STRADS: A Distributed Framework for Scheduled Model Parallel Machine Learning

Jin Kyu Kim[1]   Qirong Ho[2]   Seunghak Lee[1]   Xun Zheng[1]   Wei Dai[1]
Garth A. Gibson[1]   Eric P. Xing[1]

[1] School of Computer Science, Carnegie Mellon University, USA
[2] Institute of Infocomm Research, A*STAR, Singapore

Abstract

Machine learning (ML) algorithms are commonly applied to big data, using distributed systems that partition the data across machines and allow each machine to read and update all ML model parameters — a strategy known as data parallelism. An alternative and complimentary strategy, model parallelism, partitions the model parameters for non-shared parallel access and updates, and may periodically repartition the parameters to facilitate communication. Model parallelism is motivated by two challenges that data-parallelism does not usually address: (1) parameters may be dependent, thus naive concurrent updates can introduce errors that slow convergence or even cause algorithm failure; (2) model parameters converge at different rates, thus a small subset of parameters can bottleneck ML algorithm completion. We propose scheduled model parallelism (SchMP), a programming approach that improves ML algorithm convergence speed by efficiently scheduling parameter updates, taking into account parameter dependencies and uneven convergence. To support SchMP at scale, we develop a distributed framework STRADS which optimizes the throughput of SchMP programs, and benchmark four common ML applications written as SchMP programs: LDA topic modeling, matrix factorization, sparse least-squares (Lasso) regression and sparse logistic regression. By improving ML progress per iteration through SchMP programming whilst improving iteration throughput through STRADS we show that SchMP programs running on STRADS outperform non-model-parallel ML implementations: for example, SchMP LDA and SchMP Lasso respectively achieve 10x and 5x faster convergence than recent, well-established baselines.

1. Introduction

A variety of machine learning (ML) algorithms are used to explore and exploit big data originating from diverse sources such as activities on the internet, profiling and transactions in finance, advertisement and healthcare, or experiments in the physical sciences, to name a few. The demand for distributed ML programs that run over a cluster is driven by two trends: (1) **big data:** a single machine's computational power is inadequate for running ML algorithms on big datasets in a reasonable amount of time; (2) **large models:** even if a dataset size is not increased, ML application running times increase when their underlying ML model is given more parameters — which recent ML applications have been striving towards (e.g., at least hundreds of millions of parameters [19]).

The trend towards richer, larger ML models with more parameters (up to tens of billions [21]) is driven by the need for more "explanatory power": it has been observed that big datasets contain "longer tails" (rare-yet-unique events) than smaller datasets, and detection of such events can be crucial to downstream tasks [34, 37]. Many of these *big models* are extremely slow to converge when trained with a sequential algorithm, thereby motivating *model parallelism* [9, 20, 22, 37] — which as the name suggests, splits ML model parameters across machines, and makes each machine responsible for updating only its assigned portion of parameters (either using the full data, or a data subset). Even when the model is relatively small, model parallel execution can still mean the difference between hours or days of compute on a single machine, versus minutes on a cluster [5, 33].

Model parallelism can be contrasted with *data parallelism*, where each machine gets one partition of the data, and iteratively generates sub-updates that are applied to *all* ML model parameters (which are accessible with read/write operations from all machines; that is, distributed shared memory or replicated memory), until convergence is reached. This is possible because most ML algorithms adopt the so-called *i.i.d.* assumption on data — that is, under a given estimate of the model parameters, the data instances (or subsets of data instances) are *independent and identi-*

cally distributed. As a result, all machines' sub-updates can be easily aggregated. This convenient property does not apply to model parallel algorithms, which introduce new subtleties: (1) the model parameters are *not independent*, and (2) different model parameters may take different numbers of iterations to converge (i.e., *uneven convergence*). Hence, the effectiveness of a model-parallel algorithm is greatly affected by its *schedule* – which parameters are updated in parallel, and how they are prioritized [22, 23, 41].

Unlike conventional computer programs, whose progress can be measured by *throughput* (operations per unit time), the *progress* of an ML program is measured by a numerically explicit *objective function* specified by the ML application. More progress is made when the objective increases (or decreases) to approach an optimum at a faster rate. Poorly-chosen schedules inhibit progress made during an ML algorithm iteration, or may even cause the algorithm to fail (i.e., the objective function never approaches an optimum value). It is important to note that progress per iteration is distinct from iteration throughput (number of iterations executed per unit time); effective ML implementations combine high progress per iteration with high iteration throughput, yielding high progress per unit time. Despite these challenges, model-parallel algorithms have shown promising speedups over their data-parallel counterparts [34, 37].

In the ML literature, there is a strong focus on verifying the safety or correctness of parallel algorithms via statistical theory [5, 26, 42], but this is often done under simple assumptions about distributed environments — for example, network communication and synchronization costs are often ignored. On the other hand, the systems literature is focused on developing distributed systems [10, 21, 25, 38], with high-level programming interfaces that allow ML developers to focus on the ML algorithm's core routines. Some of these systems enjoy strong fault tolerance and synchronization guarantees that ensure correct ML execution [10, 38], while others [17, 21] exploit the error-tolerance of data-parallel ML algorithms, and employ relaxed synchronization guarantees in exchange for higher iterations-per-second throughput. These systems do not directly support fine-grained control over parameter updates that takes into account parameter dependencies and uneven convergence, which are vital for algorithm correctness and speed.

In an earlier work [20], we showed that specific model parallel strategies could be created for three well-established ML algorithms, and proposed programming primitives to represent the key operations of each model parallel strategy — however, we did not explain how model parallel strategies could be developed for general ML applications, and the system design used in the earlier work lacked critical system optimizations for achieving high throughput. We address these issues in this work by describing general principles for developing new model parallel ML applications beyond those previously addressed; furthermore, we identify

and propose system optimizations for two general categories of model parallel ML algorithms, show how several ML applications can be rewritten as instances of these categories, and present a thorough evaluation that dissects application performance into an ML metric, *progress per iteration* and a system metric, *iteration throughput*. We refer to the two categories as "static schedules" or "dynamic schedules", depending on whether their schedule depends on the dynamic values of the model parameters being trained. Our set of systems optimizations significantly improve the performance of static or dynamic schedule algorithms: (1) pipelined computation of parameter updates and schedule functions, (2) load balancing of parameter updates (which become an uneven workload as a result of accounting for parameter dependencies), (3) using a ring topology to flow parameters between machines in lockstep. The overall system design finds a "sweet spot" between algorithmic progress per iteration and iteration throughput, resulting in high progress per unit time and faster algorithm completion.

Our approach addresses the following challenges for existing systems: (1) limited appreciation for model parallel update execution order and how it can accelerate model parallel algorithms; (2) limited systems support for model parallel algorithms; (3) limited understanding of the safety and correctness of model parallel algorithms under realistic systems conditions. Specifically, this paper proposes **scheduled model parallelism (SchMP)**, where an ML application scheduler generates model parallel *schedules* that improve model parallel algorithm progress per update, by considering dependency structures and prioritizing parameters. SchMP allows model parallel algorithms to be separated into (1) a *control component* responsible for dependency checking and prioritization, and (2) an *update component* that executes iterative ML updates in the parallel schedule prescribed by the control component.

To realize SchMP, we develop **a SchMP framework called STRADS** in about 10,000 lines of C++ that parallelizes SchMP ML applications over a cluster. Even as SchMP applications innately enjoy high progress per iteration, STRADS improves the number of iterations executed per second by (1) pipelining SchMP iterations, (2) overlapping SchMP computation with parameter synchronization over the network, and (3) streaming computations around a ring topology. Through SchMP on STRADS, we achieve high performance parallel ML; that is, increased progress per iteration and increased iterations per second. The result is substantially increased *progress per second*, and therefore faster ML algorithm completion. We benchmark various SchMP algorithms implemented on STRADS — Gibbs sampling for topic modeling [4, 16, 37], stochastic gradient descent for matrix factorization [13], and coordinate descent for sparse linear (i.e., Lasso [11, 31]) and logistic [12] regressions — and show that SchMP programs on STRADS outperform non-model parallel ML implementations.

2. Model Parallelism

Although machine learning problems exhibit a diverse spectrum of model forms, their computer program implementations typically take the form of an iterative convergent procedure, meaning that they are optimization or Markov Chain Monte Carlo (MCMC [32]) algorithms that repeat some set of fixed-point update routines until convergence (i.e. a stopping criterion has been reached):

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

where index t refers to the current iteration, A the model parameters, D the input data, and $\Delta()$ the model update function[1]. Such iterative-convergent algorithms have special properties that we shall explore: tolerance to numeric errors in model parameters during iteration, dependency structures that must be respected during parallelism, and uneven convergence across model parameters.

In *model parallel* ML programs, parallel workers recursively update subsets of model parameters until convergence, refining Eq. (1) to the following form:

$$A^{(t)} = A^{(t-1)} + \sum_{p=1}^{P} \Delta_p(D, A^{(t-1)}, S_p(D, A^{(t-1)})),$$

where $\Delta_p()$ is the model update function executed at parallel worker p. The "schedule" $S_p()$ identifies a subset of parameters in A, instructing the p-th parallel worker which parameters it should work on *sequentially* (i.e. workers may not further parallelize within $S_p()$). Since the data D is unchanging, we drop it from the notation for clarity:

$$A^{(t)} = A^{(t-1)} + \sum_{p=1}^{P} \Delta_p(A^{(t-1)}, S_p(A^{(t-1)})). \qquad (2)$$

Our goal is to develop re-usable strategies for model parameter scheduling, based on an understanding of ML mathematical principles. To this end, Eq.(2) presents a generic form for ML algorithms, in which an application-specific update function Δ and a schedule function S can be plugged in. Because ML applications depend on a small number of "workhorse" algorithms (e.g. gradient descent, coordinate descent, and Gibbs sampling, to name just a few), and because these workhorses exhibit the additive model updates seen in Eq. 2, SchMP is thus generally applicable to many ML algorithms. SchMP can either reproduce the behavior of existing ML implementations (for example, by using a basic schedule function S that iterate sequentially across all parameters in A), or improve upon them by developing model parallel schedules for ML application instances and system optimizations that apply to such instances.

