is normalized, i.e., $\|\mathbf{x}_i\|_2 = 1, i = 1, \ldots, d$. 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 (3), we can map the model $A = \mathbf{w}$, the data $D = X$, and the update equation $\Delta(A, S_p(A))$ to

$$w_{j_p}^{+} \leftarrow \arg\min_{z \in \mathbb{R}} \tfrac{\beta}{2}[z - (w_{j_p} - \tfrac{1}{\beta} g_{j_p})]^2 + r(z), \qquad (12)$$

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 [25, 5]. An effective and cheaply-computable schedule $S_{RRP,p}^{(t)}()$ involves randomly proposing a small set of $Q > P$ features $\{j_1, \ldots, j_Q\}$, 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* $\epsilon := \frac{d(\mathbb{E}P^2)(\rho-1)}{d^2(\mathbb{E}P-1)} \approx \frac{(\mathbb{E}P-1)(\rho-1)}{d} < 1$*, then after* t *steps, we have*

$$\mathbb{E}[F(\mathbf{w}^{(t)}) - F(\mathbf{w}^\star)] \leq \frac{Cd\beta}{\mathbb{E}[P(1-\epsilon)]} \frac{1}{t}, \qquad (13)$$

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 (e.g. consider a problem where all dimensions are correlated with each other). For most real-world data sets, this is not a problem, and* $\mathbb{E}P$ *is equal to the number of workers.*

For reference, the Petuum Lasso scheduler uses $S_{RRP}()$, augmented with a prioritizer we will describe soon.

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

$$E[|\mathbf{w}_{ideal}^{(t)} - \mathbf{w}_{RRP}^{(t)}|] \leq \frac{2dPm}{(t+1)^2 \hat{P}} L^2 X^\top X C. \qquad (14)$$

Proofs for both theorems are in an online supplement[3].

$S_{RRP}()$ is different from Scherrer *et al.* [25], who precluster 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 [11]. The proposed $S_{RRP}()$ only needs to evaluate a small number of $|\mathbf{x}_i^\top \mathbf{x}_j|$ every iteration, and we explain next, the random selection can be replaced with *prioritization* to exploit non-uniform convergence in ML problems.

[3] `http://petuum.github.io/papers/kdd15_supp.pdf`

5.3 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 [18]. For instance, this happens in Lasso 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 [18].

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 [34].

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 [5]: $F(\boldsymbol{\beta}) := \min_{\boldsymbol{\beta}} \frac{1}{2} \|\mathbf{y} - \mathbf{X}\boldsymbol{\beta}\|_2^2 + \lambda \sum_{j=1}^{2J} \beta_j$. Here, $\mathbf{X}$ contains $2J$ features and $\beta_j \geq 0$, for all $j = 1, \ldots, 2J$.

THEOREM 4 (ADAPTED FROM [18]). **Optimality of Lasso priority scheduler:** *Suppose* $\mathcal{B}$ *is the set of indices of coefficients updated in parallel at the* t*-th iteration, and* $\delta\beta_j^{(t)}$ *is the change in* β_j *from iteration* $t-1$ *to* t*. Let* ρ *be a 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 speeds up Lasso convergence by decreasing the objective as much as possible every iteration. For efficiency, the pipelined Petuum scheduler system approximates $p(j) \propto (\delta\beta_j^{(t)})^2$ with information from iteration $t-1$; we use $p'(j) \propto (\beta_j^{(t-1)})^2 + \eta$. This is necessary because $\delta\beta_j^{(t)}$ is available only after iteration t. Finally, the constant η serves as a prior, to ensure all β_j's have a non-zero probability of being updated.

6. 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 implementations of DML and Lasso achieve significantly faster convergence rate than baselines (i.e., DML implemented on single machine, and Shotgun [5]); (2) Petuum ML implementations can run faster than other platforms (e.g. Spark, GraphLab[4]), because Petuum can exploit model dependencies, uneven convergence and error tolerance; (3) Petuum ML implementations can reach larger model sizes than other platforms, because Petuum stores ML program variables in a lightweight

[4] We omit Hadoop, as it is well-established that Spark and GraphLab significantly outperform it [32, 20].

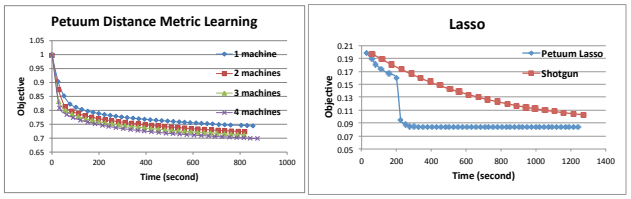

Figure 8: Left: Petuum DML convergence curve with different number of machines from 1 to 4. Right: Lasso convergence curve by Petumm Lasso and Shotgun.

