Exploiting bounded staleness to speed up Big Data analytics

Name1
Affiliation1
Email1

Name2 Name3
Affiliation2/3
Email2/3

Abstract

Many modern machine learning (ML) algorithms are iterative, converging on a final solution via many iterations over the input data. This paper explores approaches to exploiting these algorithms' convergent nature to improve performance, by allowing parallel and distributed threads to use loose consistency models for shared algorithm state. Specifically, we focus on *bounded staleness*, in which each thread can see a view of the current intermediate solution that may be a limited number of iterations out-of-date. Allowing staleness reduces communication costs (batched updates and cached reads) and synchronization (less waiting for locks or straggling threads). One approach is to increase the number of iterations between barriers in the oft-used Bulk Synchronous Parallel (BSP) model of parallelizing, which mitigates these costs when all threads proceed at the same speed. A more flexible approach, called Stale Synchronous Parallel (SSP), avoids barriers and allows threads to be a bounded number of iterations ahead of the current slowest thread. Extensive experiments with ML algorithms for topic modeling, collaborative filtering, and PageRank show that both approaches significantly increase convergence speeds, behaving similarly when there are no stragglers, but SSP outperforms BSP in the presence of stragglers.

1. Introduction

Large-scale machine learning (ML) has become a critical building block for many applications and services, as the Big Data concept gains more and more momentum. Parallel ML implementations executed on clusters of servers are increasingly common, given the total computation work often involved. These implementations face the same challenges as any parallel computing activity, including performance overheads induced by inter-thread communication and by synchronization among threads.

Among the many ML approaches being used, many fall into a category often referred to as *iterative convergent algorithms*. These algorithms start with some guess at a solution and refine this guess over a number of iterations over the input data, improving a goodness-of-solution objective function until sufficient convergence or goodness has been reached. The key property is convergence, which allows such algorithms to find a good solution given an initial guess. Likewise, minor errors in the adjustments made by any given iteration will not prevent success.

Distributed implementations of iterative convergent algorithms tend to shard the input data and follow the Bulk Synchronous Parallel (BSP) model. The current intermediate solution is shared by all threads, and each worker thread processes its subset of the input data (e.g., news documents or per-user movie ratings). Each worker thread makes adjustments to the current solution, as it processes each of its input data items, to make it match that item better. In a BSP execution, coordination happens whenever all threads have completed a certain amount of work, which we will refer to as a "clock". All threads work on clock N with a snapshot of the shared state that includes all updates from clock $N-1$, which requires exchange of data updates and a barrier synchronization at the end of each clock.

Although not often discussed as such, BSP relies on the algorithm having a tolerance of staleness. During a given clock, worker threads do not see the adjustments made by others; each of them determines adjustments independently, and those adjustments are aggregated only at the end of the clock. Indeed, these independent adjustments are a source of error that may require extra iterations. But, by coordinating only once per clock, BSP reduces communication costs (by batching updates) and synchronization delays (by reducing their frequency). While most ML practitioners equate an iteration (one pass over the input data) with a BSP clock, doing so fails to recognize staleness as a parameter to be tuned for improved performance.

This paper examines two approaches to more fully exploiting staleness to improve ML convergence speeds. Allowing more staleness often results in faster convergence, but only to a point. While it reduces communication and synchronization,

making iterations faster, it can also increase the error in any given iteration. So, there is a tradeoff between decreasing iteration times and increasing iteration counts, determined by the *staleness bound*. In BSP, the maximum staleness corresponds to the work per clock. We find that the best value is often *not* equal to one iteration, and we use the term *Arbitrarily-sized BSP* (A-BSP) to highlight this fact.[1]

A different, recently proposed, approach to exploiting staleness is the *Stale Synchronous Parallel* (SSP) model [15], which generalizes BSP by relaxing the requirement that all threads be working on the same clock at the same time. Instead, threads are allowed to progress a bounded number (the "slack") of clocks ahead of the slowest thread. Like the BSP model, the SSP model bounds staleness (to the product of the slack and the work per clock). But, unlike the BSP model, SSP's more flexible executions can better mitigate many transient straggler effects.

We describe a system, called NanTable[2], 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 two primary contributions. First, it provides the first detailed description of a system that implements BSP, A-BSP, and SSP, including techniques used and lessons learned in tuning its performance. Second, it provides the first comparative evaluations of A-BSP and SSP, exploring their relative merits when using the same staleness bound and factors contributing to the best staleness bound.

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 bounded staleness.