2.1 Properties of ML Algorithms

Intrinsic properties of ML algorithms (i.e., Eq. (1)) provide powerful opportunities for and limitations on effective parallelism. (1) **Model Dependencies**: the elements of A (the parameters) are not necessarily independent of each other,

and updates to one element may strongly affect later updates to other parameters [14, 20, 22]. (2) **Uneven Convergence**: different model parameters may converge at different rates, leading to new speedup opportunities via parameter prioritization [22, 41]. Finally, (3) **Error-Tolerant**: a limited amount of stochastic error during computation of $\Delta(D, A^{t-1})$ in each iteration does not lead to algorithm failure (though it might slow down convergence speed) [5, 17, 21].

Sometimes, it is not practical or possible to find a "perfect" parallel execution scheme for an ML algorithm, which means that some dependencies will be violated, leading to incorrect update operations. But, unlike classical computer science algorithms where incorrect operations usually lead to failure, iterative-convergent ML programs (which can also be viewed as "fixed-point iteration" algorithms) can self-correct under a limited amount of incorrect updates or other errors (though at the cost of potentially slower convergence). There remains a strong incentive to minimize errors: the more dependencies the system finds and avoids, the more progress the ML algorithm will make each iteration — however, searching for and serializing those dependencies may incur non-trivial computational costs and delays, reducing iteration throughput. Because an ML program's convergence speed is essentially progress per iteration multiplied by iteration throughput, it is important to balance these two considerations. Below, we explore this idea by explicitly discussing variations within model parallelism, in order to expose possible ways by which model parallelization can be made efficient.

2.2 Variations of Model Parallelism

We restrict our attention to model parallel programs that partition M model parameters across P worker threads in an approximately load-balanced manner. Here, we introduce variations on model parallelism, which differ on their partitioning quality. Concretely, partitioning involves constructing a size-M^2 dependency graph, with weighted edges e_{ij} that measure the dependency between parameters A_i and A_j. This measure of dependency differs from algorithm to algorithm: e.g., in Lasso regression e_{ij} is the correlation between the i-th and j-th data dimensions. The total violation of a partitioning is the sum of weights of edges that cross between the P partitions, and we wish to minimize this.

Ideal Model Parallel: Theoretically, there exists an "ideal" load-balanced parallelization over P workers which gives the highest possible progress per iteration; this is indicated by an ideal (but might be computationally intractable) schedule $S_p^{ideal}()$ that replaces the generic $S_p()$ in Eq. (2). There are two points to note: (1) even this "ideal" model parallelization can still violate model dependencies and incur errors (compared to sequential execution) because of cross-worker coupling; (2) computing $S_p^{ideal}()$ is expensive in general because graph-partitioning is NP-hard. The quality of S^{ideal} ultimately depends on the potential for parallelism

---

[1] The summation between $\Delta()$ and $A^{(t-1)}$ can be generalized to a general aggregation function $F(A^{(t-1)}, \Delta())$; for concreteness we restrict our attention to the summation form, but the techniques proposed in this paper can be applied to F.

in the ML problem. If the potential parallelism is less than the degree of parallelism P allowed by the available computing resources, then the ideal schedule on P workers will inevitably break dependencies. Ideal model parallelization achieves the highest progress per iteration amongst load-balanced model parallel programs, but may incur a large one-time or even every-iteration partitioning cost, which can greatly reduce iteration throughput.

Random Model Parallel: At the other extreme is random model parallelization, in which a schedule $S_p^{rand}()$ simply chooses one parameter at random for each worker p [5]. As the number of workers P increases, the expected number of violated dependencies will also increase, leading to poor progress per iteration (or even algorithm failure). However, there is practically no cost to iteration throughput.

Approximate Model Parallel: As a middle ground between ideal and random model parallelization, we may *approximate* $S_p^{ideal}()$ via a cheap-to-compute schedule $S_p^{approx}()$. A number of strategies exist: one may partition small subsets of parameters at a time (instead of the M^2-size full dependency graph), or apply approximate partitioning algorithms [29] such as METIS [18] (to avoid NP-hard partitioning costs), or even use strategies that are unique to a particular ML program's structure.

In this paper, we explore strategies for efficient and effective approximate model parallelization:

Static Partitioning: A fixed, static schedule $S_p^{fix}()$, based on ML application domain knowledge, hard-codes the partitioning for every iteration beforehand. Progress per iteration varies depending on how well $S_p^{fix}()$ matches the ML program's dependencies, but like random model parallel, this has little cost to iteration throughput.

Dynamic Partitioning: Dynamic partitioning $S_p^{dyn}()$ tries to select independent parameters for each worker, by performing pair-wise dependency tests between a small number L of parameters (which can be chosen differently at different iterations, based on a priority policy as discussed in Section 4.1); the GraphLab system achieves a similar outcome via graph consistency models [22]. The idea is to only do L^2 computational work per iteration, which is far less than M^2 (where M is the total number of parameters), based on a priority policy that selects the L parameters that matter most to the program's convergence. Dynamic partitioning can achieve high progress per iteration, similar to ideal model parallelism, but may suffer from poor iteration throughput on distributed clusters: because only a small number of parameters are updated each iteration, the time spent computing $\Delta_p()$ at the P workers may not amortize network latencies and the cost of computing $S_p^{dyn}()$.

Pipelining: This is not a different type of model parallelism per se, but a complementary technique that can be applied to any model parallel strategy. Pipelining allows the next iteration(s) to start before the current one finishes, ensuring that computation is always fully utilized; however,

this introduces *staleness* into the model parallel execution:

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

Note how the model parameters $A^{(t-s)}$ being used for $\Delta_p(), S_p()$ come from iteration $(t - s)$, where s is the pipeline depth. In system parlance, we intentionally allow such data-hazards in exchange for higher iteration throughput. Unlike traditional programs, where data-hazards lead to whole program failure, ML applications can still converge under stale model images (up to a practical limit) [8, 17] because ML applications are intrinsicially error-tolerant. Pipelining therefore sacrifices some progress per iteration to increase iteration throughput, and is a good way to raise the throughput of dynamic partitioning. In [20], we proposed that scheduling operations S_p could be pipelined to hide scheduling overhead, but did not provide a detailed system design, and also did not consider pipelining the updates to model parameters Δ_p, which make up the bulk of an ML algorithm's execution time. In Section 3.3.2, we present a system design that pipelines both scheduling S_p and update operations Δ_p based on stale execution (Eq. 3) — in essence, a system optimization that overlaps communication with update computation.

Prioritization: Like pipelining, prioritization is complementary to model parallel strategies. The idea is to modify $S_p()$ to prefer parameters that, when updated, will yield the most convergence progress [22], while avoiding parameters that are already converged [21]; this is effective because ML algorithms exhibit uneven parameter convergence. Since computing a parameter's potential progress can be expensive, we may employ cheap-but-effective approximations or heuristics to estimate the potential progress (as shown in Section 4.1). Prioritization can thus greatly improve progress per iteration, at a small cost to iteration throughput.

2.3 Scheduled Model Parallelism for Programming

Model parallelism accommodates a wide range of partitioning and prioritization strategies (i.e. the schedule $S_p()$), from simple random selection to complex, dependency-calculating functions that can be more expensive than the updates $\Delta_p()$. In existing ML program implementations, the schedule is often written as part of the update logic, ranging from simple for-loops that sweep over all parameters one at a time, to sophisticated systems such as GraphLab [22], which "activates" a parameter whenever one of its neighboring parameters changes. We contrast this with **scheduled model parallelism** (SchMP), in which the schedule $S_p()$ computation is explicitly separated from update $\Delta_p()$ computation. The rationale behind SchMP is that the schedule can be a distinct object for systematic investigation, separate from the updates, and that a model parallel ML program can be improved by simply changing $S_p()$ without altering $\Delta_p()$.

In order to realize SchMP programming, we have developed a framework called STRADS, that exposes param-
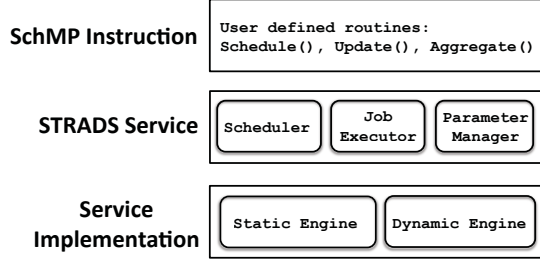
Figure 1: STRADS: To create an SchMP program, the user codes the SchMP Instructions, similar to MapReduce. The Services are system components that execute SchMP Instructions over a cluster. We provide two Implementations of the Services: a Static Engine and a Dynamic Engine, specialized for high performance on static-schedule and dynamic-schedule SchMP programs respectively. The user chooses which engine (s)he would like to use.

eter schedules $S_p()$ and parameter updates $\Delta_p()$ as separate functions for the user to implement (analogous to how MapReduce requires the user to implement Map and Reduce). This separation allows generic optimizations to be applied and enjoyed by many model parallel programs: e.g., our STRADS implementation performs automatic pipelining for dynamic model parallelism, and uses a ring communication topology for static model parallelism; we believe further yet-unexplored optimizations are possible.

3. System Implementation

STRADS is a system to execute SchMP programs, in which low-level machine/traffic coordination issues are abstracted away. The goal is to improve ML convergence speed in two ways: (1) users can easily experiment with new model parallel schedules for ML programs, using the aforementioned techniques to improve ML algorithm *convergence per iteration*; (2) the STRADS provides systems optimizations such as pipelining to automatically increase the *iteration throughput* of SchMP programs.

Conceptually, STRADS is divided into three parts (Figure 1): (1) SchMP Instructions (`schedule()`, `update()`, `aggregate()`), which the *user implements* to create an SchMP program; (2) Services, which execute SchMP Instructions over a cluster (**Scheduler**, **Job Executors**, **Parameter Manager**); (3) Implementations of the Services, specialized for high performance on different types of SchMP programs (Static Engine and Dynamic Engine). The ML programmer writes code within the SchMP instruction layer, while lower-level system considerations, such as inter-machine communication and job execution, are managed automatically by the STRADS system.

