Addressing the straggler problem for iterative convergent parallel ML

Aaron Harlap, Henggang Cui, Wei Dai, Jinliang Wei, Gregory R. Ganger, Phillip B. Gibbons, Garth A. Gibson, Eric P. Xing *Carnegie Mellon University*

Abstract

FlexRR provides a scalable, efficient solution to the straggler problem for iterative machine learning (ML). The frequent (e.g., per iteration) barriers used in traditional BSP-based distributed ML implementations cause every transient slowdown of any worker thread to delay all others. FlexRR combines a more flexible synchronization model with dynamic peer-to-peer re-assignment of work among workers to address straggler threads. Experiments with real straggler behavior observed on Amazon EC2 and Microsoft Azure, as well as injected straggler behavior stress tests, confirm the significance of the problem and the effectiveness of FlexRR's solution. Using FlexRR, we consistently observe near-ideal run-times (relative to no performance jitter) across all real and injected straggler behaviors tested.

1. Introduction

Statistical machine learning (ML) is emerging as a powerful building block for modern services, scientific endeavors, and enterprise processes. ML algorithms determine parameter values that make an assumed mathematical model fit a set of input (observation) training data, as closely as possible, such that the model can then be used on other data points. Increasingly, distributed implementations of ML algorithms are used, even when the model fits in a single machine's RAM, because the precision and complexity of many models create large computation time requirements for determining parameter values.

ML algorithms vary, and this paper focuses on a major subset: iterative convergent algorithms solved in an input-data-parallel manner. Such algorithms begin with a guess of the solution and proceed through multiple iterations over the input data to improve the solution. Most distributed implementations of such algorithms follow the Bulk Synchronous Parallel (BSP) computational model. Input data is divided among worker threads, each of which iterates over its subset of the input data and determines solution adjustments based on its local view of the latest parameter values. All workers execute the same iteration at the same time, enforced by barriers, and solution adjustments from one iteration are exchanged among workers before the next iteration begins. When many workers are involved, regular barrier synchronization often induces large slowdowns, due to straggler problems.

A straggler problem arises whenever worker threads experience uncorrelated performance jitter. In each iteration, under BSP, all workers must wait for the slowest worker in that iteration, so one slowed worker causes unproductive wait time for all the others. Unfortunately, even when load is balanced, transient slowdowns are common in real systems (especially in shared clouds) and have many causes, such as resource contention, garbage collection, background OS activities, and (for ML) stopping criteria calculations. Worse, the frequency of such issues rises significantly when executing on multi-tenant computing infrastructures rather than

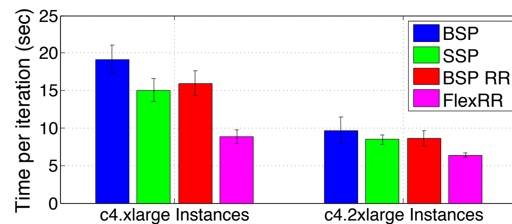


Figure 1: Comparison of MF performance on EC2. The graph shows average time-per-iteration for a collaborative filtering application (*MF*) running on 64 EC2 instances. For each of two EC2 machine classes, four approaches are compared: "BSP" and "FlexRR" represent the traditional approach and our solution, which combines flexible consistency bounds with our temporary work re-assignment technique. The "SSP" and "BSP RR" bars show use of individual ones of these two primary techniques, demonstrating that neither alone addresses the straggler problem for iterative convergent ML. FlexRR outperforms BSP and SSP by 53% and 39% (left) and 35% and 25% (right), for these two EC2 machine classes. Experimental details are in Section 5.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

SoCC '16, October 05-07, 2016, Santa Clara, CA, USA.
© 2016 ACM. ISBN 978-1-4503-4525-5/16/10…$15.00.
DOI: http://dx.doi.org/10.1145/2987550.2987554

dedicated clusters (as is becoming increasingly common) and as the number of workers and machines increases.

Straggler mitigation techniques based on *redundant task execution* [5, 6, 19, 52] have been applied successfully to data processing jobs that fit map-reduce-style BSP execution (e.g., in Hadoop [7] or Spark [51]), relying on the *idempotency* of redundantly executed tasks. But, the most efficient frameworks for distributed ML do not work that way. Instead, these frameworks share state and exploit ML-specific properties to reduce coordination overheads and converge far faster [1, 3, 11, 12, 15, 35, 38]. Because the changes to shared state are *not idempotent*, new approaches to straggler mitigation are needed.

This paper describes FlexRR, a new approach to straggler mitigation without the correctness problems of redundant task execution, for iterative convergent ML on efficient ML frameworks. FlexRR combines flexible consistency bounds with a new temporary work reassignment mechanism we call RapidReassignment. Flexible consistency bounds via SSP[1] remove the barriers of BSP, allowing fast workers to proceed ahead of slowed workers by a bounded amount [13, 31, 35]. The flexibility improves efficiency, while the bound enables convergence to be proven [31, 36]. With RapidReassignment, a slowed worker can offload a portion of its work for an iteration to workers that are currently faster, helping the slowed worker catch up. The two techniques complement each other, and both are necessary to address the straggler problem for iterative convergent ML. Flexible consistency bounds provide FlexRR with enough slack to detect slowed workers and address them with RapidReassignment, before any worker reaches the bound and is blocked.

FlexRR's RapidReassignment is a specialized form of work shedding optimized for large-scale data-parallel iterative ML: (1) It takes advantage of the well-defined notion of progress through an iteration to identify slowed workers quickly. (2) It uses P2P communication among workers to detect slowed workers and perform work re-assignment, bounding communication overhead and avoiding a central decision-making bottleneck. (3) It uses explicit *helper groups*, limiting to which other workers any given work can be offloaded, to minimize data movement and enable input data preloading. (4) Optionally, it exploits iteration knowledge to further reduce how much data needs to be preloaded on helpers. Overall, RapidReassignment's design enables efficient and scalable temporary work reassignment for state-of-the-art efficient ML frameworks.

Extensive experiments demonstrate that FlexRR successfully addresses the straggler problem for three real ML applications and various real and injected straggler behaviors. As illustrated in Figure 1, FlexRR reduces time-per-iteration by 35–53% on average, compared to the traditional BSP approach, and by 25–39% over SSP, even for relatively small ML problems on relatively expensive Amazon EC2 instances (which would be expected to have minimal resource sharing with other tenant activities). Similar results are observed in experiments on Microsoft Azure. In addition, various synthetic stragglers behaviors drawn from prior studies are used for controlled study of a wider range of scenarios; FlexRR consistently nearly matches an "ideal" lower bound in which all work is at all times perfectly balanced at no overhead, resulting in up to 5–10× improvement over BSP and SSP in extreme cases.

This paper makes three primary contributions: (1) It describes the first runtime solution, to our knowledge, to the straggler problem for iterative ML on efficient frameworks. Previous work on stragglers focuses on impractical solutions (e.g., preventing them entirely) or different parallel computations (e.g., sets of idempotent tasks with no iteration or shared state). (2) It describes FlexRR, including its novel RapidReassignment design that specializes work shedding for parameter server systems to avoid significant inefficiencies that would arise with traditional realizations. (3) It demonstrates the efficacy of FlexRR, and its advantages over existing approaches, with three real ML applications. We measure real straggler effects and show FlexRR effectively mitigates them on each of Amazon EC2, Microsoft Azure, and a large dedicated cluster, as well as under various injected straggler patterns. Overall, FlexRR provides a simple, practical, and effective solution to an important and long-unsolved problem in parallel ML.

2. Background and Related Work

This section presents background on iterative convergent ML algorithms and prior work addressing stragglers.

