Managed Communication and Consistency for Fast Data-Parallel Iterative Analytics

Jinliang Wei, Wei Dai, Aurick Qiao, Qirong Ho*, Henggang Cui
Gregory R. Ganger, Phillip B. Gibbons, Garth A. Gibson, Eric P. Xing
*Carnegie Mellon University, *Institute for Infocomm Research, A*STAR*

CMU-PDL-15-105
April 2015

Parallel Data Laboratory
Carnegie Mellon University
Pittsburgh, PA 15213-3890

Acknowledgements: This research is supported in part by Intel as part of the Intel Science and Technology Center for Cloud Computing (ISTC-CC), the National Science Foundation under awards CNS-1042537, CNS-1042543 (PRObE, www.nmc-probe.org), IIS-1447676 (Big Data), the National Institute of Health under contract GWAS R01GM087694, and DARPA under contracts FA87501220324 and FA87501220324 (XDATA). We also thank the members companies of the PDL Consortium (including Actifio, APC, EMC, Facebook, Google, Hewlett-Packard, Hitachi, Huawei, Intel, Microsoft, NetApp, Oracle, Samsung, Seagate, Symantec, Western Digital) for their interest, insights, feedback, and support.

Keywords: distributed systems, machine learning, parameter server

Abstract

At the core of Machine Learning (ML) analytics applied to Big Data is often an expert-suggested model, whose parameters are refined by iteratively processing a training dataset until convergence. The completion time (i.e. convergence time) and quality of the learned model not only depends on the rate at which the refinements are generated but also the quality of each refinement. While data-parallel ML applications often employ a loose consistency model when updating shared model parameters to maximize parallelism, the accumulated error may seriously impact the quality of refinements and thus delay completion time, a problem that usually gets worse with scale. Although more immediate propagation of updates reduces the accumulated error, this strategy is limited by physical network bandwidth. Additionally, the performance of the widely used stochastic gradient descent (SGD) algorithm is sensitive to initial step size, simply increasing communication without adjusting the step size value accordingly fails to achieve optimal performance.

This paper presents Bösen, a system that maximizes the network communication efficiency under a given inter-machine network bandwidth budget to minimize accumulated error, while ensuring theoretical convergence guarantees for large-scale data-parallel ML applications. Furthermore, Bösen prioritizes messages that are most significant to algorithm convergence, further enhancing algorithm convergence. Finally, Bösen is the first distributed implementation of the recently presented adaptive revision algorithm, which provides orders of magnitude improvement over a carefully tuned fixed schedule of step size refinements. Experiments on two clusters with up to 1024 cores show that our mechanism significantly improves upon static communication schedules.

1 Introduction

Machine learning (ML) analytics are an increasingly important cloud workload. At the core of many important ML analytics is an expert-suggested model, whose parameters must be refined starting from an initial guess. For example, deep learning applications refine their inter-layer weight matrices to achieve higher prediction accuracy for classification and regression problems; topic models refine the word composition and weights in each global topic to better summarize a text corpus; sparse coding or matrix factorization models refine their factor matrices to better de-noise or reconstruct the original input matrix. The parameter refinement is performed by algorithms that repeatedly/iteratively compute updates from many data samples, and push the model parameters towards an optimum value. When the parameters are close enough to an optimum, the algorithm is stopped and is said to have converged—therefore such algorithms are called *iterative-convergent*.

Some of these iterative-convergent algorithms are well-known, because they have been applied to a wide variety of ML models. Examples include stochastic gradient descent (SGD) with applications in deep learning [19, 27], coordinate descent with applications in regression [18, 9], and Markov chain Monte Carlo sampling with applications in topic modeling [21, 23]. Even when the size of the data or model is not in the terabytes, the computational cost of these applications can still be significant enough to inspire programmers to parallelize the application over a cluster of machines (e.g., in the cloud). *Data-parallelism* is one common parallelization scheme, where the data samples are partitioned across machines, which all have shared access to the model parameters. In each data-parallel iteration, every machine computes a “sub-update” on its data subset (in parallel with other machines), following which the sub-updates are aggregated and applied to the parameters.

In the ML literature, it is often the case that a data-parallel algorithm is executed in a Bulk Synchronous Parallel (BSP) fashion, where computation uses local model copies that are synchronized only at the end of each iteration and the next iteration may not start until all machines have received up-to-date model parameters. The consistency guarantees provided by BSP-style execution allows convergence of the distributed algorithms to be ensured, though BSP-style execution may suffer from significant overheads due to the synchronization barriers separating iterations [24, 12]. On the other hand, asynchronous systems have been proposed, in which machines can enter the next iteration before receiving the fully-updated model parameters [4, 10, 15], hiding the synchronization latency. As asynchronous communication incurs much less overhead, asynchronous systems typically communicate model refinements and up-to-date parameters in much finer granularity, showing improved per-data-sample convergence rate due to fresher parameter values as long as the network can cope with communication. However, they no longer enjoy the assurance of formal convergence guarantees.

One exception is systems that satisfy the Bounded Staleness consistency model, also called Bounded Delay [29] or Stale Synchronous [24], which allows computations to use stale model parameters (to reduce synchronization overheads), but strictly upper-bounds the number of missing iterations, restoring formal convergence guarantees [24]. While using (possibly stale) local model parameters improves computation throughput (number of data samples processed per second), it degrades algorithm throughput (convergence per data sample processed) due to staleness (from missing updates).

Another challenge in applying ML algorithms, originally designed for sequential settings, to distributed systems is tuning the parameters in the algorithm itself (distinct from model parameters tuning). For example, stochastic gradient descent (SGD), which has been applied to a wide range of large-scale data-parallel ML applications such as ads click through rate (CTR) prediction [35], deep neural networks [15], and collaborative filtering [25], is highly sensitive to the step size that modulates the gradient magnitude. Many existing distributed ML systems requires manual tuning of step size by the users [32], while some provides heuristic tuning mechanism [19]. While increased freshness of model parameter values may speed up convergence, the model parameter has to be tuned accordinly to fully exploit the benefit of improved parameter freshness.

This paper presents Bösen, a system designed to achieve maximal network communication efficiency by incorporating knowledge of network bandwidth and ML heuristics. Through bandwidth management, Bösen fully utilizes, but never exceeds, a pre-specified amount of network bandwidth when communicating ML sub-updates and up-to-date model parameters. Bösen also satisfies bounded staleness model's requirements and thus inherits its formal convergence guarantees. Moreover, our solution maximizes communication efficiency by prioritizing network bandwidth for messages most significant for algorithm progress, which further enhance algorithm throughput for a fixed network bandwidth budget. We demonstrate the effectiveness of managed communication on Matrix Factorization with SGD and Topic Modeling (LDA) with Gibbs sampling on up to 1024 core compute cluster. These applications represent methods from optimization and sampling, as well as large latent-space models.

To our knowledge, Bösen provides the first distributed implementation of Adaptive Revision [34], a principled step-size tuning algorithm that takes communication delays into consideration. Adaptive Revision achieves theoretical convergence guarantees by adaptively adjusting the step size to account for errors caused by delayed updates. Our experiments on Matrix Factorization show orders of magnitude of improvements in the number of iterations needed to achieve convergence, compared to the best hand-tuned fixed-schedule step size. With Adaptive Revision, algorithm throughput of SGD can be significantly improved by Bösen's managed communication without manual parameter tuning.

2 Background

2.1 Data Parallelism

Although ML programs come in many forms, such as topic models, deep neural networks, and sparse coding (to name just a few), almost all seek a set of parameters (typically a vector or matrix) to a global model A that best summarizes or explains the input data $\mathscr{D}$ as measured by an explicit objective function such as likelihood or reconstruction loss [8]. Such problems are usually solved by *iterative-convergent* algorithms, many of which can be abstracted as the following additive form:

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

where, $A^{(t)}$ is the state of the model parameters at iteration t, and $\mathscr{D}$ is all input data, and the update function $\Delta()$ computes the model updates from data, which are added to form the model state of the next iteration. This operation repeats itself until A stops changing, i.e., converges — known as the fixed-point (or attraction) property in optimization.

Furthermore, ML programs often assume that the data $\mathscr{D}$ are *independent and identically distributed (i.i.d)*; that is to say, the contribution of each datum D_i to the estimate of model parameter A is independent of other data D_j[1]. This in turn, implies the validity of a *data-parallel* scheme *within each iteration* of the iterative convergent program that computes $A^{(t)}$, where data $\mathscr{D}$ is split over different threads and worker machines, in order to compute the update $\Delta(A^{(t-1)}, \mathscr{D})$ based on the globally-shared model state from the previous iteration, $A^{(t-1)}$. Because $\Delta()$ depends on the latest value of A, workers must communicate updates $\Delta()$ to each other. It is well-established that fresher model parameters (A or Δ) improves the *per-data-sample* progress of ML algorithms, i.e. each data sample processed makes more progress towards the final solution [27].

