

# 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<sup>†</sup> Garth A. Gibson Eric P. Xing

Carnegie Mellon University, \*Institute for Infocomm Research, A\*STAR, <sup>†</sup>Intel Labs

## Abstract

At the core of Machine Learning (ML) analytics 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 step size. Simply increasing communication often fails to bring improvement without tuning step size accordingly and tedious hand tuning is usually needed 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 parallel error, while ensuring theoretical convergence guarantees for large-scale data-parallel ML applications. Furthermore, Bösen prioritizes messages 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 for some SGD

algorithms. Experiments on two clusters with up to 1024 cores show that our mechanism significantly improves upon static communication schedules.

**Categories and Subject Descriptors** C.2.4 [Distributed Systems]: Client/server, Distributed applications, Distributed databases; D.4.4 [Communications Management]: Network communication; D.4.7 [Organization and Design]: Batch processing systems, Distributed systems; G.1.6 [Optimization]: Gradient methods

## 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 [15, 36], coordinate descent with applications in regression [7, 14], and Markov chain Monte Carlo sampling with applications in topic modeling [18, 20]. 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 par-

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s). Copyright is held by the owner/author(s).

SoCC '15, August 27 - 29, 2015, Kohala Coast, HI, USA  
ACM 978-1-4503-3651-2/15/08.  
<http://dx.doi.org/10.1145/2806777.2806778>

tioned 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. However, BSP-style execution suffers from overheads due to the synchronization barriers separating iterations [10, 21]; therefore, asynchronous systems have been proposed, in which machines can enter the next iteration before receiving the fully-updated model parameters [3, 8, 13]. Sometimes these systems show faster convergence times than BSP-style implementations, though 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 [24] or Stale Synchronous [21], 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 [21]. 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 accumulated parallel errors (from missing updates). There is theoretical and empirical evidence showing that fresher parameters lead to higher algorithm throughput [12].

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 [29], deep neural networks [13], and collaborative filtering [22], is highly sensitive to the step size that modulates the gradient magnitude. Many existing distributed ML systems require manual tuning of step size by the users [26], while some provides heuristic tuning mechanism [15]. An effective large-scale ML system would need to address these important algorithmic challenges.

This paper presents Bösen, a system designed to achieve maximal network communication efficiency by incorporating knowledge of network bandwidth. 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. More-

over, our solution maximizes communication efficiency by prioritizing network bandwidth for messages most significant for algorithm progress, which further enhances algorithm throughput for a fixed network bandwidth budget. We demonstrate the effectiveness of managed communication on three applications: Matrix Factorization with SGD, Topic Modeling (LDA) with Gibbs sampling, and Multiclass Logistic Regression with SGD on an up to 1024 core compute cluster.

To our knowledge, Bösen is the first distributed implementation of Adaptive Revision [28], a principled step-size tuning algorithm tolerant of delays in distributed systems. 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. Even with a delay-tolerant algorithm, Bösen’s managed communication still improves the performance of SGD with Adaptive Revision.

## 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  $\mathcal{D}$  as measured by an explicit objective function such as likelihood or reconstruction loss [6]. Such problems are usually solved by *iterative-convergent* algorithms, many of which can be abstracted as the following additive form:

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

where,  $A^{(t)}$  is the state of the model parameters at iteration  $t$ , and  $\mathcal{D}$  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  $\mathcal{D}$  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  $D_j$ <sup>1</sup>. 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  $\mathcal{D}$  is split over different threads and worker machines, in order to compute the update  $\Delta(A^{(t-1)}, \mathcal{D})$  based on the globally-shared model state from the previous iteration,  $A^{(t-1)}$ . Because  $\Delta()$  depends on the

<sup>1</sup> More precisely, each  $\mathcal{D}$  is *conditionally independent* of other  $D_j$  given knowledge of the true  $A$ .

latest value of  $A$ , workers must communicate updates  $\Delta()$  to each other.

It is well-established that fresher model parameters ( $A$  or  $\Delta$ ) improves the *per-iteration* progress of ML algorithms, i.e. each round of update equations makes more progress towards the final solution [36].

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

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

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

In each iteration, a subset of data  $\mathcal{D}_p$  is used for computing  $\Delta()$  on worker  $p$ ;  $\mathcal{D}_p$  is often called a “mini-batch” in the ML literature, and can be as small as one data sample  $D_i$ .  $\Delta$  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  $\mathcal{D}$ . Physically, multiple server machines, or co-location of server/client machines can be used to facilitate greater memory and network bandwidth. 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 ([3, 8, 13]) as well as general-purpose systems ([10, 24, 31]).

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 results. However, BSP suffers from the well-studied *stragglers* problem [4, 5, 9, 10], 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 some-

times 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* [21, 24], 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 dramatically improve times to convergence [10, 24], and theoretical analysis on convergence rates and error bounds is beginning to be reported for some cases [12, 21].

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. Previous implementations of bounded staleness often bundled communication with the staleness threshold and synchronization barriers or relied on explicit application control of communication. Our goal is to implement a system that supports sophisticated yet effective consistency models and resource-driven communication, while providing a PS API that abstracts these mechanisms away from the user. 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 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 (unlike TAP), while enjoying high iteration throughput that is better than BSP and close to TAP systems. Additionally, Bösen transmits model updates and up-to-date model parameters proactively 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 **parameter server partitions** (Figure 1); the former provides the Application Programming Interface (API) for reading/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 ML program process is linked against the client library, any thread in that process may read/update model parameters concurrently. 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 multiple machines.