2.1 Iterative convergent algorithms and 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 [22]. 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

[1] A-BSP is not a different model than BSP, but we use the additional term to distinguish traditional use of BSP (by ML practitioners) from explicit tuning of staleness in BSP.

[2] NanTable (**N**ot **a n**ame **Table**) is a placeholder name for blind submission.

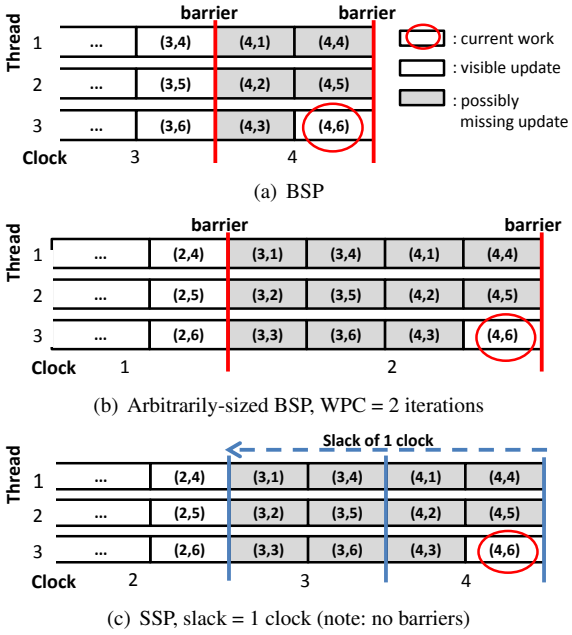


Figure 1. BSP, A-BSP, and SSP models. A block with (i, j) represents the j-th work-unit of the i-th iteration. Focusing on thread 3's execution of the circled work-unit, the shaded blocks indicate the updates that it may not see, under each model. SSP is a pipelined version of A-BSP, where the work of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

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 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 coordinating/synchronizing on every single update to the shared state.

Accepting some staleness allows batching of updates, fewer cache misses, 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 [25]. 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 [3] 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 the approaches we consider) provide no mechanisms to enforce such bounds.

2.3 Expanding staleness exploitation

This section describes two approaches to more fully exploit bounded 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 [24], hardware failures [6], imbalanced data distribution among tasks, garbage collection in high-level languages, and even operating system effects [8, 30]. 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) [15]) was proposed as an approach to mitigating the straggler effect. SSP keeps the WPC concept from BSP, but eliminates the 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), since 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 $\{wpc \times (s+1), 0\}$. This fact is important in understanding the relative merits of these approaches, whereas previous papers unfairly compare SSP only to BSP.

The right comparison is SSP vs. A-BSP, where both use an equivalent staleness bound, as described. A-BSP requires a barrier at the end of each clock, so it's very sensitive to stragglers in the system. SSP, however, 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 one 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. NanTable Design and Implementation

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

NanTable holds globally shared data in a cluster of *tablet servers*, and application programs access the data through the *client library*, as depicted in 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.

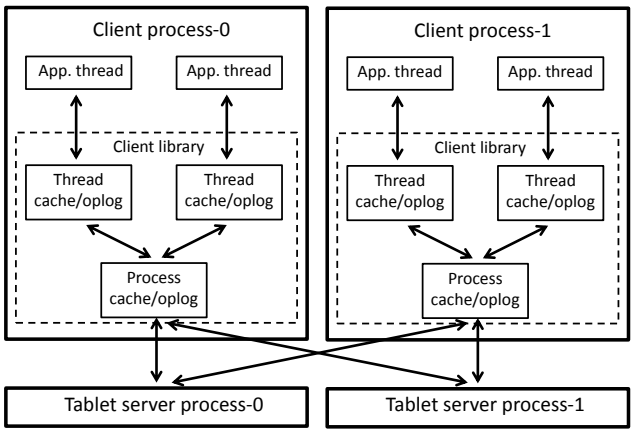


Figure 2. NanTable running two application processes with two application threads each.

3.1 NanTable Abstraction

This subsection describes the data model, user interface, and consistency guarantees provided by NanTable.

Data Model. Globally shared data is organized in a collection of *rows* in NanTable. 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. Each row is uniquely indexed by a sparse (table_id, row_id) tuple. Rows with the same table_id should have the same type.

Having each data unit being a row, NanTable simplifies the implementation of many ML algorithms, which can usually be expressed as operations on matrices and vectors. Each vector can naturally be stored as a row in NanTable.

Operations. NanTable 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 NanTable client library, which borrow from Piccolo [31] and add staleness awareness.