2.1 Iterative ML, BSP, and Stragglers

Iterative convergent algorithms. Many ML tasks (e.g., sparse matrix factorization, multinomial logistic regression, DNN, K-means, Sparse Coding and latent Dirichlet allocation) are solved by searching a space of potential solutions for one with a large (or, for minimization, small) objective value, using an iterative convergent algorithm. Such algorithms start with an initial solution (model parameter values) and proceed through a sequence of iterations, each seeking to produce a new solution with an improved objective value. Typically, each iteration considers each input datum individually and adjusts current model parameters to more accurately reflect it. Eventually, the algorithm reaches a stopping criterion, such as surpassing a target objective value or leveling off of objective value improvements, and outputs a solution. A key property of these algorithms is that they converge to a good solution even if there are minor inconsistencies in parameter updates. This makes them amenable to efficient distributed execution using flexible consistency bounds, with up to 99× speedups reported over fast single-thread implementations [15].

BSP. Most distributed implementations of iterative convergent algorithms follow the Bulk Synchronous Parallel (BSP) computational model with an input-data-parallel approach.

[1] We refer to Stale Synchronous Parallel (SSP) in the paper, but the concept has also been described as Bounded Delay consistency [35].

The input data is divided among worker threads that execute in parallel, performing the work associated with their input data, and executing barrier synchronizations at the end of each iteration. For an ML algorithm, the model parameters are stored in a shared data structure (often distributed among the workers) that all workers update during each iteration. BSP guarantees that all workers see all updates from the previous iteration, but not that they will see updates from the current iteration, enabling workers to use cached copies of model parameters for efficiency. Typically, the assignment of work to workers stays the same from one iteration to the next, to avoid the overheads of input data movement.

Stragglers: A primary performance issue for BSP is *stragglers*, because in each iteration, all workers must wait for the slowest worker in that iteration. The straggler problem grows with the level of parallelism, as random variations in execution times increase the probability that at least one worker will run unusually slowly in a given iteration. Even when it is a different straggler in each iteration, due to uncorrelated transient effects, the entire application can be slowed significantly.

Stragglers can occur for a number of reasons [5, 6], including hardware heterogeneity [32, 45, 49], hardware failures [6], unbalanced data distribution among tasks, garbage collection in high-level languages, and various OS effects [8, 42]. Resource contention is another common cause, especially in shared cloud infrastructures. Additionally, for ML algorithms, expensive stopping criteria computations can lead to straggler effects, when performed on a different worker machine every so many iterations.

2.2 Related Work Addressing Stragglers

Stragglers have long plagued parallel computing, and many techniques have been developed to mitigate them.

Eliminating performance variation. The HPC community—which frequently runs applications using the BSP model—puts significant effort into identifying and removing sources of performance jitter from the hardware and OSs of their supercomputers [23, 42]. Naiad [40] used the same approach. While this approach can be effective at reducing performance "jitter" in specialized and dedicated machines, it does not solve the more general straggler problem. For instance, it is not applicable to programs written in garbage-collected languages, does not handle algorithms that inherently cause stragglers during some iterations, and does not work for today's multi-tenant computing infrastructures [17, 22, 30].

Blacklisting is a limited form of performance variation elimination, which attempts to mitigate stragglers by ceasing to assign work to workers that are falling behind. However, this approach is fragile. Stragglers caused by temporary slowdowns (e.g., due to resource contention with a background activity) often occur on non-blacklisted machines [18]. Worse, good workers that have such a temporary slowdown may then be blacklisted, unnecessarily reducing the computing power available.

Speculative execution and task cloning. Speculative execution is used to mitigate stragglers in data processing systems like MapReduce, Hadoop, and Spark [5–7, 19, 52]. Jobs in these systems consist of stateless, idempotent tasks like "map" and "reduce", and speculative execution runs slow tasks redundantly on multiple machines. While this consumes extra resources, it can significantly reduce job completion delays caused by stragglers, because the output from the first instance of any given task can be used without waiting for slower ones.

State-of-the-art frameworks for high-performance parallel ML use a *parameter server* architecture [1, 3, 11, 12, 15, 16, 29, 35], which does not accommodate computation redundancy. While iterative ML can be built as a series of collections of idempotent tasks, doing so precludes many effective techniques for reducing overhead and speeding convergence [14, 15, 35]. For example, the highly-tuned Spark-based GraphX [27] was shown to approximately match PowerGraph [26], which has been shown [15] to be 2–10X slower than two recent parameter server systems (IterStore [15] and LazyTable [14]) for collaborative filtering via sparse matrix factorization. In parameter server systems, worker processing involves shared state and is not idempotent. Applying the same adjustments more than once can affect convergence negatively or even break algorithm invariants. FlexRR uses peer-to-peer interactions among workers to offload work when necessary, avoiding the wasted resources and potentially incorrect behavior of redundant work.

Work stealing, work shedding. Work stealing and work shedding are mirror approaches for adaptively rebalancing work queues among workers [2, 10, 20, 21]. The concept is to move work from a busy worker to an idle worker. FlexRR's temporary work reassignment mechanism is a form of work shedding, specialized to the nature of data-parallel iterative ML. There are several key differences. First, FlexRR takes advantage of the well-defined notion of progress through an iteration to identify slowed workers early on and avoid delays; work stealing, in contrast, waits for a worker to idle before looking to steal work, incurring additional delays until work is found. Second, while work stealing is computation-centric (e.g., data is moved to the thread that steals the work), FlexRR carefully avoids data movement by limiting and pre-determining reassignment patterns to avoid expensive on-demand loading of input data and parameter state. Third, because of its focus on *transient* stragglers, FlexRR's reassignments are temporary—only for the remainder of an iteration. Finally, it is designed explicitly to work in conjunction with flexible consistency bounds, as discussed below.

Using less strict progress synchronization. The strict barriers of BSP can be replaced with looser coordination models. One approach is to reduce the need for synchronization by restricting communication patterns. For example, GraphLab [38, 39] programs structure computation as

a graph, where data can exist on nodes and edges. All communication occurs along the edges of this graph, so non-neighboring nodes need not synchronize. However, GraphLab requires the application programmer to know and specify the communication pattern.

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.

Yahoo! LDA [3] and Project Adam [12], as well as most solutions based around NoSQL databases, allow workers to run asynchronously, relying on a best-effort model for updating shared data. While such approaches can work well in some cases, they provide no guarantees of convergence for ML algorithms and indeed can readily diverge.

FlexRR uses flexible consistency bounds, which has been recently proposed and studied under two different names: "Stale Synchronous Parallel" (SSP) [13, 31] and "Bounded Delay consistency" [35]. These schemes generalize BSP by allowing any worker to be up to a bounded number of iterations ahead of the slowest worker. So, for BSP, the bound (which we will refer to as the *slack-bound*) would be zero. With a slack-bound of b, a worker at iteration t is guaranteed to see all updates from iterations 1 to $t-b-1$, and it may see (not guaranteed) the updates from iterations $t-b$ to t. Such a bound admits proofs of convergence [31, 35, 36]. Consistent with our results, SSP has been shown to mitigate small transient straggler effects [14, 35] but not larger effects. FlexRR combines SSP with temporary work reassignment to address the straggler problem for iterative ML.

3. FlexRR Design & Implementation

FlexRR provides parallel execution control and shared state management for input-data-parallel iterative convergent ML algorithms. This section overviews FlexRR's API, basic execution architecture, shared state management approach, and solution to the straggler problem.

FlexRR's design relies on a few assumptions about application behavior. It assumes data-parallel processing, with worker threads processing assigned input data items independently and without order-dependence. It assumes iterations (or mini-batches, when used) are not too short to detect and react to slowed workers and that a worker's progress through an iteration can be measured, such as by the fraction of its input data items processed, and that the overall training time is greater than the loading time. Also, RapidReassignment's performance relies on being able to reassign work quickly; so, it assumes that there is either no cross-iteration data-item-specific local state or that there is a way to avoid needing to transfer it with reassigned work.[2] These characteristics are common to most data-parallel iterative ML applications, including our benchmark apps as well as deep neural networks, K-means clustering, Sparse Coding and many others.

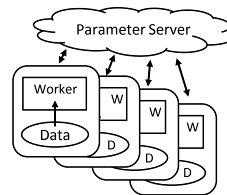


Figure 2: FlexRR architecture. This high-level picture illustrates FlexRR's logical structure, for the case of one core (so, one worker thread) per node. The worker is assigned some input data to process in each iteration, making adjustments to the model parameters stored in a parameter server, which is itself sharded among the same nodes as the worker threads.