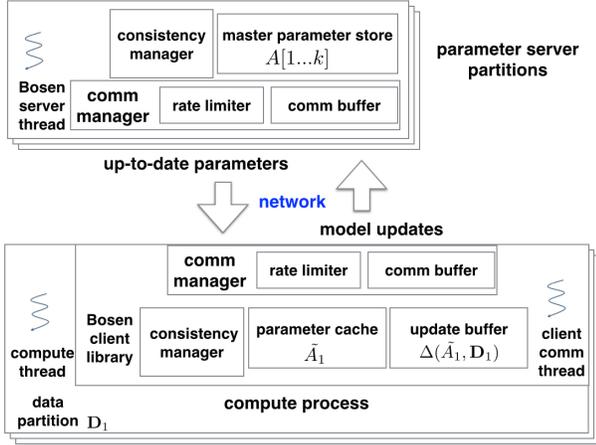


Figure 1: Parameter Server Architecture

<b>Get (key)</b>
Read a parameter indexed by <i>key</i> .
<b>GetRow (row_key)</b>
Read a row of parameters indexed by <i>row_key</i> . A row consists of a static group of parameters.
<b>Inc (key, delta)</b>
Increment the parameter <i>key</i> by <i>delta</i> .
<b>IncRow (row_key, deltas)</b>
Increment the row <i>row_key</i> by <i>deltas</i> .
<b>Clock ()</b>
Signal end of iteration.

Table 1: Bösen Client API

### 3.1.1 Bösen Client API

Bösen’s API abstracts consistency management and networking operations away from application, and presents a simple key-value interface (Table 1). `Get ()` is used to read parameters and `Inc ()` is used to increment the parameter by some delta. To maintain consistency, the client application signals the end of a unit of work 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 *rows* – a row is a set of parameters that are usually accessed together. A row can also be 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. 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).

### 3.1.2 Bounded Staleness Consistency

The Bösen client library and server partitions have a consistency manager, which is responsible for enforcing con-



Figure 2: Exemplar execution under bounded staleness (without communication management). The system consists of 5 workers, with staleness threshold  $S = 3$ ; “iteration  $t$ ” refers to iteration starting from  $t$ . Worker 2 is currently running in iteration 4 and thus according to bounded staleness, it is guaranteed to observe all updates generated before (exclusively) iteration  $4 - 3 = 1$  (black). It may also observe local updates (green) as updates can be optionally applied to local parameter cache. Updates that are generated in completed iterations (i.e. clock ticks) by other workers (blue) are highly likely visible as they are propagated at the end of each clock. Updates generated in incomplete iterations (white) are not visible as they are not yet communicated. Such updates could be made visible under managed communication depending on the bandwidth budget.

sistency requirements on the local parameter image  $\tilde{A}$ , thus ensuring correct ML program execution even under *worst-case* delays (whether computation or communication). The ML program’s tolerance to stale parameters is specified as the *staleness threshold*, a non-negative integer supplied by user.

The consistency manager works by blocking client process worker threads when reading parameters, until the local model image  $\tilde{A}$  has been updated to meet the consistency requirements. Bounded staleness puts constraints on parameter age; Bösen will block if  $\tilde{A}$  is older than the worker’s current iteration by  $S$  or more (i.e.,  $CurrentIteration(worker) - Age(\tilde{A}) \geq S$ ), where  $S$  is the user-defined staleness threshold.  $\tilde{A}$ ’s age is defined as the oldest iteration such that some updates generated within that iteration are missing from  $\tilde{A}$ .

The bounded staleness model enjoys BSP-like ML execution guarantees, theoretically explored in [12, 21], which are absent from total asynchronous parallelization (TAP). Bösen’s bounded staleness consistency is closest to ESSP [12]. Similar to ESSP, Bösen propagates updates upon completion of each iteration even though they are not required yet. With eager end-of-clock communication, parameters read typically have staleness 1 regardless of the staleness threshold as the end-of-clock communication typically completes within 1 iteration. Managed communication allows updates to be propagated even earlier, before completing an iteration. Our experiments used a staleness threshold of 2 unless otherwise mentioned, which has been proved to be effective in [10]. An exemplar execution of 5 workers under Bounded Staleness is depicted in Fig 2.

**Bulk Synchronous Parallel (BSP).** When the staleness threshold is set to 0, bounded staleness consistency reduces to the classic BSP model. The BSP model is a gold standard for correct ML program execution; it requires all updates computed in previous iterations to be made visible before the current iteration starts. A conventional BSP implementation may use a global synchronization barrier; Bösen’s consistency manager achieves the same result by requiring calls to `PS.Get()` and `PS.GetRow()` at iteration  $t$  to reflect all updates, made by any thread, before its  $(t - 1)$ -th call to `PS.Clock()` — otherwise, the call to `PS.Get()` or `PS.GetRow()` blocks until the required updates are received.

## 3.2 System Architecture

This section describes Bösen’s system architecture and focuses on its realization of the bounded staleness consistency. The system described in this section sufficiently ensures the consistency guarantees without communication management. Bounded staleness consistency without communication management serves as our baseline in evaluation and is referred to as “Bounded Staleness” in Section 5.

### 3.2.1 Client Library

The client library provides access to the model parameters  $A$  on the server partitions, and also caches elements of the model  $A$  for faster access, while cooperating with server processes 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 compute threads; (2) an *update buffer* that buffers updates applied by compute threads via `PS.Inc()` and `PS.RowInc()`; (3) a group of *client communication threads* (distinct from compute threads) that perform synchronization of the local model cache and buffered updates with the servers’ master copies, while the compute threads executes the application algorithm.

The parameters cached at a client are hash partitioned among the client communication threads. Each client communication thread needs to access only its own parameter partition when reading the computed updates and applying up-to-date parameter values to minimize lock contention. The client parameter cache and update buffer allow concurrent reads and writes from worker threads, and similar to [11], 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.

In each compute process, locks are needed for shared access to parameters and buffered update entries. In order to amortize the runtime cost of concurrency control, we allow applications to define parameter key ranges we call *rows* (as noted above). Parameters in the same row share one lock for accesses to their parameter caches, and one lock for accesses to their update buffers.