- `read(table_id, row_id, slack)`: Atomically retrieves a row. If the available version of the row is not within the given `slack` bound, the calling thread is blocked. This is the only function that blocks the calling thread.
- `update(table_id, row_id, delta)`: Atomically updates a row by `delta` by aggregating it to the current row value 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 cache copy 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. NanTable 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.

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

3.2 System Components

This section describes the NanTable tablet servers and client library. The NanTable prototype is written in C++, using the ZeroMQ [1] 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, and the current implementation does not replicate row data or support multi-row updates. As a result, tablet server processes are independent. The current implementation uses a simple, static row-server mapping: $tablet\_id = row\_id \mod num\_of\_tablets$. The current implementation 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 will put it into a *pending writes list*. Updates in this list will only be applied to master data 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, which indicates the updates from that client that 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 NanTable 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.

NanTable has two levels of caches/oplogs in the client library: 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 whose access frequencies exceed a given threshold. 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-writes", 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 push all updates of clock t in the process oplog to

the appropriate tablet servers. The background thread will also send a `clock` message to *all* tablet servers, which is like a `commit` that tells each server to apply the updates to the data. 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 a 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. Along with the row data, 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-writes" for the application thread, it applies to the received data row 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, NanTable 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 to overlap fetch time with computation. While this requires an application to know beforehand what it will read in the current clock, the ML applications in our study all have this property because of their iterative nature. 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 that property, NanTable 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 strategy 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. So we propose two prefetching strategies. NanTable 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 NanTable. 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. NanTable 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 will have likely 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. NanTable currently only implements 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 if we want to restore a failed execution from a snapshot, we still need to make assumptions about the application. For example, client-1 might have generated its updates for clock t based on client-2's updates from clock $t + 1$. When we restore 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 that can tolerate data staleness can usually tolerate this kind of error as well.[3]

4. Example ML applications

This section describes three ML applications that employ different algorithms representing a range of oft-used ML approaches. For each, we also explain how we implement it on NanTable for our experiments.

[3] The very recent convergence proofs for several ML algorithms under SSP [21] appear to extend readily to this setting, providing some theoretical justification for this observation.

Topic Modeling (TM). Topic Modeling (TM) is a class of problems that assign *topic vectors* to documents. The topic vector represents how well a document is represented by a topic. For instance, a document about a financial scandal may be classified as "50% politics, 30% finance, 20% legal", while another document about the professional baseball draft may be "80% sports, 20% finance".

The specific model on which we focus is known as Latent Dirichlet Allocation [9] (LDA). LDA is a generative model. It assumes that each document can be summarized as a multinomial distribution over topics. Similarly, each topic is a multinomial distribution over words. So, to generate a document, words would be generated one-by-one: first a topic is chosen from the document's topic distribution, then a word is chosen from the topic's word distribution. This procedure would continue, generating an infinite stream of words for each document.

The goal of LDA is to learn 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, the example application uses Gibbs sampling [20], a Monte Carlo technique. Gibbs sampling is a technique to estimate probability distribution via sampling in cases where direct estimation is difficult. It requires that the algorithm designer specify how to sample from any conditional distribution. That is, given fixed values for all variables except one, sample from the distribution of the remaining variable. Gibbs sampling iterates through all variables, updating the current value of each variable by sampling in this way.

We implement topic modeling on NanTable as follows. We divide the set of input documents evenly among all application threads. The application creates two tables, a document-topic table and a word-topic table. Each row of the document-topic table represents the topic assignment of one document, while each row of the word-topic table represents the topic assignment of one unique word. The row data type is an STL vector, and the size of each row equals the number of topics. The word-topic table also contains a summation row that is the summation of all the other rows. This summation row is necessary for the collapsed Gibbs sampling algorithm to calculate the probability that a word belongs to a certain topic. In each iteration, all application threads pass through all the words in their input documents. For a word with $word\_id$ in a document with doc_id, the thread will read the doc_id-th row of the document-topic table and the $word\_id$-th row plus the summation row of the word-topic table, and it will update these rows based on the new topic assignment calculated.

Matrix factorization (MF). The second application is an example of collaborative filtering. Collaborative filtering attempts to predict user's preferences based on the known preferences of similar users. A familiar example of this is found in online movie services, where users rate movies they have seen. The service uses these ratings to predict the users' ratings for other movies. These predictions are used to present the user with a personalized list of movie recommendations.

