the work per clock). But, unlike BSP, SSP's more flexible executions can better mitigate transient straggler effects.

We describe a system, called LazyTable, that supports BSP, A-BSP, and SSP. Using three diverse, real ML applications (topic modeling, collaborative filtering, and PageRank) running on 500 cores, we study the relative merits of these models under various conditions. Our results expose a number of important lessons that must be considered in designing and configuring such systems, some of which conflict with prior work. For example, as expected, we find that tuning the staleness bound significantly reduces convergence times. But, we also find that A-BSP and SSP, when using the (same) best staleness bound, perform quite similarly in the absence of significant straggler effects. In fact, SSP involves some extra communication overheads that can make it slightly slower than A-BSP in such situations. In the presence of transient straggler effects, however, SSP provides much better performance.

This paper makes three primary contributions over previous work, including our workshop paper that proposed SSP. First, it provides the first detailed description of a system that implements SSP, as well as BSP and A-BSP, including techniques used and lessons learned in tuning its performance. Second, to our knowledge, it is the first to introduce the concept of tuning the BSP work-per-clock in the context of parallel ML, allowing A-BSP to be viewed (and evaluated) in the same bounded staleness model as SSP. Third, it provides the first comparative evaluations of A-BSP and SSP, exploring their relative merits when using the same staleness bound, whereas previous papers (e.g., [12]) only compared SSP to BSP. Importantly, these comparisons clarify when SSP does and does not provide value over a simpler A-BSP implementation.

2 Parallel ML and bounded staleness

This section reviews iterative convergent algorithms for ML, the traditional BSP model for parallelizing them, why staleness helps performance but must be bounded, and the A-BSP and SSP approaches to exploiting staleness.

2.1 Iterative convergent algorithms & BSP

Many ML tasks (e.g., topic modeling, collaborative filtering, and PageRank) are mapped onto problems that can be solved via iterative convergent algorithms. Such algorithms typically search a space of potential solutions (e.g., N-dimensional vectors of real numbers) using an *objective function* that evaluates the goodness of a potential solution. The goal is to find a solution with a large (or in the case of minimization, small) objective value. For some algorithms (e.g., eigenvector and shortest path), the objective function is not explicitly defined or evaluated. Rather, they iterate until the solution does not change (significantly) from iteration to iteration.

These algorithms start with an initial state S_0 that has some objective value $f(S_0)$. They proceed through a set of iterations, each one producing a new state S_{n+1} with a potentially improved solution (e.g., greater objective value $f(S_{n+1}) > f(S_n)$). In most ML use-cases, this is done by considering each input datum, one by one, and adjusting the current state to more accurately reflect it. Eventually, the algorithm reaches a stopping condition and outputs the best known state as the solution. A key property of these algorithms is that they will converge to a good state, even if there are minor errors in their intermediate calculations.

Iterative convergent algorithms are often parallelized with the Bulk Synchronous Parallel (BSP) model. In BSP, a sequence of computation work is divided among multiple computation threads that execute in parallel, and each thread's work is divided into *clock periods* by barriers. The clock period usually corresponds to an amount of work, rather than a wall clock time, and the predominant ML practice is to perform one full iteration over the input data each clock [19]. For an iterative convergent algorithm, the algorithm state is stored in a shared data structure (often distributed among the threads) that all threads update during each iteration. BSP guarantees that all threads see all updates from the previous clock, but not that they will see updates from the current clock, so computation threads can experience *staleness errors* when they access the shared data.

2.2 Bounded staleness

Parallel execution always faces performance challenges due to inter-thread communication overheads and synchronization delays. They can be mitigated by having threads work independently, but at the expense of threads not seeing the latest solution improvements by other threads. This lack of awareness of updates to the shared state by other threads is what we mean by "staleness".

In the BSP model, threads work independently within a clock period, with no guarantee of seeing updates from other threads until the next barrier. Figure 1(a) illustrates a BSP execution with 3 threads. In the original sequential execution, each iteration has 6 *work-units*, which are finer-grained divisions of the original sequential execution. We denote (i, j) as the j-th work-unit of the i-th iteration. In this example, when thread-3 is doing work (4, 6), which is circled in the illustration, BSP only guarantees that it will see the updates from work completed in the previous clocks (clocks 3 and lower, not shaded). It may or may not see updates from the five shaded work-units.

Because iterative convergent algorithms can tolerate some error in the adjustments made to the algorithm state, independent work by threads can be acceptable even though the algorithm state is shared. That is, even if a thread incorrectly assumes that other threads have made

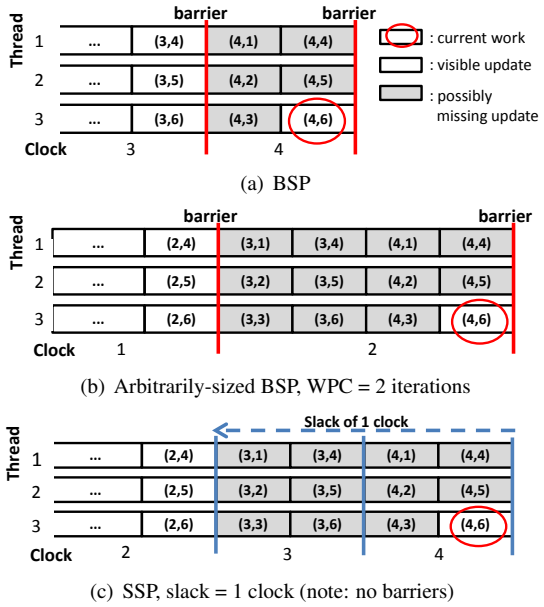


Figure 1: BSP, A-BSP, and SSP models. A block with (i, j) represents the j-th work-unit of the i-th iteration. Focusing on thread 3's execution of the circled work-unit, the shaded blocks indicate the updates that it may not see, under each model. SSP is more flexible than A-BSP, allowing the work of later clocks to start before the work of earlier clocks complete, up to the slack bound.

no relevant modifications to the shared state, causing it to produce a somewhat imperfect adjustment, the algorithm will still converge. Exploiting this fact, such as with a BSP implementation, allows parallel execution without synchronizing on every update to the shared state.

Accepting some staleness allows batching of updates, more reading from local (possibly out-of-date) views of the shared state, and less frequent synchronization, all of which helps iterations to complete faster. But, it may take more iterations to converge, because each iteration is less effective. In theory, a number of algorithms have been shown to converge given reasonable bounds on staleness [23]. Empirically, our experiments show that there is a sweet spot in this staleness tradeoff that maximizes overall convergence speed for a given execution, considering both the time-per-iteration and the effectiveness-per-iteration aspects.