3.1 User-implemented SchMP Instructions

Table 1 shows the three SchMP Instructions, which are abstract functions that a user implements in order to create an SchMP program. All SchMP programs are iterative, where each iteration begins with `schedule()`, followed by parallel instances of `update()`, and ending with

`aggregate()`; Figure 1 shows the general form of an SchMP program.

3.2 STRADS Services

STRADS executes SchMP Instructions across a cluster via three Services: the **Scheduler**, **Job Executors**, and the **Parameter Manager**. The **Scheduler** is responsible for computing `schedule()` and passing the output jobs $\{\mathbf{S}_p\}$ on; most SchMP programs only require one machine to run the **Scheduler**, others may benefit from parallelization and pipelining over multiple machines. The **Scheduler** can keep local program state between iterations (e.g. counter variables or cached computations).

The P jobs $\{\mathbf{S}_p\}$ are distributed to P **Job Executors**, which start worker processes to run `update()`. On non-distributed file systems, the **Job Executors** must place worker processes exactly on machines with the data. Global access to model variables A is provided by the **Parameter Manager**, so the **Job Executors** do not need to consider model placement. Like the **Scheduler**, the **Job Executors** may keep local program state between iterations.

Once the worker processes finish `update()` and generate their intermediate results R_p, the aggregator process on **scheduler** (1) performs `aggregate()` on $\{R_p\}$, and (2) commit model updates and thus reach the next state $A_p^{(t+1)}$. Control is then passed back to the **Scheduler** for the next iteration $(t+1)$. Finally, the **Parameter Manager** supports the **Scheduler** and **Job Executors** by providing global access to model parameters A. The Static Engine and Dynamic Engine implement the **Parameter Manager** differently.

3.3 Service Implementations (Engines)

Many ML algorithms use a "static" schedule, where the order of parameter updates is known or fixed in advance (e.g. Matrix Factorization and Topic Modeling). One may also write "dynamic" schedules that change in response to the model parameters, and which can outperform static-schedule equivalents — our SchMP-Lasso program is one example. These two classes of schedules pose different systems requirements; static `schedule()` functions tend to be computationally light, while dynamic `schedule()` functions are computationally intensive.

Static-schedule algorithms usually generate jobs $\mathbf{S}_p$ with many parameters; it is not uncommon to cover the whole parameter space A in a single SchMP iteration, and communication of parameters A across the network can easily become a bottleneck. On the other hand, dynamic-schedule algorithms prefer to create small parameter update jobs $\mathbf{S}_p$, which not only reduces the computational bottleneck at the scheduler, but also allows the ML algorithm to quickly react to and exploit uneven parameter convergence. However, this makes the SchMP iterations very short, and therefore latency (from both scheduler computation and network communication) becomes a major issue.

SchMP Function	Purpose	Available Inputs	Output
schedule()	Select parameters A to update	model A, data D	P parameter jobs $\{\mathbf{S}_p\}$
update()	Model parallel update equation	one parameter job $\mathbf{S}_p$, local data D_p, parameters A	one intermediate result R_p
aggregate()	Collect R_p and update model A	P intermediate results $\{R_p\}$, model A	new model state $A_p^{(t+1)}$

Table 1: **SchMP Instructions**. To create an SchMP program, the user implements these Instructions. The available inputs are optional — e.g. schedule() does not necessarily have to read A, D (such as in static partitioning).
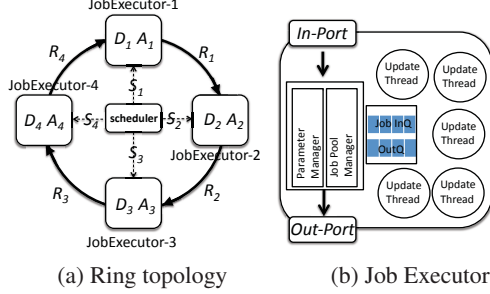

(a) Ring topology   (b) Job Executor

Figure 2: **Static Engine:** (a) Parameters A_p and intermediate results R_p are exchanged over a ring topology. (b) **Job Executor** architecture: the **Parameter Manager** and a job pool manager receive and dispatch jobs to executor threads; results R_p are immediately forwarded without waiting for other jobs to complete.

(a) Non-pipelined   (b) Pipelined

Figure 3: **Dynamic Engine** pipelining: (a) Non-pipelined execution: network latency dominates; (b) Pipelining overlaps networking and computation.

Because static- and dynamic-schedule SchMP algorithms have different needs, we provide two distinct but complete Implementations ("engines") of the three Services: a Static Engine specialized for high performance on static-schedule algorithms, and a Dynamic Engine specialized for dynamic-schedule algorithms. For a given ML program, the choice of Engine is primarily driven by domain knowledge — e.g. it is known that coordinate descent-based regressions benefit greatly from dynamic schedules [20, 29]. Once the user has chosen an Engine, STRADS provides default schedule() implementations appropriate for that Engine (described in the next Section) that can used as-is. These defaults cover a range of ML programs, from regressions through topic models and matrix factorization.

3.3.1 Static Engine

In static-schedule algorithms, every iteration reads/writes to many parameters, causing bursty network communication. To avoid network hot spots and balance communication, the Static Engine's **Parameter Manager** connects **Job Executors** into a logical ring (Figure 2), used to transfer parameters A_p and intermediate results R_p.

The **Job Executors** forward received parameters A_p and results R_p to their next ring neighbor, making local copies of needed A_p, R_p as they pass by. Once A_p, R_p return to their originator on the ring, they are removed from circulation. The Static Engine uses a straightforward, single-threaded implementation for its **Scheduler** (because static schedule()s are not computationally demanding).

3.3.2 Dynamic Engine

Dynamic-schedule algorithms have short iterations, hence computation time by **Job Executors** is often insufficient to amortize away network communication time (Figure 3a). To address this, the Dynamic Engine uses pipelining (Figure 3b) to overlap communication and computation; the **Scheduler**
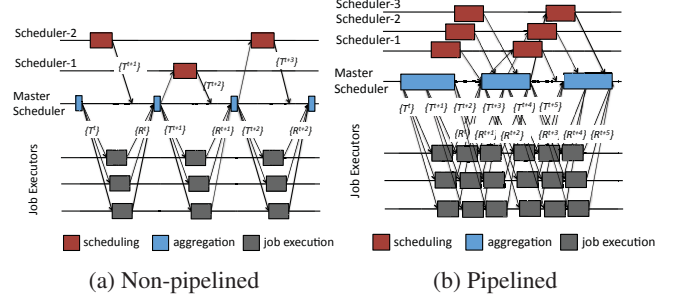
will start additional iterations before waiting for the previous one to finish. The pipeline depth (number of in-flight iterations) can be set by the user. When the scheduling latency is higher than the update latency, scheduling may become a bottleneck to iteration throughput. To address this problem, the scheduling can be distributed over multiple scheduler instances as shown in Figure 3a, 3b by running an individual scheduler instance on a partition of model parameters.

Although pipelining improves iteration throughput and overall convergence speed, it may lower progress per iteration due to (1) using the old model state $A^{(t)}$ instead of new updates $A^{(t+s)}$, and (2) dependencies between pipelined iterations due to overlapping of update jobs $\mathbf{S}_p$. This does not lead to ML program failure because ML algorithms can tolerate some error and still converge — albeit more slowly. Pipelining is basically execution with stale parameters, $A^{(t)} = F(A^{(t-s)}, \{\Delta_p(A^{(t-s)}, S_p(A^{(t-s)}))\}_{p=1}^{P})$ where s is the pipeline depth.

3.4 Other Considerations

Fault tolerance: STRADS execution can be made fault-tolerant, by checkpointing the model parameters A every x iterations. Because ML programs are error-tolerant, background checkpointing (which may span several iterations), is typically sufficient.

Avoiding lock contention: To avoid lock contention, the STRADS **Scheduler** and **Job Executors** avoid sharing data structures between threads in the same process. For example, when jobs $\mathbf{S}_p$ are being assigned by a **Job Executor** process to individual worker threads, we use a separate, dedicated queue for each worker thread.

Dynamic Engine parameter reordering: Within each Dynamic Engine iteration, STRADS re-orders the highest priority parameters to the front of the iteration, which improves the performance of pipelining. The intuition is as follows: because high-priority parameters have a larger effect

Algorithm 1 Generic SchMP ML program template
———————————————————————————
A: model parameters
D_p: local data stored at worker p
P: number of workers

Function schedule(A, D):
　　Generate P parameter subsets $[\mathbf{S}_1, \ldots, \mathbf{S}_P]$
　　Return $[\mathbf{S}_1, \ldots, \mathbf{S}_P]$

Function update($p, \mathbf{S}_p, D_p, A$):　　// In parallel over $p = 1..P$
　　For each parameter a in $\mathbf{S}_p$:
　　　　$R_p[a]$ = updateParam(a, D_p)
　　Return R_p

Function aggregate($[R_1, \ldots, R_P], A$):
　　Combine intermediate results $[R_1, \ldots, R_P]$
　　Apply intermediate results to A
———————————————————————————

Algorithm 2 SchMP Dynamic, Prioritized Lasso
———————————————————————————
$\mathbf{X}, \mathbf{y}$: input data
$\{\mathbf{X}\}^p, \{\mathbf{y}\}^p$: rows/samples of $\mathbf{X}, \mathbf{y}$ stored at worker p
β: model parameters (regression coefficients)
λ: ℓ_1 regularization penalty
τ: $\mathcal{G}$ edges whose weight is below τ are ignored

Function schedule($\beta, \mathbf{X}$):
　　Pick $L > P$ params in β with probability $\propto (\Delta\beta_a)^2$
　　Build dependency graph $\mathcal{G}$ over L chosen params:
　　　　edge weight of (β_a, β_b) = correlation($\mathbf{x}^a, \mathbf{x}^b$)
　　$[\beta_{\mathcal{G}_1}, \ldots, \beta_{\mathcal{G}_K}]$ = findIndepNodeSet($\mathcal{G}, \tau$)
　　For $p = 1..P$:
　　　　$\mathbf{S}_p = [\beta_{\mathcal{G}_1}, \ldots, \beta_{\mathcal{G}_K}]$
　　Return $[\mathbf{S}_1, \ldots, \mathbf{S}_P]$

Function update($p, \mathbf{S}_p, \{\mathbf{X}\}^p, \{\mathbf{y}\}^p, \beta$):
　　For each param β_a in $\mathbf{S}_p$, each row i in $\{\mathbf{X}\}^p$:
　　　　$R_p[a]$ += $x_a^i y^i - \sum_{b \neq a} x_a^i x_b^i \beta_b$
　　Return R_p

Function aggregate($[R_1, \ldots, R_P], \mathbf{S}_1, \beta$):
　　For each parameter β_a in $\mathbf{S}_1$:
　　　　temp = $\sum_{p=1}^{P} R_p[a]$
　　　　$\beta_a = \mathcal{S}(\text{temp}, \lambda)$
———————————————————————————

on subsequent iterations, we should make their updated values available as soon as possible, rather than waiting until the end of the pipeline depth s.