One technique to perform such recommendations is by low-rank matrix factorization. In the movie example, this technique assumes that there is a large matrix, the *user-movie matrix*, with a row for each user and a column for each movie. The values in the matrix represent the users' ratings for each movie. However, only some values are known, and our goal is to fill in the rest of the matrix.

This is done by assuming that the matrix has low rank. That is, it is the product of a tall skinny matrix (the *left-matrix*) and a short wide matrix (the *right-matrix*). Once this factorization is found, any rating can be predicted by simply computing the dot product of the user's row in the user-genre matrix, and the movie's column in the genre-movie matrix. This matrix factorization problem can be solved by the stochastic gradient descent algorithm [18, 32].

To implement such matrix factorization on NanTable, we define the calculation on each non-zero element in the input sparse matrix data as one work-unit, and we partition them evenly among all application threads. The application creates two tables, one for the left-matrix and the other for the right-matrix. The row data type is an STL vector, and each row represents one row vector of the skinny left-matrix or one column vector of the short right-matrix. So, the size of a row equals the rank of the matrices. In each iteration, all application threads pass through all elements in their input data. For an element (i, j) in the original matrix, the thread reads and adjusts the i-th row of the left-matrix table and the j-th row of the right-matrix table to better fit that element.

PageRank. The PageRank algorithm assigns a weighted score (PageRank) to every vertex in a graph [13, 26]. A vertex's score measures its importance in the graph, with higher scores indicating higher importance. In particular, high-scoring vertices must have incoming links from many other high-scoring vertices; thus, the PageRank score can be viewed as a measure of authority (e.g., high-scoring webpages are more likely to have genuine content).

There are several methods to compute PageRank scores; we focus on the power iteration approach. This method essentially simulates web surfers navigating web pages, assuming that surfers are more likely to click on higher-scoring pages than lower-scoring ones. In each iteration, every web surfer clicks on a new page, and the PageRank score of each vertex is updated based on the proportion of visitors. Eventually, the PageRank scores stop changing, and the algorithm is terminated.

We implement an edge-scheduling version of PageRank on NanTable. In each iteration, the algorithm passes through all edges in the graph and updates the rank of the destination node according to the current rank of the source node. The set of edges are partitioned evenly among application threads. The application uses only one table, where each row holds the information of one node, including its in-degree, out-degree,

and current rank. In order to calculate the update on a page's rank, each thread also holds a vector locally to memorize the rank contribution of the node pairs for which it is responsible.

5. Evaluation

This section evaluates the A-BSP and SSP approaches via experiments with real ML applications on our NanTable prototype. The results show that (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; (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 NanTable configuration. Except where otherwise stated, the experiments use an 8-node cluster of 64-core machines. Each node has four 2.1 GHz 16-core Opteron 6272s and 128GB of RAM. 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 describe 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 non-zero 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. The quality of the result is defined as the summation of squared errors, and a lower value indicates a better solution.

For PageRank, we use the *web-BerkStan* dataset [27], which is a web graph with 685k nodes and 7.6m edges. We configure the application to use a damping factor of 0.85. Since there's 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). A low value indicates a better solution.