Mathematically, when an ML program is executed in a *perfectly synchronized* data-parallel fashion on P workers, the iterative-convergent Eq. 1 becomes

$$A^{(t)} = A^{(t-1)} + \textstyle\sum_{p=1}^{P} \Delta(A^{(t-1)}, \mathscr{D}_p), \tag{2}$$

where $\mathscr{D}_p$ is the data partition allocated to worker p (distinct from D_i, which means the i-th single data point).

[1]More precisely, each $\mathscr{D}$ is *conditionally independent* of other D_j given knowledge of the true A.

In each iteration, a subset of data $\mathscr{D}_p$ is used for computing $\Delta()$ on worker p; $\mathscr{D}_p$ is often called a "mini-batch" in the ML literature, and can be as small as one data sample D_i. Δ may include a "step size" common in gradient descent algorithms which requires manual or automatic tuning for the algorithm to work well.

2.2 Parameter Server and Synchronization Schemes

A Parameter Server (PS) is essentially a distributed shared memory system that enables clients to easily share access to the global model parameters via a key-value interface, in a logically bipartite server-client architecture for storing model parameter A and data $\mathscr{D}$. Physically, multiple server machines, or co-location of server/client machines can be used to balance hardware resource utilization. Typically, a large number of clients (i.e., workers) are deployed, each storing a subset of the big data. A data parallel ML program can be easily implemented on the PS architecture, by letting the execution of the update $\Delta()$ take place only on each worker over data subsets therein, and the application of the updates to model parameters (e.g. addition) take place on the server. This strategy has been widely used in a variety of specialized applications ([4, 15, 10]) as well as general-purpose systems ([12, 29, 37]). Some advanced algorithms (such as Adaptive Revision) require server-side computation to derive the additive delta update from the $\Delta()$'s computed by workers, and Eq. 2 can be generalized to:

$$A^{(t)} = A^{(t-1)} + \textstyle\sum_{p=1}^{P} f(\Delta(A^{(t-1)}, \mathscr{D}_p)), \tag{3}$$

where $\Delta()$ is computed by worker and $f()$ is computed on server.

A major utility of the PS system is to provide a vehicle to run data-parallel programs using *stale* parameters $\tilde{A}_p \approx A$ that reside in each client p, thereby trading off expensive network communication (between client-server) with cheap CPU-memory communication (of cached parameters within a client) via different synchronization mechanisms. For example, under *Bulk Synchronous Parallelization* (BSP), $\tilde{A}_p$ is made precisely equal to $A^{(t-1)}$, so that Eq. 2 is faithfully executed and hence yields high-quality computation. However, BSP suffers from the well-studied *stragglers* problem [7, 6, 11, 12], in which the synchronization barrier between iterations means that the computation proceeds only at the rate of the slowest worker in each iteration. Another example is *Total Asynchronous Parallelization* (TAP) where $\tilde{A}_p$ is whatever instantaneous state in the server results from the aggregation of out-of-sync (hence inconsistent) updates from different workers. Although highly efficient and sometimes accurate, TAP does not enjoy a theoretical guarantee and can diverge.

In this paper, we explore a recently proposed middle ground between BSP and TAP, namely *bounded staleness parallelization* [24, 29], in which each worker maintains a possibly *stale* local copy of A, where the degree of staleness is bounded by a target staleness threshold S, i.e., no worker is allowed to be more than S clock units ahead of the slowest worker. This idea has been shown experimentally to improve times to convergence [12, 29], and theoretical analysis on convergence rates and error bounds is beginning to be reported for some cases [24, 14].

Although ML algorithms may tolerate bounded staleness and still achieve correct convergence, the algorithm performance (i.e., convergence per data sample processed) may suffer from staleness, resulting in suboptimal performance. Existing implementations of distributed ML systems or applications propagate model updates and fetch up-to-date model parameters when certain amount of computation has been done or when it is needed for computation to proceed under the consistency requirements. For example, previous implementations of bounded staleness bundled communication with the staleness threshold and synchronization barriers or relied on explicit application control of communication. In light of that increased parameter value freshness improves algorithm performance (i.e. per-data-sample convergence rate), our goal is to maximize parameter value freshness under limited hardware resource by maximizing commuication efficiency, in order to speed up ML application convergence. This is achieved by bandwidth-driven communication for model

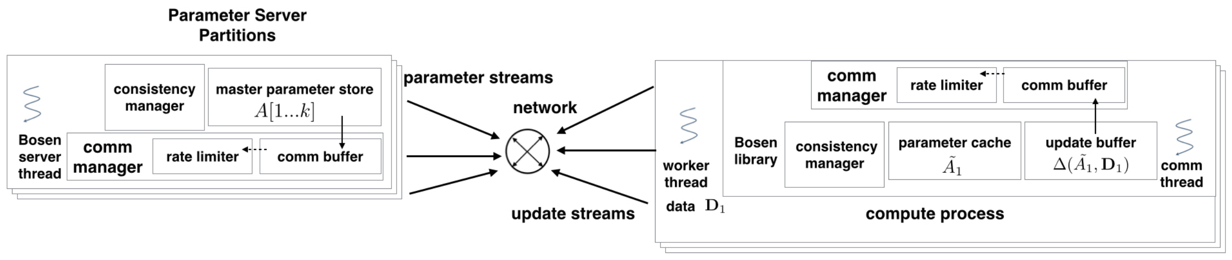


Figure 1: Parameter Server Architecture

`Get(key), RowGet(row_key)`	Get a parameter or a row indexed by 'key' or 'row_key'
`Inc(key, delta), RowInc(row_key, key_set, delta_set)`	Add `delta` to 'key' or add a set of deltas to a row of parameters
`Clock()`	Signal end of iteration

Table 1: Bösen Client API

updates and model parameters that fully utilizes the bandwidth budget and prioritization that allocates the limited bandwidth to messages that contributes most to convergence. Our sophisticated communication management is provided under a simple storage abstraction that asbtracts communication away from application developer. For ease of reference in the rest of the paper, we call this system "Bösen", in honor of one author's favorite musical instrument.

3 The *Bösen* PS Architecture

Bösen is a parameter server architecture with a ML-consistent, bounded-staleness parallel scheme and bandwidth-managed communication mechanisms. It realizes bounded staleness consistency, which offers theoretical guarantees for iterative convergent ML programs, while enjoying high iteration throughput that is better than BSP and same as TAP systems in common cases. Additionally, Bösen transmits model updates and up-to-date model parameters without exceeding a bandwidth limit, while making better use of the bandwidth by scheduling the bandwidth budget based on the contribution of the messages to algorithm progress — thus improving per-data-sample convergence compared to an agnostic communication strategy.

3.1 API and Bounded Staleness Consistency

Bösen PS consists of a **Client Library** and **Server Partitions** (Figure 1); the former provides the Application Programming Interface (API) for reading and updating model parameters, and the latter stores and maintains the model *A*. In terms of usage, Bösen closely follows other key-value stores: once a programmer links her ML program against the client library, any worker thread in any program process may read and update model parameters concurrently, whose master copy is stored on the server partitions. The user runs a Bösen ML program by invoking as many server partitions and ML application **compute processes** (which use the client library) as needed, across a cluster of machines.

Bösen's API abstracts consistency management and networking operations away from the user, and presents a simple key-value interface (Table 1). `Get()` is used to read parameters and `Inc()` is used to increment a parameter by some delta. To maintain consistency, the user signals the end of a unit of work (i.e. refered to as a "clock tick" or an "iteration") via `Clock()`. In order to exploit locality in ML applications and thus amortize the overhead of operating on concurrent data structures and network messaging, Bösen allows applications to statically partition the parameters into batches called *row*s – a row is a set of parameters that are usually accessed together. A row is also the unit of communication between client and server: `RowGet()` is provided to read a row by its key, and `RowInc()` applies a set of deltas to multiple elements in a row.