Note that having a bound on staleness is important, at least in theory. There have been *Asynchronous Parallel* systems [1] that allow threads to work completely asynchronously, with best-effort communication of updates among them, but their robustness is unknown. While they have worked in some empirical evaluations, the convergence proofs associated with such efforts assume there are bounds on how out-of-synch threads will get, even though such systems (in contrast to those we consider) provide no mechanisms to enforce such bounds.

2.3 Expanding staleness exploitation

This section describes two approaches for more fully exploiting staleness to improve ML convergence speeds.

Arbitrarily-sized Bulk Synchronous Parallel (A-BSP). Because the staleness bound represents a tradeoff, tuning it can be beneficial. Focusing first on the BSP model, we define the amount of work done in each clock period as *work-per-clock (WPC)*. While the traditional ML approach equates iteration and clock, it is not necessary to do so. The WPC could instead be a multiple of or a fraction of an iteration over the input data. To distinguish BSP executions where WPC is not equal to one iteration from the current ML practice, we use the term "Arbitrarily-sized BSP" (A-BSP) in this paper.

Figure 1(b) illustrates an A-BSP execution in which the WPC is two full iterations. That is, the barriers occur every two iterations of work, which approximately halves the communication work and doubles the amount of data staleness compared to base BSP. Manipulating the A-BSP WPC in this manner is a straightforward way of controlling the staleness bound.

Stale Synchronous Parallel (SSP). While A-BSP amortizes per-clock communication work over more computation, it continues to suffer from BSP's primary performance issue: stragglers. All threads must complete a given clock before the next clock can begin, so a single slow thread will cause all threads to wait. This problem grows with the level of parallelism, as random variations in execution times increase the probability that at least one thread will run unusually slowly in a given clock. Even when it is a different straggler in each clock, due to transient effects, the entire application can be slowed significantly (see Section 5 for examples).

Stragglers can occur for a number of reasons including heterogeneity of hardware [21], hardware failures [3], imbalanced data distribution among tasks, garbage collection in high-level languages, and even operating system effects [5, 27]. Additionally, there are sometimes algorithmic reasons to introduce a straggler. Many algorithms use an expensive computation to check a stopping criterion, which they perform on a different one of the machines every so many clocks.

Recently, a model called "Stale Synchronous Parallel" (SSP) [12] was proposed as an approach to mitigate the straggler effect. SSP uses work-per-clock as defined above, but eliminates A-BSP's barriers and instead defines an explicit *slack* parameter for coordinating progress among the threads. The slack specifies how many clocks out-of-date a thread's view of the shared state can be, which implicitly also dictates how far ahead of the slowest thread that any thread is allowed to progress. For example, with a slack of s, a thread at clock t is guaranteed to see all updates from clocks 1 to $t - s - 1$, and it may see (not guaranteed) the updates from clocks $t - s$ to $t - 1$.

Figure 1(c) illustrates an SSP execution with a slack of 1 clock. When thread-3 is doing work (4, 6), SSP guarantees that it sees all the updates from clocks 1 and 2, and it might also see some updates from clocks 3 and 4.

Relationship of A-BSP and SSP. In terms of data staleness, SSP is a generalization of A-BSP (and hence of BSP), because SSP's guarantee with slack set to zero matches A-BSP's guarantee when both use the same WPC. (Hence, SSP's guarantee with slack set to zero and WPC set to 1 iteration matches BSP's guarantee.) For convenience, we use the tuple $\{wpc, s\}$ to denote an SSP or A-BSP configuration with work per clock of wpc and slack of s. (For A-BSP, s is always 0.) The data staleness bound for an SSP execution of $\{wpc, s\}$ is $wpc \times (s+1) - 1$. This SSP configuration provides the same staleness bound as A-BSP with $\{wpc \times (s+1), 0\}$.

A-BSP requires a barrier at the end of each clock, so it is very sensitive to stragglers in the system. SSP is more flexible, in that it allows some slack in the progress of each thread. The fastest thread is allowed to be ahead of the slowest by $wpc \times s$. As a result, the execution of SSP is like a pipelined version of A-BSP, where the work of later clocks can start before the work of earlier clocks complete. Intuitively, this makes SSP better at dealing with stragglers, in particular when threads are transient stragglers that can readily resume full speed once the cause of the slowness mitigates (e.g., the stopping criterion calculation or the OS/runtime operation completes).

But, SSP involves additional communication costs. The SSP execution of $\{wpc, s\}$ will have $s+1$ times more clocks than its A-BSP counter-part. In other words, SSP is a finer-grained division of the execution, and updates will be propagated at a higher frequency. As a result, SSP requires higher network throughput and incurs extra CPU usage for communication. When there are no stragglers, A-BSP can perform slightly better by avoiding the extra communication. Our evaluation explores this tradeoff.

3 LazyTable design and implementation

This section describes LazyTable, which provides for shared global values accessed/updated by multiple threads across multiple machines in a *staleness-aware* manner. It can be configured to support BSP, A-BSP, and SSP.

LazyTable holds globally shared data in a cluster of *tablet servers*, and application programs access the data through the *client library*. (See Figure 2.) Each tablet server holds a partition (shard) of the data, and the client library services data access operations from the application by communicating with the appropriate tablet server. The client library also maintains a hierarchy of caches and operation logs in order to reduce network traffic.

3.1 LazyTable data model and API

Data model. Globally shared data is organized in a collection of *rows* in LazyTable. A row is a user-defined data type and is usually a container type, such as an STL vector or map. The row type is required to be serializable and be defined with an associative aggregation operation, such as plus, multiply, or union, so that updates from different threads can be applied in any order. Each row is uniquely indexed by a (table_id, row_id) tuple.

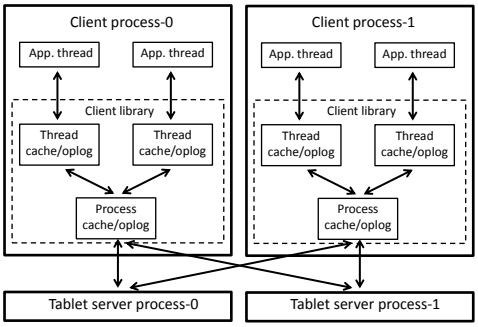


Figure 2: LazyTable running two application processes with two application threads each.

Having each data unit be a row simplifies the implementation of the many ML algorithms that are naturally expressed as operations on matrices and vectors. Each vector can naturally be stored as a row in LazyTable.