When serving read requests (`Get()` and `RowGet()`) from worker threads, the client parameter cache is searched first, and a read request is sent to the server processes only if either the requested parameter is not in the cache or the cached parameter’s staleness is not within the staleness threshold. The reading compute thread blocks until the parameter’s staleness is within the threshold. When writes are invoked, updates are inserted into the update buffer, and, optionally, the client’s own parameter cache is also updated.

Once all compute threads in a client process have called `PS.Clock()` to signal the end of a unit of work (e.g. an iteration), the client communication threads release buffered model updates to servers. Note that buffered updates may be released sooner under managed communication if the system detects spare network bandwidth to use.

### 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; 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.

### 3.2.3 Ensuring Bounded Staleness

Bounded staleness is ensured by coordination of clients and server partitions using *clock messages*. On an individual client, as soon as all updates generated before and in iteration  $t$  are sent to server partitions and no more updates before or in that iteration can be generated (because all compute threads have advanced beyond that iteration), the client’s communication threads send an client clock message to each server partition, indicating “all updates generated before and in iteration  $t$  by this client have been made visible to this server partition” (assuming reliable ordered message delivery).

After a server partition sends out all dirty parameters modified in iteration  $t$ , it sends an server clock message to each client communication thread, indicating “all updates generated before and in iteration  $t$  in the parameter partition have been made visible to this client”. Upon receiving such a clock message, the client communication thread updates the age of the corresponding parameters and permits the relevant blocked compute threads to proceed on reads if any.

### 3.2.4 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  $\sqrt{2T_s T_f / N}$  [34], 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.

As Bösen targets offline batch training, restarting the system (disrupting its availability) is not critical. With tens or hundreds of machines, such training tasks typically complete in hours or tens of hours. Considering the MTTF of modern hardware, it is not necessary to create many checkpoints and the probability of restarting is low. In contrast, a replication-based fault tolerance mechanism inevitably costs  $2\times$  or even more memory on storing the replicas and additional network bandwidth for synchronizing them.

### 3.3 Managed Communication

Bösen’s client library and server partitions feature a communication manager whose purpose is to improve ML progress per iteration through careful use of network bandwidth in communicating model updates/parameters. Communication 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 communication manager has two objectives: (1) communicate as many updates per second as possible (full utilization of the bandwidth budget) *without* overusing the network (which could delay update delivery and increase message processing computation overhead); and (2) prioritize more important model updates to improve ML progress per iteration. The first objective is achieved via bandwidth-driven communication with rate limiting, while the second is achieved by choosing a proper prioritization strategy.

#### 3.3.1 Bandwidth-Driven Communication

Similar to the leaky bucket model, the Bösen communication manager models the network as a bucket that leaks bytes at certain rate and the leaky rate corresponds to the node’s bandwidth consumption. Thus the leaky rate is set to the given bandwidth budget to constrain the average bandwidth consumption. In order to fully utilize the given bandwidth budget, the communication manager permits communication of updates or updated parameters whenever the bucket becomes empty (and thus communication may happen be-

fore the completion of an iteration). The communication manager keeps track of the number of bytes sent last time to monitor the state of the bucket. In our prototype implementation, the communication threads periodically query the communication manager for opportunities to communicate. The size of each send is limited by the size of the bucket (referred to as “queue size”) to control its burstiness.

**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 budget  $B$ . Hence, the Bösen communication 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 communication manager assumes that the network is overused, and waits until the window becomes clear before permitting new messages to be sent.

**Update quantization:** Since ML applications are error tolerant, Bösen applications have the option to use 16-bit floating point numbers for communication, reducing bandwidth consumption in half compared to 32-bit floats. The lost information often has negligible impact on progress per iteration.

#### 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. When there is opportunity for communication due to spare bandwidth, the server or client communication threads pick a subset of parameter values or updates to send. The prioritization strategy determines which subset is picked at each communication event. By picking the right subset to send, the prioritization strategy alters the communication frequency of different parameters, effectively allocating more network bandwidth to more important updates. It should be noted that the end-of-clock communication needs to send all up-to-date parameters or updates older than a certain clock number to ensure the consistency guarantees.

Bösen’s bandwidth manager supports multiple prioritization strategies. The simplest possible strategies are **Randomized**, where communications threads send out randomly-chosen rows and **Round-Robin**, where communication threads repeatedly walk through the rows following a fixed order, and sends out all non-zero updates or updated parameters encountered. These strategies are baselines; better strategies prioritize according to significance to convergence progress. We study the following two better strategies.

**Absolute Magnitude prioritization:** Updates/parameters are sorted by their accumulated change in the buffer,  $|\delta|$ .

**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  $|\delta/a|$  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 prioritization.

### 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  $\eta$ . The update equation is thus:

$$A^{(t)} = A^{(t-1)} + \sum_{p=1}^P \eta_p^{(t-1)} \nabla(A^{(t-1)}, \mathcal{D}_p). \quad (3)$$

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. There exist principled strategies for adaptively adjusting the stochastic gradient step size, reducing sensitivity to the initial step size  $\eta^{(1)}$  and achieving good algorithm performance using any initial step size from a reasonable range. The current state of the art is Adaptive Revision (i.e. AdaRevision) [28], which computes the step size for each model dimension and takes into account both timeliness and magnitude of the update.

AdaRevision differs from regular SGD in that it maintains an accumulated sum of historical gradients for each parameter; a gradient atomically updates the parameter and the accumulated sum. When a parameter is read out of the parameter store for computation, a snapshot of the accumulated sum is taken and returned along with the parameter value. A client will compute a gradient using that parameter, and then apply it back to the parameter store — when this happens, the snapshot associated with that parameter is also supplied. The difference between the snapshot value and the latest parameter value indicates the timeliness of the update, and is used to adjust the step size: the longer the updates are delayed, the smaller the step size, so that long-delayed gradients do not jeopardize model quality. Our implementation stores the accumulated sum of historical gradients on the server partitions, and thus the updates are only applied on the server, while the client parameter cache is made read-only to the compute threads.