4. SchMP Implementations of ML Programs

We describe how two ML algorithms can be written as Scheduled Model Parallel (SchMP) programs. The user implements schedule(), update(), aggregate(); alternatively, STRADS provides pre-implemented schedule() functions for some classes of SchMP programs. Algorithm 1 shows a typical SchMP program.

4.1 Parallel Coordinate Descent for Lasso

Lasso, or ℓ_1-regularized least-squares regression, is used to identify a small set of important features from high-dimensional data. It is an optimization problem

$$\min_\beta \quad \tfrac{1}{2}\sum_{i=1}^{n}\left(\mathbf{y}^i - \mathbf{x}^i\beta\right)^2 + \lambda\|\beta\|_1 \qquad (4)$$
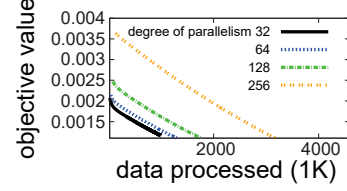

Figure 4: **Random Model Parallel Lasso:** Objective value (lower the better) versus processed data samples, with 32 to 256 workers performing concurrent updates. Under naive (random) model parallel, higher degree of parallelism results in worse progress.

where $\|\beta\|_1 = \sum_{a=1}^{d} |\beta_a|$ is a sparsity-inducing ℓ_1-regularizer, and λ is a tuning parameter that controls the sparsity level of β. $\mathbf{X}$ is an N-by-M design matrix ($\mathbf{x}^i$ represents the i-th row, $\mathbf{x}_a$ represents the a-th column), $\mathbf{y}$ is an N-by-1 observation vector, and β is the M-by-1 coefficient vector (the model parameters). The Coordinate Descent (CD) algorithm is used to solve Eq. (4), and thus learn β from the inputs $\mathbf{X}, \mathbf{y}$; the CD update rule for β_a is

$$\beta_a^{(t)} \leftarrow \mathcal{S}(\mathbf{x}_a^\top \mathbf{y} - \textstyle\sum_{b \neq a} \mathbf{x}_a^\top \mathbf{x}_b \beta_b^{(t-1)}, \lambda), \qquad (5)$$

where $\mathcal{S}(\cdot, \lambda)$ is a soft-thresholding operator [11].

Algorithm 2 shows an SchMP Lasso that uses dynamic, prioritized scheduling. It expects that each machine locally stores a subset of data samples (which is common practice in parallel ML), however the Lasso update Eq. (5) uses a feature/column-wise access pattern. Therefore every worker $p = 1..P$ operates on the same scheduled set of L parameters, but using their respective data partitions $\{\mathbf{X}\}^p, \{\mathbf{y}\}^p$. Note that update() and aggregate() are a straightforward implementation of Eq. (5).

We direct attention to schedule(): it picks (i.e. prioritizes) L parameters in β with probability proportional to their squared difference from the latest update (their "delta"); parameters with larger delta are more likely to be non-converged. Next, it builds a dependency graph over these L parameters, with edge weights equal to the correlation[2] between data columns $\mathbf{x}^a, \mathbf{x}^b$. Finally, it removes all edges in $\mathcal{G}$ below a threshold $\tau > 0$, and extracts nodes $\beta_{\mathcal{G}_k}$ that do not have common edges. All chosen $\beta_{\mathcal{G}_k}$ are thus pairwise independent and safe to update in parallel.

Why is such a sophisticated schedule() necessary? Suppose we used random parameter selection [5]: Figure 4 shows its progress, on the Alzheimer's Disease (AD) data [40]. The total compute to reach a fixed objective value goes up with more concurrent updates — i.e. progress per unit computation is decreasing, and the algorithm has poor scalability. Another reason is uneven parameter convergence: Figure 5 shows how many iterations different parameters took to converge on the AD dataset; $> 85\%$ of parameters converged in < 5 iterations, suggesting that the prioritization in Algorithm 2 should be very effective.
Default schedule() **functions:** The squared delta-based parameter prioritization and dynamic dependency check-

———————————————————————————
[2] On large data, it suffices to estimate the correlation with a data subsample.

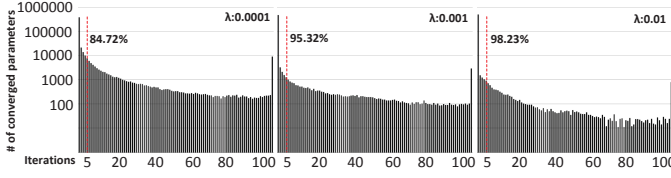Figure 5: **Uneven Parameter Convergence:** Number of converged parameters at each iteration, with different regularization parameters λ. Red bar shows the percentage of converged parameters at iteration 5.

ing in SchMP Lasso's `schedule()` (Algorithm 2) generalize to other regression problems — for example, we also implement sparse logistic regression using the same `schedule()`. STRADS allows ML programmers to reuse Algorithm 2's `schedule()` via a library function `scheduleDynRegr()`.

4.2 Parallel Gibbs Sampling for Topic Modeling

Topic modeling, a.k.a. Latent Dirichlet Allocation (LDA), is an ML model for document soft-clustering; it assigns each of N text documents to a probability distribution over K topics, and each topic is a distribution over highly-correlated words. Topic modeling is usually solved via a parallel Gibbs sampling algorithm involving three data structures: an N-by-K document-topic table U, an M-by-K word-topic table V (where M is the vocabulary size), and the topic assignments z_{ij} to each word "token" j in each document i. Each topic assignment z_{ij} is associated with the j-th word in the i-th document, w_{ij} (an integer in 1 through M); the z_{ij}, w_{ij} are usually pre-partitioned over worker machines [1].

The Gibbs sampling algorithm iteratively sweeps over all z_{ij}, assigning each one a new topic via this probability distribution over topic outcomes $k = 1..K$:

$$P(z_{ij} = k \mid U, V) \propto \frac{\alpha + U_{ik}}{K\alpha + \sum_{\ell=1}^{K} U_{i\ell}} + \frac{\beta + V_{w_{ij},k}}{M\beta + \sum_{m=1}^{M} V_{mk}}, \quad (6)$$

where α, β are smoothing parameters. Once a new topic for z_{ij} has been sampled, the tables U, V are updated by (1) decreasing $V_{i,oldtopic}$ and $U_{w_{ij},oldtopic}$ by one, and (2) increasing $U_{i,newtopic}$ and $V_{w_{ij},newtopic}$ by one.

Eq. (6) is usually replaced by a more efficient (but equivalent) variant called SparseLDA [36], which we also use. We will not show its details within `update()` and `aggregate()`; instead, we focus on how `schedule()` controls which z_{ij} are being updated by which worker. Algorithm 3 shows our SchMP LDA implementation, which uses a static "word-rotation" schedule, and partitions the documents over workers. The word-rotation schedule partitions the rows of V (word-topic table), so that workers never touch the same rows in V (each worker just skips over words w_{ij} associated with not-assigned rows). The partitioning is "rotated" P times, so that every word w_{ij} in each worker is touched exactly once after P invocations of `schedule()`.

As with Lasso, one might ask why this `schedule()` is useful. A common strategy is to have workers sweep over all their z_{ij} every iteration [1], however, as we show later in

---

Algorithm 3 SchMP Static-schedule Topic Modeling

U, V: doc-topic table, word-topic table (model params)
N, M: number of docs, vocabulary size
$\{z\}_p, \{w\}_p$: topic indicators and token words stored at worker p
c: persistent counter in `schedule()`

Function `schedule()`:
  For $p = 1..P$:   // "word-rotation" schedule
    $x = (p - 1 + c) \bmod P$
    $\mathbf{S}_p = (xM/P, (x+1)M/P)$ // p's word range
  $c = c + 1$
  Return $[\mathbf{S}_1, \dots, \mathbf{S}_P]$

Function `update`$(p, \mathbf{S}_p, \{U\}_p, V, \{w\}_p, \{z\}_p)$:
  [lower,upper] = $\mathbf{S}_p$ // Only touch w_{ij} in range
  For each token z_{ij} in $\{z\}_p$:
    If $w_{ij} \in$ range(lower,upper):
      old = z_{ij}
      new = SparseLDAsample(U_i, V, w_{ij}, z_{ij})
      Record old, new values of z_{ij} in R_p
  Return R_p

Function `aggregate`$([R_1, \dots, R_P], U, V)$:
  Update U, V with changes in $[R_1, \dots, R_P]$

Section 5, this causes concurrent writes to the same rows in V, breaking model dependencies.

Default `schedule()` functions: Like SchMP Lasso, SchMP LDA's `schedule()` (Algorithm 3) can be generically applied to ML program where each data sample touches just a few parameters (Matrix Factorization is one example). The idea is to assign disjoint parameter subsets across workers, who only operate on data samples that "touch" their currently assigned parameter subset. For this purpose, STRADS provides a generic `scheduleStaticRota()` that partitions the parameters into P contiguous (but disjoint) blocks, and rotates these blocks amongst workers at the beginning of each iteration.