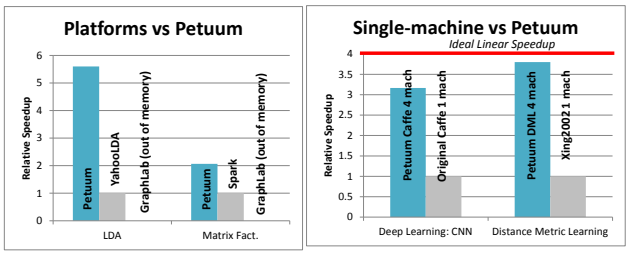

Figure 9: Left: Petuum performance: relative speedup vs popular platforms (larger is better). Across ML programs, Petuum is at least 2-10 times faster than popular implementations. Right: Petuum is a good platform for writing cluster versions of existing single-machine algorithms, achieving near-linear speedup with increasing number of machines (Caffe CNN and DML).

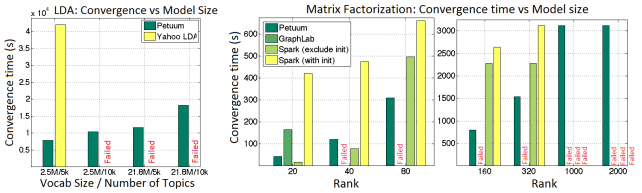

Figure 10: Left: LDA convergence time: Petuum vs YahooLDA (lower is better). Petuum's data+model-parallel LDA converges faster than YahooLDA (data-parallel-only), and scales to more parameters (larger vocab size, number of topics). Right panels: Matrix Factorization convergence time: Petuum vs GraphLab 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.

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; Petuum currently lacks features that are necessary for clusters with ≥ 1000 machines, such as automatic recovery from machine failure. Our experiments are therefore focused on clusters with 10-100 machines, in accordance with our target users.

Experimental settings We used 3 clusters with varying specifications, to show Petuum's adaptability to different hardware: "Cluster-1" machines have 2 AMD cores, 8GB RAM, 1Gbps Ethernet; "Cluster-2" machines have 64 AMD cores, 128GB RAM, 40Gbps Infiniband; "Cluster-3" machines have 16 Intel cores, 128GB RAM, 10Gbps Ethernet.

LDA was run on 128 Cluster-1 nodes, using 3.9m English Wikipedia abstracts with unigram ($V = 2.5$m) and bigram ($V = 21.8$m) vocabularies. MF and Lasso were run on 10 Cluster-2 nodes, respectively using the Netflix data and a synthetic Lasso dataset with $N = 50$k samples and 100m features/parameters. CNN was run on 4 Cluster-3 nodes, using a 250k subset of Imagenet with 200 classes, and 1.3m model parameters. The DML experiment was run on 4 Cluster-2 nodes, using the 1-million-sample Imagenet [10] dataset with 1000 classes (220m model parameters), and 200m similar/dissimilar statements. In all experiments, we sharded the data over local hard disks.

Performance of Distance Metric Learning and Lasso We investigate Petuum's DML and Lasso performance: Figure 8 shows the convergence of Petuum and baselines, using a fixed model size (21504×1000 distance matrix for DML; 100M features for Lasso). For DML, increasing the machine count consistently increases the convergence speed. Petuum DML achieves 3.8 times speedup with 4 machines and 1.9 times speedup with 2 machines, with potential to continue scaling well with more machines. For Lasso, given the same number of machines, Petuum achieved a significantly faster convergence rate than Shotgun (which randomly selects a subset of parameters to be updated). In the initial stage, Petuum lasso and Shotgun show similar convergence rates because Petuum updates every parameter in the first iteration to "bootstrap" the scheduler (at least one iteration is required to initialize all parameters). After this initial stage, Petuum dramatically decreases the Lasso objective compared to Shotgun, by taking advantage of dependency structures and non-uniform convergence via the scheduler.

Platform Comparison Figure 9 (left) compares Petuum to popular ML platforms (Spark and GraphLab) and well-known cluster implementations (YahooLDA [2]). For two common ML programs of LDA and MF, we show the relative speedup of Petuum over the other platforms' implementations. In general, Petuum is between 2-6 times faster than other platforms; the differences help to illustrate the various data/model-parallel features in Petuum. 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. On the other hand, Petuum LDA is nearly 6 times faster than YahooLDA; the speedup mostly comes from scheduling $S()$, which enables correct, dependency-aware model-parallel execution.

Scaling to Larger Models We show that Petuum supports larger ML models for the same amount of cluster memory. Figure 10 shows 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[5], 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 is the result of two factors: (1) model-parallelism, which divides the model across machines; (2) a lightweight parameter server system with minimal storage overhead (from using simple arrays and hashmaps). Compared to Petuum, we observed that GraphLab's vertex representation and Spark's RDD representation consumed $\sim$ 10x and $\sim$ 2-3x memory (respectively) to store model variables (Figure 10).

Fast Cluster Implementations of New ML Programs We show that Petuum facilitates the development of new ML programs without existing cluster implementations. In

[5]LDA model size equals vocab size times number of topics.