Whereas a naive implementation of AdaRevision might let clients fetch the accumulated sum (which is generally not needed for computing gradients) along with the parameters from the server (and send the sum back along with the computed gradients), Bösen instead supports **parameter versioning** to reduce the communication overhead. The server maintains a version number for each parameter row,

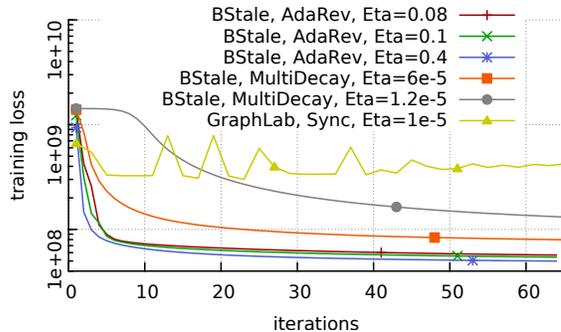


Figure 3: Compare Bösen’s SGD MF w/ and w/o adaptive revision with GraphLab SGD MF. Eta denotes the initial step size. Multiplicative decay (MultiDecay) used its optimal initial step size.

which is incremented every time the row is updated. The version number is sent to clients along with the corresponding parameters, to be 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. The use of version number to indicate timeliness of writes is similar to optimistic concurrency control.

The AdaRevision algorithm is implemented as a **user-defined stored procedure (UDF)** on the server. The user-defined stored procedure contains a set of user-implemented functions that are invoked at various events to control the server’s behavior. Most notably, the UDF takes snapshots of the accumulated sum of the historical gradients and increments the version number upon sending parameters to clients and computes the step size when applying gradients. In order to bound the overall memory usage, the UDF imposes an upper limit on the number of snapshots that can be kept. Bandwidth-triggered communication is canceled upon exceeding this limit. The snapshots are freed when no client cache still contains this version of the parameter

We demonstrate the importance of step size tuning and effectiveness of adaptive revision using the SGD MF application on the Netflix dataset, with rank = 50, using one node in the PROBE Susitna cluster (see Sec 5). As shown in Fig. 3, we compared adaptive revision (AdaRev) with Multiplicative Decay (MultiDecay) using various initial step sizes. We also ran GraphLab’s SGD MF using its synchronous engine (the asynchronous engine converges slower per iteration) with a range of initial step sizes from  $1e-4$  to  $1e-6$  and showed its best convergence result.

Firstly, we observed that multiplicative decay is sensitive to the initial step size. Changing the initial step size from  $6e-5$  to  $1.2e-5$  reduces the number of iterations needed to reach training loss of  $1e8$  by more than  $3\times$ . However, convergence with adaptive revision is much more robust and the

difference between initial step size of 0.08 and 0.4 is negligible. Secondly, we observed that SGD MF under adaptive revision converges  $2\times$  faster than using multiplicative decay with the optimal initial step size that our manual parameter tuning could find. Even though GraphLab also applies multiplicative decay to its step size, it does not converge well.

The adaptive revision algorithm becomes more effective when scaling the SGD application as it adapts the step size to tolerate the communication delay. An experiment (not shown) using 8 Susitna nodes shows that adaptive revision reduces the number of iterations to convergence by  $10\times$ .

#### 4. ML program instances

We study how bandwidth management improves the convergence time and final solution quality for three commonly-used data-parallel ML programs: matrix factorization, multi-class 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  $r \ll \min(M, N)$  is the rank which determines the model size (along with  $M$  and  $N$ ). We partition observations  $D$  to  $P$  worker threads and solve MF via stochastic gradient descent (SGD) using adaptive revision. All MF experiments in Section 5 used adaptive revision, and thus the updates (i.e. gradients) are not directly added to the parameters. Instead they are scaled on servers by the step size computed from AdaRevision UDFs.

**Multiclass Logistic Regression (MLR)** Logistic Regression is a classical method used in large-scale classification [35], natural language processing [16], and ad click-through-rate prediction [29] among others. Multiclass Logistic Regression generalizes LR to multi-way classification, seen in large-scale text classification [25], 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 [23]. For each observation, MLR produces a categorical distribution over the label classes. The model size is  $J \times d$  where  $d$  is the input dimension and  $J$  is the number of output labels. We solve MLR using regular stochastic gradient descent (SGD). In our experiments, MLR with regular SGD converges within 50 iterations.

**Topic Modeling (LDA)** *Topic Modeling (Latent Dirichlet Allocation)* 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  $w_{ij}$  ( $j$ -th token in the  $i$ -th document) is assigned with a latent topic  $z_{ij}$  from totally  $K$  topics. We use Gibbs

sampling to infer the topic assignments  $z_{ij}$ .<sup>2</sup> The sampling step involves three sets of parameters, known as “sufficient statistics”: (1) document-topic vector  $\theta_i \in \mathbb{R}^K$  where  $\theta_{ik}$  the number of topic assignments within document  $i$  to topic  $k = 1 \dots K$ ; (2) word-topic vector  $\phi_w \in \mathbb{R}^K$  where  $\phi_{wk}$  is the number of topic assignments to topic  $k = 1, \dots, K$  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  $k$ . The corpus  $(w_{ij}, z_{ij})$  is partitioned to worker nodes (i.e. each node has a set of documents), and  $\theta_i$  is computed on-the-fly before sampling tokens in document  $i$ .  $\phi_w$  and  $\tilde{\phi}$  are stored as rows in PS.

#### 5. Evaluation

We evaluated Bösen using the above ML applications.

**Cluster setup:** Most of our experiments were conducted on PROBE Nome [17] consisting of 200 high-end computers running Ubuntu 14.04. Our experiments used different number of computers, varying 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 a 1 Gb Ethernet and 20 Gb Infiniband. A few experiments were conducted on PROBE Susitna [17]. 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<sup>3</sup>. The ML models and datasets are described in Table 2 and the system and application configurations are described in Table 3.