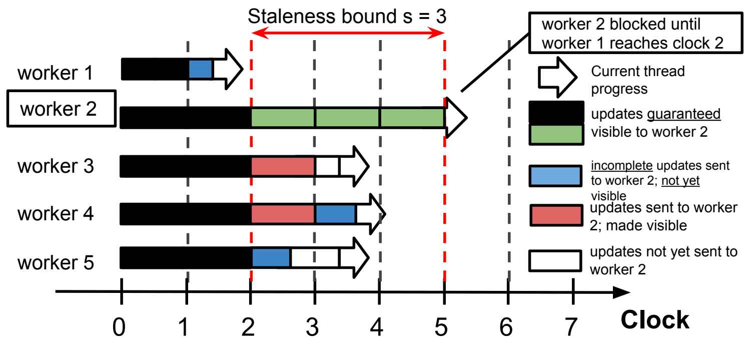

Staleness bound s = 3
worker 2 blocked until worker 1 reaches clock 2
worker 1
worker 2
worker 3
worker 4
worker 5
Current thread progress
updates guaranteed visible to worker 2
incomplete updates sent to worker 2; not yet visible
updates sent to worker 2; made visible
updates not yet sent to worker 2
0
1
2
3
4
5
6
7
Clock

Figure 2: Consistency guarantees for bounded staleness. We show a system with 5 workers, using staleness $S = 3$; here "iteration t" refers to progress made between x-axis (clock) ticks $t-1$ and t. From the perspective of worker 2, all updates from iteration 1 are visible (colored black). However, the Bösen consistency manager blocks worker 2 during iteration $t = 6$, because worker 2 has not received worker 1's updates from iteration $t - S - 1 = 2$. Bösen continuously propagates updates before they are strictly required, such as the red updates from iteration 3 at workers 3 and 4 — these updates have been made visible to worker 2 even though it has not reached iteration 7. Blue bars are updates that have been propagated to worker 2, but Bösen does not make them visible to worker threads yet, because the iteration is still incomplete.

Bösen supports user-defined "stored procedures" to be executed on each server—these can be used to alter the default increment behavior of `Inc()` and `RowInc()` (see Sec 3.4).
Bösen enforces **bounded staleness** on model parameter accesses. Under bounded staleness, all workers are expected to execute the same number of iterations (i.e. clock ticks) of computation. During its computation, a worker updates the shared records via accumulative and associative updates, and thus the updates can be applied in arbitrary order. An update is refered to as "missing to worker p" if it is generated in the past or concurrently according to logical time but is not visible to worker p. The difference between the age of the oldest missing update and the worker's current clock number is refered to as staleness of that parameter. Bounded staleness as its name suggests, bounds the staleness of all parameters, that is, it gurantees that for a worker at logical clock T, updates that are older than $T - S - 1$ must be visible, where S is a user-defined parameter refered to as *staleness threshold* (Figure 2). Theoretical convergence guarantees of ML algorithms under bounded staleness have been explored in [24, 14]. Bounded staleness becomes Bulk Synchronous Parallel (BSP) when the staleness threshold S is configured to 0 and workers are guaranteed to not see others' updates until the end of an iteration.

3.2 System Architecture

3.2.1 Client library

The client library provides access to model parameters A stored on the server partitions, and also caches them for faster access, while cooperating with server partitions in order to maintain consistency guarantees and manage bandwidth. This is done through three components: (1) a *parameter cache* that caches a partial (or complete) image of the model, $\tilde{A}$, at the client, in order to serve read requests made by worker threads; (2) an *update buffer* that buffers updates computed by worker threads and applied via `Inc()` or `RowInc()`; (3) a group of *client library communication threads* (distinct from worker threads) that perform synchronization of the local model cache and buffered updates with the master copy of server partitions.

The parameters cached at a client are hash partitioned among the client library threads, and each thread is allowed to access only its own parameter partition to minimize lock contention. The client parameter cache and update buffer allow concurrent reads and writes from worker threads, and client library threads and locks are needed for concurrency control. Thus batched access via *row* is encouraged to ammortize the overhead.

Parameters in the same row share one lock for accesses to their parameter caches, and one lock for accesses to their update buffers. Similar to [13], the cache and buffer use static data structures, and pre-allocate memory for repeatedly accessed parameters to minimize the overhead of maintaining a concurrent hash table.

When serving `Get()` or `RowGet()` from worker threads, the client parameter cache is searched first, and a read request is sent to the corresponding server partition only if the requested parameter is not in the cache. The read request registers a callback on the server partition. Via the callback, the server partition may update the clients' cache whenever parameter values are updated.

The **client consistency manager** enforces bounded staleness consistency by blocking worker thread reads if locally cached parameter value is too stale. When a parameter value is found in the cache, its age (age of the oldest missing update minus 1) is compared against the clock tick nunmber of the accessing worker thread. When and only when the parameter value's age is large enough, it is returned to the worker thread, otherwise the worker is blocked waiting for more up-to-date parameter values to be received from server partitions. The server partition guarantees to eventually send the up-to-date parameter values to clients so the worker threads do not block indefinitely.

When `Inc()` and `RowInc()` is invoked, updates are inserted into the update buffer, and, optionally, the client's own parameter cache is also updated. The client communication thread may send the buffered updates to server partitions when spare bandwidth is available, in order to improve freshness of the parameter values.

When a clock tick is completed, the worker thread notifies the client library by calling `Clock()`. Once all worker threads in the compute process complete a clock tick, the client threads read all model updates from the buffer and communicate them to the server partitions. The client threads send a clock message to each server partition, notifying the server partition the completion of a certain clock tick, once all model updates generated in that clock tick have been sent to the server partition. The clock message allow server partition to determine whether or not the master copy of parameters have contained all updates up to certain clock tick.

3.2.2 Server partitions

The master copy of the model's parameters, A, is hash partitioned, and each partition is assigned to one server thread. The server threads may be distributed across multiple server processes and physical machines. As model updates are received from client processes, the addressed server thread updates the master copy of its model partition. When a client read request is received, the corresponding server thread registers a callback for that request; the server thread responds to a read request only if the master copy of the parameter is fresh enough. Once a server thread has applied all updates from all clients for a given unit of work, it walks through its callbacks and sends up-to-date model parameter values, in order to ensure progress of all workers. The server thread may send fresh parameter values eariler

3.2.3 Fault tolerance

Bösen provides fault tolerance by checkpointing the server model partitions; in the event of failure, the entire system is restarted from the last checkpoint. A valid checkpoint contains the model state *strictly* right after iteration t — the model state includes all model updates generated before and during iteration t, and excludes all updates after the t-th `PS.Clock()` call by any worker thread. With bounded staleness, clients may asynchronously enter new iterations and begin sending updates; thus, whenever a checkpointing clock event is reached, each server model partition will copy-on-write protect the checkpoint's parameter values until that checkpoint has been successfully copied externally. Since taking a checkpoint can be slow, a checkpoint will not be made every iteration, or even every few iterations. A good estimate of the amount of time between taking checkpoints is $\sqrt{2T_sT_f/N}$ [42], where T_s is the mean time to save a checkpoint, there are N machines involved and T_f is the mean time to failure (MTTF) of a machine, typically estimated as the inverse of the average fraction of machines that fail each year.

3.3 Bandwidth-Driven Communication

Bösen differs from other ML systems in its intermachine network management, whose purpose is to improve ML per-data-sample convergence through careful use of network bandwidth in communicating model updates/parameters. Bandwidth management is complementary to consistency management: the latter prevents worst-case behavior from breaking ML consistency (correctness), while the former improves convergence time (speed).

The bandwidth manager has two objectives: (1) communicate model updates and dirty model parameters as quickly as possible *without* overusing the network bandwidth budget (full network utilization); and (2) allocate network bandwidth according to the messages' contribution to convergence. The first objective is achieved via a *rate limiter*, while the second is achieved by choosing one of several prioritization strategies.

3.3.1 Rate limiting

In Bösen, each client and server partition is allocated with a bandwidth budget, set by user. When idle, the communication thread periodically queries the rate limiter for availability of bandwidth. When and only when spare bandwidth is available, the communication thread sends out model updates or dirty model parameters. The rate limiter's response also include the maximum allowed send size in order to control burstinessThe message sizes are relayed back to the bandwidth manager, so it can compute network availability. Implementation-wise, the Bösen bandwidth manager uses a *leaky bucket* model (leaky bucket as a queue): the bucket records the number of bytes sent, and is constantly drained at rate equal to the bandwidth limit. New messages may only be sent if they can fit in the bucket, otherwise they must wait until it has been drained sufficiently. We now discuss some additional refinements to our rate-limiting strategy.

Coping with network fluctuations: In real cloud data centers with multiple users, the available network bandwidth may fluctuate and fail to live up to the bandwidth limit B. Hence, the Bösen bandwidth manager regularly checks to see if the network is overused, by monitoring how many messages were sent without acknowledgement in a recent time window (i.e. message non-delivery). If too many messages fail to be acknowledged, the bandwidth manager assumes that the network is overused, and waits until the window becomes clear before permitting new messages to be sent.