FlexRR (Figure 2) is implemented as a C++ library linked by an ML application using it. During execution, FlexRR consists of one process executing on each node being used. Each FlexRR process starts a worker thread for each core on the node and a number of background threads for its internal functionality. The worker threads execute the ML application code for adjusting model parameters based on input data and possibly local state. The shared model parameters, which may be read and adjusted by all worker threads, are stored in a so-called "parameter server" maintained by the set of FlexRR processes.

3.1 Workers and Execution Management

During initialization, an ML application provides FlexRR with the list of nodes/cores to be used, the input data file path, several functions called by FlexRR, and a stopping criterion. The input file contains data items in an understood format (e.g., rows that each contain one input data item in an easy-to-process format). The stopping criterion may be a number of iterations, an amount of time, or a determination of convergence. The most important function provided (`process-input`) is for processing a single input data item, taking the data item value as input and processing it to determine and apply model parameter adjustments as needed.

Each worker thread is assigned a unique ID, from zero to $N-1$, and a disjoint subset of the input data items. The default assignment is a contiguous range of the input data, determined based on the worker ID, number of workers, and number of data items. Each worker has an outer loop for iterating until the stopping criterion is reached and an inner loop for each iteration[3] that calls `process-input` on each of its data items.

[2] An example of the latter is our *LDA* application, which originally relied on local state. Instead of transferring local state between workers, which we found to be too inefficient, we designed an *LDA*-specific mechanism to avoid dependence on the local state.

[3] We have simplified the description a bit. For greater flexibility, FlexRR actually provides a notion of a *clock of work* that gets executed on each inner loop, which may range from some number of data items (a "mini-batch" of an iteration) to some number of complete iterations.

3.2 Parameter Server for Shared State

FlexRR uses the increasingly popular parameter server approach [3, 31, 35, 43] to storing and managing the shared state (i.e., the model parameters being computed) among worker threads. A parameter server provides a simple read-update interface to the shared state, greatly simplifying the application ML code by efficiently and scalably handling communication and consistency.

The FlexRR parameter server is derived from LazyTable [14, 15]. It exposes a simple key-value interface to the ML application code, which uses `read-param` and `update-param` functions to read or update (apply a delta to) a model parameter value specified by the key. The value type is application-defined, but must be serializable and have a commutative and associative aggregation function, such as plus, union, or multiply. With this property different worker threads can apply changes in any order without affecting the result. For the ML applications used in this paper, the values are vectors and the aggregation function is addition.

The parameter server implementation reduces cross-node traffic by including a client-side cache for model parameter entries. While logically separate, the parameter server is part of the same FlexRR processes as the worker threads. Every process has a number of parameter server threads and maintains a shard of the shared state. Each iteration updates are write-back cached, and asynchronously sent to the appropriate parameter server shards.

FlexRR supports both the BSP and SSP models discussed in Section 2. Each cached value is associated with an iteration number that indicates the latest iteration for which all workers' updates have been applied to it. During a read, the cached value is returned only if it reflects all updates up to the slack-bound (zero, for BSP); that is, the value is up-to-date enough if it reflects all updates from iterations more than "slack-bound" before the worker's current local iteration number. Otherwise, the read must proceed to the appropriate server shard to retrieve the value, possibly waiting there for other workers' updates. For fault tolerance, FlexRR supports the same checkpointing mechanism as LazyTable [14]–its RapidReassignment mechanism does not affect it.

3.3 Straggler Mitigation

FlexRR combines two mechanisms, flexible consistency bounds via the SSP model and temporary work reassignments via our *RapidReassignment*, to address the straggler problem for iterative ML. The SSP model allows each worker thread to be ahead of the slowest worker by up to a specified slack-bound number of iterations. This flexibility mitigates stragglers to some extent [14] (see also the SSP bars in Figure 1 and Section 5), but more importantly provides enough flexibility for RapidReassignment to be highly effective. RapidReassignment uses peer-to-peer communication to enable workers to self-identify as stragglers and temporarily offload work to workers that are ahead. While FlexRR's support for SSP is similar to recent systems [14, 35], its scalable, data-centric work reassignment mechanism is new.

4. RapidReassignment Design

The goal of RapidReassignment is to detect and temporarily shift work from stragglers before they fall too far behind, so that workers never have to wait for one another. Workers exchange progress reports, in a peer-to-peer fashion, allowing workers to compare their progress to that of others. If a worker finds that it is falling behind, it can send a portion of its work to its potential helpers (a subset of other workers), which can confirm that they are indeed progressing faster and provide assistance (see Figure 3). Combined with SSP, RapidReassignment is highly effective in mitigating straggler delays of all intensities.

4.1 Worker Groups

RapidReassignment is designed for scalability, using peer-to-peer coordination among workers instead of a central arbiter. Like overlay networks [46, 48], workers exchange progress reports and offloaded work with only a few other workers, avoiding the scalability problems that would arise from all-to-all progress tracking or a centralized work reassignment approach, especially for short iterations when progress tracking and reassignment are more frequent.[4] During initialization, each worker is assigned a group of workers that are eligible to provide assistance, referred to as its *helper group*, and a group of workers to whom the worker is eligible to provide assistance, referred to as its *helpee group*. The size of each group is set at start up and can be configured by the ML application. Each worker is assigned one helper on the same machine, and its other helpers are spread across different machines. While helper and helpee groups may overlap, they are usually not identical. For example, in a system containing 64 workers assigned round-robin to 8 machines and 4 helpers assigned to every worker, worker 14 might be eligible to assist workers (8,9,15,22) while workers (6,11,12,13) would be designated as its helpers. A waterfall effect results, whereby a worker providing a lot of assistance to some workers can in turn offload its own work to others, and so on, such that all workers make similar progress.

Worker groups also improve work reassignment efficiency. A helper needs to access the input data associated with the reassigned work. While it could fetch that data on demand, the helper's help is much more efficient if the data is fetched in advance. Indeed, based on our experiments, work reassignment is too slow to be helpful without doing so. Toward that end, each worker under RapidReassignment prefetches a copy of the input data of its helpee group members after

[4] For small scale systems and longer iterations, our design could be readily adapted to use a central master to handle inter-worker coordination and reassignment. However, this approach would not outperform our P2P design in general. Moreover, the computation and communication of the master would compete with doing real work on that server, adding an additional straggler effect.

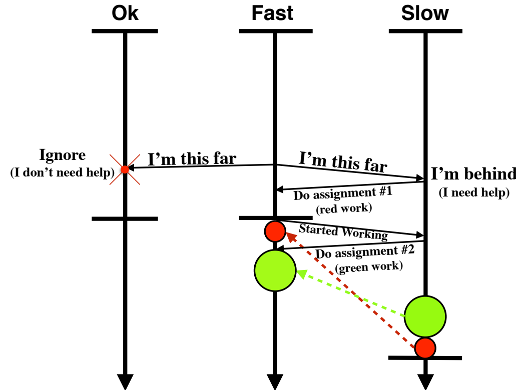


Figure 3: RapidReassignment example. The middle worker sends progress reports to the other two workers (its helpee group). The worker on the left is running at a similar speed, so it ignores the message. The worker on the right is running slower, so it sends a `do-this` message to re-assign an initial work assignment. Once the faster worker finishes its own work and begins helping, it sends a `begun-helping` message to the slow worker. Upon receiving this, the slow worker sends a `do-this` with a follow-up work assignment to the fast worker.

loading its own data. The limited set of helpees bounds the cache space needed, which can be further reduced by caching only the tail of each helpee's iteration. Our experiments show that FlexRR suffers minimal performance loss from workers caching only the tail fraction of their helpees' input data (see Section 5.6).

4.2 Worker Communication

RapidReassignment uses non-blocking Message Passing Interface (MPI) for communication among workers. Workers explicitly poll for messages during each iteration, in order to compare their respective progress. The *message check frequency* parameter specifies how many times during each iteration a worker checks for incoming messages. The default setting is 100 checks per iteration, which our sensitivity experiments (Section 5.7) show is a good value.