**Performance metrics:** Our evaluation measures performance as the absolute convergence rate on the training objective value; that is, our goal is to reach convergence to an estimate of the model parameter  $A$  that best represents the training data (as measured by the training objective value) in the shortest time.

Bösen is executed under different modes in this section:

**Single Node:** The ML application is run on one shared-memory machine linked against one Bösen client library instance with only consistency management. The parameter cache is updated upon write operations. Thus updates be-

<sup>2</sup> Specifically, we use the SparseLDA variant in [33] which is also used in YahooLDA [3] that we compare with.

<sup>3</sup> 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.

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
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
MLR	ImageNet5%	65K samples	1000 classes, 21K of feature dimensions	1K	84KB	5.1GB

Table 2: Descriptions of ML models and datasets. Data size refers to the input data size. Workload refers to the total number of data samples in the input data set. The overall model size is thus # Rows multiplied by row size.

Application & Dataset	Cluster	# Machines	Per-node Bandwidth Budgets	Queue Size	Initial Step Size	Figures
SGD MF, Netflix	Nome	8	200Mbps, 800Mbps	100, 100	0.08	4a, 6a, 7a
LDA, NYTimes	Nome	16	320Mbps, 640Mbps, 1280Mbps	5000, 500	N/A	4b, 5, 6b, 8
LDA, ClueWeb10%	Nome	64	800Mbps	5000, 500	N/A	not shown
MLR, ImageNet5%	Susistna	4	100Mbps, 200Mbps, 1600Mbps	1000, 500	1	7b

Table 3: Bösen system and application configurations. The queue size (in number of rows) upper bounds the send size to control burstiness; the first number denotes that for client and the second for server. LDA experiments used hyper-parameters  $\alpha = \beta = 0.1$ . Most of our experiments used the 1GbE network on both clusters and only experiments whose bandwidth budget exceeds 1Gbps used 20Gb infiniband or 40Gb Ethernet.

come immediately visible to compute threads. It represents a gold standard when applicable. It is denoted as “SN”.

**Linear Scalability:** It represents an ideal scenario where the single-node application is scaled out and linear scalability is achieved. It is denoted as “LS”.

**Bounded Staleness:** Bösen is executed with only consistency management enabled, communication management is disabled. It is denoted as “BS”.

**Bounded Staleness + Managed Communication:** Bösen is executed with both consistency and communication management enabled. It is denoted as “MC- $X$ -P”, where  $X$  denotes the per-node bandwidth budget (in Mbps) and P denotes the prioritization strategy: “R” for Randomized, “RR” for Round-Robin, and “RM” for Relative-Magnitude.

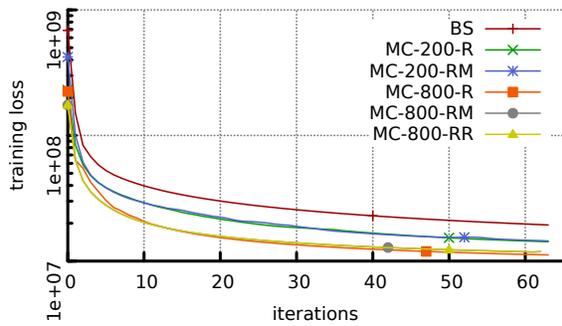
**Bounded Staleness + Fine-Grained Clock Tick Size:** Bösen is executed with only consistency management enabled, communication management is disabled. In order to communicate updates and model parameters more frequently, a full pass over data is divided into multiple clock ticks. It is denoted as “BS- $X$ ”, where  $X$  is the number of clock ticks that constitute a data pass.

Unless otherwise mentioned, we used a staleness threshold of 2 and we found that although bounded staleness converges faster than BSP, changing the staleness threshold does not affect average-case performance as the actual staleness is usually 1 due to the eager end-of-clock communication (Section 3.1). The network waiting time is small enough that a staleness threshold of 2 ensures no blocking. The bounded staleness consistency model allows computation to proceed during synchronization. As long as the workload is balanced and synchronization completes within one iteration of computation (which is typically the case), the network waiting time can be completely hidden.

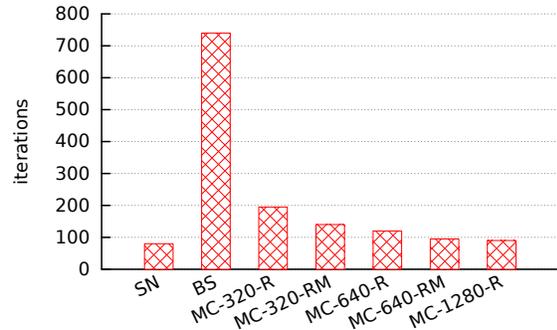
## 5.1 Communication Management

In this section, we show that the algorithm performance improves with more immediate communication of updates and model parameters. Moreover, proper bandwidth allocation based on the importance of the messages may achieve better algorithm performance with less bandwidth consumption. To this end, we compared managed communication with non-managed communication (i.e. only the consistency manager is enabled). The communication management mechanism was tested with different per-node bandwidth budgets (see Table 3) and different prioritization strategies (Section 3.3.2). Each node runs the same number of client and server communication threads and the bandwidth budget is evenly divided among them.

**Effect of increasing bandwidth budget.** Under Bösen’s communication management, increasing bandwidth budget permits more immediate communication of model updates and parameters and thus improves algorithm performance (higher convergence per iteration) given a fixed prioritization policy. We demonstrate this effect via the MF and LDA experiments (Fig. 4). First of all, we observed that enabling communication management significantly reduces the number of iterations needed to reach convergence (objective value of  $2e7$  for MF and  $-1.022e9$  for LDA). In MF, communication management with bandwidth budget of 200Mbps reduces the number of iterations needed to reach  $2e7$  from 64 (BS) to 24 (MC-200-R). In LDA, a bandwidth budget of 320Mbps reduces the number of iterations to convergence from 740 (BS) to 195 (MC-320-R). Secondly, increasing the bandwidth budget further reduces the number of iterations needed. For example, in LDA, increasing the bandwidth budget from 320Mbps (MC-320-R) to 640Mbps (MC-640-R) reduces the number iterations needed from 195 to 120.