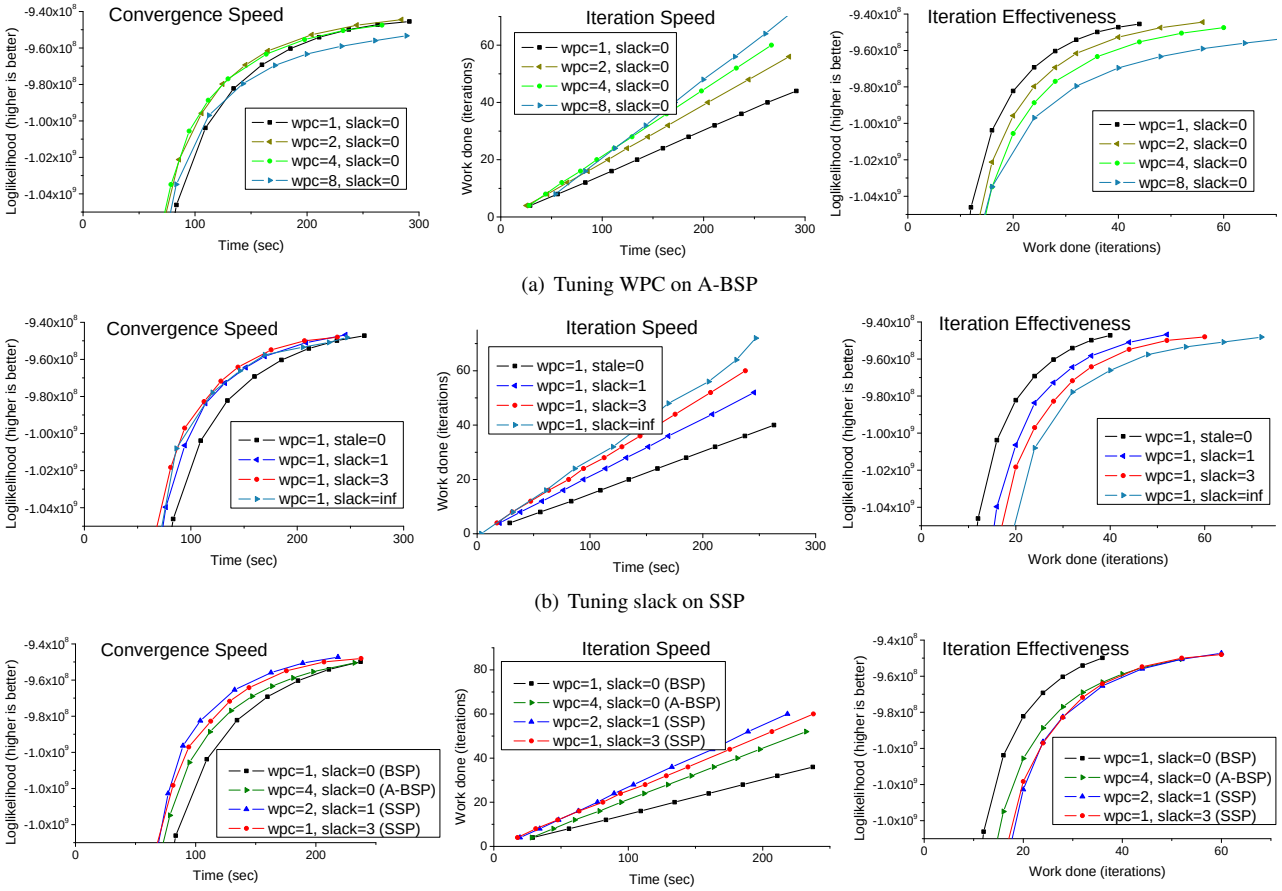
We extract result quality data 16 times, evenly spaced, during each execution. To do so, the application creates a background thread in each client process and rotates the extraction among them. For TM and PR, the result quality value is computed during the execution, based on reading the current solution with slack=0. For MF, the computation is very time-consuming (several minutes), so we instead take a snapshot (see Section 3.3) and compute the result quality off-line.

5.2 Exploiting staleness with A-BSP and SSP

A-BSP. Figure 3(a) shows performance effects on Topic Modeling of using A-BSP with different WPC settings. The leftmost graph shows overall *convergence speed*, which is the result quality as a function of execution time, as the application converges. The data shows that 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. Since 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



(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

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's 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 in NanTable `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 comes from a combination of waiting for stragglers and waiting for fetching fresh-enough data. Since there is minimal straggler behavior in these experiments, almost all of the benefit comes from reducing the frequency with which freshness misses in the client cache occur. The diminishing returns are also visible, caused by the smaller remaining opportunity (i.e., the wait time) for improvement as the staleness bound grows.

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 tol-

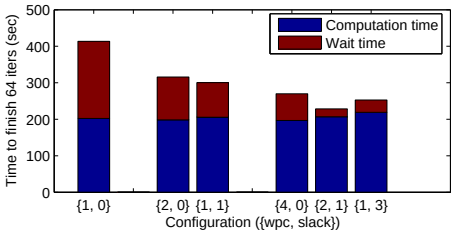
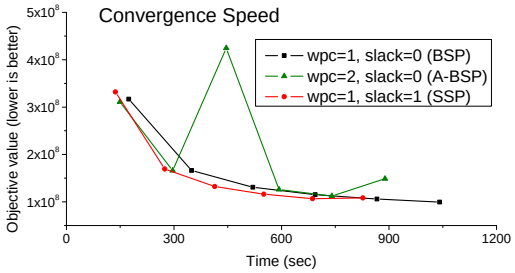
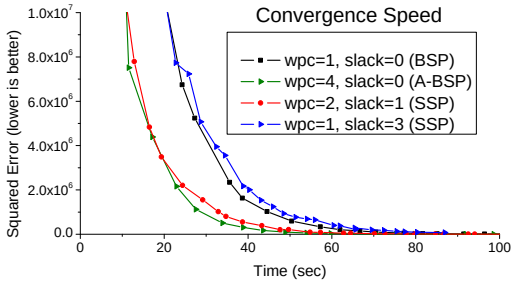


Figure 4. Time consumption distribution of TM.