To determine speed differences between workers, each worker keeps a *runtime timer* to track how long it has been running. These timers are launched during initialization, following a joint barrier. Because RapidReassignment addresses relatively large differences in progress (e.g., 20% of a multi-second iteration), with smaller differences mitigated by flexible consistency bounds, these timers are sufficiently precise and need to be resynchronized only infrequently (e.g., hourly).

4.3 RapidReassignment Actions

This section describes the five primary RapidReassignment actions in FlexRR, designed to quickly offload work from slow workers to faster helpers (Figure 3). As will be seen, RapidReassignment is carefully tuned to the properties of data-parallel iterative convergent algorithms, such as the freedom to delay and reorder updates on the one hand, yet the need to avoid duplicating work on the other (recall that duplicating work can lead to incorrect behavior).

Identifying Stragglers. Upon reaching the *progress checkpoint* in the current iteration, which by default is set to 75% completion, a worker sends out a `progress-report` message to its helpee group, containing its current iteration number and the local time in its runtime timer. During each message check, a worker checks for `progress-report` messages from its helpers. Upon receiving such a message, the worker calculates its progress compared to the progress of the eligible helper. The logic for calculating the progress difference is shown in Algorithm 1. The result informs the worker how far ahead or behind (as a percentage of the iteration) it is relative to the `progress-report` sender.

Algorithm 1 Progress Difference Calculation

1: *completion_diff* ← progress in the message *minus* progress of the current worker
2: *current_avg* ← weighted average time it takes the current worker to complete an iteration
3: *time_diff* ← timer value of the current worker *minus* timer value contained in progress message
4: $progress_difference \leftarrow completion_diff + \frac{time_diff}{current_avg}$

Reassigning Work. If a worker finds that it has fallen behind the sender of a `progress-report` by more than a set threshold (the *straggler trigger threshold*, with a default of 20%), it will send an initial work assignment in a `do-this` message back to the sender. This initial work assignment is a percentage of its work for the current iteration. Section 5.7 shows that a default of 2.5% is a good setting for this tunable. The `do-this` message contains the current iteration number of the slow worker, beginning and end of the work assignment (a range of the input data), and a local timestamp of the message. A sample message is `do-this (iteration: 4, start: 140, end: 160, timer: 134.43)`. Note that, as shown in Figure 3, the slow worker reassigns ranges of work starting from the end of its current iteration. Assigning from the end is the least disruptive to the slow worker's progression through its input data items, and takes advantage of the robustness of iterative convergent algorithms to processing data items in any order despite the data dependencies (such dependencies between item processing—resulting from their `update-param` calls—would make this reordering unsafe for general code).

Helping with Work. Workers check for `do-this` messages on every message check. Upon receiving a `do-this` message, the worker (the potential helper, in this case) will compare the timestamp of the message to the timestamp of the latest `cancel-help` message (see below) from the same worker. If the timestamp in the `do-this` message is greater, the potential helper will compare its current iteration to the iteration number contained in the `do-this` message. If the iteration number contained in the message is smaller than the helper's current iteration, the helper will immediately

Algorithm 2 Helping Decision

1: *msg* ← check for helping requests
2: **if** *msg.timestamp* ≤ *last_cancellation_timestamp* OR *msg.iteration* > *current_iteration* **then**
3: Discard *msg*
4: **else if** *msg.iteration* < *current_iteration* OR finished its own work this iteration **then**
5: Send `begun-helping` and do the help
6: **else**
7: Save *msg* for the end of this iteration
8: **end if**

send a `begun-helping` message and begin working on the work assignment. Upon completing the work assignment, the worker will send a `help-completed` message to the original worker and check for additional `do-this` messages prior to returning to its own work.

If the iteration number in the `do-this` message equals the helper's current iteration (as in assignment #1 in Figure 3), then the helper will put aside the work assignment until the end of the current iteration. If at the end of the iteration the worker has yet to receive a `cancel-help` message containing a timestamp greater than the timestamp of the `do-this` message, the helper will send out a `begun-helping` message and begin working on the work assignment. Upon completing the work assignment, the helper will send a `help-completed` message to the original worker (the helpee) and, after checking for additional valid `do-this` messages, will move on to its own next iteration. Algorithm 2 shows the pseudo-code for the worker's decision about if and when to provide assistance.

Assigning Additional Work. After much experimentation, we have found that a good strategy for balancing various concerns is to first assign a relatively small amount of initial work, immediately followed by a larger amount (double) of additional work once a helper begins processing the initial assignment. To that end, after a worker sends out a `do-this` message, it will check for `begun-helping` messages during every message check in that iteration. If such a message is received, and more help is needed, the worker will send an additional `do-this` message to the faster worker, containing a follow-up work assignment of twice the size (see Figure 3). For the rest of the iteration, the worker will send another follow-up work assignment each time it receives a `begun-helping` message.

Cancelling Work Reassignments. After reassigning a portion of its work, a worker will continue working on its current iteration until it completes all the work it has not given away. At this point, for all pending `do-this` messages the worker has sent out, the worker will check for `begun-helping` messages. If there is a work assignment for which a `begun-helping` message has yet to be received, the worker will send out a `cancel-help` message containing the current timestamp and complete the work on its own. Upon completing all such messages, the worker will wait to receive a `help-completed` message for all work assignments before moving on to the next iteration. This is is done to guarantee the *slack-bound*. There is a small window in which both the helpee and a helper may begin the same work, which can be addressed by having the helpee only commit changes corresponding to reassigned work after it confirms that the helper acknowledged the `cancel-help` message. Again, we are relying here on the robustness of data-parallel iterative convergent algorithms to delayed and out-of-order updates.

5. Evaluation

This section evaluates the effectiveness of FlexRR. Results are reported for sets of Amazon EC2 and Microsoft Azure instances as well as for local clusters. The results support a number of important findings: (1) significant straggler problems occur in real cloud infrastructures, (2) when straggler problems occur, FlexRR greatly outperforms BSP and SSP, achieving near-ideal performance for all of the various straggler patterns studied; (3) to achieve ideal performance, the RapidReassignment and SSP techniques need to be combined, as is done by FlexRR, as neither alone is sufficient; (4) FlexRR is not sensitive to the choices of run-time configuration parameters, within a wide range of reasonable settings.

5.1 Experimental Setup

Experimental Platforms. We use a variety of clusters for our experiments. **Cluster-A** is 16 virtual machines running on a dedicated cluster of 16 physical machines, each with a 2 quad-core Intel Xeon E5430 processor running at 2.66GHz, connected via 1 Gbps Ethernet (≈700 Mbps observed). Each VM runs on one physical machine, and is configured with 8 vCPUs and 15 GB memory, running Debian Linux 7.0. **Cluster-B** is a cluster of 64 Amazon EC2 c4.2xlarge instances. Each instance has 8 vCPUs and 15 GB memory, running 64-bit Ubuntu Server 14.04 LTS (HVM). **Cluster-C** is a cluster of 64 Amazon EC2 c4.xlarge instances, a lower class version of Cluster-B. Each instance has 4 vCPUs and 7.5 GB memory, running 64-bit Ubuntu Server 14.04 LTS (HVM). From our testing using `iperf`, we observe a bandwidth of 1 Gbps between each pair of EC2 instances. **Cluster-D** is a cluster of 64 Microsoft Azure A4 Standard instances. Each instance has 8 vCPUs and 15 GB memory, running 64-bit Ubuntu Server 14.04 LTS (HVM) on Intel Xeon E5507 processors. **Cluster-E** is a cluster of 64 Microsoft Azure A3 Standard instances. Each instance has 4 vCPUs and 7 GB memory, running 64-bit Ubuntu Server 14.04 LTS (HVM) on AMD Opteron 4171 HE processors. From our testing using `iperf`, we observed a bandwidth of 1.1 Gbps between each pair of Azure instances. **Cluster-F** is a PRObE Nome [25] dedicated cluster of 128 high-end computers running Ubuntu 14.04. Each machine contains 4 quad-core AMD Opteron 8354 CPUs (16 physical cores per machine) and 32GB of RAM. The machines are connected via 1Gb Ethernet.