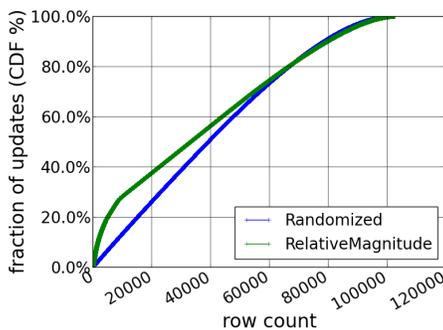


(a) SGD Matrix Factorization

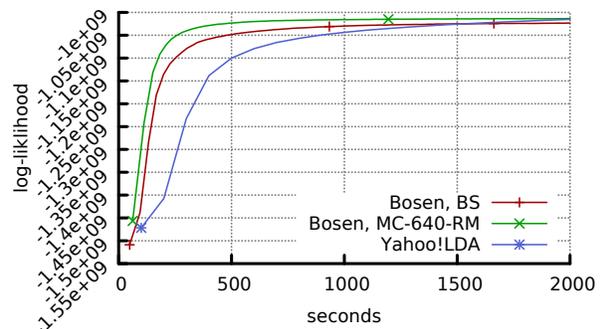


(b) Topic Modeling (LDA), number of iterations to convergence

Figure 4: Algorithm performance under managed communication



(a) Model Parameter Communication Frequency CDF



(b) Compare Bösen LDA with Yahoo!LDA on NYTimes Data

Figure 5: Topic Modeling with Latent Dirichlet Allocation

**Effect of prioritization.** As shown Fig. 4b, 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 320Mbps of per-node bandwidth budget MC-320-RM reduces the number of iterations needed to reach  $-1.022e9$  from 195 (MC-320-R) to 145.

Relative-Magnitude prioritization improves upon Randomized prioritization as it differentiates updates and model parameters based on their significance to algorithm performance. It allocates network bandwidth accordingly and communicates different updates and model parameters at different frequencies. Fig. 5a shows the CDFs of communication frequency of LDA’s model parameters, under different policies. For the NYTimes dataset, we observed that Relative-Magnitude and Absolute-Magnitude prioritization achieve similar effect, where a small subset of keys are communicated much more frequently. Random and Round-Robin achieve similar effect where all keys are communicated at roughly the same frequency.<sup>4</sup>

<sup>4</sup>On a skewed dataset, it’s possible to observe a skewed communication frequency distribution even with Randomized or Round-Robin policy when some words appear much more frequently than others. Even then the pri-

oritization appears to be less effective for MF. The server UDF computes the step size which scales the gradient, altering the gradient by up to orders of magnitude. Since the adaptive revision algorithm tends to [28] apply a larger scaling factor for smaller gradients, the raw gradient magnitude is a less effective indicator of significance.

**Overhead of communication management and absolute convergence rate.** Under managed communication, the increased volume of messages incurs noticeable CPU overheads due to sending and receiving the messages and serializing and deserializing the content. Computing importance also costs CPU cycles. Fig. 6 presents the per-iteration runtime and network bandwidth consumption corresponding to Fig. 4. For example, enabling communication management with a 200Mbps bandwidth budget (MC-200-R) incurs a 12% per-iteration runtime overhead.

However, the improved algorithm performance significantly outweighs such overheads and results in much higher absolute convergence rate in wall clock time, as shown in Fig. 7 (MF and MLR) and Fig. 8a. For example, for MF, we observed a  $2.5\times$  speedup in absolute convergence rate us-

oritization scheme can still alter the communication frequency to prioritize the most important updates.

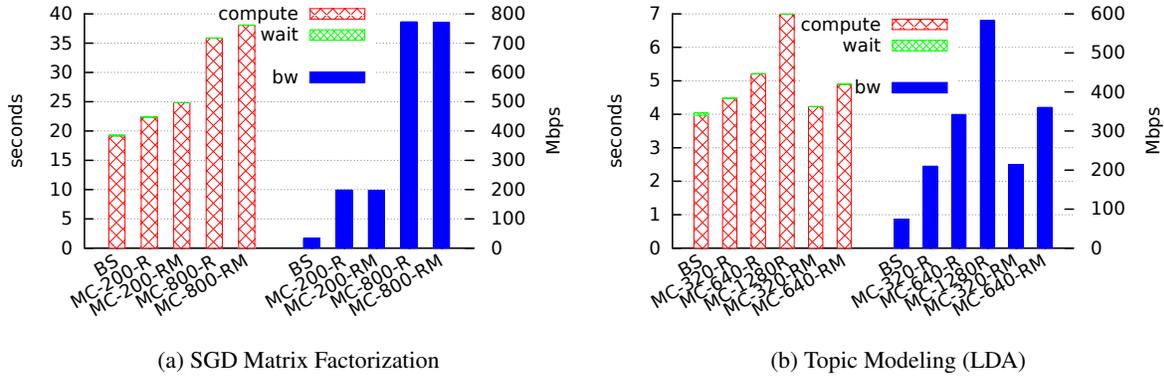


Figure 6: Overhead of communication management: average per-iteration time and bandwidth consumption