(a) Matrix Factorization



(b) PageRank

Figure 5. Synchronization-staleness tradeoff in MF and PR.

erance 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 synchronization 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.

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

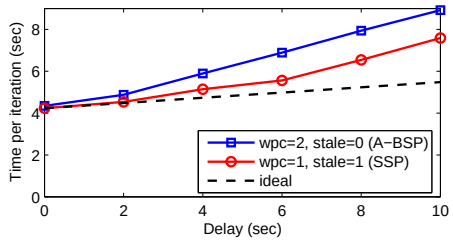


Figure 6. Influence of delayed threads

5.3 Influence of Stragglers

The original motivation for SSP was to tolerate intermittent stragglers [15], 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. Our first set of experiments repeats the straggler experiments of Cipar et al. [15] to confirm that our system matches the behavior that they claim. They induced stragglers by having application threads sleep in a particular schedule. 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 the time of which only one delay of d is experienced. For SSP, the effect depends on the magnitude of d relative to the un-delayed time per iteration (≈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. Our second set of experiments induces stragglers with competing computation, as might occur with background activities like garbage collection or short high-priority jobs. We do this experiment 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 perform CPU-intensive work when activated; so, the CPU scheduler will give the disrupter half of the CPU resources, when it is active. Using discretized time slots of size t, each machine's disrupter is active in a time slot with a probability of 10%, independently determined. Figure 7 shows the increase in iteration time as a function of t. Ideally, the increase would be just 5%, because the disrupters 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

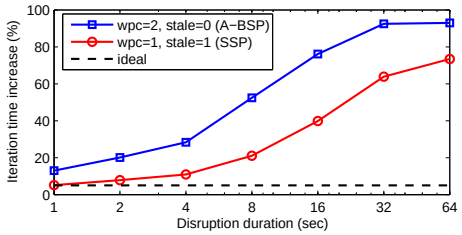


Figure 7. Influence of background disrupting work. When there's no disruption, each iteration takes about 4.2 seconds.

consistently able to mitigate the stragglers' impact better than A-BSP.

5.4 Examining communication overhead

Table 2 lists the network traffic of SSP {1, 3}, SSP {2, 1}, and A-BSP {4, 0} 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, which makes the updates sent to the server twice as frequently. 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.

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/received per client per iteration of TM.

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 exploit the ability of to configure the number of cores used in the VM cluster to 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). Since we have fewer application threads, using a WPC of one 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

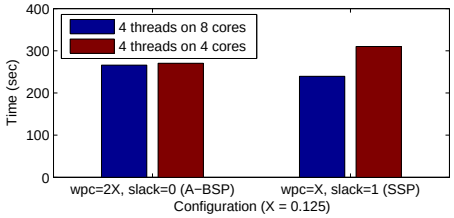


Figure 8. CPU overhead of communication

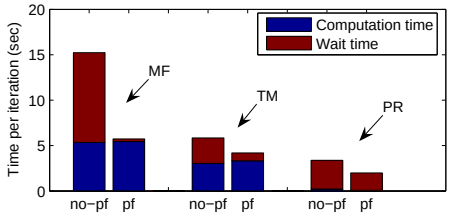


Figure 9. Time per iteration with/without prefetching for all three applications with {ipc=1, slack=1}.

the CPU overhead of communication were reduced in our existing prototype.

5.5 Prefetching and Throughput

Figure 9 shows the importance of prefetching, highlighting the value of NanTable's iteration-aware prefetching scheme. The time per iteration, partitioned into computation 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're two reasons for why the speed up of MF is higher than that of TM and PR. First, the set of rows accessed by different application threads overlap less in MF than in TM; so, when there's 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 NanTable is that it tends to reduce the best-choice staleness bound. Since 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 right staleness bound. Before adding prefetching, we observed that higher staleness bounds were better than those reported here.

Figure 10 compares conservative prefetching and aggressive prefetching, using a large slack value (7) to highlight the difference. Matching the intuition, aggresive prefetching increases iteration efficiency (higher is better) without significantly affecting iteration speeds.