For experiments that control and/or instrument straggler causes, we primarily use Cluster-A. The other five clusters are used to experiment with naturally occurring stragglers in two public clouds and for a larger-scale example. We use our limited access to Cluster-F to experiment on a large problem and dataset that does not fit on the other clusters.

Naturally-occurring and Injected Straggler Patterns. Our goal is to experiment with a wide variety of straggler patterns that are likely to be encountered in practice, as well as with more extreme patterns that provide stress tests. Our experiments with EC2, Azure, and Nome provide evaluation in the presence of a variety of naturally-occurring stragglers in several real infrastructures. But, we are unable to instrument these systems to evaluate the causes or particular nature of those stragglers; we consistently observe straggler problems, but whatever happens happens when using public clouds. To evaluate a broader range of straggler effects and intensities beyond what arose during the particular times of the experiments, and to directly measure attributable delays, we also perform more controlled experiments with injected transient stragglers, using three distinct methodologies:

Slow Worker Pattern: Models transient worker slowdown by inserting *sleep* commands into worker threads. At each of 10 possible delay points within an iteration, each worker decides (independently) to be slowed, with 1% probability, for a period uniformly randomly chosen between 0–2× the duration of an iteration. Naturally, multiple (or no) workers may be slowed at any given time. We denote the *transient delay intensity %* (*delay %* for short) to be the percentage by which a worker is slowed (e.g., 100% delay means runs twice as slow). To simulate the effect of a worker being slow, we divide each iteration into 1000 parts and insert milliseconds-long *sleep* commands at each of these 1000 points. For a delay % of d within a t second iteration, each of these sleeps are $d \times t$ milliseconds long. For example, for a 50% delay within a 6 second iteration, we insert a 3ms sleep at each point.

Disrupted Machine Pattern: Models transient resource contention (e.g., due to sub-machine allocation or a background OS process) by running a disruptor process that takes away CPU resources. Every 20 seconds, each machine independently starts up a disruptor process with 20% probability. The *disruptor* launches a number of threads that each executes a tight computational loop for 20 seconds. For a transient delay intensity % of d on a p-core machine running p application threads, the disruptor launches $d \times p$ processes of its own. For example, on our 8-core machines running 8 worker threads, a 200% delay means having the disruptor launch 16 threads. Such a delay experienced by a worker for a whole iteration will cause it to run roughly 200% slower than without delays.

Power-Law Pattern: Based on a real-world straggler pattern [42], this uses a power-law distribution [44] to model the time t for a worker to complete an iteration: $p(t) \propto t^{\alpha}$, where α is the parameter that controls the "skewness" of the distribution. `sleep` commands are used, as in the *Slow Worker Pattern*, to extend an iteration as determined. Smaller α makes the distribution more "flat" and leads to more delay on average. Each experiment uses a fixed α, and the iteration time of each worker is chosen independently from this distribution. When we set the α parameter to 11, the iteration times in our emulated environment without FlexRR have the same distribution as was measured on real clusters in [42].

We also study a *persistent* straggler pattern where half the machines get 75% of the work per iteration—such uneven workloads could arise in cases where data processing skew is correlated with data placement.

Systems Compared. We compare the speed and convergence rates of four modes implemented in FlexRR: [5]

BSP	Classic BSP execution
SSP	SSP execution
BSP RR	BSP with our RapidReassignment
FlexRR	Our solution
Ideal	Best possible (computed lower bound)

We also compute a value termed "Ideal", which represents the speed that should be achieved if all work is at all times perfectly balanced with no overhead. Reporting results for BSP RR and SSP (without RapidReassignment) enables us to study the impact of RapidReassignment and flexible consistency bounds in isolation versus in combination, as in FlexRR. For SSP, we use a slack-bound of 1 in all experiments after verifying that it leads to the fastest convergence on these benchmarks.

FlexRR features several run-time configuration parameters such as *helper group size* and *work assignment sizes*. Table 1 lists these parameters, the range of values studied, and their default values. The default values are the best settings obtained after extensive experimentation over the range of parameters shown. Section 5.7 provides a sensitivity analysis on the parameter settings, showing that FlexRR performs well over a broad range of settings.

Table 1: FlexRR Parameter Settings

Parameter	**Range**	**Default**
Helper group size	2–16	4
Initial work assignment	1.25%–15%	2.5%
Follow-up work assignment	2.5%–30%	5%
Message checks/iteration	20–50k	100
Straggler trigger threshold	10%–40%	20%

[5] Although we do not show results of comparisons to other systems, the base system in which we integrated FlexRR compares favorably to state-of-the-art frameworks, as noted in Section 2.2. For example, its BSP mode is faster than GraphLab [26, 38] by 10–14× for MF and 50–100% for LDA [15], which in turn has been shown to outperform Hadoop and Spark implementations [27, 38]. It also outperforms efficient single-threaded implementations of MF and LDA by 99× and 62×, respectively, when using 8 64-core machines [15].

Experimental Methodology. Every experiment was run at least thrice, and we report arithmetic means. In experiments that had injected stragglers, the first run was conducted from smallest delay injections to largest, the second in reverse order, and the third in random order.

5.2 Application Benchmarks

We use three popular iterative ML applications.

Matrix Factorization (MF) is a technique commonly used in recommendation systems, such as recommending movies to users on Netflix (a.k.a. collaborative filtering). The key idea is to discover latent interactions between the two entities (e.g., users and movies) via matrix factorization. Given a partially filled matrix X (e.g., a rating matrix where entry (i,j) is user i's rating of movie j), matrix factorization factorizes X into factor matrices L and R such that their product approximates X (i.e., $X \approx LR$). Like many other systems [14, 15, 24, 34], we implement *MF* using the stochastic gradient descent (SGD) algorithm. Each worker is assigned a subset of the observed entries in X; in every iteration, each worker processes every element of its assigned subset and updates the corresponding row of L and column of R based on the gradient. L and R are stored in the parameter server.

Our *MF* experiments use the *Netflix* dataset, which is a 480k-by-18k sparse matrix with 100m known elements. They are configured to factor it into the product of two matrices with rank 500 for Cluster-A and rank 1000 for Cluster-B, Cluster-C, Cluster-D and Cluster-E. We also conduct an experiment on Cluster-F using a synthetically enlarged version of the *Netflix* dataset that is 256 times the original. It's a 7683k-by-284k sparse matrix with 4.24 billion known elements with rank 100.

Multinomial Logistic Regression (MLR) is a popular model for multi-way classification, such as used in the last layer of deep learning models for image classification [33] or text classification [37]. In *MLR*, the likelihood that each (d-dimensional) observation $x \in \mathbb{R}^d$ belongs to each of the K classes is modeled by *softmax* transformation $p(\text{class}{=}k|x) = \frac{\exp(w_k^T x)}{\sum_j \exp(w_j^T x)}$, where $\{w_j\}_{j=1}^K$ is the linear (d-dimensional) weights associated with each class and are considered the model parameters. The weight vectors are stored in the parameter server, and we train the *MLR* model using SGD where each gradient updates the full model [9].

Our *MLR* experiments use the *ImageNet* dataset [47] with LLC features [50], containing 64k observations with a feature dimension of 21,504 and 1000 classes.

Latent Dirichlet Allocation (LDA) is an unsupervised method for discovering hidden semantic structures (*topics*) in an unstructured collection of *documents*, each consisting of a bag (multi-set) of *words*. *LDA* discovers the topics via word co-occurrence. For example, "Obama" is more likely to co-occur with "Congress" than "super-nova", and thus "Obama" and "Congress" are categorized to the same topic associated with political terms, and "super-nova" to another

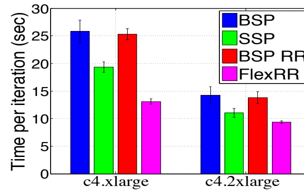


Figure 4: EC2, LDA, no injected delay

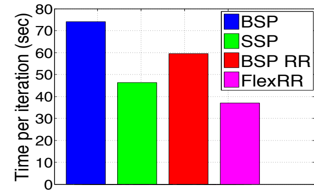


Figure 5: PRObE Nome, large MF, no injected delay

topic associated with scientific terms. Further, a document with many instances of "Obama" would be assigned a topic distribution that peaks for the politics topics. *LDA* learns the hidden topics and the documents' associations with those topics jointly. It is often used for news categorization, visual pattern discovery in images, ancestral grouping from genetics data, and community detection in social networks.