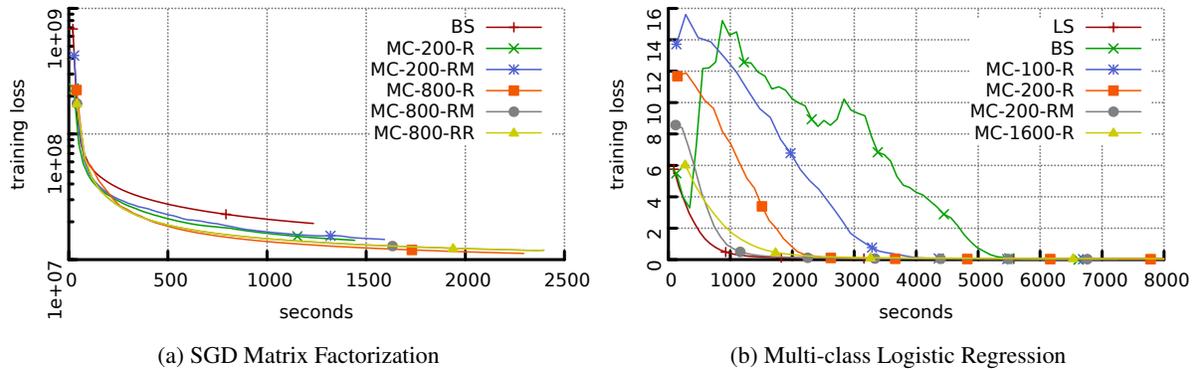


Figure 7: Absolute convergence rate under managed communication

ing bandwidth budget of 800Mbps and Relative-Magnitude prioritization compared the bounded staleness baseline.

**Comparison with Yahoo!LDA.** We compare Bösen LDA with the popular Yahoo!LDA using the NYTimes and 10% of the ClueWeb data set, using 1Gbe and 20 Gb Infiniband respectively. The former is plotted in Fig. 5b. Yahoo!LDA employs a parameter server architecture that’s similar to Bösen’s, but uses total asynchronous parallelization. The compute threads of Yahoo!LDA process roughly the same number of data points as Bösens. Each Yahoo!LDA worker (node) runs one synchronizing thread that iterates over and synchronizes all cached parameter in a predefined order. We observed that Bösen significantly outperformed Yahoo!LDA on the NYTimes dataset, but converged at similar rate on the ClueWeb10% data set.

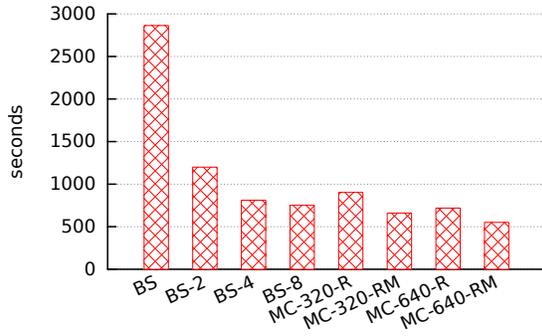
In summary, by making full use of the 800Mbps and 640Mbps bandwidth budget, communication management with Randomized prioritization improved the time to convergence of the MF and LDA application by  $2.5\times$  and  $2.8\times$  in wall clock time and  $5.3\times$  and  $6.1\times$  in number of iterations, compared to a bounded staleness execution. Relative-Magnitude prioritization further improves the convergence time of LDA by 25%. Communication management with

bandwidth budget of 200Mbps and Relative-Magnitude prioritization improved the convergence time of MLR by  $2.5\times$ .

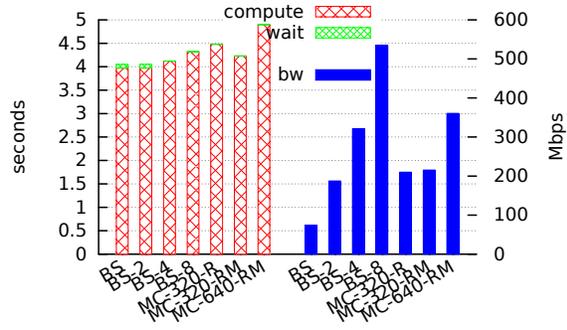
## 5.2 Comparison with Clock Tick Size Tuning

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 8. 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.

Firstly, from Fig. 8b 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. 8a, 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, MC-640-RM used only 63% of the



(a) time to convergence



(b) average per-iteration time and bandwidth consumption

Figure 8: Comparing Bosen with simply tuning clock tick size

bandwidth compared to BS-8 but converged 28% faster. 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: a. special-purpose solvers for particular categories of ML algorithms, and b. general-purpose, programmable frameworks and ML libraries that encompasses a broad range of ML problems. Examples of special-purpose systems are Yahoo!LDA [3], DistBelief [13], ProjectAdam [8]. General-purpose systems include MLlib [2], Mahout [1], various parameter server systems [10, 21, 24, 31], graph computation frameworks [26, 27] and data flow engines [30]. Bösen is an instance of the parameter server family, although the managed communication mechanism is general and could be applied to other distributed ML solutions.

In terms of consistency, many distributed ML frameworks adopt the Bulk Synchronous Parallel (BSP) model originally developed for parallel simulation [32], such as MapReduce, Hadoop, Spark, and Pregel. At the other extreme there are fully asynchronous systems, including Yahoo!LDA [3], Project Adam [8], and DistBlief [13]. Distributed GraphLab [26] supports serializability using two-phase locking which incurs high overhead. Recently the bounded staleness consistency model has aroused from theoretical results in the ML community and has been exploited in several parameter-server-based systems [10, 21, 24].

The systems discussed above tie communication to computation. For example, BSP systems communicate at and only at clock boundaries. Even fully asynchronous systems like ProjectAdam [8] communicates once for each minibatch. PowerGraph [19] communicates upon invocation of the vertex APIs. Our work decouples communication and computation and autonomously manages communication based on available network bandwidth and consistency re-

quirements to take maximal advantage of the network while reducing application developers’ and users’ burden.

## 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 to maximize the communication efficiency of 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 2 – 3× speedup relative to a well implemented bounded staleness system.

Our prototype implementation has certain limitations. While a production system should address these limitations, our evaluation nevertheless demonstrates the importance of managing network bandwidth. Our implementation assumes all nodes have the same inbound and outbound bandwidth and each nodes inbound/outbound bandwidth is evenly shared among all nodes that it communicates with. Such assumption is broken in a hierarchical network topology typically seen in today’s data centers, leading to underutilized network bandwidth. Although update magnitude serves as a good indicator of update importance for some applications, there are cases, such as when stored procedures are used, where it may be insufficient. Future research should look into exploiting more application-level knowledge and actively incorporating server feedbacks to clients.