4.3 Other ML Programs

In our evaluation, we consider two more SchMP ML Programs — sparse Logistic Regression (SLR) and Matrix Factorization (MF). SchMP SLR uses the same dynamic, prioritized `scheduleDynRegr()` as SchMP Lasso, while the `update()` and `aggregate()` functions are slightly different to accommodate the new LR objective function. SchMP MF uses `scheduleStaticRota()` that, like SchMP LDA, rotates disjoint (and therefore dependency-free) parameter assignments amongst the P distributed workers.

5. Evaluation

We compare SchMP ML programs implemented on STRADS against existing parallel execution schemes — either a well-known publicly-available implementation, or if unavailable, we write our own implementation — as well as sequential execution. Our intent is to show (1) SchMP implementations executed by STRADS have significantly improved progress per iteration over other parallel execution schemes, coming fairly close to "ideal" sequential execution in some cases. At

ML app	Data set	Workload	Feature	Model Parameters	Data size
MF	Netflix	100M ratings	480K users, 17K movies (rank=40)	19×10^6 (74MB)	2.2GB
MF	x256 Netflix	25B ratings	7.6M users, 272K movies (rank=40)	318×10^6 (1.2GB)	563GB
LDA	NYTimes	99.5M tokens	300K documents, 100K words 1K topics	400×10^6 (1.5GB)	0.5GB
LDA	PubMed	737M tokens	8.2M documents, 141K words 1K topics	8.3×10^9 (31GB)	4.5GB
LDA	ClueWeb	10B tokens	50M webpages, 2M words, 1K topics	52×10^9 (193GB)	80GB
Lasso	AlzheimerDisease(AD)	235M nonzeros	463 samples, 0.5M features	0.5×10^6 (2MB)	6.4GB
Lasso	LassoSynthetic	2B nonzeros	50K samples, 100M features	100×10^6 (381MB)	60 GB
Logistic	LogisticSynthetic	1B nonzeros	50K samples, 10M features	10×10^6 (3.8MB)	29 GB

Table 2: Data sets used in our evaluation.

the same time, (2) the STRADS system can sustain high iteration throughput (i.e. model parameters and data points processed per second) that is competitive with existing systems. Together, the high progress per iteration and high iteration throughput lead to faster ML program completion times (i.e. fewer seconds to converge).

While [20] demonstrated that SchMP can achieve fast ML program completion times, they did not break down the results into algorithmic progress per update and update throughput, which are important metrics that distinguish between the performance gains due to SchMP (which improves algorithm progress per update) versus the system design (which improves update throughput). Our experiments provide this crucial breakdown, in order to distinguish between the two sources of performance gain.

Cluster setup: Unless otherwise stated, we used 100 nodes each with 4 quad-core processors (16 physical cores) and 32GB memory; this configuration is similar to Amazon EC2 c4.4xlarge instances (16 physical cores, 30GB memory). The nodes are connected by 1Gbps ethernet as well as a 20Gbps Infiniband IP over IB interface. Most experiments were conducted on the 1Gbps ethernet; we explicitly point out those that were conducted over IB.

Datasets: We use several real and synthetic datasets — see Table 2 for details. All real datasets except AD are public.

Performance metrics: We compare ML implementations using three metrics: (1) *objective function value* versus *total data samples operated upon*[3], abbreviated **OvD**; (2) *total data samples operated upon* versus *time (seconds)*, abbreviated **DvT**; (3) *objective function value* versus *time (seconds)*, referred to as **convergence time**. The goal is to achieve the best objective value in the least time — i.e. fast convergence.

OvD is a uniform way to measure ML progress per iteration across different ML implementations, as long as they use identical parameter update equations — we ensure this is always the case, unless otherwise stated. Similarly, DvT measures ML iteration throughput across comparable implementations. Note that high OvD and DvT imply good (i.e. small) ML convergence time, and that measuring OvD or DvT alone (as is sometimes done in the literature) is *insufficient* to show that an algorithm converges quickly.

5.1 Static SchMP Evaluation

Our evaluation considers static-schedule SchMP algorithms separately from dynamic-schedule SchMP algorithms, because of their different service implementations (Section 3.3). We first evaluate static-schedule SchMP algorithms running on the STRADS Static Engine.

ML programs and baselines: We evaluate the performance of LDA (a.k.a. topic model) and MF (a.k.a collaborative filtering). STRADS uses Algorithm 3 (**SchMP-LDA**) for LDA, and a scheduled version of the Stochastic Gradient Descent (SGD) algorithm[4] for MF (**SchMP-MF**). For baselines, we used **YahooLDA**, and **BSP-MF** – our own implementation of the classic BSP SGD for MF ; both are data parallel algorithms, meaning that they do not use SchMP schemes. These baselines were chosen to analyze how SchMP affects OvD, DvT and convergence time; later will we show convergence time benchmarks against the GraphLab system which does use model parallelism.

To ensure a fair comparison, YahooLDA was modified to (1) dump model state at regular intervals for later objective (log-likelihood) computation[5]; (2) keep all local program state in memory, rather than streaming it off disk, because it fits for our datasets. All LDA experiments were performed on the 20Gbps Infiniband network, so that bandwidth would not be a bottleneck for the parameter server used by YahooLDA. Note that in LDA OvD and DvT measurements, we consider each word token as one data sample.

5.1.1 Improvement in convergence times

Static SchMP has high OvD: In the LDA experiments, YahooLDA's OvD decreases substantially when going from 25 to 50 machines (NYT/PubMed data sets), or to 100 machines (ClueWeb data set). In contrast, SchMP-LDA maintains the same OvD (Figures 6a, 6b, 6c). For MF, Figure 6f shows that BSP-MF is sensitive to step size[6]; if BSP-MF employs the ideal step size determined for serial execution, it does not properly converge on ≥ 32 machines. In contrast, SchMP-MF can safely use the ideal serial step size (Figures 6d,6e),

---

[3] ML algorithms operate upon the same data point many times. The total data samples operated upon exceeds N, the number of data samples.

[4] Due to space limits, we could not provide a full Algorithm figure. Our SchMP-MF divides up the input data such that different workers never update the same parameters in the same iteration.

[5] With overhead less than 1% of total running time.

[6] A required tuning parameter for SGD MF implementations; higher step sizes lead to faster convergence, but step sizes that are too large can cause algorithm divergence/failure.
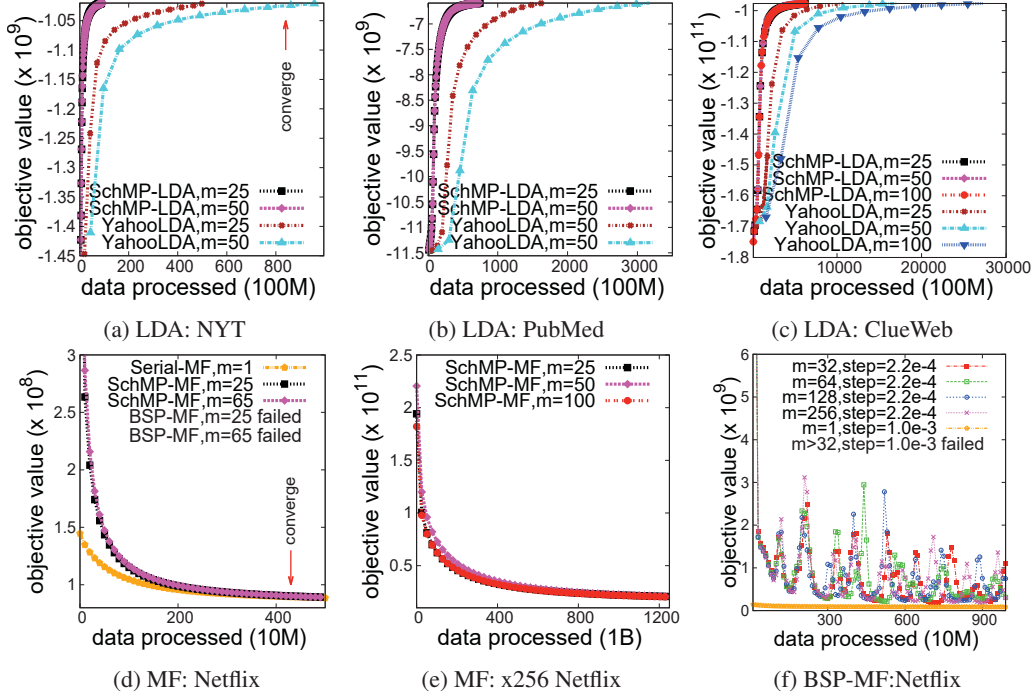
Figure 6: **Static SchMP:** OvD. (a-c) SchMP-LDA vs YahooLDA on three data sets; (d-e) SchMP-MF vs BSP-MF on two data sets; (f) parallel BSP-MF is unstable if we use an ideal sequential step size. m denotes number of machines.
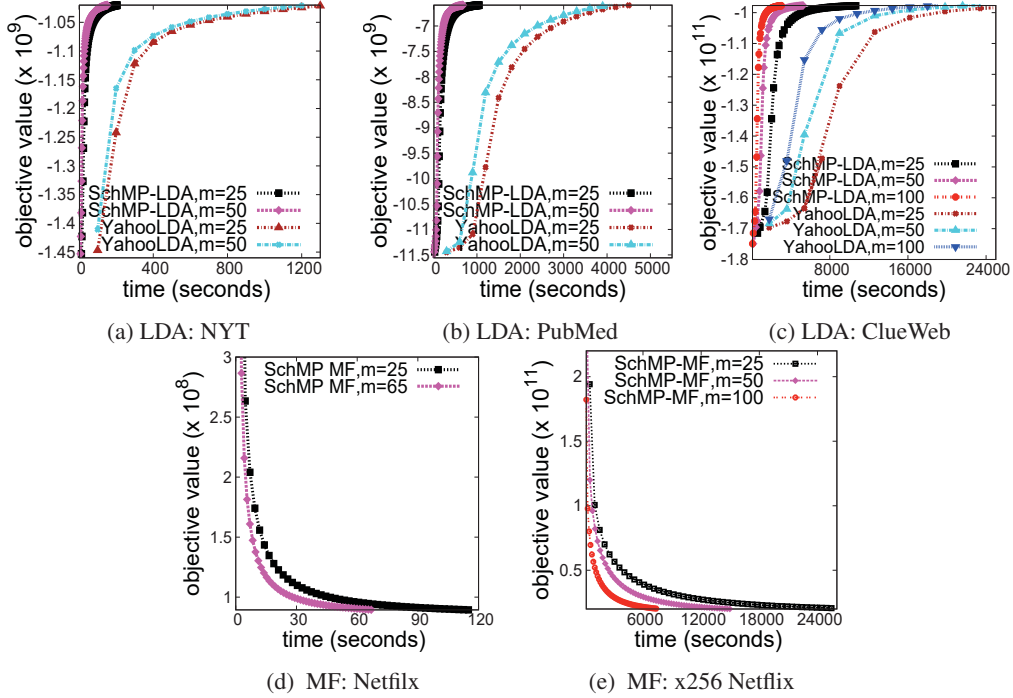

Figure 7: **Static SchMP:** convergence times. (a-c) SchMP-LDA vs YahooLDA; (d-e) SchMP-MF with varying number of machines m.

and approaches the same OvD as serial execution within 20 iterations.