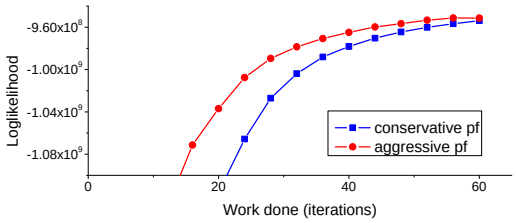


Figure 10. Iteration effectiveness comparison between conservative PF and aggressive PF for {ipc=1, slack=7}.

6. Related Work

Cipar et al. [15] proposed SSP in their HotOS workshop paper focused on addressing the straggler problem. They described an early prototype of a system called LazyTables that implemented SSP, and did a couple of experiments to show that it helps mitigate delayed thread stragglers. We obtained a copy of the now-accepted paper [21] submitted to NIPS that follows on that work, which provides evidence of convergence for several ML algorithms under SSP, including proofs. Such proofs provide theoretical justification for SSP's bounded staleness model. Some experiments comparing SSP to BSP show performance improvement, but little detail about the system is provided, and the comparisons fail to recognize that A-BSP is significantly better than BSP as commonly used in the ML community. Our work goes beyond those papers in several ways: 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.

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 [16, 17, 30, 36]. 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. [4] 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 some ML applications.

Another class of solutions attempts to reduce the need for synchronization by restricting the structure of the communication patterns. For example, GraphLab [28, 29] 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 [35] describes consistency along three dimensions: numerical error, order error and staleness. Other work [34] attempts to classify 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 databases literature, bounded staleness has been applied to improve update and query performance. LazyBase [14] allows staleness bounds to be configured on a per-query basis, and uses this relaxed staleness to improve both query and update performance. FAS [33] keeps data replicated in a number of databases, each providing a different freshness/performance tradeoff. Data stream warehouses [19] 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 [23], dynamic web content generation [5], web caching [12], and information systems [10].

It is possible to ignore consistency and synchronization altogether, and rely on a best-effort model for updating shared data. Yahoo! LDA [2] as well as most solutions based around NoSQL databases rely on this model. While this approach can work well in some cases, it may require careful design (and luck) to ensure that the algorithm is operating correctly.

7. Conclusion

Bounded staleness reduces communication and synchronization overheads, allowing parallel ML algorithms to converge more quickly. NanTable 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.

References

[1] ZeroMQ: the intelligent transport layer. http://www.zeromq.org/.

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

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

[4] J. Albrecht, C. Tuttle, A. C. Snoeren, and A. Vahdat. Loose synchronization for large-scale networked systems. In *USENIX Annual Tech*, pages 301–314, 2006.

[5] N. R. Alexandros Labrinidis. Balancing performance and data freshness in web database servers. pages pp. 393 – 404, September 2003.

[6] 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 USENIX conference on Operating systems design and implementation (OSDI)*, pages 1–16, 2010.

[7] Apache Mahout. Apache Mahout, http://mahout.apache.org. URL `http://mahout.apache.org`.

[8] P. Beckman, K. Iskra, K. Yoshii, and S. Coghlan. The influence of operating systems on the performance of collective operations at extreme scale. *Cluster Computing, 2006 IEEE International Conference on*, pages 1–12, 2006.

[9] D. M. Blei, A. Y. Ng, and M. I. Jordan. Latent dirichlet allocation. *J. Mach. Learn. Res.*, 3:993–1022, Mar. 2003. ISSN 1532-4435.

[10] M. Bouzeghoub. A framework for analysis of data freshness. In *Proceedings of the 2004 international workshop on Information quality in information systems*, IQIS '04, pages 59–67, 2004. ISBN 1-58113-902-0. .

[11] J. Bradley, A. Kyrola, D. Bickson, and C. Guestrin. Parallel coordinate descent for l1-regularized loss minimization. In L. Getoor and T. Scheffer, editors, *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, ICML '11, pages 321–328, New York, NY, USA, June 2011. ACM. ISBN 978-1-4503-0619-5.

[12] L. Bright and L. Raschid. Using latency-recency profiles for data delivery on the web. In *Proceedings of the 28th international conference on Very Large Data Bases*, VLDB '02, pages 550–561, 2002.

[13] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. *Computer networks and ISDN systems*, 30 (1):107–117, 1998.

[14] J. Cipar, G. Ganger, K. Keeton, C. B. Morrey, III, C. A. Soules, and A. Veitch. LazyBase: trading freshness for performance in a scalable database. In *Proceedings of the 7th ACM european conference on Computer Systems*, pages 169–182, 2012. ISBN 978-1-4503-1223-3. .