## Acknowledgments

This research is supported in part by Intel as part of the Intel Science and Technology Center for Cloud Comput-

ing (ISTC-CC), the National Science Foundation under awards CNS-1042537, CNS-1042543 (PROBE, [www.nmc-probe.org](http://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, Avago, EMC, Facebook, Google, Hewlett-Packard, Hitachi, Huawei, Intel, Microsoft, NetApp, Oracle, Samsung, Seagate, Symantec, Western Digital) for their interest, insights, feedback, and support. We thank Mu Li, Jin Kyu Kim, Aaron Harlap, Xun Zheng and Zhiting Hu for their suggestions and help with setting up other third-party systems for comparison. We thank our shepherd Jinyang Li and the anonymous SoCC reviewers.

## References

- [1] Apache Mahout. <http://mahout.apache.org/>.
- [2] Apache Spark MLlib. <https://spark.apache.org/mllib/>.
- [3] A. Ahmed, M. Aly, J. Gonzalez, S. Narayanamurthy, and A. J. Smola. Scalable inference in latent variable models. In *WSDM '12: Proceedings of the fifth ACM international conference on Web search and data mining*, pages 123–132, New York, NY, USA, 2012. ACM.
- [4] 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.
- [5] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica. Effective straggler mitigation: Attack of the clones. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 185–198, Lombard, IL, 2013. USENIX.
- [6] C. M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.
- [7] J. K. Bradley, A. Kyrola, D. Bickson, and C. Guestrin. Parallel coordinate descent for l1-regularized loss minimization. In *International Conference on Machine Learning (ICML 2011)*, Bellevue, Washington, June 2011.
- [8] 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.
- [9] J. Cipar, Q. Ho, J. K. Kim, S. Lee, G. R. Ganger, G. Gibson, K. Keeton, and E. P. Xing. Solving the straggler problem with bounded staleness. In *Presented as part of the 14th Workshop on Hot Topics in Operating Systems*, Berkeley, CA, 2013. USENIX.
- [10] 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.
- [11] 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.
- [12] W. Dai, A. Kumar, J. Wei, Q. Ho, G. A. Gibson, and E. P. Xing. High-performance distributed ML at scale through parameter server consistency models. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence, January 25-30, 2015, Austin, Texas, USA.*, pages 79–87, 2015.
- [13] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, Q. V. Le, M. Z. Mao, M. Ranzato, A. W. Senior, P. A. Tucker, K. Yang, and A. Y. Ng. Large scale distributed deep networks. In *Advances in Neural Information Processing Systems 25: 26th Annual Conference on Neural Information Processing Systems 2012. Proceedings of a meeting held December 3-6, 2012, Lake Tahoe, Nevada, United States.*, pages 1232–1240, 2012.
- [14] J. H. Friedman, T. Hastie, and R. Tibshirani. Regularization paths for generalized linear models via coordinate descent. *Journal of Statistical Software*, 33(1):1–22, 2 2010.
- [15] 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.
- [16] A. Genkin, D. D. Lewis, and D. Madigan. Large-scale bayesian logistic regression for text categorization. *Technometrics*, page 2007.
- [17] G. Gibson, G. Grider, A. Jacobson, and W. Lloyd. Probe: A thousand-node experimental cluster for computer systems research. volume 38, June 2013.
- [18] W. R. Gilks. *Markov chain monte carlo*. Wiley Online Library, 2005.
- [19] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 17–30, Hollywood, CA, 2012. USENIX.
- [20] T. L. Griffiths and M. Steyvers. Finding scientific topics. *PNAS*, 101(suppl. 1):5228–5235, 2004.
- [21] Q. Ho, J. Cipar, H. Cui, S. Lee, J. K. Kim, P. B. Gibbons, G. A. Gibson, G. Ganger, and E. P. Xing. More effective distributed ml via a stale synchronous parallel parameter server. In C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Weinberger, editors, *Advances in Neural Information Processing Systems 26*, pages 1223–1231. Curran Associates, Inc., 2013.
- [22] Y. Koren, R. Bell, and C. Volinsky. Matrix factorization techniques for recommender systems. *Computer*, 42(8):30–37, Aug. 2009.
- [23] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. Burges, L. Bottou, and K. Weinberger, editors,

- Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012.
- [24] 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.
- [25] 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, KDD '09*, pages 547–556, New York, NY, USA, 2009. ACM.
- [26] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein. Distributed graphlab: A framework for machine learning and data mining in the cloud. *Proc. VLDB Endow.*, 5(8):716–727, Apr. 2012.
- [27] 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.
- [28] H. B. McMahan and M. Streeter. Delay-tolerant algorithms for asynchronous distributed online learning. *Advances in Neural Information Processing Systems (NIPS)*, 2014.
- [29] 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.
- [30] 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.
- [31] 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.
- [32] L. G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, Aug. 1990.
- [33] 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.
- [34] J. W. Young. A first order approximation to the optimum checkpoint interval. *Commun. ACM*, 17(9):530–531, Sept. 1974.
- [35] H.-F. Yu, H.-Y. Lo, H.-P. Hsieh, J.-K. Lou, T. G. McKenzie, J.-W. Chou, P.-H. Chung, C.-H. Ho, Y.-H. Chang, Chun-Fu and Wei, et al. Feature engineering and classifier ensemble for kdd cup 2010. *KDD Cup*, 2010.
- [36] M. Zinkevich, J. Langford, and A. J. Smola. Slow learners are fast. In Y. Bengio, D. Schuurmans, J. Lafferty, C. Williams, and A. Culotta, editors, *Advances in Neural Information Processing Systems 22*, pages 2331–2339. Curran Associates, Inc., 2009.