STRADS Static Engine has high DvT: For LDA, table 3 shows that SchMP-LDA enjoys higher DvT than YahooLDA; we speculate that YahooLDA's lower DvT is primarily due to lock contention on shared data structures between application and parameter server threads (which the STRADS Static Engine tries to avoid).

Static SchMP on STRADS has low convergence times: Thanks to high OvD and DvT, SchMP-LDA's convergence

(a) DvT   (b) Time   (c) OvD
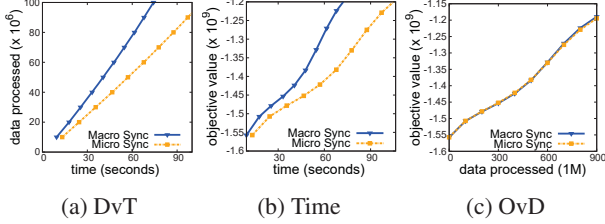
Figure 8: **Static Engine:** synchronization cost optimization. (a) macro synchronization improves DvT by 1.3 times; (b) it improves convergence speed by 1.3 times; (c) This synchronization strategy does not hurt OvD.
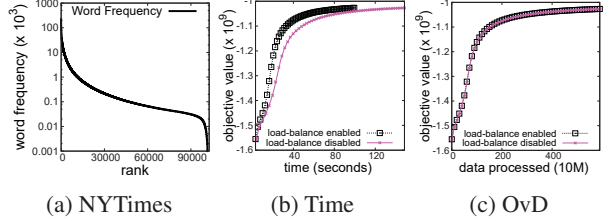

(a) NYTimes   (b) Time   (c) OvD

Figure 9: **Static Engine:** Job pool load balancing. (a) Biased word frequency distribution in NYTimes data set; (b) by dispatching the 300 heaviest words first, convergence speed improves by 30 percent to reach objective value -1.02e+9; (c) this dispatching strategy does not hurt OvD.

times are not only lower than YahooLDA, but also scale better with increasing machine count (Figures 7a, 7b, 7c). SchMP-MF also exhibits good scalability (Figure 7d, 7e).

5.1.2 Benefits of Static Engine optimizations

The STRADS Static Engine achieves high DvT (i.e iteration throughput) via two system optimizations: (1) reducing synchronization costs via the ring topology; (2) using a job pool to perform load balancing across **Job Executors**.

Reducing synchronization costs: Static SchMP programs (including SchMP-LDA and SchMP-MF) do not require all parameters to be synchronized across all machines, and this motivates the use of a ring topology. For example, consider SchMP-LDA Algorithm 3: the word-rotation `schedule()` directly suggests that **Job Executors** can pass parameters to their ring neighbor, rather than broadcasting to all machines; this applies to SchMP-MF as well.

STRADS's Static Engine implements this parameter-passing strategy via a ring topology, and only performs a global synchronization barrier after all parameters have completed one rotation (i.e. P iterations) — we refer to this as "Macro Synchronization". This has two effects: (1) network traffic becomes less bursty, and (2) communication is effectively overlapped with computation; as a result, DvT is improved by 30% compared to a naive implementation that invokes a synchronization barrier every iteration ("Micro Synchronization", Figure 8a). This strategy does not negatively affect OvD (Figure 8c), and hence time to convergence improves by about 30% (Figure 8b).

Job pool load balancing: Uneven workloads are common in Static SchMP programs: Figure 9a shows that the word

Data set(size)	#machines	YahooLDA	SchMP-LDA
NYT(0.5GB)	25	38	43
NYT(0.5GB)	50	79	62
PubMed(4.5GB)	25	37	70
PubMed(4.5GB)	50	75	107
ClueWeb(80GB)	25	39.7	58.3
ClueWeb(80GB)	50	78	114
ClueWeb(80GB)	100	151	204

Table 3: **Static SchMP:** DvT for topic modeling (million tokens operated upon per second).

distribution in LDA is highly skewed, meaning that some SchMP-LDA `update()` jobs will be much longer than others. Hence, STRADS dispatches the heaviest jobs first to the **Job Executor** threads. This improves convergence times by 30% on SchMP-LDA (Figure 9b), without affecting OvD.

5.1.3 Comparison against other systems:

GraphLab: We compare SchMP-MF with GraphLab's SGD MF implementation, on a different set of 8 machines — each with 64 cores, 128GB memory. On Netflix , GL-SGDMF converged to objective value 1.8e+8 in 300 seconds, and SchMP-MF converged to 9.0e+7 in 302 seconds (i.e. better objective value in the same time). In terms of DvT, SchMP-MF touches 11.3m data samples per second, while GL-MF touches 4.5m data samples per second.

Comparison against single-core LDA: We compare SchMP-LDA with a single-core LDA implementation (Single-LDA)[7] on PubMed. Single-LDA converges in 24.6 hours while SchMP-LDA takes 17.5 minutes and 11.5 minutes on 25 machines (400 cores) and 50 machines (800 cores) respectively. Both Single-LDA and SchMP-LDA show similar OvD results. In DvT, Single-LDA processes 830K tokens per second while SchMP-LDA processes 70M tokens on 25 machines (175K tokens per core), and 107M tokens on 50 machines (133K tokens per core). The locking contention on a shared data structure within a machine accounts for the reduced per-core efficiency of SchMP versus Single-LDA. Even so, the distributed approach of SchMP achieves substantial speed-up gains (84 times on 25 machines, 128 times on 50 machines) over Single-LDA. We leave further machine-level optimizations, such as relaxed consistency on shared data structures within a machine, as future work.

Bösen: We compare SchMP-LDA against an implementation of LDA on a recent parameter server, Bösen[35], which prioritizes model parameter communication across the network, based on each parameter's contribution to algorithm convergence. Thus, Bösen improves convergence rate (OvD) over YahooLDA while achieving similar token processing throughput (DvT). On the NYT data with 16 machines[8],

---

[7] For fair comparison, Single-LDA implements the same sampling algorithm and the same data structure of SchMP-LDA, and is lock-free. We use C++11 STL library for implementing the sampling algorithm routine from scratch without third-party library.

[8] For fair comparison, we set the stopping log-likelihood value to -1.0248e+09 for all experiments: Bösen, YahooLDA, SchMP-LDA with 16 machines.
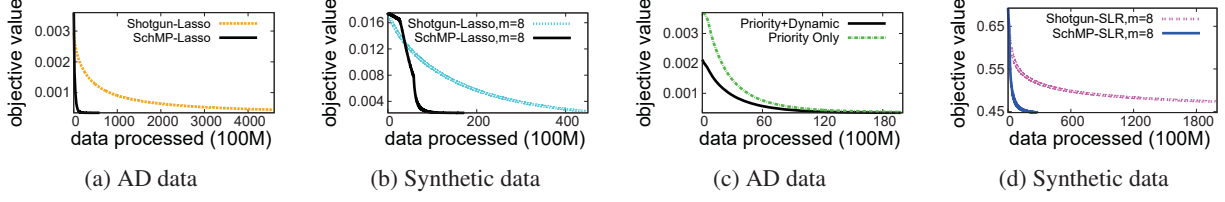
Figure 10: **Dynamic SchMP:** OvD. (a) SchMP-Lasso vs Shotgun-Lasso [5] on one machine (64 cores); (b) SchMP-Lasso vs Shotgun-Lasso on 8 machines; (c) SchMP-Lasso with & w/o dynamic partitioning on 4 machines; (d) SchMP-SLR vs Shotgun-SLR on 8 machines. m denotes number of machines.
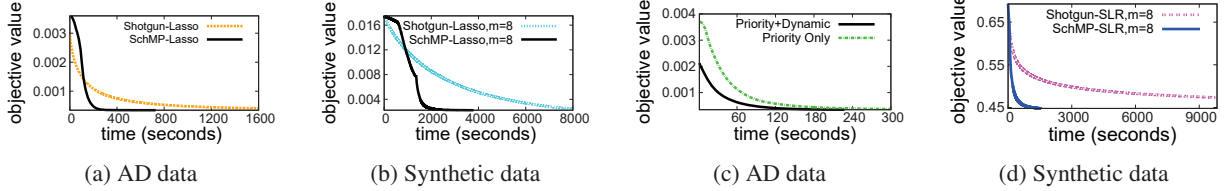

Figure 11: **Dynamic SchMP:** convergence time. Subfigures (a-d) correspond to Figure 10.

SchMP-LDA and Bösen are 7 and 3 times faster, respectively, than YahooLDA, and SchMP-LDA is about 2.3 times faster than Bösen. The SchMP-LDA improvement comes from the static model-parallel schedule() in Algorithm 3 (that avoids violating model dependencies in LDA), which the Bösen data-parallel LDA implementation does not have.

5.2 Dynamic SchMP Scheduling

Our evaluation of dynamic-schedule SchMP algoritms on the STRADS Dynamic Engine shows significantly improved OvD compared to random model parallel scheduling. We also show that (1) in the single machine setting, Dynamic SchMP comes at a cost to DvT, but overall convergence speed is still superior to random model parallel; and (2) in the distributed setting, this DvT penalty mostly disappears.