Our *LDA* solver implements collapsed Gibbs sampling [28]. In every iteration, each worker goes through its assigned documents and makes adjustments to the topic assignment of the documents and the words. The *LDA* experiments use the *Nytimes* dataset [41], containing 100m words in 300k documents with a vocabulary size of 100k. They are configured to classify words and documents into 500 topics for Cluster-A and 1000 topics for Cluster-B, Cluster-C, Cluster-D and Cluster-E.

5.3 Naturally-occurring Straggler Results

We performed experiments on Amazon EC2, Microsoft Azure and PRObE Nome to evaluate FlexRR in the presence of naturally-occurring straggler effects observed during the particular times of the experiments. No synthetic straggler effects are injected during these experiments.

Amazon EC2 results. Figure 1 (on page 1) and Figure 4 show the results for *MF* and *LDA*, respectively, Cluster-B (c4.2xlarge VMs) and Cluster-C (c4.xlarge VMs). Using c4.2xlarge VMs, FlexRR reduces time-per-iteration by 35% (25%) for *MF* and by 34% (15%) for *LDA* relative to BSP (SSP, respectively). Using c4.xlarge VMs, the reductions are 53% (39%) for *MF* and 49% (32%) for *LDA*. The improvements are larger for c4.xlarge VMs, because these less expensive VMs experience more transient straggler effects.

The improvements on EC2 come despite executing relatively short experiments on relatively expensive EC2 instances that would be expected to have minimal resource sharing with other tenant activities, highlighting the real-ness of transient stragglers in cloud infrastructures.

Microsoft Azure results. Figure 6 shows the results of *MF* on Cluster-D (A4 VMs) and Cluster-E (A3 VMs) on Microsoft Azure. Using the A4 VMs, FlexRR reduces time-per-iteration by 43% (32%) relative to BSP (SSP, respectively). Using the A3 VMs, FlexRR reduces time-per-iteration by 56% (38%) relative to BSP (SSP). While the A4 instances are bigger and more expensive VMs, the times-per-iteration are larger than on the A3 instances because the A3 CPUs

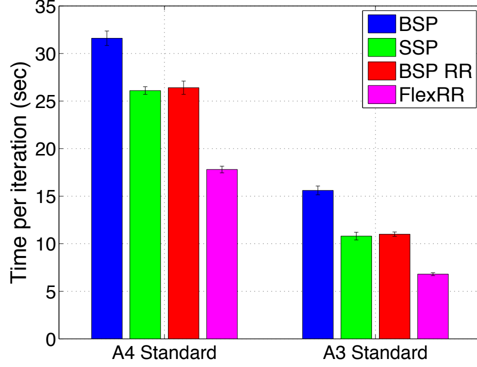


Figure 6: Microsoft Azure, MF, no injected delay

perform better on MF's floating point computations. Nonetheless, significant straggler effects are observed for both setups, and the results are akin to those for the c4.xlarge VMs on EC2.

PRObE Nome large MF experiment. To verify FlexRR's effectiveness for a larger workload, we used the *Netflix * 256* synthetically enlarged dataset on Cluster-F. Figure 5 shows that FlexRR reduces time per iteration by 21% over SSP and 51% over BSP, even on this dedicated cluster with no injected delays. As expected, more straggler effects are observed as cluster size increases, and FlexRR effectiveness is not hampered by the increased problem size. Cluster-F displayed less of a straggler problem due to the dedicated nature of the cluster and a higher quality network then the AWS EC2 Clusters.

5.4 Slow Worker Pattern Results

This section studies the speed and convergence rate of the ML applications under the *Slow Worker Pattern*.

5.4.1 Speed Tests

For each application, we measured the time-per-iteration, $\frac{Overall\ run\ time}{Number\ of\ iterations}$, for the four modes, varying the transient delay intensity %. Each experiment is run for 20 iterations. (Running more iterations yields the same results.)

Results on Cluster-A. Figure 7a and Figure 7b show the results for the *MF* and *LDA* applications running on Cluster-A. *MLR* results looks similar to *MF* (not shown due to space constraints). BSP slows down linearly with delay intensity. By controlling straggler intensity, we see that SSP can mitigate delays below its slack-bound (e.g., see the 50% delay intensity points), but then too suffers linearly. (The natural stragglers from Section 5.3 were clearly too big for SSP alone.) FlexRR, on the other hand, nearly matches Ideal even up to 400% delays, which are more extreme that should be expected in practice. We measured the percentage of work that gets reassigned by FlexRR: it ranges from 8–9% of the work at 0% delay (i.e., no injected delays) to 19–22% at 400% delay. Even at 0% delay, FlexRR runs 18% faster than BSP on *MF* and *LDA* and 13% faster than BSP on *MLR*. The figures also show that our RapidReassignment technique can be used in BSP to decrease its straggler penalty, but it is FlexRR's combination of flexible consistency bounds and RapidReassignment that nearly matches Ideal.

At high delay % values, there is some divergence from Ideal for *LDA*. That is because *LDA* uses a special mechanism to handle its local state, which involves two extra model parameter updates for each work re-assignment. At higher delays, more work is re-assigned, thus these extra updates begin to have an effect on the run-time, causing FlexRR to deviate from Ideal.

Results on Cluster-B. Figure 7c shows the results for *MF* on the larger Amazon EC2 cluster, which are qualitatively the same as on Cluster-A. As on Cluster-A, BSP slows down linearly with delay intensity, SSP can mitigate stragglers only up to its slack-bound, and FlexRR nearly matches Ideal. FlexRR reassigns 21% of the work at 0% delay and 31% at 400% delay. The main difference between the Cluster-B and Cluster-A results is that on Cluster-B there is an even larger separation between FlexRR and the next best approach (BSP with our RapidReassignment). E.g., at 400% delay, BSP RR is 10 times slower than FlexRR. At 0% delay (no injected delays, corresponding to Figure 1 (right)), FlexRR is 35% faster than BSP and 25% faster than SSP, because of non-injected performance jitter. The results for *LDA* on Cluster-B are qualitatively as on Cluster-A (not shown due to space constraints).

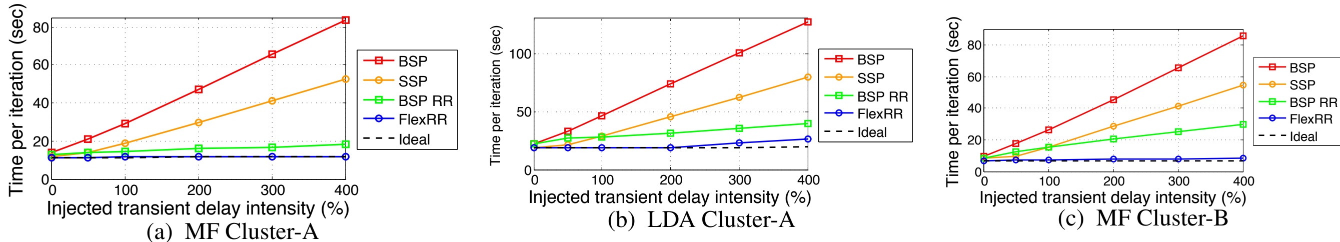
5.4.2 Convergence Tests

We also measure the time to convergence for the ML applications running in each of the modes. We calculate Ideal by multiplying the Ideal time-per-iteration values from Section 5.4.1 by the number of iterations needed to reach convergence by the BSP experiment.[6]

Criteria for Convergence. We use the following stopping criterion, based on guidance from our ML experts: If the objective value (for *MF*) or log-likelihood (for *LDA*) of the solution changes less than 2% over the course of 10 iterations, then convergence is considered to have been reached. We also verified that they reached the same objective value. Because the objective value calculation is relatively expensive, and we wanted to observe it frequently, we did it offline on FlexRR checkpoints.