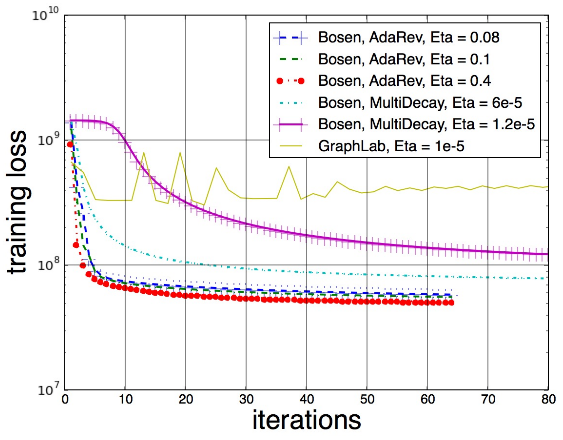
3.3.2 Update prioritization

Bösen spends available bandwidth on communicating information that contributes the most to convergence. For example, gradient-based algorithms (including Logistic Regression) are iterative-convergent procedures in which the fastest-changing parameters are often the largest contributors to solution quality — in this case, we prioritize communication of fast-changing parameters, with the largest-magnitude changes going out first.

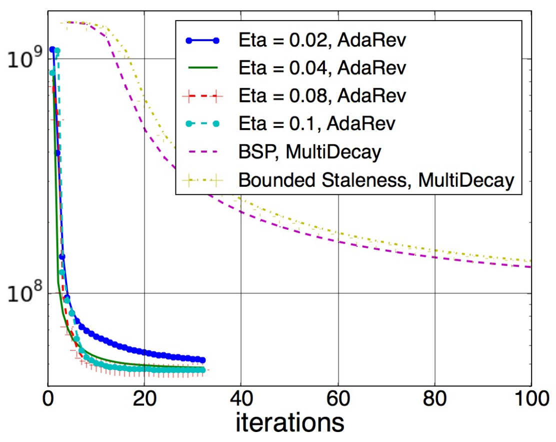
Bösen's bandwidth manager supports multiple prioritization strategies, all of which depend on ordering the *communication buffer*. The simplest possible strategies are **Round-Robin**, where communications threads send out the oldest rows in the buffer first, and **Randomized**, where communications threads send out randomly-chosen rows. These strategies are baselines; better strategies prioritize according to convergence progress. We study the following two strategies.

Absolute magnitude prioritization: Updates/parameters are sorted by their accumulated change in the buffer, $|\delta|$ — this strategy is expected to work well for gradient-based algorithms.

Relative magnitude prioritization: Same as absolute magnitude, but the sorting criteria is $|\delta/a|$, i.e. the accumulated change normalized by the current parameter value, a. For some ML problems, relative change



(a) Compare Bösen SGD MF with GraphLab SGD MF



(b) Adaptive Revision (AdaRev) vs. Multiplicative Decay

Figure 3: Adaptive Revision Effectiveness. (a) Adaptive Revision vs. Multiplicative Decay on 1 node; (b) Adaptive Revision vs. Multiplicative Decay on 8 nodes. AdaRev uses BSP, and Eta denotes the inital step size. MultiDecay used their respective optimal initial step size, which was chosen as the one that achieves fastest convergence without oscillating objective value within the first 40 iterations.

$|\delta/a|$ may be a better indicator of progress than absolute change $|\delta|$. In cases where $a = 0$ or is not in the client parameter cache, we fall back to absolute magnitude.

3.4 Adaptive Step Size Tuning

Many data-parallel ML applications use the stochastic gradient descent (SGD) algorithm, whose updates are gradients multiplied by a scaling factor, referred to as "step size" and typically denoted as η. The update equation is thus:

$$A^{(t)} = A^{(t-1)} + \textstyle\sum_{p=1}^{P} \eta_p^{(t-1)} \nabla(A^{(t-1)}, \mathscr{D}_p). \tag{4}$$

The SGD algorithm performance is very sensitive to the step size used. Existing distributed SGD applications (i.e., GraphLab's SGD MF, MLlib's SGD LR, etc.) apply the same step size for all dimensions and decay the step size each iteration according to a fixed schedule. Achieving ideal algorithm performance requires a great amount of tuning to find an optimal initial step size. When using a fixed schedule, the parameter server workers compute the gradients and scale them with the predetermined step size, and the parameters are updated via addition.

There exist principled strategies for adaptively adjusting the stochastic gradient step size, reducing sensitivity to the initial step size $\eta^{(1)}$ and they may achieve good algorithm performance using any step size from a reasonable range of step sizes. The popular Adaptive Gradient (AdaGrad) strategy [17] adaptively adjusts the step size for each coordinate separately and converges better than using a fixed schedule. However, it was designed for serial execution, and the algorithm performance is still sensitive to the degree of parallelism and network delays. The recently-developed Adaptive Revision (AdaRevision) strategy [34] is theoretically valid under distributed execution, provided that the staleness of each update is bounded (which is guaranteed by bounded staleness, but not under TAP). It was shown analytically and via single machine simulation that AdaRevision performs better than AdaGrad when delay is present.

With AdaRevision, computing the step size for each model dimension requires: 1) $\bar{g}$, the accumulated sum of all historical gradients on that dimension when the update is applied (implicitly assumes updates are applied in a serializable order) and 2) $\bar{g}^{old}$, the value of $\bar{g}$ on that dimension that corresponds to the parameter value used for computing the gradient. When using AdaRevision in a parameter server, the workers compute gradients which are sent to servers to update the parameters. The updates can only be applied on a server as $\bar{g}$ is known only by servers. The client parameter cache is therefore read-only to workers, i.e. the updates are not seen until fresh parameters are received from a server. While a naive implementation may let clients fetch $\bar{g}$ (which is generally not needed for computing gradients) when fetching parameters from a server and send it back when sending the derived gradients, B̈osen supports **versioning** of parameters to reduce the communication overhead. The AdaRevision algorithm is implemented as a **user-defined stored procedure** on the servers.

Parameter Versioning. The server maintains a version number per dimension (i.e. parameter), which is incremented every time the parameter is updated. In B̈osen, as updates are applied in the unit of a row, there is one version number per row. The version number is sent to clients along with the corresponding parameters and is stored in the client's parameter cache. The update computed (by a worker) for parameter i is tagged with the version number of parameter i in the parameter cache. The updates tagged with the same version number are aggregated via addition as usual, but updates with different version numbers are stored separately.

User-defined Stored Procedure. The user-defined stored procedure (UDF) contains a set of user-implemented functions that are invoked at various events to control the server's behavior. We illustrate the use of user-defined stored procedures via our implementation of the AdaRevision algorithm. Firstly, when parameters are created, the UDF creates and initializes $\bar{g}$ along with other bookkeeping variables needed to compute the step size.

When a row is to be sent to clients, the UDF makes a copy of the current values of $\bar{g}$'s for the parameters in that row and inserts this into a hash table indexed by the row ID and the version number. The copy is freed when no client cache still contains this version of the parameter. In order to bound the overall memory usage, the UDF imposes an upper limit on the number of copies that can be kept. Bandwidth-triggered communication is canceled upon exceeding this limit. Clock-driven communication generates only a bounded number of copies due to the staleness threshold. When gradients are received by the server, the UDF is invoked with the gradient value and version number. The UDF computes the proper step size and applies the update.

Effectiveness & Comparison We firstly conduct a single-node experiment to verify the effectiveness of our algorithm implementation. We used the matrix factorization (MF) application on the Netflix data set and rank = 50. We run one Bosen MF process using BSP on a single 64-core node in cluster-B (Sec. 5) with adaptive revision (AdaRev) and with multiplicative decay respectively (MultiDecay), each using various step sizes. On a single node, MF-MultiDecay workers share updates with local workers via the shared parameter cache. However, MF-AdaRev may observe the updates only at the end of an iteration due to the read-only parameter cache. It is shown in Fig. 3a that MF-MultiDecay shows better convergence with careful step size tuning, however MF-AdaRev constantly outperforms MF-MultiDecay with a wide range of initial step sizes. We also ran GraphLab's SGD MF using a range of initial step sizes from $1e-4$ to $1e-6$: the convergence behaves similarly and does not converge to an acceptable solution. It should also be noted that while it took GraphLab 4400 seconds to perform 80 iterations (i.e. data passes), it took only 600 seconds on Bosen.

We compare our implementation of AdaRevision with multiplicative decay of step size using 8 nodes in cluster-B. As shown in Fig. 3b, we observe that AdaRevsion significantly reduces the number of iterations to achieve convergence. Also, varying the initial step size within a reasonable range makes little difference to the algorithm performance. Under AdaRevision, the per-iteration run time is increased by nearly 30% for MF due to the computation of step size.

4 ML program instances

We study how consistency and bandwidth management improves the convergence time and final solution quality for four commonly-used data-parallel ML programs: matrix factorization, logistic regression, multiclass logistic regression, and topic modeling.