Operations. LazyTable provides a simple API for accessing the shared data and for application threads to synchronize their progress. Listed below are the core methods of the LazyTable client library, which borrow from Piccolo [28] and add staleness awareness:

`read(table_id, row_id, slack)`: Atomically retrieves a row. Ideally, the row is retrieved from a local cache. If no available version of the row is within the given `slack` bound, the calling thread waits. This is the only function that blocks the calling thread.

`update(table_id, row_id, delta)`: Atomically updates a row by `delta` using the defined aggregation operation. The `delta` should have the same type as row data, so it is usually a vector instead of a single value. If the target row does not exist, the row data will be set to `delta`.

`refresh(table_id, row_id, slack)`: Refreshes the process cache entry of a row, if it is too old. This interface can be used for the purpose of prefetching.

`clock()`: Increases the "clock time" of the calling thread by one. Although this is a synchronization-purposed function, it does not block the calling thread, so it is different from a barrier in a BSP or A-BSP system.

Data freshness and consistency guarantees. LazyTable does not have explicit barriers at the end of each clock. Instead, it bounds staleness by attaching a *data age* field to each row. If a row has a data age of τ, it is guaranteed to contain all updates from all application threads for clocks $1, 2, ..., \tau$. For the case of BSP, where each thread can only use the output from the previous

clock, a thread at clock t can only use the data of age $t-1$. For the case of SSP, when a thread at clock t issues a `read` request with a slack of s, only row data with data age $\tau \geq t-1-s$ can be returned.

LazyTable also enforces the *read-my-updates* property, which ensures that the data read by a thread contains all its own updates. Read-my-updates often makes it much easier to reason about program correctness. It also enables applications to store local intermediate data in LazyTable.

3.2 LazyTable system components

The LazyTable prototype is written in C++, using the ZeroMQ [32] socket library for communication.

3.2.1 Tablet servers

The tablet servers collectively store the current "master" view of the row data. Data is distributed among the tablet servers based on row ID, by default using a simple, static row-server mapping: $tablet\_id = row\_id \mod num\_of\_tablets$. The current implementation does not replicate row data or support multi-row updates, and also requires the tablet server data to fit in memory.

Each tablet server uses a vector clock to keep track of the version of all its row data. Each entry of the vector clock represents the server's view of the progress of each client process, which starts from zero and is increased when a `clock` message is received. The minimal clock value among the vector clock entries is referred to as the *global clock value*. A global clock value of t indicates that all application threads on all client machines have finished the work up to clock t, and that all updates from these threads have been merged into the master data. The `update` and `read` requests are serviced by the tablet servers as follows:

Proposing updates. When the tablet server receives a row update from a client, it puts it into a *pending updates list*. Updates in this list are applied to the master data only after a `clock` message is received from that client. This mechanism guarantees that the vector clock values can uniquely determine the version of the master data.

Reading values. When the tablet server receives a read request from a client, it looks at its global clock value. If the clock value is at least as large as the requested data age, the request is serviced immediately. Otherwise, the request will be put in the *pending read list*, which is sorted by the requested data age. When the global clock value of the tablet server advances to a required data age, the server then replies to those pending requests in the list. Along with the requested row data, the server also sends two clock fields: data age and *requester clock*. The data age is simply the server's global clock. The requester clock is the server's view of the requesting client's clock—this indicates which updates from that client have been applied to the row data. The client uses this information to clear its oplogs (discussed below).

3.2.2 Client library

The client library runs in the same process as the application code and translates the LazyTable API calls to server messages. It also maintains different levels of *caches* and *operation logs*. The client library creates several *background worker threads* that are responsible for jobs such as propagating updates and receiving data.

The LazyTable client library has two levels of caches/oplogs: the process cache/oplog and the thread cache/oplog, as depicted in Figure 2. The process cache/oplog is shared by all threads in the client process, including the application threads and background worker threads. Each thread cache/oplog, on the other hand, is exclusively associated with one application thread. Thread cache entries avoid locking at the process cache, and they are used only for the few rows that would suffer contention. The use of these caches/oplogs in servicing `update` and `read` operations is described below.

The client library tracks the progress of its application threads using a vector clock. Each entry of the vector clock represents the progress of one application thread, which starts at zero and will be increased by one each time the application thread calls the `clock()` operation.

Proposing updates. Application threads propose updates to the globally shared data using the `update()` operation. Suppose a thread at clock t wants to update the value of a row by `delta`. If the corresponding thread cache/oplog entry exists, the update will be logged in the thread oplog. To guarantee "read-my-updates", it will also be applied to the data in the thread cache immediately. When this thread calls the `clock` function, all updates in the thread oplog will be pushed to the process oplog and also applied to the row data in the process cache. If there is no thread cache/oplog entry, then the update will be pushed to the process cache/oplog immediately.

When all application threads in a client process have finished clock t, the client library will signal a background thread to send the `clock` messages, together with the batched updates of clock t in the process oplog, to the tablet servers. For robustness, the process oplogs are retained until the next time the client receives row data containing that update.

Reading values. When an application thread at clock t wants to read row r with a slack of s, the client library will translate the request to "read row r with data age $age \geq t-s-1$". To service this request, the client library will first look in the thread cache, and then the process cache, for a cached entry that satisfies the data age requirement. If not, it will send a request to the tablet server for row r and block the calling thread to wait for the new data. A per-row tag in the process cache tracks whether a request is in progress, in order to squash

redundant requests to the same row.

When the server replies, the row data is received by a background worker thread. The server also sends a requester clock value rc, which tells the client which of its updates have been applied to this row data. The receiver thread erases from its process oplog any operation for that tablet server's shard with clock $\leq rc$. Then, to guarantee "read-my-updates" for the application thread, it applies to the received data row (r) any operations for row r remaining in the process oplog (i.e., with clock $> rc$). Finally, the receiver thread sets the data age field of the row and signals the waiting application threads.

3.3 Prefetching and fault-tolerance

Prefetching. Not unexpectedly, we find that prefetching makes a huge difference in performance, even when bounded staleness is allowed (e.g., see Section 5.5). But, the access patterns often do not conform to traditional application-unaware policies, such as sequential prefetching. So, LazyTable provides an explicit `refresh()` interface for prefetching. The parameters for `refresh()` are the same as `read()`, but the calling thread is not blocked and row data is not returned.

Applications usually prefetch row data at the beginning of each clock period. The general rule-of-thumb is to refresh every row that will be used in that clock period so as to overlap fetch time with computation. While this requires an application to know beforehand what it will read in the current clock, many iterative ML applications have this property. Generally, each application thread processes the same input data in the same order, accessing the same rows in the same order as well. To leverage this property, LazyTable provides an automatic prefetcher module. The access pattern can be captured after one iteration, and the prefetcher can automatically refresh the needed data at the beginning of each clock.