ML programs and baselines: We evaluate ℓ_1-regularized linear regression (Lasso) and ℓ_1-regularized Logistic regression (sparse LR, or SLR) – STRADS uses Algorithm 2 (SchMP-Lasso) for the former, and we solve the latter using a minor modification to SchMP-Lasso [9] (called SchMP-SLR). To the best of our knowledge, there are no open-source distributed Lasso/SLR baselines that use coordinate descent, so we implement the Shotgun Lasso/SLR algorithm [5] (Shotgun-Lasso, Shotgun-SLR), which uses random model parallel scheduling.

5.2.1 Improvement in convergence times

Dynamic SchMP has high OvD: Dynamic SchMP achieves high OvD, in both single-machine (Figure 10a) and distributed, 8-machine (Figure 10b) configurations; here we have compared SchMP-Lasso against random model parallel Lasso (Shotgun-Lasso) [5]. In either case, Dynamic SchMP decreases the data samples required for convergence by an

---

[9] Lasso and SLR are solved via the coordinate descent algorithm, hence SchMP-Lasso and SchMP-SLR only differ slightly in their update equations. We use coordinate descent rather gradient descent because it has no step size tuning and more stable convergence [27, 28].

order of magnitude. Similar observations hold for distributed SchMP-SLR versus Shotgun-SLR (Figure 10d).

STRADS Dynamic Engine DvT analysis: Table 4 shows how STRADS Dynamic Engine's DvT scales with increasing machines. We observe that DvT is limited by dataset density — if there are more nonzeros per feature column, we observe better DvT scalability with more machines. The reason is that the Lasso and SLR problems' model parallel dependency structure (Section 4.1) limits the maximum degree of parallelization (number of parameters that can be correctly updated each iteration), thus Dynamic Engine scalability does not come from updating more parameters in parallel (which may be mathematically impossible), but from processing more data per feature column.

Dynamic SchMP on STRADS has low convergence times: Overall, both SchMP-Lasso and SchMP-SLR enjoy better convergence times than their Shotgun counterparts. The worst-case scenario is a single machine using a dataset (AD) with few nonzeros per feature column (Figure 11a) — when compared with Figure 10a, we see that SchMP DvT is much lower than Shotgun (Shotgun-Lasso converges faster initially), but ultimately SchMP-Lasso still converges 5 times faster. In the distributed setting (Figure 11b Lasso, Figure 11d SLR), the DvT penalty relative to Shotgun is much smaller, and the curves resemble the OvD analysis (SchMP exhibits more than an order of magnitude speedup).

5.2.2 Benefits of Dynamic Engine optimizations

The STRADS Dynamic Engine improves DvT (data throughput) via iteration pipelining, while improving OvD via dynamic partitioning and prioritization in schedule().

Impact of dynamic partitioning and prioritization: Figures 10c (OvD) and 11c (OvT) show that the convergence speedup from Dynamic SchMP comes mostly from prioritization — we see that dependency checking approximately doubles SchMP-Lasso's OvD over prioritization alone, im-
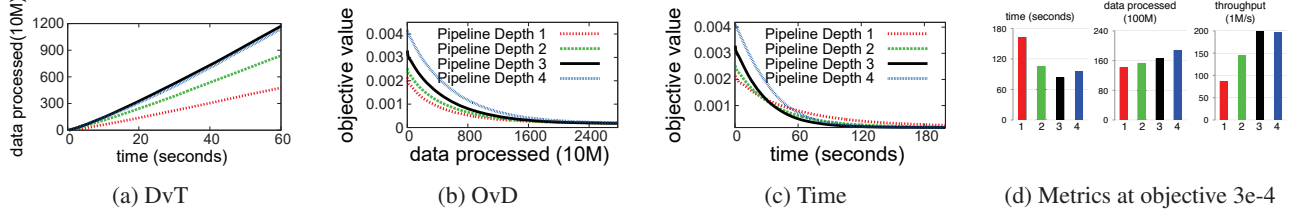
Figure 12: **Dynamic Engine:** iteration pipelining. (a) DvT improves $2.5\times$ at pipeline depth 3, however (b) OvD decreases with increasing pipeline depth. Overall, (c) convergence time improves $2\times$ at pipeline depth 3. (d) Another view of (a)-(c): we report time to converge to objective value 0.0003, OvD and DvT.

	nonzeros per column	1K	10K	20K
Application				
SchMP-Lasso 4 machines		125	212	202
SchMP-Lasso 8 machines		162	306	344
SchMP-LR 4 machines		75	98	103
SchMP-LR 8 machines		106	183	193

Table 4: **Dynamic SchMP:** DvT of SchMP-Lasso and SchMP-LR, measured as data samples (millions) operated on per second, for synthetic data sets with different column sparsity.

plying that the rest of the order-of-magnitude speedup over Shotgun-Lasso comes from prioritization. Additional evidence is provided by Figure 5; under prioritization most parameters converge within just 5 iterations.

Pipelining improves DvT at a small cost to OvD: The STRADS Dynamic Engine can pipeline iterations to improve DvT (iteration throughput), at some cost to OvD. Figure 12c shows that SchMP-Lasso (on 8 machines) converges most quickly at a pipeline depth of 3, and Figure 12d provides a more detailed breakdown, including the time take to reach the same objective value (0.0003). We make three observations. First, DvT improvement saturates at pipeline depth 3. Second, OvD, expressed as the number of data samples to convergence, gets proportionally worse as pipeline depth increases. Hence, the sweet spot for convergence time is pipeline depth 3, which halves convergence time compared to no pipelining (depth 1). Third, pipelining at depth 3 hurts OvT in the initial stage, but gets ahead of pipelining at depth 1,2 in the middle of running time due to the property of iterative algorithm. As iterations increases, delta of parameter update becomes smaller so that the erros from pipeline diminishes while the iteration throughput gain of pipeline is being kept constantly.

STRADS dynamic engine sets the pipeline depth to 3 by default. Since `update()` consists of three stages (Figure 3b), depth 3 is usually the sweet spot for convergence speed. There are two possible cases where a smaller pipeline depth could be more beneficial: 1) the latency of updates is far longer than the communication latency; 2) the problem has very strong dependency structures. In these cases, the OvD performance gains from setting a pipeline depth less than 3

can outweigh the DvT gains at depth 3, and manual depth tuning may be required to maximize covergence speed.

5.2.3 Comparisons against other systems

We compare SchMP-Lasso/SLR with Spark MLlib (Spark-Lasso, Spark-SLR), which uses the SGD algorithm. As with the earlier GraphLab comparison, we use 8 nodes with 64 cores and 128GB memory each. On the AD dataset (which has complex gene-gene correlations), Spark-Lasso reached objective value 0.0168 after 1 hour, whereas SchMP-Lasso achieved a lower objective (0.0003) in 3 minutes. On the LogisticSynthetic dataset (which was constructed to have few correlations), Spark-SLR converged to objective 0.452 in 899 seconds, while SchMP-SLR achieved a similar result.[10] This confirms that SchMP is more effective in the presence of more complex model dependencies.

Finally, we want to highlight that the STRADS system can significantly reduce the code required for an SchMP program compared to a standalone SchMP program implementation using MPI[24]: our SchMP-Lasso implementation (Algorithm 2) has 390 lines in `schedule()`, 181 lines in `update()` and `aggregate()`, and another 209 lines for miscellaneous uses like setting up the program environment while the standalone MPI implementation consists of about six thousand lines of code.

6. Related work

Early systems for scaling up ML focus on data parallelism [6] to leverage multi-core and multi-machine architectures, following the ideas in MapReduce [10]. Along these lines, Mahout[3] on Hadoop [2] and more recently MLI [30] on Spark [39] have been developed. The second generation of distributed ML systems — e.g. parameter servers (PS, [1, 9, 17, 21]) — address the problem of distributing large shared models across multiple workers. Early systems were designed for a particular class of ML problems, e.g., [1] for LDA and [9] for deep neural nets. More recent works [17, 21] have generalized the parameter server concept to support a wide range of ML algorithms. There are counterparts to parameter server ideas in STRADS:

---

[10] In the SchMP-Lasso/LR experiments, we did not include the overhead of checkpointing. We found that it is negligible ($< 1\%$ of total execution time) and dominated by update computation time.

for instance, stale synchronous parallel (SSP, [7, 8, 17]) and STRADS both control parameter staleness; the former through bookkeeping on the deviation between workers, and the latter through pipeline depth. Another example is filtering [21], which resembles parameter scheduling in STRADS, but is primarily for alleviating synchronization costs, e.g., their KKT filter suppresses transmission of "unnecessary" gradients, while STRADS goes a step further and uses algorithm information to make update choices (not just synchronization choices).

None of the above systems directly address the issue of conflict updates, which leads to slow convergence or even algorithmic failure [20]. Within the parallel ML literature, there have been several approaches: some works choose to ignore the dependencies (while also cautioning that this may cause algorithm failure) [5, 26], while others focus on understanding the dependency structure of a specific ML model, such as Matrix Factorization [13] or Latent Dirichlet Allocation [37]. The first systematic approach was proposed by GraphLab [14, 22], where ML computational dependencies are encoded by the user in a graph, so that the system may select disjoint subgraphs to process in parallel — thus, graph-scheduled model parallel ML algorithms can be written in GraphLab. Intriguing recent work, GraphX [15], combines these sophisticated GraphLab optimizations with database-style data processing and runs on a BSP-style MapReduce framework, sometimes without significant loss of performance.

Task prioritization (to exploit uneven convergence) was studied by PrIter [41] and GraphLab [22]. The former, built on Hadoop [2], prioritizes *data samples* that contribute most to convergence, while GraphLab ties prioritization to the program's graph representation. STRADS prioritizes the most promising *model parameter* values.