Matrix Factorization (MF) MF is commonly used in recommender systems, such as recommending movies to users on Netflix. Given a matrix $D \in \mathbb{R}^{M \times N}$ which is partially filled with observed ratings from M users on N movies, MF factorizes D into two factor matrices L and R such that their product approximate the ratings: $D \approx LR^T$. Matrix L is M-by-r and matrix R is N-by-r where $r << \min(M,N)$ is the rank which determines the model size (along with M and N). Our experiments use $r \geq 1000$ as higher ranks lead to more accurate recommendations [45]. We partition observations D to workers and solve MF via stochastic gradient descent (SGD) as illustrated in Algorithm 1.

Algorithm 1 Matrix Factorization via SGD

```
Function threadWorker(p):
  For iteration $t = 1..T$:
    For each observed entry $D_{ij}$ at this thread $p$:
      $\boldsymbol{L}_{i*} \leftarrow$ PS.Get('$L_{i*}$')   // $\boldsymbol{L}_{i*}$ is $i$-th row of $\boldsymbol{L}$
      $\boldsymbol{R}_{*j} \leftarrow$ PS.Get('$R_{*j}$')   // $\boldsymbol{R}_{*j}$ is $j$-th col of $\boldsymbol{R}$
      $e_{ij} = D_{ij} - \boldsymbol{L}_{i*}\boldsymbol{R}_{*j}$
      Lgrad = $(e_{ij}\boldsymbol{R}_{*j}^T - \lambda \boldsymbol{L}_{i*})$
      Rgrad = $(e_{ij}\boldsymbol{L}_{i*}^T - \lambda \boldsymbol{R}_{*j})$
      PS.Inc('$L_{i*}$', $-\eta_t$*Lgrad)     // $\eta_t$ is step-size
      PS.Inc('$R_{*j}$', $-\eta_t$*Rgrad)
    PS.Clock()
```

Logistic Regression (LR) & Multiclass Logistic Regression (MLR) Logistic Regression is a classical method used in large-scale classification [43], natural language processing [20], and ad click-through-rate prediction [35], among others. It models the likelihood of a d-dimensional observation $\boldsymbol{x}$ being from class 1 with $p(\text{class=1}|\boldsymbol{x}) = \frac{1}{1+\exp(-\boldsymbol{w}^T\boldsymbol{x})}$, where $\boldsymbol{w}$ is the d-dim model parameters. Multiclass Logistic Regression generalizes LR to multi-way classification, seen in large-scale text classification [31], and the ImageNet challenge involving 1000 image categories (i.e. each labeled images comes from 1 out of 1000 classes) where MLR is employed as the final classification layer [26]. MLR models the likelihood that a d-dimensional data instance $\boldsymbol{x}$ belonging to class j is $p(\text{class} = j|\boldsymbol{x}) = \frac{\exp(\boldsymbol{w}_j^T\boldsymbol{x})}{\sum_{k=1}^J \exp(\boldsymbol{w}_k^T\boldsymbol{x})}$ where J is the number of classes and $\boldsymbol{w}_1, ..., \boldsymbol{w}_J$ are the d-dim weights associated with each class (thus making the model size $J \times d$). We solve both LR and MLR using stochastic gradient descent (SGD). Due to space, we only provide pseudo-code for MLR in Algorithm 2. Note that in both LR and MLR each gradient update is "dense" as it modifies all parameters (d for LR, $J \times d$ for MLR).

Topic Modeling (Latent Dirichlet Allocation) *Topic Modeling (LDA)* is an unsupervised method to uncover hidden semantics ("topics") from a group of documents, each represented as a multi-set of tokens (bag-of-words). In LDA each token w_{ij} in the document is associated with a latent topic assignment $z_{ij} \in \{1, ..., K\}$, where K is the number of topics, i indexes the documents and j indexes each token in the document. We use Gibbs sampling to infer the topic assignments z_{ij}. (Specifically, we use the SparseLDA variant in [41] which is also used in YahooLDA [39] that we compare with.) The sampling step requires three parameters, known as "sufficient statistics": (1) document-topic vector $\theta_i \in \mathbb{R}^K$ where θ_{ik} the number of topic assignments within document i to topic $k = 1...K$; (2) word-topic vector $\phi_w \in \mathbb{R}^K$ where ϕ_{wk} is the number

Algorithm 2 Multi-class Logistic Regression via SGD

```
Function threadWorker(p):
  For iteration $t = 1..T$:
    For each data minibatch $m$ at this thread $p$:
      For class $j = 1..J$:
        $\boldsymbol{w}_j \leftarrow$ PS.Get('$w_j$')
      For each observation $(\boldsymbol{x}_n, y_n)$ in minibatch $m$:
        For class $j = 1..J$:
          grad$_j$ = $(\text{softmax}(\boldsymbol{x}_n)_j - \mathbb{I}(j = y_n))\boldsymbol{x}_n$
          note:  softmax$(\boldsymbol{x})_j := \frac{\exp(\boldsymbol{w}_j^T \boldsymbol{x})}{\sum_{k=1}^{J} \exp(\boldsymbol{w}_k^T \boldsymbol{x})}$
          PS.Inc('$w_j$', $-\eta_t$*grad$_j$) // $\eta_t$ is step-size
    PS.Clock()
```

Application	Dataset	Workload	Description	# Rows	Row Size	Data Size
SGD MF	Netflix	100M ratings	480K users, 18K movies, rank=400	480K	1.6KB	1.3GB
SGD MF	Netflix×256	25.6B ratings	7.68M users, 288K movies, rank=100	7.68M	1.6KB	332GB
LDA	NYTimes	99.5M tokens	300K documents, 100K words 1K topics	100K	dynamic	0.5GB
LDA	ClueWeb10%	10B tokens	50M webpages, 160K words, 1K topics	160K	dynamic	80GB

Table 2: ML applications and datasets

of topic assignments to topic $k = 1, ..., K$ for word (vocabulary) w across all documents; (3) $\tilde{\phi} \in \mathbb{R}^K$ where $\tilde{\phi}_k = \sum_{w=1}^{W} \phi_{wk}$ is the number of tokens in the corpus assigned to topic k. We partition the corpus (w_{ij}, z_{ij}) to worker nodes (i.e each node has a set of documents), and compute θ_i on-the-fly before sampling tokens in document i (thus we do not store θ). We store ϕ_w and $\tilde{\phi}$ as rows in PS. The pseudo-code is in Algorithm 3.

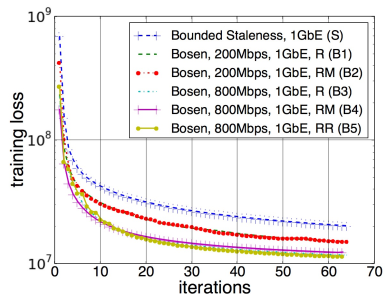
Algorithm 3 LDA via Gibbs Sampling

```
Input: $w_{ij}, z_{ij}$ partitioned to worker's local memory, $\phi_w$ stored as rows in PS, Number of topics $K$, Hyperparameters
$\alpha, \beta$ (Dirichlet priors)
Function threadWorker(p):
  For iteration $t = 1..T$:
    For each document $i$ at this thread $p$:
      Compute $\theta_i$ from $z_{ij}, j = 1, ..., d_i$ ($d_i$ is doc length)
      For each token $w_{ij}, z_{ij}$ in doc $i$:
        $\phi_{w_{ij}} \leftarrow$ PS.Get('$\phi_{w_{ij}}$')
        $k_1 \leftarrow z_{ij}$
        $k_2 \leftarrow$ Gibbs$(\theta_i, \phi_{w_{ij}, \tilde{\phi}}, \alpha, \beta)$
        If $k_1 \neq k_2$
          PS.Inc('$\phi_{w_{ij}, k_1}$',-1)
          PS.Inc('$\phi_{w_{ij}, k_2}$',+1)
          PS.Inc('$\tilde{\phi}_{k_1}$',-1)
          PS.Inc('$\tilde{\phi}_{k_2}$',+1)
          Update $\theta_i$ based on $k_1, k_2$
    PS.Clock()
```

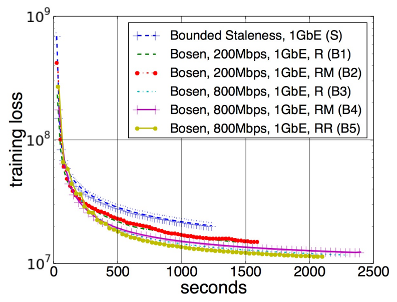
5 Evaluation

We evaluate Bösen on several data-parallel machine learning algorithms described above: matrix factorization (SGD MF) and latent dirichlet allocation (LDA).