Figure 9 (right), we present two instances: first, a cluster version of the open-source Caffe CNN toolkit, created by adding $\sim$ 600 lines of Petuum code. The basic data-parallel strategy was left unchanged, so the Petuum port directly tests Petuum's efficiency. Compared to the original single-machine Caffe *with no network communication*, Petuum achieves approaching-linear speedup (3.1-times speedup on 4 machines) due to the parameter server's SSP consistency for managing network communication. Second, we compare the Petuum DML program against the original DML algorithm proposed in [29] (denoted by Xing2002), which is optimized with SGD on a single-machine (with parallelization over matrix operations). The intent is to show that a fairly simple data-parallel SGD implementation of DML (the Petuum program) can greatly speed up execution over a cluster. The Petuum implementation converges 3.8 times faster than Xing2002 on 4 machines — this provides evidence that Petuum enables data/model-parallel algorithms to be efficiently implemented over clusters.

7. CONCLUSION

Petuum is a system for writing data- and model-parallel iterative-convergent ML programs, which takes advantage of their properties to speed up convergence: limited error tolerance, dependency structures between parameters, and non-uniform parameter convergence. We have shown promising results on 100+ machines, though several important practical issues remain to be addressed: fault tolerance strategies for 1000s of machines, a standard interface for accessing input data, integration with the Hadoop/YARN ecosystem, and out-of-core model storage for memory-limited situations. We are investigating these as future work.

8. ACKNOWLEDGMENTS

We thank Garth Gibson, Greg Ganger, Phillip Gibbons, and members of the Parallel Data Lab and Intel Science and Technology Center for Cloud Computing, for the insights and discussions during this project. This work is supported by the following grants to Eric P. Xing: NSF IIS 1447676, 1111142, 1218282, 1115313; NIH R01GM087694, P30DA035778, R01GM093156.

9. REFERENCES

[1] A. Agarwal and J. C. Duchi. Distributed delayed stochastic optimization. In *NIPS*, 2011.

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

[3] L. Bottou. Large-scale machine learning with stochastic gradient descent. In *Proceedings of COMPSTAT'2010*, pages 177–186. Springer, 2010.

[4] S. Boyd, N. Parikh, E. Chu, B. Peleato, and J. Eckstein. Distributed optimization and statistical learning via the alternating direction method of multipliers. *Foundations and Trends in Machine Learning*, 3:1–124, 2011.

[5] J. K. Bradley, A. Kyrola, D. Bickson, and C. Guestrin. Parallel coordinate descent for l1-regularized loss minimization. In *ICML*, 2011.

[6] X. Chen, Q. Lin, S. Kim, J. Carbonell, and E. Xing. Smoothing proximal gradient method for general structured sparse learning. In *UAI*, 2011.

[7] 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.

[8] 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*, pages 209–216. ACM, 2007.

[9] 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.

[10] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. Imagenet: A large-scale hierarchical image database. In *CVPR*, 2009.

[11] H. B. M. et. al. Ad click prediction: a view from the trenches. In *KDD*, 2013.

[12] J. Friedman, T. Hastie, H. Höfling, and R. Tibshirani. Pathwise coordinate optimization. *Annals of Applied Statistics*, 1(2):302–332, 2007.

[13] T. L. Griffiths and M. Steyvers. Finding scientific topics. *PNAS*, 101(Suppl 1):5228–5235, 2004.

[14] 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.

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

[16] A. Kumar, A. Beutel, Q. Ho, and E. P. Xing. Fugue: Slow-worker-agnostic distributed learning for big models on big data. In *AISTATS*, 2014.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] 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*, 2010.

[22] F. Niu, B. Recht, C. Ré, and S. J. Wright. Hogwild!: A lock-free approach to parallelizing stochastic gradient descent. In *NIPS*, 2011.

[23] R. Power and J. Li. Piccolo: building fast, distributed programs with partitioned tables. In *OSDI*. USENIX Association, 2010.

[24] P. Richtárik and M. Takáč. Parallel coordinate descent methods for big data optimization. *arXiv:1212.0873*, 2012.

[25] C. Scherrer, A. Tewari, M. Halappanavar, and D. Haglin. Feature clustering for accelerating parallel coordinate descent. *NIPS*, 2012.

[26] 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:1405.4402*, 2014.

[27] T. White. *Hadoop: The definitive guide*. O'Reilly Media, Inc., 2012.

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

[29] E. P. Xing, M. I. Jordan, S. Russell, and A. Y. Ng. Distance metric learning with application to clustering with side-information. In *NIPS*, 2002.

[30] H.-F. Yu, C.-J. Hsieh, S. Si, and I. Dhillon. Scalable coordinate descent approaches to parallel matrix factorization for recommender systems. In *ICDM*, 2012.

[31] 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 *WWW*. 2015.

[32] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: cluster computing with working sets. In *HotCloud*, 2010.

[33] Y. Zhang, Q. Gao, L. Gao, and C. Wang. Priter: A distributed framework for prioritized iterative computations. In *SOCC*, 2011.

[34] Y. Zhang, Q. Gao, L. Gao, and C. Wang. Priter: A distributed framework for prioritizing iterative computations. *IEEE Transactions on Parallel and Distributed Systems*, 24(9):1884–1893, 2013.

[35] 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.

[36] M. Zinkevich, J. Langford, and A. J. Smola. Slow learners are fast. In *NIPS*, 2009.