The prefetching described so far addresses read miss latencies. But, for SSP, where multiple versions of the data can be accepted by a `read`, prefetching can also be used to provide fresher data. LazyTable supports two prefetching strategies. `Conservative prefetching` only refreshes when necessary; if $cache\_age < t - s - 1$, the prefetcher will send a request for (row $= r$, age $\geq t - s - 1$). `Aggressive prefetching` will always refresh if the row is not from the most recent clock, seeking the freshest possible value.

The conservative prefetching strategy incurs the minimal amount of traffic in order to avoid freshness misses. The aggressive prefetching strategy refreshes the data frequently, even with an infinite slack, at the cost of extra client-server communication. As a result, we can use infinite slack plus aggressive prefetching to emulate Asynchronous Parallel systems (as discussed in Section 2.2) in LazyTable. In our experiments to date, it is usually worthwhile to pay the cost of aggressive prefetching, so we use it as the default prefetching strategy.

Fault tolerance. LazyTable provides fault-tolerance via checkpointing. The tablet servers are told (in advance) about the clock at which to make a checkpoint, so that they can do so independently. Suppose the tablet servers are planned to make a checkpoint at the end of clock t. One way of checkpointing is to have each tablet server flush all its master data to the storage (take a snapshot) as soon as it has received a `clock=`t message from all of the clients. But, when slack > 0, the snapshot taken from this approach is not a *pure* one that contains exactly the clock 1 through t updates from all clients: some clients may have run ahead and already applied their clock $t + 1, \ldots$ updates to the master data. We call this approach an *approximate snapshot*. Approximate snapshots can be used to do off-line data processing after the computation, such as expensive objective value computation. LazyTable currently implements only approximate snapshot.

An alternative is to have each tablet server keep a *pure copy* of the master data, and have the tablet server flush it to storage instead of the latest master data. If a tablet server is going to checkpoint at the end of clock t, we can keep all updates beyond clock t out of the pure copy, so that it contains exactly the updates from clock 1 through t. But, some effects from SSP slack can be present. For example, client-1 might have generated its updates for clock t based on client-2's updates from clock $t + 1$. When one restores the execution from clock t using that snapshot, there will still be a residual influence from the "future". Moreover, in the view of client-1, client-2 goes backwards. Fortunately, iterative convergent algorithms suitable for SSP can tolerate this kind of error as well [18].

4 Example ML applications

This section describes three ML apps with different algorithms, representing a range of ML approaches.

Topic modeling. Topic modeling is a class of problems that assign *topic vectors* to documents. The specific model on which we focus is known as Latent Dirichlet Allocation [6] (LDA). LDA learns the parameters of the document-topic and topic-word distributions that best explain the input data (a corpus of documents). While there are a number of ways to do this, our example application uses the Gibbs sampling [17] algorithm.

Our implementation works as follows. We divide the set of input documents among threads and use two tables to store document-topic assignment and word-topic respectively. In each iteration, each thread passes through the words in their input documents. For the word $word\_id$ in document doc_id, the thread reads the doc_id-th row of the document-topic table and the $word\_id$-th row plus a summation row[2] of the word-topic table, and updates

[2]The sum of all word-topic rows, used by the algorithm to calculate

them based on the calculated topic assignment.

Matrix factorization. Matrix factorization can be used to predict missing values in a sparse matrix.[3] One example application is to predict user's preferences based on the known preferences of similar users, where the sparse matrix represents users' preference ranking to items. Matrix factorization assumes the matrix has low rank and can be expressed as the product of a tall skinny matrix (the *left-matrix*) and a short wide matrix (the *right-matrix*). Once this factorization is found, any missing value can be predicted by computing the product of these two matrices. This problem can be solved by the stochastic gradient descent algorithm [14].

Our implementation on LazyTable partitions the known elements in the sparse matrix among threads and uses two tables to store the left-matrix and right-matrix, respectively. In each iteration, each thread passes through the elements, and for the element (i, j) in the sparse matrix, it reads and adjusts the i-th row of the left-matrix table and the j-th row of the right-matrix table.

PageRank. The PageRank algorithm assigns a weighted score (PageRank) to every vertex in a graph [10], the score of a vertex measures its importance in the graph.

We implement an edge-scheduling version of PageRank on LazyTable. In each iteration, the algorithm passes through all the edges in the graph and updates the rank of the dst node according to the rank of the src node. The set of edges are partitioned evenly among threads, and the application stores the ranks of each node in LazyTable.

5 Evaluation

This section evaluates the A-BSP and SSP approaches via experiments with real ML applications on our LazyTable prototype; using the same system for all experiments enables us to focus on these models with all else being equal. The results support a number of important findings, some of which depart from previous understandings, including: (1) the staleness bound controls a tradeoff between iteration speed and iteration goodness, in both A-BSP and SSP, with the best setting generally being greater than a single iteration; (2) SSP is a better approach when transient straggler issues occur; (3) SSP requires higher communication throughput, including CPU usage on communication; and (4) iteration-aware prefetching significantly lowers delays for reads and also has the effect of reducing the best-choice staleness bound.

5.1 Evaluation setup

Cluster and LazyTable configuration. Except where otherwise stated, the experiments use 8 nodes of the NSF PRObE cluster [15]. Each node has four 2.1 GHz 16-core Opteron 6272s and 128GB of RAM (512 cores in total). The nodes run Ubuntu 12.04 and are connected via an Infiniband network interface (40Gbps spec; ≈13Gbps observed).

A small subset of experiments, identified explicitly when described, use a second cluster (the "VM cluster"). It consists of 32 8-core blade servers running VMware ESX and connected by 10 Gbps Ethernet. We create one VM on each physical machine, configured with 8 cores (either 2.3GHz or 2.5GHz each) and 23GB of RAM, running Debian Linux 7.0.

Each node executes a client process, with one application thread per core, and a tablet server process. The default aggressive prefetching policy is used, unless otherwise noted. The staleness bound configuration for any execution is described by the "wpc" and "slack" values. The units for the WPC value is iterations, so wpc=2 means that two iterations are performed in each clock period. The units for slack is clocks, which is the same as iterations if wpc=1. Recall that, generally speaking, BSP is wpc=1 and slack=0, A-BSP is wpc=N and slack=0, and SSP is wpc=1 and slack=N.