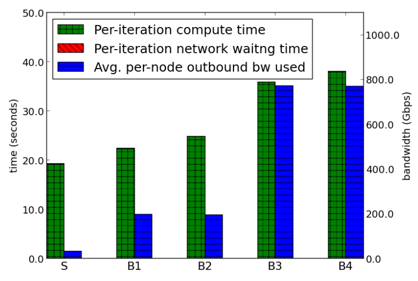
Cluster setup: Most of our experiments were conducted on a research cluster (cluster-A) consisting of 200 high-end computers running Ubuntu 14.04. Our experiments used different number of computers, varying



(a) Convergence per iteration

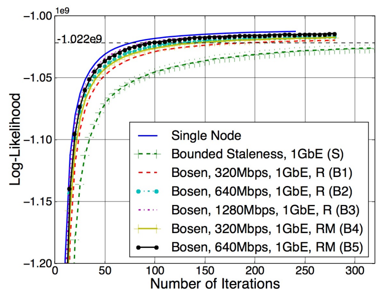


(b) Convergence per second

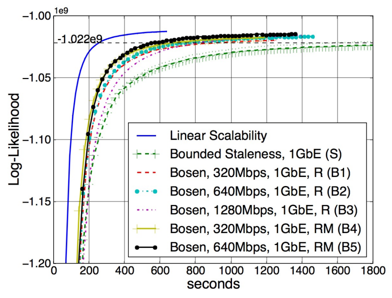


(c) Execution time & bandwidth

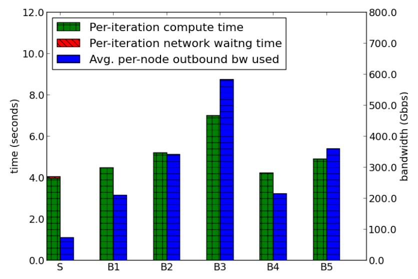
Figure 4: Matrix Factorization under Managed Communication



(a) Convergence per iteration



(b) Convergence Per Second



(c) Execution time & bandwidth

Figure 5: LDA Topic Modeling under Managed Communication

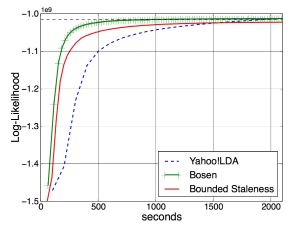
from 8 to 64. Each machine contains 4 $\times$ quad-core AMD Opteron 8354 CPUs (16 physical cores per machine) and 32GB of RAM. The machines are distributed over multiple racks and connected via two networks: 1 Gb Ethernet and 20 Gb Infiniband. A few experiments were conducted on another research cluster (cluster-B). Each machine contains 4 $\times$ 16-core AMD Opteron 6272 CPUs (64 physical cores per machine) and 128GB of RAM. The machines are distributed over two racks and connected to two networks: 1 GbE and 40 GbE. In both clusters, every machine is used to host Bösen server, client library, and worker threads (i.e. servers and clients are collocated and evenly distributed).

ML algorithm setup: In all ML applications, we partition the data samples evenly across the workers. Unless otherwise noted, we adopted the typical BSP configuration and configured 1 logical clock tick (i.e. iteration) to be 1 pass through the worker's local data partition. See Table 2 for dataset details.

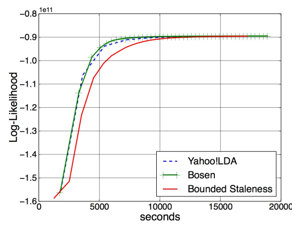
Performance metrics:

Fast convergence comes from two sources: (1) high convergence progress per iteration, measured by changes in ML algorithm's objective value (i.e. loss function) towards its asymptote per unit of computation (e.g. number of data samples processed, or number of iterations[2]), and (2) high iteration throughput, measured as the average time spent per iteration. To measure the absolute ML program convergence speed, we report the objective value versus time (seconds).

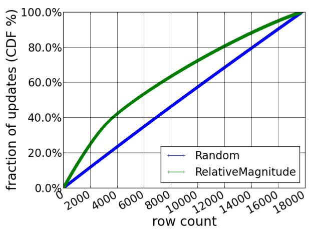
[2]In one iteration we compute parameter updates using each of the N data samples in the dataset exactly once, regardless of the number of parallel workers. With more workers, each worker will touch fewer data samples per iteration.



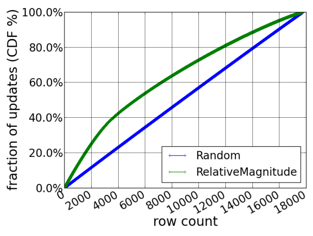
(a) NYTimes



(b) ClueWeb10%



(c) Update communication



(d) Parameter communication

Figure 6: (a), (b): Comparing Bösen LDA with Yahoo!LDA on two datasets; (c), (d): Update and parameter communication frequency CDF in Matrix Factorization

5.1 Communication Management

We evaluate the Bösen communication management schemes. We show that algorithm performance improves with more immediate communication of updates and messages. Moreover, proper bandwidth allocation based on the importance of the messages may achieve better algorithm performance with less bandwidth consumption.

Experiment setup: Fig. 4 and Fig. 5 demonstrate the effectiveness of communication management for MF and LDA, respectively.

The MF experiments use 8 machines (128 processors) on cluster-A. For LDA, we use 16 machines (256 processors), except the single-node experiment. The MF experiments use the Netflix dataset; LDA experiments use the NYTimes dataset (Table 2). As entailed by the adaptive revision of step size algorithm described in Sec. 3.4, the client parameter cache is read-only to SGD MF workers; updates are scaled on the server before they are applied to client caches, and workers can observe updates only when the refreshed parameters are received from server. On the other hand, the LDA application directly applies updates to its client parameter cache so that the current update becomes visible to local workers immediately. Thus the single-node experiment of LDA shows the algorithm performance with updates seen at memory speed, which constitutes an unachievable upper bound for the distributed program.

For both MF and LDA, we compare managed communication (labelled B1, B2, ...) with non-managed communication (i.e. only the consistency manager is turned on, labelled S) to show the effectiveness of communication management. The communication management mechanism was tested with different per-node bandwidth budgets (200Mbps and 800Mbps for MF; 320Mbps, 640Mbps and 1280Mbps for LDA) and different prioritization strategies (section 3.3.2): Random (R), Round-Robin (RR), and Relative-Magnitude (RM). Each node runs the same number of communication and server threads and the bandwidth budget is evenly divided between communication and server threads.

For all experiments in this section, we fixed the queue size of the client and server thread rate limiter (100 rows for MF; 500 rows for LDA client threads and 5000 rows for LDA server threads). In all experiments, we used a staleness parameter of 2 and we found that although bounded staleness converges faster than BSP, changing the staleness level does not affect average-case performance. For MF, we used the same initial step size (0.08) for all runs and for LDA, we used the same hyper-parameters ($\alpha = \beta = 0.1$) for all runs.

Convergence speed under managed communication: Fig. 4a and Fig. 5a shows that managed-communication execution (Bösen) always achieve better progress per iteration (leftmost panels) than non-managed-communication executions (i.e. bounded staleness). While the time required per iteration (Fig. 4c, Fig. 5c) increases with more bandwidth use, the overall convergence speed (middle panels) is generally improved with increasing bandwidth usage. For example, in the case of MF, managed communication with a 200Mbps per-node bandwidth budget reached the training loss of $2.0e7$ within 27 iterations while that took bounded staleness 64 iterations, which suggests an over 50% improvement in algorithm performance. In the case of LDA, it took bounded staleness 875 iterations to reach a log-likelihood of $-1.022e9$ while only 195 iterations under managed communication with a 320Mbps per-node bandwidth budget and randomization.

Effect of additional bandwidth: Under a fixed prioritization policy, increasing the bandwidth budget consistently improves algorithm performance, due to more frequent communication of updates and up-to-date parameters. For example, in MF 4, when the bandwidth budget is increased to 800Mbps from 200Mbps, it took only 12 iterations to reach $2.0e7$ instead of 27. In LDA, the number of iterations to reach $-1.022e9$ is reduced to 120 from 195 when the bandwidth budget is increased from 320Mbps to 640Mbps, corresponding to a 1959 seconds reduction in time.

We observe diminishing returns from additional network bandwidth: for example, in MF, when network bandwidth usage is increased from 33Mbps to 200Mbps, the number of iterations needed to reach the desired training loss is reduced by 37, while that is only reduced by 15 when the bandwidth usage is increased from 200Mbps to 770Mbps.

Effect of prioritization: In the case of LDA, prioritization by Relative-Magnitude (RM) consistently improves upon Randomization (R) when using the same amount of bandwidth. For example, with 200Mbps of per-node bandwidth budget RM reduces the number of iterations needed to reach $1.022e9$ from 195 to 145.