7. Conclusion

We developed STRADS to improve the convergence speed of model parallel ML at scale, achieving both high progress per iteration (via dependency checking and prioritization through SchMP programming), and high iteration throughput (via STRADS system optimizations such as pipelining and the ring topology). Consequently, SchMP programs running on STRADS achieve a marked performance improvement over recent, well-established baselines: to give two examples, SchMP-LDA converges 10x faster than YahooLDA, while SchMP-Lasso converges 5x faster than randomly-scheduled Shotgun-Lasso.

There are issues we would like to address in the future: chief amongst them is automatic `schedule()` creation, so users only have to implement `update()` and `aggregate()`, while scheduling is left to the system. We would also like to apply SchMP to ML programs that were not in our experiments, but offer similar scheduling opportunities due to complex model structure — such as deep neu-

ral nets and network models. A final issue we would like to explore is hybrid approaches combining data-parallelism and model-parallelism. [9] presented a hybrid approach specialized for training a large scale deep neural network, that showed promise for handling problems with extremely large data and model sizes. We would like to generalize the hybrid approach to a wider range of ML algorithms, beyond deep neural networks. While the ML community has developed the theoretical foundations for hybrid approaches, we have not yet observed substantial system support for these approaches from the systems community.

Finally, we believe that SchMP can improve other systems' ML application performance. Since SchMP is an approach to ML algorithm design, it is mostly orthogonal to the underlying system implementation, and therefore it should be possible to implement SchMP algorithms on Spark [39], MapReduce [10], and the Parameter Server system [21] [7]. We believe such implementations will benefit from improved algorithm progress per iteration (a property of SchMP), but improving iteration throughput (like in STRADS) may require additional system optimizations for SchMP algorithms (e.g. schedule and update pipelining).

8. Acknowledgements

References

[1] AHMED, A., ALY, M., GONZALEZ, J., NARAYANA-MURTHY, S., AND SMOLA, A. J. Scalable inference in latent variable models. In *WSDM* (2012).

[2] Apache Hadoop, http://hadoop.apache.org.

[3] Apache Mahout, http://mahout.apache.org.

[4] BLEI, D. M., NG, A. Y., AND JORDAN, M. I. Latent Dirichlet allocation. *Journal of Machine Learning Research 3* (2003), 993–1022.

[5] BRADLEY, J. K., KYROLA, A., BICKSON, D., AND GUESTRIN, C. Parallel coordinate descent for l1-regularized loss minimization. In *ICML* (2011).

[6] CHU, C., KIM, S. K., LIN, Y.-A., YU, Y., BRADSKI, G., NG, A. Y., AND OLUKOTUN, K. Map-reduce for machine learning on multicore. *NIPS* (2007).

[7] CUI, H., CIPAR, J., HO, Q., KIM, J. K., LEE, S., KUMAR, A., WEI, J., DAI, W., GANGER, G. R., GIBBONS, P. B.,

GIBSON, G. A., AND XING, E. P. Exploiting bounded staleness to speed up big data analytics. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)* (Philadelphia, PA, June 2014), USENIX Association, pp. 37–48.

[8] DAI, W., KUMAR, A., WEI, J., HO, Q., GIBSON, G. A., AND XING, E. P. High-performance distributed ML at scale through parameter server consistency models. In *AAAI* (2015).

[9] DEAN, J., CORRADO, G., MONGA, R., CHEN, K., DEVIN, M., MAO, M., AURELIO RANZATO, M., SENIOR, A., TUCKER, P., YANG, K., LE, Q. V., AND NG, A. Y. Large scale distributed deep networks. In *NIPS*. 2012.

[10] DEAN, J., AND GHEMAWAT, S. Mapreduce: Simplified data processing on large clusters. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6* (Berkeley, CA, USA, 2004), OSDI'04, USENIX Association, pp. 10–10.

[11] FRIEDMAN, J., HASTIE, T., HOFLING, H., AND TIBSHIRANI, R. Pathwise coordinate optimization. *Annals of Applied Statistics 1*, 2 (2007), 302–332.

[12] FU, W. Penalized regressions: the bridge versus the lasso. *Journal of Computational and Graphical Statistics 7*, 3 (1998), 397–416.

[13] GEMULLA, R., NIJKAMP, E., HAAS, P. J., AND SISMANIS, Y. Large-scale matrix factorization with distributed stochastic gradient descent. In *KDD* (2011).

[14] GONZALEZ, J. E., LOW, Y., GU, H., BICKSON, D., AND GUESTRIN, C. Powergraph: Distributed graph-parallel computation on natural graphs. In *OSDI* (2012), vol. 12, p. 2.

[15] GONZALEZ, J. E., XIN, R. S., DAVE, A., CRANKSHAW, D., FRANKLIN, M. J., AND STOICA, I. Graphx: Graph processing in a distributed dataflow framework. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2014).

[16] GRIFFITHS, T. L., AND STEYVERS, M. Finding scientific topics. *Proceedings of National Academy of Science 101* (2004), 5228–5235.

[17] HO, Q., CIPAR, J., CUI, H., LEE, S., KIM, J. K., GIBBONS, P. B., GIBSON, G. A., GANGER, G. R., AND XING, E. P. More effective distributed ML via a stale synchronous parallel parameter server. In *NIPS* (2013).

[18] KARYPIS, G., AND KUMAR, V. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput. 20*, 1 (Dec. 1998), 359–392.

[19] KRIZHEVSKY, A., SUTSKEVER, I., AND HINTON, G. E. Imagenet classification with deep convolutional neural networks. In *NIPS* (2012).

[20] LEE, S., KIM, J. K., ZHENG, X., HO, Q., GIBSON, G., AND XING, E. P. On model parallelism and scheduling strategies for distributed machine learning. In *NIPS*. 2014.

[21] LI, M., ANDERSEN, D. G., PARK, J. W., SMOLA, A. J., AHMED, A., JOSIFOVSKI, V., LONG, J., SHEKITA, E. J., AND SU, B.-Y. Scaling distributed machine learning with the parameter server. In *OSDI* (2014).

[22] LOW, Y., BICKSON, D., GONZALEZ, J., GUESTRIN, C., KYROLA, A., AND HELLERSTEIN, J. M. Distributed graphlab: A framework for machine learning and data mining in the cloud. *Proc. VLDB Endow. 5*, 8 (2012), 716–727.

[23] MCSHERRY, F. A uniform approach to accelerated pagerank computation. In *Proceedings of the 14th international conference on World Wide Web, WWW 2005, Chiba, Japan, May 10-14, 2005* (2005), pp. 575–582.

[24] MPICH, http://mpich.org.

[25] POWER, R., AND LI, J. Piccolo: Building fast, distributed programs with partitioned tables. In *OSDI* (2010).

[26] RECHT, B., RE, C., WRIGHT, S., AND NIU, F. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *NIPS* (2011).

[27] RICHTÁRIK, P., AND TAKÁČ, M. Parallel coordinate descent methods for big data optimization. *arXiv preprint arXiv:1212.0873* (2012).

[28] SCHERRER, C., HALAPPANAVAR, M., TEWARI, A., AND HAGLIN, D. Scaling up parallel coordinate descent algorithms. In *ICML* (2012).

[29] SCHERRER, C., TEWARI, A., HALAPPANAVAR, M., AND HAGLIN, D. Feature clustering for accelerating parallel coordinate descent. In *NIPS*. 2012.

[30] SPARKS, E. R., TALWALKAR, A., SMITH, V., KOTTALAM, J., PAN, X., GONZALEZ, J., FRANKLIN, M. J., JORDAN, M. I., AND KRASKA, T. MLI: An API for distributed machine learning. In *ICDM* (2013).

[31] TIBSHIRANI, R. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society. Series B (Methodological) 58*, 1 (1996), 267–288.

[32] TIERNEY, L. Markov chains for exploring posterior distributions. *the Annals of Statistics* (1994), 1701–1728.

[33] WANG, M., XIAO, T., LI, J., ZHANG, J., HONG, C., AND ZHANG, Z. Minerva: A scalable and highly efficient training platform for deep learning. In *APSys* (2014).

[34] WANG, Y., ZHAO, X., SUN, Z., YAN, H., WANG, L., JIN, Z., WANG, L., GAO, Y., LAW, C., AND ZENG, J. Peacock: Learning long-tail topic features for industrial applications. *ACM Transactions on Intelligent Systems and Technology 9*, 4 (2014).

[35] WEI, J., DAI, W., QIAO, A., HO, Q., CUI, H., GANGER, G. R., GIBBONS, P. B., GIBSON, G. A., AND XING, E. P. Managed communication and consistency for fast data-parallel iterative analytics. In *Proceedings of the Sixth ACM Symposium on Cloud Computing* (New York, NY, USA, 2015), SoCC '15, ACM, pp. 381–394.

[36] YAO, L., MIMNO, D., AND MCCALLUM, A. Efficient methods for topic model inference on streaming document collections. In *KDD* (2009).

[37] YUAN, J., GAO, F., HO, Q., DAI, W., WEI, J., ZHENG, X., XING, E. P., LIU, T.-Y., AND MA, W.-Y. LightLDA: Big topic models on modest compute clusters. In *WWW* (2015).

[38] ZAHARIA, M., CHOWDHURY, M., DAS, T., DAVE, A., MA, J., MCCAULEY, M., FRANKLIN, M. J., SHENKER, S., AND

STOICA, I. Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In *NSDI* (2012).

[39] ZAHARIA, M., DAS, T., LI, H., HUNTER, T., SHENKER, S., AND STOICA, I. Discretized streams: Fault-tolerant streaming computation at scale. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2013), SOSP '13, ACM, pp. 423–438.

[40] ZHANG, B., GAITERI, C., BODEA, L.-G., WANG, Z., MCELWEE, J., PODTELEZHNIKOV, A. A., ZHANG, C., XIE, T., TRAN, L., DOBRIN, R., ET AL. Integrated systems approach identifies genetic nodes and networks in late-onset Alzheimer's disease. *Cell 153*, 3 (2013), 707–720.

[41] ZHANG, Y., GAO, Q., GAO, L., AND WANG, C. Priter: A distributed framework for prioritized iterative computations. In *SOCC* (2011).

[42] ZINKEVICH, M., WEIMER, M., LI, L., AND SMOLA, A. J. Parallelized stochastic gradient descent. In *NIPS* (2010).