Application benchmarks. We use the three example applications described in Section 4: Topic Modeling (TM), Matrix Factorization (MF), and PageRank (PR). Table 1 summarizes the problem sizes.

App.	# of rows	Row size (bytes)	# of row accesses per iteration
TM	400k	800	600m
MF	500k	800	400m
PR	685k	24	15m

Table 1: Problem sizes of the ML application benchmarks.

For TM, we use the *Nytimes* dataset, which contains 300k documents, 100m words, and a vocabulary size of 100k. We configure the application to generate 100 topics on this dataset. The quality of the result is defined as the loglikelihood of the model, which is a value that quantifies how likely the model can generate the observed data. A higher value indicates a better model.

For MF, we use the *Netflix* dataset, which is a 480k-by-18k sparse matrix with 100m known elements. We configure the application to factor it into the product of two matrices with rank 100, using an initial step size of 5e-10. Result quality is defined as the summation of squared errors, and a lower value indicates a better solution.

For PageRank, we use the *web-BerkStan* dataset [24], which is a web graph with 685k nodes and 7.6m edges. We configure the application to use a damping factor of 0.85. Because there is no result quality criteria for PageRank, we define it to be the summation of squared errors from a "ground truth result", which we obtain by running a sequential PageRank algorithm on a single machine for a relatively large number of iterations (100).

the probability that a word belongs to a certain topic.

[3] Here "sparse" means most elements are unknown.

A low value indicates a better solution.

We extract result quality data 16 times, evenly spaced, during each execution. To do so, the application creates a background thread in each client process and rotates the extraction among them. For TM and PR, the result quality is computed during the execution by a background thread in each client. For MF, the computation is very time-consuming (several minutes), so we instead have LazyTable take snapshots (see Section 3.3) and compute the quality off-line.

5.2 Exploiting staleness w/ A-BSP and SSP

A-BSP. Figure 3(a) shows performance effects on TM of using A-BSP with different WPC settings. The leftmost graph shows overall *convergence speed*, which is result quality as a function of execution time, as the application converges. WPC settings of 2 or 4 iterations significantly outperform settings of 1 (BSP) or 8, illustrating the fact that both too little and too much staleness is undesirable. One way to look at the data is to draw a horizontal line and compare the time (X axis) required for each setting to reach a given loglikelihood value (Y axis). For example, to reach -9.5e8, WPC=1 takes 236.2 seconds, while WPC=2 takes only 206.6 seconds.

The middle and rightmost graphs help explain this behavior, showing the *iteration speed* and *iteration effectiveness*, respectively. The middle graph shows that, as WPC increases, iterations complete faster. The rightmost graph shows that, as WPC increases, each iteration is less effective, contributing less to overall convergence such that more iterations are needed. The overall convergence speed can be thought of as the combination of these two metrics. Because the iteration speed benefit exhibits diminishing returns, and the iteration effectiveness does not seem to, there ends up being a sweet spot.

SSP. Figure 3(b) shows the same performance effects when using SSP with different slack settings. Similar trends are visible: more staleness increases iteration speed, with diminishing returns, and decreases iteration effectiveness. Empirically, the sweet spot is at slack=3.

We show the behavior for *infinite* slack, for comparison, which is one form of non-blocking asynchronous execution. Although it provides no guarantees, it behaves reasonably well in the early portion of the execution, because there are no major straggler issues and the aggressive prefetching mechanism refreshes the data even though it is not required. But, it struggles in the final (fine-tuning) stages of convergence. Even the BSP baseline (slack=0) finishes converging faster.

A-BSP vs. SSP. Figure 3(c) uses the same three-graph approach to compare four configurations: BSP, the best-performing A-BSP (wpc=4), the best-performing SSP from above (slack=3), and the best-performing overall configuration (a hybrid with wpc=2 and slack=1). Other than BSP, they all use the same staleness bound: wpc $\times$ (slack + 1). The results show that SSP outperforms both BSP and A-BSP, and that the best configuration is the hybrid. For example, to reach -9.5e8, A-BSP takes 237.0 seconds, while the two SSP options take 222.7 seconds and 193.7 seconds, respectively. Indeed, across many experiments with all of the applications, we generally find that the best setting is slack=1 with wpc set to one-half of the best staleness bound value. The one clock of slack is enough to mitigate intermittent stragglers, and using larger WPC amortizes communication costs more.

Looking a bit deeper, Figure 4 shows a breakdown of iteration speed into two parts: computation time and "wait time", which is the time that the client application is blocked by a LazyTable `read` due to data freshness misses. As expected, the iteration speed improvements from larger staleness bounds (from the first bar to the second group to the third group) come primarily from reducing wait times. Generally, the wait time is a combination of waiting for stragglers and waiting for fetching fresh-enough data. Because there is minimal straggler behavior in these experiments, almost all of the benefit comes from reducing the frequency of client cache freshness misses. The diminishing returns are also visible, caused by the smaller remaining opportunity (i.e., the wait time) for improvement as the staleness bound grows.

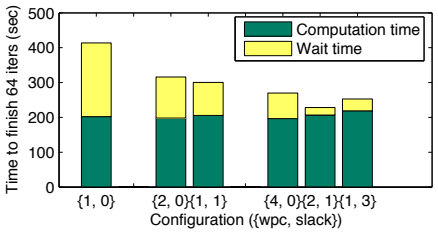
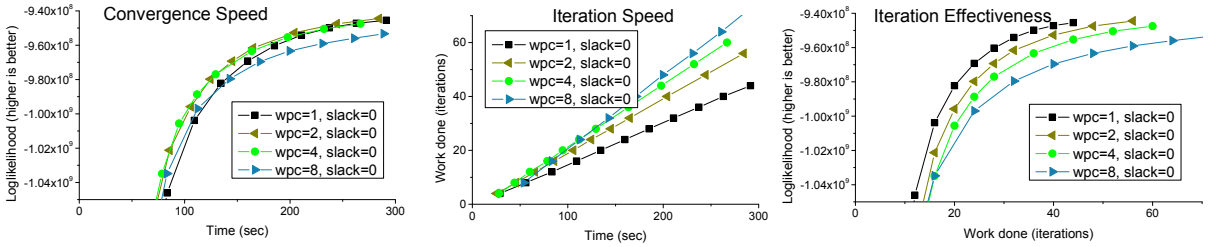


Figure 4: Time consumption distribution of TM.