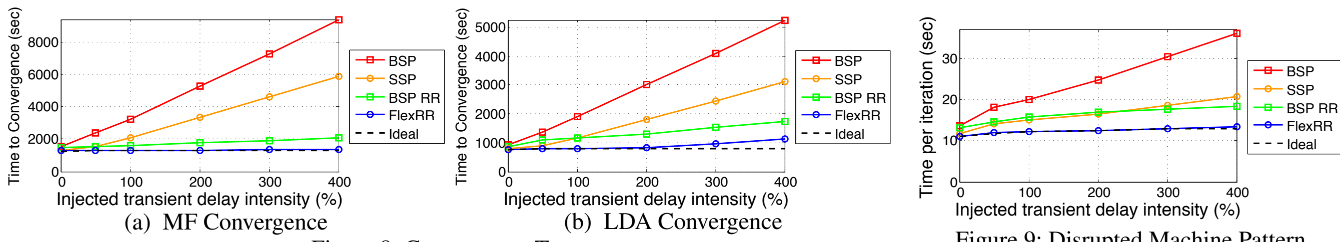
Convergence Test Results. Figure 8a shows the results for *MF*. For all delay %, BSP (and BSP RR) required 112 iterations to reach convergence and SSP required 113 iterations. FlexRR required 114 iterations, with the exception of 400% delay, where it took 115 iterations. Even with the extra iterations required to reach convergence, FlexRR converged 10% faster than BSP at 0% delay injected. With delays injected, BSP suffered from linear increase in convergence time, while FlexRR effectively matched the Ideal convergence time even

[6] We use BSP iterations in this lower bound because the flexible consistency bounds of FlexRR and SSP can lead to a (modest) increase in the number of iterations needed [13, 31], e.g., 2-3 extra iterations in our experiments.



(a) MF Cluster-A (b) LDA Cluster-A (c) MF Cluster-B

Figure 7: Slow Worker Pattern Speed Tests



(a) MF Convergence (b) LDA Convergence

Figure 8: Convergence Tests

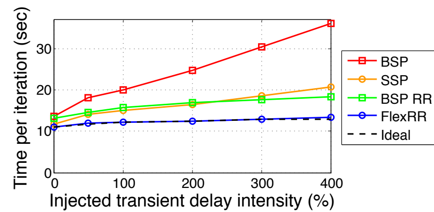


Figure 9: Disrupted Machine Pattern

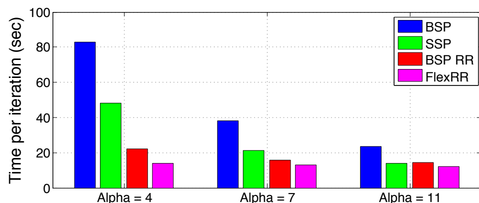


Figure 10: MF Speed, Power-Law Pattern

at 400% delay. As expected, adding RapidReassignment to BSP improves its convergence times, to faster than SSP but still much slower than FlexRR.

Figure 8b shows the results for *LDA*. For all delay %, BSP required 41 iterations to converge. For 0%, 50%, and 100% delays, FlexRR required 42 iterations, for 200% and 300% delays it required 43 iterations, and for 400% delay it required 44 iterations. Despite the need for these extra iterations, FlexRR converges significantly faster than BSP. With no injected delays, FlexRR converged 18% faster than BSP, and maintains near-Ideal convergence time with increasing delays. BSP, on the other hand, suffers from a linear increase in convergence time when delays are injected. *LDA* deviates from Ideal at higher delays for the same local state issue discussed in Section 5.4.1.

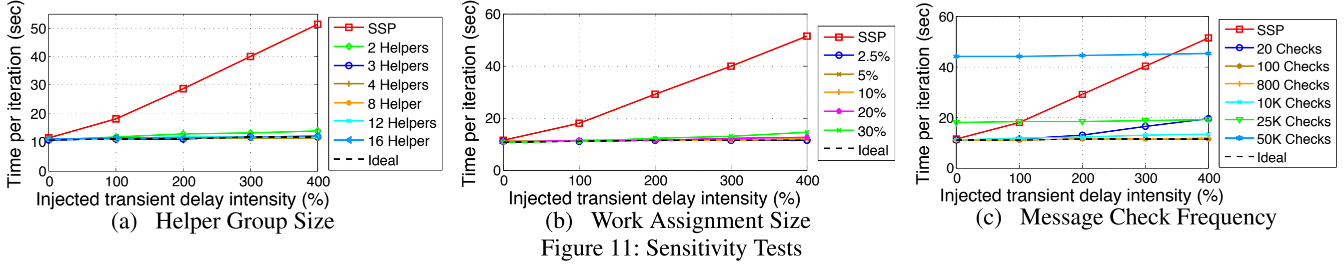
5.5 Other Straggler Patterns

Disrupted Machine Pattern. We compare the average time-per-iteration (20 iterations) of FlexRR to the alternative modes for the *Disrupted Machine Pattern*. Figure 9 shows results for *MF* on Cluster-A—results for *LDA* and *MLR* are qualitatively similar. SSP and BSP RR individually reduce the delay experienced by BSP by up to 49% and 42%, respectively. The combination of the two techniques in FlexRR matches Ideal, reducing the run-time by up to 63%.

Power-Law Pattern Results. Next, We compare the average time-per-iteration (20 iterations) of FlexRR to the alternative modes for the *Power-Law Pattern*. We present results on Cluster-A for each of our applications, setting α to 4, 7, and 11. Recall that $\alpha = 11$ emulates a real cluster measured in [42], and the configurations with smaller α values yield more severe delay. Figure 10 shows the results for *MF*. For $\alpha = 11$, SSP and BSP RR are faster than BSP by 39% and 40%, respectively. When the two techniques are combined in FlexRR, the run-time is 48% faster than BSP. Similarly to experiments conducted in earlier sections, with increasing delays (smaller α), the other three modes experienced significant increases in run-times, while FlexRR experienced only slight increases.

The results for *MLR* and *LDA* show similar trends. For $\alpha = 11$, SSP and BSP RR were 36% and 31% respectively faster than BSP for *MLR* and 37% and 42% respectively faster than BSP for *LDA*. FlexRR was 43% and 52% faster than BSP on *MLR* and *LDA* respectively. With increasing delays (smaller α), the other three modes experienced significant increases in run-times for both *MLR* and *LDA*. FlexRR experienced only modest delays for *MLR* and somewhat larger delays for *LDA* (not shown due to space constraints). In all cases, FlexRR significantly outperforms the other three modes.

Uneven Workload Distribution. While FlexRR was originally designed to mitigate transient stragglers, it is also effective at dealing with long-term workload differences among workers. Figure 12 shows an experiment on Cluster-A where half of the machines are assigned 75% of the workload, and the remaining half of machines are assigned 25% of the workload. FlexRR was able to mitigate the straggler effects of the uneven workload distribution, running at close to ideal speed



(a) Helper Group Size (b) Work Assignment Size (c) Message Check Frequency

Figure 11: Sensitivity Tests

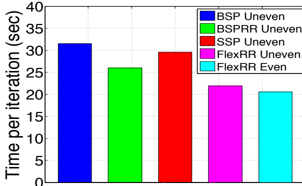


Figure 12: Uneven Workload

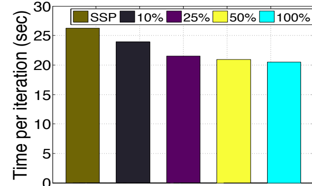


Figure 13: Partial Replication

shown in the FlexRR Even Bar, while SSP experienced a 54% slowdown.

5.6 Partial Replication

In all previous experiments, workers replicated 100% of the input data belonging to the workers that they are helping. We found that FlexRR is still effective when workers replicate only a portion of the input data, and thus are only eligible to help with that portion. Figure 13 shows the *MF* application with no injected delays run on Cluster-A with different percentages of the input data replicated on the helper workers. FlexRR with just 25% replication is close to FlexRR with 100% replication and much better than SSP (and BSP—not shown).