[15] J. Cipar, Q. Ho, J. K. Kim, S. Lee, G. R. Ganger, G. Gibson, K. Keeton, and E. Xing. Solving the straggler problem with bounded staleness. In *USENIX conference on Hot topics in operating systems (HotOS)*, 2013.

[16] A. C. Dusseau, R. H. Arpaci, and D. E. Culler. Effective distributed scheduling of parallel workloads. In *Proceedings of the ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 25–36, 1996. ISBN 0-89791-793-6. .

[17] K. B. Ferreira, P. G. Bridges, R. Brightwell, and K. T. Pedretti. The impact of system design parameters on application noise sensitivity. In *Proceedings of the 2010 IEEE International Conference on Cluster Computing*, CLUSTER '10, pages 146–155, Washington, DC, USA, 2010. IEEE Computer Society. ISBN 978-0-7695-4220-1. .

[18] R. Gemulla, E. Nijkamp, P. J. Haas, and Y. Sismanis. Large-scale matrix factorization with distributed stochastic gradient descent. In *KDD*, pages 69–77. ACM, 2011.

[19] L. Golab and T. Johnson. Consistency in a stream warehouse. In *CIDR 2011*, pages 114–122. .

[20] T. L. Griffiths and M. Steyvers. Finding scientific topics. *Proceedings of the National Academy of Sciences of the United States of America*, 101(Suppl 1):5228–5235, 2004.

[21] Q. Ho, J. Cipar, and et al. More effective distributed ml via a stale synchronous parallel parameter server. In *NIPS*, 2013, to appear.

[22] M. Hoffman, F. R. Bach, and D. M. Blei. Online learning for latent dirichlet allocation. In *advances in neural information processing systems*, pages 856–864, 2010.

[23] C.-T. Huang. Loft: Low-overhead freshness transmission in sensor networks. In *SUTC 2008*, pages 241–248, Washington, DC, USA, 2008. IEEE Computer Society. ISBN 978-0-7695-3158-8. .

[24] E. Krevat, J. Tucek, and G. R. Ganger. Disks are like snowflakes: no two are alike. In *Proceedings of the USENIX conference on Hot topics in operating systems*, pages 14–14, 2011.

[25] J. Langford, A. J. Smola, and M. Zinkevich. Slow learners are fast. In *Advances in Neural Information Processing Systems*, pages 2331–2339, 2009.

[26] P. Lawrence, B. Sergey, R. Motwani, and T. Winograd. The pagerank citation ranking: Bringing order to the web. Technical report, Stanford University, 1998.

[27] J. Leskovec, K. J. Lang, A. Dasgupta, and M. W. Mahoney. Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. *Internet Mathematics*, 6(1):29–123, 2009.

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

[29] Y. Low, G. Joseph, K. Aapo, D. Bickson, C. Guestrin, and M. Hellerstein, Joseph. Distributed GraphLab: A framework for machine learning and data mining in the cloud. *PVLDB*, 2012.

[30] F. Petrini, D. J. Kerbyson, and S. Pakin. The case of the missing supercomputer performance: Achieving optimal performance on the 8,192 processors of asci q. In *Proceedings of the ACM/IEEE conference on Supercomputing*, pages 55–, 2003. ISBN 1-58113-695-1. .

[31] R. Power and J. Li. Piccolo: building fast, distributed programs with partitioned tables. In *Proceedings of the USENIX conference on Operating systems design and implementation (OSDI)*, pages 1–14, 2010.

[32] H. Robbins and S. Monro. A stochastic approximation method. *Annals of Mathematical Statistics*, 22:400–407, 1951.

[33] U. Röhm, K. Böhm, H.-J. Schek, and H. Schuldt. Fas: a freshness-sensitive coordination middleware for a cluster of olap components. In *VLDB 2002*, pages 754–765. VLDB Endowment, 2002.

[34] D. Terry. Replicated data consistency explained through baseball. Technical Report MSR-TR-2011-137, Microsoft Research, October 2011.

[35] H. Yu and A. Vahdat. Design and evaluation of a conit-based continuous consistency model for replicated services. *ACM Transactions on Computer Systems*, 20(3):239–282, Aug. 2002.

[36] R. Zajcew, P. Roy, D. L. Black, C. Peak, P. Guedes, B. Kemp, J. LoVerso, M. Leibensperger, M. Barnett, F. Rabii, and D. Netterwala. An osf/1 unix for massively parallel multicomputers. In *USENIX Winter*, pages 449–468, 1993.