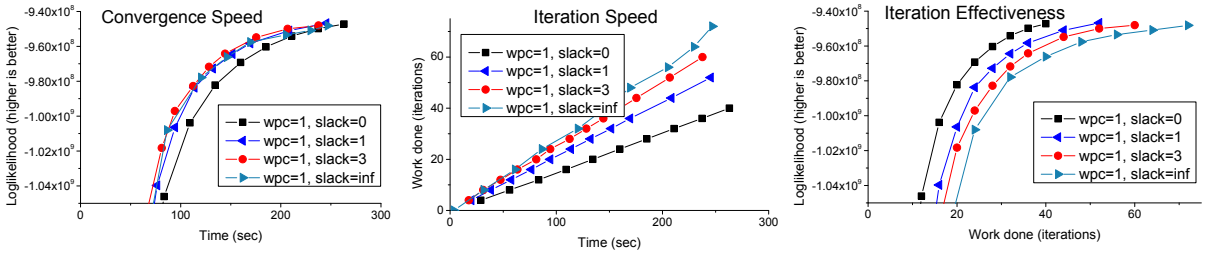
Matrix factorization and PageRank. Space limits preclude us from including all results, but Figure 5 shows the convergence speed comparison. For MF, we observe less tolerance for staleness, and SSP with slack=1 performs the best, while A-BSP with wpc=2 actually struggles to converge.[4]

PageRank behaves more similarly to Topic Modeling, but we found that A-BSP {4,0} slightly outperforms SSP {2,1} and significantly outperforms SSP {1,3} in this case. This counter-intuitive result occurs because of the increased communication overheads associated with SSP, which aggressively sends and fetches data every clock, combined with PageRank's lower computation work. When the communication throughput, not syn-

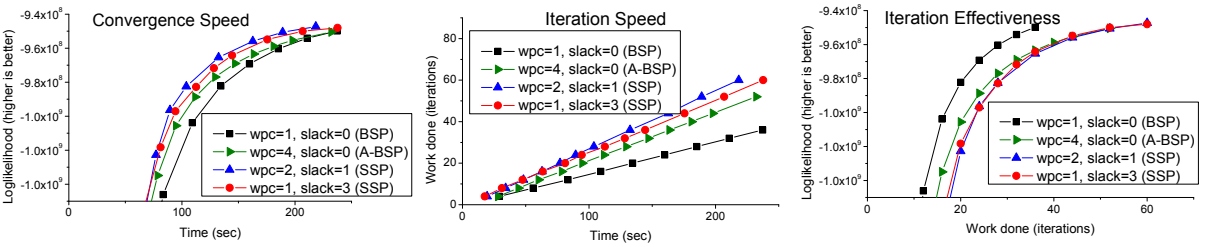
[4]We found, empirically, that MF does not diverge as shown with smaller step sizes, which bound the error from staleness to lower values, but we show this data because it is for the best step size for BSP.



(a) Tuning WPC on A-BSP



(b) Tuning slack on SSP



(c) Direct comparison of SSP with A-BSP

Figure 3: Different ways of exploiting data staleness in Topic Modeling.

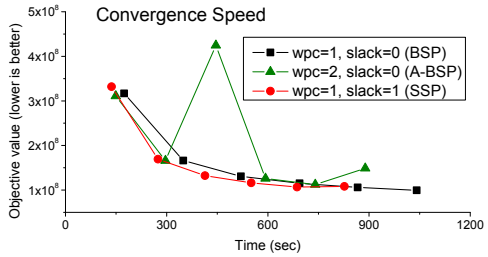
chronization overheads or straggler delays, limit iteration speeds, SSP's tendency to use more communication can hurt performance if it does so too aggressively. Note, however, that maximizing WPC with slack=1 remains close to the best case, and is more robust to larger straggler issues. We will examine this communication overhead of SSP in more detail in Section 5.4.

5.3 Influence of stragglers

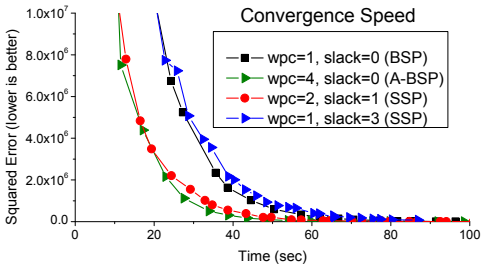
The original motivation for SSP was to tolerate intermittent stragglers [12], but our carefully controlled experimental setup exhibits minimal execution speed variation. This section examines Topic Modeling performance under two types of straggler behavior.

Delayed threads. First, we induced stragglers by having application threads sleep in a particular schedule, confirming that a complete system addresses such stragglers as predicted in the HotOS paper [12]. Specifically, the threads on machine-1 sleep d seconds at iteration-1, and then the threads on machine-2 sleep d seconds at iteration-2, and so on. Figure 6 shows the average time per iteration, as a function of d, for Topic Modeling via BSP, A-BSP, and SSP. Ideally, the average time per iteration would increase by only $\frac{d}{N}$ seconds, where N is the number of machines, because each thread is delayed d seconds every N iterations. For A-BSP, the average iteration time increases linearly with a slope of 0.5, because the threads synchronize every two iterations (wpc=2), at which time one delay of d is experienced. For SSP, the effect depends on the magnitude of d relative to the un-delayed time per iteration ($\approx$4.2 seconds). When d is 6 or less, the performance of SSP is close to ideal, because the delays are within the slack. Once d exceeds the amount that can be mitigated by the slack, the slope matches that of A-BSP, but SSP stays 1.5 seconds-per-iteration faster than A-BSP.

Background work. Second, we induced stragglers with competing computation, as might occur with background activities like garbage collection or short high-priority jobs. We did these experiments on 32 machines of the VM cluster, using a background "disrupter" process. The disrupter process creates one thread per core (8 in the VM cluster) that each perform CPU-intensive work when activated; so when the disrupter is active, the CPU scheduler will give it half of the CPU resources (or more, if the TM thread is blocked). Using discretized time slots of size t, each machine's disrupter is active in a time slot with a probability of 10%, independently determined.



(a) Matrix Factorization



(b) PageRank

Figure 5: Synchronization-staleness tradeoff examples.

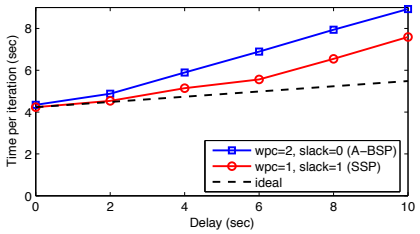


Figure 6: Influence of delayed threads.

Figure 7 shows the increase in iteration time as a function of t. Ideally, the increase would be just 5%, because the disrupters would take away half of the CPU 10% of the time. The results are similar to those from the previous experiment. SSP is close to ideal, for disruptions near or below the iteration time, and consistently able to mitigate the stragglers' impact better than A-BSP.