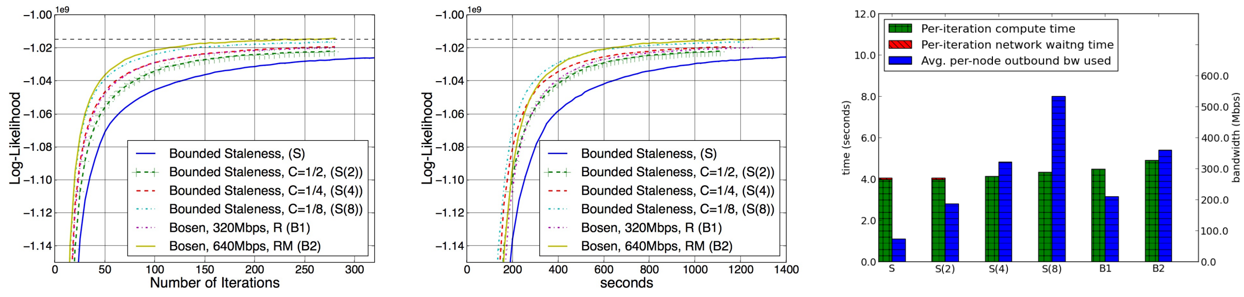
However, prioritization appears to be less effective for MF. The server computes a scaling factor to the gradient, which alters the gradient by orders of magnitude. Moreover, as the adaptive revision algorithm [34] entails, the server tends to use a larger scaling factor for small gradients, thus the gradient magnitue alone doesn't constitute a good indicator for significance.

Relative-Magnitude prioritization improves upon Random prioritization as it differentiates updates and model parameters based on their significance to algorithm performance. Thus it schedules the communication accordingly and communicates different updates and model parameters at different frequencies. Fig 6c and Fig 6d show the CDFs of communication frequency of MF's model updates and parameters respectively, in number of rows, under different policies. We observed that Relative-Magnitude and Absolute-Magnitude prioritization achieve similar effect, where a large number of communication were spent on a small subset of parameters or updates. Random and Round-Robin achieves similar effect where all parameters and updates are communicated at roughly the same frequency. We verified (not shown) that Round-Robin and Random achieve the same level of algorithm performance while Relative-Magnitude and AbsoluteMagnitdue achieve the same level of algorithm performance.

Overhead of communication management: As shown in Fig. 4c and Fig. 5c, communication management incurs compute overhead This overhead comes from processor cycles spent computing prioritization weights, sorting updates and parameters, and constructing and parsing additional messages; it also comes from increased lock contention in the client parameter cache and update buffer. With high bandwidth budgets, we observed a 100% increase in compute overhead for both MF and MLR, though the improved progress per iteration usually more than makes up for this.

Comparison with Yahoo!LDA. We compared the Bösen LDA implementation with the popular Yahoo!LDA [5] using the NYTimes and 10% of ClueWeb data set, using 1GbE and 20 Gb Inifiniband respectively. The NYTimes experiment used 16 machines for both systems and the ClueWeb experiment used 64 machines, both in cluster-B. The result is shown in Fig 6a and Fig 6b. Yahoo!LDA is an early example of parameter server specialized for the LDA application. In our experiments, servers and clients are colocated on the same machines, as in Bösen. Each machine runs one client, which has 16 worker threads for computation. The communication in Yahoo!LDA is totally asynchronous. The communication thread walks through the set of parameters and for each parameter, it sends the delta update to server and fetches the fresh parameter value. In both experiments, Bösen LDA processes roughly the same number of tokens per second as YahooLDA! does, so the result also suggests the relative performance on convergence per data sample.

NYTimes is a relatively small dataset. The application worker threads completes $3-4$ passes of the dataset while the communication thread sweeps through the parameter set only once. That is one complete



(a) Convergence per iteration (b) Convergence per second (c) Runtime breakdown and bandwidth

Figure 7: Comparing Bosen with simply tuning clock tick size

synchronization per $3-4$ passes through the dataset, equivalent to an effective staleness of $3-4$ in Bösen. Bösen's communication thread is a lot more efficient, which performs 1 complete synchronization per pass of the data under bounded staleness without bandwidth management, while synchronizes more than that under bandwidth management. Thus we observe that Bösen bounded staleness converges faster than Yahoo!LDA while bandwidth management further improves upon that. On the other hand, ClueWeb is a $100\times$ larger dataset, while the number of parameters is only $1.6\times$ more. Yahoo!LDA completes 3 synchronizations per pass of the data, converging faster than bounded staleness. However, Bösen slightly improves upon Yahoo!LDA due to its prioritization scheme.

5.2 Comparison with Clock Tick Size Tuning

Another way of reducing parallel error on a BSP or bounded staleness system is to divide a full data pass into multiple clock ticks to achieve more frequent synchronization, while properly adjusting the staleness threshold to ensure the same staleness bound. This approach is similar to mini-batch size tuning in ML literature. In this section, we compare Bösen's communication management with application-level clock tick size tuning via the LDA application and the result is plotted in Fig 7.

For application-level clock tick size tuning, we run Bösen LDA with the communication manager disabled (labelled S(i) for *i* clock ticks per data pass) using different numbers of clock ticks (1, 2, 4, 8) per full data pass. For each number of clock ticks per data pass, we adjust the staleness threshold so all runs share the same staleness bound of 2 data passes. We then compare the manually tuned clock tick execution with managed communication, which was executed using 320Mbsp of network bandwidth limit with the randomization policy (labelled (B1)) and using 640Mbps with the Relative-Magnitude bandwidth scheduling policy (B2).

Firstly, from Fig. 7c we observe that as the clock tick size halves, the average bandwidth usage over the first 280 iterations doubles but the average time per iteration doesn't change significantly. From Fig. 7a and Fig. 7b, we observe that the increased communication improves the algorithm performance. Although simply tuning clock tick size also improves algorithm behavior, it doesn't enjoy the benefit of prioritization. For example, B2 used only 60% of the bandwidth compared to S(8) but achieves better algorithm performance. The difference is due to careful optimization which cannot be achieved via application-level tuning.

6 Related Work

Existing distributed systems for machine learning can be broadly divided into two categories: (1) special-purpose solvers for a particular categories of ML algorithms, and (2) general-purpose, programmable frameworks and ML libraries that encompasses a broad range of ML problems. Examples of special-purpose systems are YahooLDA [4], DistBelief [15], ProjectAdam [10]. General-purpose systems include libraries like MLlib [3] and Mahout [2] on MapReduce [16, 1, 44], Parameter Server systems [24, 12, 30, 37], graph-based frameworks [32, 33] and data flow engines [36]. Bösen is an instance of the parameter server family, although the managed communication mechanism is general and could be implemented in most existing ML systems.

In terms of communication mechanisms, many distributed ML frameworks adopt the Bulk Synchronous Parallel (BSP) model originally developed for parallel simulation [40] , such as MapReduce, Hadoop, Spark, and Pregel. BSP programs are easy for programmers to reason about but are sensitive to stragglers, also known as operating system jitter, and can incur high synchronization overhead. At the other extreme there are fully asynchronous systems [4, 10, 15] that exhibit good empirical convergence speed due to the error-tolerant nature of ML applications, but forgo any theoretical guarantees. Graph-based frameworks such as GraphLab [32] support serializability by providing strong isolation around dependent parallel vertex executions.

Recently a new wave of consistency models and synchronization schemes arise from theoretical results in the ML community. Many ML algorithms are error-tolerant, i.e., bounded intermediate errors during execution will be self-corrected and will not affect the final outcome. Stochastic optimization technique such as stochastic gradient descent (SGD), coordinate descent, and sampling methods all make use of these fundamental properties. Recent works show that ML applications are also robust under various bounded delays in distributed systems [24, 27, 38, 14]. These theoretical results have inspired various bounded-staleness frameworks that enjoys both the theoretical guarantees akin to that of BSP and the high performance close to fully-asynchronous systems.

Most systems discussed here ties communication to computation. For example, BSP systems communicate at clock boundaries, and even some async system like PowerGraph [22] communicates upon when vertex operations (compute task) are invoked, or ProjectAdam [10] where communication occurs after each minibatch. Our work largely *decouples* communication and computation, which in many cases improves performance while reducing application programmer burdens. Similar idea has been explored in database systems.

7 Conclusion

While tolerance to bounded staleness reduces communication and synchronization overheads for distributed machine learning algorithms and thus improves system throughput, the accumulated error may, sometimes heavily, harm algorithm performance and result in slower convergence rate. More frequent communication reduces staleness and parallel error and thus improves algorithm performance but it is ultimately bound by the physical network capacity. This paper presents a communication management technique that makes efficient use of a bounded amount of network bandwidth to improve algorithm performance. Experiments with several ML applications on over 1000 cores show that our technique significantly improves upon static communication schedules and demonstrate $3-5\times$ speedup relative to a well implemented bounded staleness system.

There are some areas which could be improved as future work. In Machine Learning, model parallelism, where client updates $\Delta()$ are applied to subsets of A (instead of the whole A), can be used to speed up the convergence time of large, expensive-to-compute models [15, 28]. This is a complementary idea to data parallelism (in which $\Delta()$ operates on subsets of the data $\mathscr{D}$), and we intend to investigate how model parallelism can be used to improve the efficiency of each data parallel update, particularly on very large clusters.

Bösen provides fault tolerance in the form of model checkpoints, and restarts the entire system from the latest checkpoint upon failure. For large clusters, it is more desirable to restart just those clients and servers which have failed, as future work. Another scalability-related issue is communication topologies: Bösen currently assumes that all clients enjoy equal bandwidth to servers, but this is rarely the case in clusters or datacenters with multiple racks and switches. We would like to extend Bösen's bandwidth management strategies to take into account the available bandwidth on each link within the network, or to use pre-specified communications topologies that route updates and parameters through multiple machines and thus reduce congestion.

References

[1] Apache hadoop. https://hadoop.apache.org/.

[2] Apache mahout. http://mahout.apache.org/.

[3] Spark mllib. https://spark.apache.org/mllib/.

[4] A. Ahmed, M. Aly, J. Gonzalez, S. Narayanamurthy, and A. J. Smola. Scalable inference in latent variable models. In *WSDM*, pages 123–132, 2012.

[5] A. Ahmed, M. Aly, J. Gonzalez, S. M. Narayanamurthy, and A. J. Smola. Scalable inference in latent variable models. In E. Adar, J. Teevan, E. Agichtein, and Y. Maarek, editors, *WSDM*, pages 123–132. ACM, 2012.

[6] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica. Effective straggler mitigation: Attack of the clones. In *NSDI*, pages 185–198, 2013.

[7] G. Ananthanarayanan, S. Kandula, A. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris. Reining in the outliers in map-reduce clusters using mantri. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 1–16, Berkeley, CA, USA, 2010. USENIX Association.

[8] C. M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[9] J. K. Bradley, A. Kyrola, D. Bickson, and C. Guestrin. Parallel coordinate descent for l1-regularized loss minimization. In *ICML*, pages 321–328, 2011.

[10] T. Chilimbi, Y. Suzue, J. Apacible, and K. Kalyanaraman. Project adam: Building an efficient and scalable deep learning training system. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 571–582, Broomfield, CO, Oct. 2014. USENIX Association.

[11] J. Cipar, Q. Ho, J. K. Kim, S. Lee, G. R. Ganger, G. Gibson, K. Keeton, and E. Xing. Solving the straggler problem with bounded staleness. In *Proc. of the 14th Usenix Workshop on Hot Topics in Operating Systems*, HotOS '13. Usenix, 2013.

[12] H. Cui, J. Cipar, Q. Ho, J. K. Kim, S. Lee, A. Kumar, J. Wei, W. Dai, G. R. Ganger, P. B. Gibbons, G. A. Gibson, and E. P. Xing. Exploiting bounded staleness to speed up big data analytics. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 37–48, Philadelphia, PA, June 2014. USENIX Association.

[13] H. Cui, A. Tumanov, J. Wei, L. Xu, W. Dai, J. Haber-Kucharsky, Q. Ho, G. R. Ganger, P. B. Gibbons, G. A. Gibson, and E. P. Xing. Exploiting iterative-ness for parallel ml computations. In *Proceedings of the ACM Symposium on Cloud Computing*, SOCC '14, pages 5:1–5:14, New York, NY, USA, 2014. ACM.

[14] W. Dai, A. Kumar, J. Wei, Q. Ho, G. A. Gibson, and E. P. Xing. Analysis of high-performance distributed ml at scale through parameter server consistency models. In *Proceedings of the 29th AAAI Conference on Artificial Intelligence (To Appear)*, Austin, TX, 2015.

[15] J. Dean, G. S. Corrado, R. Monga, K. Chen, M. Devin, Q. V. Le, M. Z. Mao, M. Ranzato, A. Senior, P. Tucker, K. Yang, and A. Y. Ng. Large scale distributed deep networks. In *NIPS*, 2012.

[16] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI 2004*, pages 137–150, 2004.

[17] J. Duchi, E. Hazan, and Y. Singer. Adaptive subgradient methods for online learning and stochastic optimization. *The Journal of Machine Learning Research*, 12:2121–2159, 2011.

[18] J. Friedman, T. Hastie, and R. Tibshirani. Regularization paths for generalized linear models via coordinate descent. *Journal of statistical software*, 33(1):1, 2010.

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

[20] A. Genkin, D. D. Lewis, and D. Madigan. Large-scale bayesian logistic regression for text categorization. *Technometrics*, 49(3):291–304, 2007.

[21] W. R. Gilks. *Markov chain monte carlo*. Wiley Online Library, 2005.

[22] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI '12)*, Hollywood, October 2012.

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

[24] Q. Ho, J. Cipar, H. Cui, S. Lee, J. K. Kim, P. B. Gibbons, G. A. Gibson, G. Ganger, and E. Xing. Distributed ml via a stale synchronous parallel parameter server. In *Advances in Neural Information Processing Systems (NIPS) 26*, pages 1223–1231. 2013.

[25] Y. Koren, R. M. Bell, and C. Volinsky. Matrix factorization techniques for recommender systems. *IEEE Computer*, 42(8):30–37, 2009.

[26] A. Krizhevsky, I. Sutskever, and G. Hinton. Imagenet classification with deep convolutional neural networks. In P. Bartlett, F. Pereira, C. Burges, L. Bottou, and K. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1106–1114. 2012.

[27] J. Langford, E. J. Smola, and M. Zinkevich. Slow learners are fast. In *In NIPS*, pages 2331–2339, 2009.

[28] S. Lee, J. K. Kim, X. Zheng, Q. Ho, G. A. Gibson, and E. P. Xing. On model parallelization and scheduling strategies for distributed machine learning. In *Advances in Neural Information Processing Systems*, pages 2834–2842, 2014.

[29] 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 *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 583–598, Broomfield, CO, Oct. 2014. USENIX Association.

[30] M. Li, L. Z. Z. Yang, A. L. F. Xia, D. G. Andersen, and A. Smola. Parameter server for distributed machine learning. *NIPS workshop*, 2013.

[31] J. Liu, J. Chen, and J. Ye. Large-scale sparse logistic regression. In *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 547–556. ACM, 2009.

[32] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. Graphlab: A new parallel framework for machine learning. In *Conference on Uncertainty in Artificial Intelligence (UAI)*, Catalina Island, California, July 2010.

[33] 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 *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, SIGMOD '10, pages 135–146, New York, NY, USA, 2010. ACM.

[34] B. McMahan and M. Streeter. Delay-tolerant algorithms for asynchronous distributed online learning. In *Advances in Neural Information Processing Systems*, pages 2915–2923, 2014.

[35] H. B. McMahan, G. Holt, D. Sculley, M. Young, D. Ebner, J. Grady, L. Nie, T. Phillips, E. Davydov, D. Golovin, S. Chikkerur, D. Liu, M. Wattenberg, A. M. Hrafnkelsson, T. Boulos, and J. Kubica. Ad click prediction: a view from the trenches. In *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, 2013.

[36] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: a timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 439–455, New York, NY, USA, 2013. ACM.

[37] R. Power and J. Li. Piccolo: building fast, distributed programs with partitioned tables. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10, pages 1–14, Berkeley, CA, USA, 2010. USENIX Association.

[38] B. Recht, C. Re, S. J. Wright, and F. Niu. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *NIPS*, pages 693–701, 2011.

[39] A. Smola and S. Narayanamurthy. An architecture for parallel topic models. *Proc. VLDB Endow.*, 3(1-2):703–710, Sept. 2010.

[40] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

[41] L. Yao, D. Mimno, and A. McCallum. Efficient methods for topic model inference on streaming document collections. In *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '09, pages 937–946, New York, NY, USA, 2009. ACM.

[42] J. W. Young. A first order approximation to the optimum checkpoint interval. *Commun. ACM*, 17(9):530–531, Sept. 1974.

[43] H.-F. Yu, H.-Y. Lo, H.-P. Hsieh, J.-K. Lou, T. G. McKenzie, J.-W. Chou, P.-H. Chung, C.-H. Ho, C.-F. Chang, Y.-H. Wei, et al. Feature engineering and classifier ensemble for kdd cup 2010. *KDD Cup*, 2010.

[44] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, pages 10–10, 2010.

[45] Y. Zhou, D. Wilkinson, R. Schreiber, and R. Pan. Large-scale parallel collaborative filtering for the netflix prize. In *Proc. 4th Int'l Conf. Algorithmic Aspects in Information and Management, LNCS 5034*, pages 337–348. Springer, 2008.