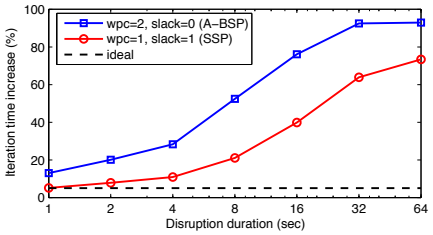


Figure 7: Influence of background disrupting work. With no disruption, each iteration takes about 4.2 seconds.

5.4 Examining communication overhead

Table 2 lists the network traffic of A-BSP {4, 0}, SSP {2, 1}, and SSP {1, 3} for Topic Modeling. We denote the traffic from client to server as "sent" and that from server to client as "received". The "sent" traffic is mainly caused by `update`, and the "received" traffic is mainly caused by `read`. For a configuration that halves the WPC, the sent traffic will be doubled, because the number of clocks for the same number of iterations is doubled. However, the received traffic is less than doubled, because for SSP, if the data received from the server is fresh enough, it can be used in more than one clock of computation.

As discussed in Section 5.2, this extra communication can cause SSP to perform worse than A-BSP in some circumstances. Much of the issue is the CPU overhead for processing the communication, rather than network limitations. To illustrate this, we emulate a situation where there is zero computation overhead for communication. Specifically, we configure the application to use just 4 application threads, and then configure the VMs to use either 4 cores (our normal 1 thread/core) or 8 cores (leaving 4 cores for doing the communication processing without competing with the application threads). Because we have fewer application threads, using WPC=1 would be a lot of computation work per clock. As a result, we make WPC smaller so that it is similar to our previous results: {wpc=0.25, slack=0} for A-BSP and {wpc=0.125, slack=1} for SSP. Figure 8 shows the time for them to complete the same amount of work. The primary takeaway is that having the extra cores makes minimal difference for A-BSP, but significantly speeds up SSP. This result suggests that SSP's potential is much higher if the CPU overhead of communication were reduced in our current implementation.

Config.	Bytes sent	Bytes received
wpc=4, slack=0	33.0 M	29.7 M
wpc=2, slack=1	61.9 M	51.0 M
wpc=1, slack=3	119.4 M	81.5 M

Table 2: Bytes sent/rec'd per client per iteration of TM.

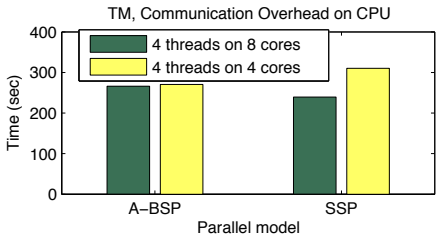


Figure 8: CPU overhead of communication.

5.5 Prefetching and throughput

Figure 9 shows the importance of prefetching, highlighting the value of LazyTable's iteration-aware prefetching scheme. The time per iteration, partitioned into com-

putation time and wait time, is shown for each of the three applications using SSP. In each case, the prefetching significantly reduces the wait time. There are two reasons that the speed up for MF is higher than for TM and PR. First, the set of rows accessed by different application threads overlap less in MF than in TM; so, when there is no prefetching, the rows used by one thread on a machine are less likely to be already fetched by another thread and put into the shared process cache. Second, each iteration is longer in MF, which means the same slack value covers more work; so, with prefetching, the threads are less likely to have data misses.

Note that, as expected, prefetching reduces wait time but not computation time. An interesting lesson we learned when prefetching was added to LazyTable is that it tends to reduce the best-choice staleness bound. Because increased staleness reduces wait time, prefetching's tendency to reduce wait time reduces the opportunity to increase iteration speed. So, the iteration effectiveness component of convergence speed plays a larger role in determining the best staleness bound. Before adding prefetching, we observed that higher staleness bounds were better than those reported here.

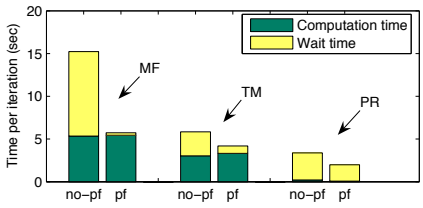


Figure 9: Time per iteration with/without prefetching for all three applications with {wpc=1, slack=1}.

6 Related work

In a HotOS workshop paper [12], we proposed SSP, briefed an early LazyTable prototype that implemented it, and did a couple of experiments to show that it helps mitigate delayed thread stragglers. We recently published follow-on work in an ML conference [18] that provides evidence of convergence for several ML algorithms under SSP, including proofs that provide theoretical justification for SSP's bounded staleness model. Some experiments comparing SSP to BSP show performance improvement, but the focus is on the ML algorithm behavior. This paper makes several important contributions beyond our previous papers: we describe a more general view of bounded staleness covering A-BSP as well as SSP, we describe in detail a system that supports both, we explain design details that are important to realizing their performance potential, and we thoroughly evaluate both and show the strengths and weaknesses for each. Importantly, whereas SSP almost always outperforms BSP (as commonly used in ML work) significantly, this paper makes clear that A-BSP can be as effective when straggler issues are minor.

The High Performance Computing community – which frequently runs applications using the BSP model – has made much progress in eliminating stragglers caused by hardware or operating system effects [13, 27]. While these solutions are very effective at reducing "operating system jitter", they are not intended to solve the more general straggler problem. For instance, they are not applicable to programs written in garbage collected languages, nor do they handle algorithms that inherently cause stragglers during some iterations.

In large-scale networked systems, where variable node performance, unpredictable communication latencies and failures are the norm, researchers have explored relaxing traditional barrier synchronization. For example, Albrecht et al. [2] describe partial barriers, which allow a fraction of nodes to pass through a barrier by adapting the rate of entry and release from the barrier. This approach does not bound how far behind some nodes may get, which is important for ML algorithm convergence.

Another class of solutions attempts to reduce the need for synchronization by restricting the structure of the communication patterns. For example, GraphLab [25, 26] programs structure computation as a graph, where data can exist on nodes and edges. All communication occurs along the edges of this graph. If two nodes on the graph are sufficiently far apart they may be updated without synchronization. This model can significantly reduce synchronization in some cases. However, it requires the application programmer to specify the communication pattern explicitly.

Considerable work has been done in describing and enforcing relaxed consistency in distributed replicated services. For example, the TACT model [31] describes consistency along three dimensions: numerical error, order error and staleness. Other work [30] classifies existing systems according to a number of consistency properties, specifically naming the concept of bounded staleness. Although the context differs, the consistency models have some similarities.

In the database literature, bounded staleness has been applied to improve update and query performance. LazyBase [11] allows staleness bounds to be configured on a per-query basis, and uses this relaxed staleness to improve both query and update performance. FAS [29] keeps data replicated in a number of databases, each providing a different freshness/performance tradeoff. Data stream warehouses [16] collect data about timestamped events, and provide different consistency depending on the freshness of the data. The concept of staleness (or freshness/timeliness) has also been applied in other fields such as sensor networks [20], dynamic web content generation [22], web caching [9], and information systems [7].

Of course, one can ignore consistency and synchronization altogether, relying on a best-effort model for updating shared data. Yahoo! LDA [1] as well as most solutions based around NoSQL databases rely on this model. While this approach can work well in some cases, having no staleness bounds makes confidence in ML algorithm convergence difficult.

7 Conclusion

Bounded staleness reduces communication and synchronization overheads, allowing parallel ML algorithms to converge more quickly. LazyTable supports parallel ML execution using any of BSP, A-BSP, or SSP. Experiments with three ML applications executed on 500 cores show that both A-BSP and SSP are effective in the absence of stragglers. SSP mitigates stragglers more effectively, making it the best option in environments with more variability, such as clusters with multiple uses and/or many software layers, or for algorithms with more variability in the work done per thread within an iteration.

Acknowledgements. We thank the members and companies of the PDL Consortium (including Actifio, APC, EMC, Emulex, Facebook, Fusion-IO, Google, Hewlett-Packard, Hitachi, Huawei, Intel, Microsoft, NEC Labs, NetApp, Oracle, Panasas, Riverbed, Samsung, Seagate, STEC, Symantec, VMWare, Western Digital). This research is supported in part by the Intel Science and Technology Center for Cloud Computing (ISTC-CC), National Science Foundation under awards CNS-1042537 and CNS-1042543 (PRObE [15]), and DARPA Grant FA87501220324.

References

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

[2] ALBRECHT, J., TUTTLE, C., SNOEREN, A. C., AND VAHDAT, A. Loose synchronization for large-scale networked systems. In *USENIX Annual Tech* (2006).

[3] ANANTHANARAYANAN, G., KANDULA, S., GREENBERG, A., STOICA, I., LU, Y., SAHA, B., AND HARRIS, E. Reining in the outliers in map-reduce clusters using Mantri. In *OSDI* (2010).

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

[5] BECKMAN, P., ISKRA, K., YOSHII, K., AND COGHLAN, S. The influence of operating systems on the performance of collective operations at extreme scale. *CLUSTER* (2006).

[6] BLEI, D. M., NG, A. Y., AND JORDAN, M. I. Latent dirichlet allocation. *JMLR* (2003).

[7] BOUZEGHOUB, M. A framework for analysis of data freshness. In *IQIS* (2004).

[8] BRADLEY, J. K., KYROLA, A., BICKSON, D., AND GUESTRIN, C. Parallel coordinate descent for L1-regularized loss minimization. In *ICML* (2011).

[9] BRIGHT, L., AND RASCHID, L. Using latency-recency profiles for data delivery on the web. In *VLDB* (2002).

[10] BRIN, S., AND PAGE, L. The anatomy of a large-scale hypertextual web search engine. *Computer Networks* (1998).

[11] CIPAR, J., GANGER, G., KEETON, K., MORREY III, C. B., SOULES, C. A., AND VEITCH, A. LazyBase: Trading freshness for performance in a scalable database. In *Eurosys* (2012).

[12] CIPAR, J., HO, Q., KIM, J. K., LEE, S., GANGER, G. R., GIBSON, G., KEETON, K., AND XING, E. Solving the straggler problem with bounded staleness. In *HotOS* (2013).

[13] FERREIRA, K. B., BRIDGES, P. G., BRIGHTWELL, R., AND PEDRETTI, K. T. The impact of system design parameters on application noise sensitivity. In *CLUSTER* (2010).

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

[15] GIBSON, G., GRIDER, G., JACOBSON, A., AND LLOYD, W. PRObE: A thousand-node experimental cluster for computer systems research. *USENIX ;login:* (2013).

[16] GOLAB, L., AND JOHNSON, T. Consistency in a stream warehouse. In *CIDR* (2011).

[17] GRIFFITHS, T. L., AND STEYVERS, M. Finding scientific topics. *Proc. National Academy of Sciences USA* (2004).

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

[19] HOFFMAN, M., BACH, F. R., AND BLEI, D. M. Online learning for latent dirichlet allocation. In *NIPS* (2010).

[20] HUANG, C.-T. Loft: Low-overhead freshness transmission in sensor networks. In *SUTC* (2008).

[21] KREVAT, E., TUCEK, J., AND GANGER, G. R. Disks are like snowflakes: No two are alike. In *HotOS* (2011).

[22] LABRINIDIS, A., AND ROUSSOPOULOS, N. Balancing performance and data freshness in web database servers. In *VLDB* (2003).

[23] LANGFORD, J., SMOLA, A. J., AND ZINKEVICH, M. Slow learners are fast. In *NIPS* (2009).

[24] LESKOVEC, J., LANG, K. J., DASGUPTA, A., AND MAHONEY, M. W. Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. *Internet Mathematics* (2009).

[25] LOW, Y., GONZALEZ, J., KYROLA, A., BICKSON, D., GUESTRIN, C., AND HELLERSTEIN, J. M. GraphLab: A new parallel framework for machine learning. In *UAI* (2010).

[26] LOW, Y., GONZALEZ, J., KYROLA, A., BICKSON, D., GUESTRIN, C., AND HELLERSTEIN, J. M. Distributed GraphLab: A framework for machine learning and data mining in the cloud. *PVLDB* (2012).

[27] PETRINI, F., KERBYSON, D. J., AND PAKIN, S. The case of the missing supercomputer performance: Achieving optimal performance on the 8,192 processors of ASCI Q. In *Supercomputing* (2003).

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

[29] RÖHM, U., BÖHM, K., SCHEK, H.-J., AND SCHULDT, H. FAS: A freshness-sensitive coordination middleware for a cluster of OLAP components. In *VLDB* (2002).

[30] TERRY, D. Replicated data consistency explained through baseball. Tech. rep., Microsoft Research, 2011.

[31] YU, H., AND VAHDAT, A. Design and evaluation of a conit-based continuous consistency model for replicated services. *ACM TOCS* (2002).

[32] ZeroMQ: The intelligent transport layer. http://www.zeromq.org/.