5.7 Sensitivity Study

This section reports on tests used to determine good settings for FlexRR parameters. We vary each parameter across its Table 1 range while using the default values for other parameters. For brevity, we show sensitivity results for *MF*'s average time-per-iteration (for 20 iterations) when running on Cluster-A under the Slow Worker straggler pattern, although similar results hold for the other two applications, convergence time, and the other clusters (same default settings used in all experiments, on all clusters, reported above).

Helper Group Size Test. Recall that the *helper group* is the set of workers to whom a worker is eligible to provide assistance. Figure 11a shows the results of varying the *helper group size* from zero helpers, which is equivalent to running in SSP mode, to sixteen helpers for each worker. The results show that, once the *helper group size* is set to 3 or higher, near-Ideal performance is achieved. Closer inspection reveals that using four helpers provides the best performance. But, the difference between settings from 3 to 16 is negligible.

Work Assignment Size Test. One of the key design decisions was the amount of work to be re-assigned in `do-this` assignment messages. Work assignments occur in two different sizes, an initial work assignment size and a follow-up work assignment size. Figure 11b shows the results of varying the *follow-up work assignment size* from 0% (equivalent to SSP) to 30%. The *initial work assignment size* is always half the *follow-up work assignment size*. Near-Ideal performance is achieved across the range of non-zero sizes, although as delays increased, the larger work assignments do perform worse. This occurs because of a rare corner case where workers that run slowly can re-assign a portion of their work to a faster worker that starts to complete the extra work but then is delayed significantly. Because the current implementation of FlexRR does not look to reassign work that has already been reassigned and accepted, other workers end up waiting in this case. This corner case is not a problem for smaller work assignments, because the helper does not fall behind significantly.

Message Check Frequency Test. FlexRR depends on messages between workers to keep track of progress and re-assign work. The *message check frequency* is the number of times a worker checks for incoming messages during an iteration. If the checks are not performed often enough, the system runs the risk of not reacting fast enough, while checking too often can cause an unnecessary overhead. Figure 11c shows that any frequency between 100 and 10K performs well, but the performance suffers once the frequency is greater than 10K.

6. Conclusion

FlexRR addresses the straggler problem for iterative convergent data-parallel ML. By integrating flexible consistency bounds with temporary peer-to-peer work reassignment, FlexRR successfully avoids having unhindered worker threads wait for workers experiencing slowdowns. Experiments with real ML applications under a variety of naturally-occurring and synthetic straggler behaviors confirm that FlexRR achieves near-ideal performance. On Amazon EC2 and Microsoft Azure, with no injected delays, this results in 15–56% reduction in runtimes, on average, over SSP and BSP. Experiments with various synthetic straggler patterns confirm that FlexRR consistently mitigates stragglers, resulting in up to 5–10× improvement over BSP and SSP in extreme cases.

Acknowledgments

We thank the members and companies of the PDL Consortium: Broadcom, Citadel, EMC, Facebook, Google, Hewlett-Packard Labs, Hitachi, Intel, Microsoft Research, MongoDB, NetApp, Oracle, Samsung, Seagate Technology, Tintri, Two Sigma, Uber, Veritas and Western Digital for their interest, insights, feedback, and support. This research is supported in part by Intel as part of the Intel Science and Technology Center for Cloud Computing (ISTC-CC), National Science Foundation under awards CNS-1042537, CCF-1533858, CNS-1042543 (PRObE [25]) and DARPA Grant FA87501220324.

References

[1] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, et al. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[2] U. A. Acar, A. Chargueraud, and M. Rainey. Scheduling parallel programs by work stealing with private deques. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, pages 219–228. ACM, 2013.

[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*, 2006.

[5] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica. Effective straggler mitigation: Attack of the clones. In *NSDI'13*, pages 185–198, 2013.

[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 9th USENIX conference on Operating systems design and implementation*, OSDI'10, pages 1–16. USENIX Association, 2010.

[7] Apache Hadoop. `http://hadoop.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. In *IEEE International Conference on Cluster Computing*, pages 1–12, 2006.

[9] C. M. Bishop et al. *Pattern recognition and machine learning*, volume 4. springer New York, 2006.

[10] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *JACM*, 46(5):720–748, 1999.

[11] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang. MXNet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274*, 2015.

[12] T. Chilimbi, Y. Suzue, J. Apacible, and K. Kalyanaraman. Project adam: Building an efficient and scalable deep learning training system. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 571–582. USENIX Association, 2014.

[13] 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.

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

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

[16] H. Cui, H. Zhang, G. R. Ganger, P. B. Gibbons, and E. P. Xing. Geeps: Scalable deep learning on distributed gpus with a gpu-specialized parameter server. In *Proceedings of the Eleventh European Conference on Computer Systems*, page 4. ACM, 2016.

[17] C. Curino, D. E. Difallah, C. Douglas, S. Krishnan, R. Ramakrishnan, and S. Rao. Reservation-based scheduling: If you're late don't blame us! In *Proceedings of the ACM Symposium on Cloud Computing*, SOCC'14, pages 2:1–2:14. ACM, 2014.

[18] J. Dean. Achieving rapid response times in large online services. In *Berkeley AMPLab Cloud Seminar*, 2012.

[19] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI*, 2004.

[20] J. Dinan, D. B. Larkins, P. Sadayappan, S. Krishnamoorthy, and J. Nieplocha. Scalable work stealing. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC'09, pages 53:1–53:11. ACM, 2009.

[21] J. Dinan, S. Olivier, G. Sabin, J. Prins, P. Sadayappan, and C.-W. Tseng. Dynamic load balancing of unbalanced computations using message passing. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1–8, 2007.

[22] A. D. Ferguson, P. Bodik, S. Kandula, E. Boutin, and R. Fonseca. Jockey: guaranteed job latency in data parallel clusters. In *Proceedings of the 7th ACM European conference on Computer Systems*, pages 99–112. ACM, 2012.

[23] 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. IEEE Computer Society, 2010.

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

[25] G. Gibson, G. Grider, A. Jacobson, and W. Lloyd. PRObE: A thousand-node experimental cluster for computer systems research. *USENIX; login*, 38(3), 2013.

[26] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *Proc. OSDI*, 2012.

[27] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica. GraphX: Graph processing in a distributed dataflow framework. In *11th USENIX Symposium*

on Operating Systems Design and Implementation (OSDI 14), pages 599–613, 2014.

[28] T. L. Griffiths and M. Steyvers. Finding scientific topics. *Proceedings of the National Academy of Sciences of the United States of America*, 2004.

[29] A. Harlap, G. R. Ganger, and P. B. Gibbons. Tier ml: Using tiers of reliability for agile elasticity in machine learning. 2016.

[30] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. H. Katz, S. Shenker, and I. Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *NSDI*, volume 11, pages 22–22, 2011.

[31] Q. Ho, J. Cipar, H. Cui, S. Lee, J. K. Kim, P. B. Gibbons, G. A. Gibson, G. R. Ganger, and E. P. Xing. More effective distributed ML via a Stale Synchronous Parallel parameter server. In *NIPS*, 2013.

[32] E. Krevat, J. Tucek, and G. R. Ganger. Disks are like snowflakes: no two are alike. In *USENIX conference on Hot topics in operating systems (HotOS)*, 2011.

[33] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[34] J. Langford, A. J. Smola, and M. Zinkevich. Slow learners are fast. In *NIPS*, 2009.

[35] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su. Scaling distributed machine learning with the parameter server. In *Proc. OSDI*, pages 583–598, 2014.

[36] M. Li, D. G. Andersen, A. J. Smola, and K. Yu. Communication efficient distributed machine learning with the parameter server. In *NIPS*, pages 19–27, 2014.

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

[38] 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)*, 2010.

[39] 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.

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

[41] New York Times dataset. `http://www.ldc.upenn.edu/`.

[42] 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 2003 ACM/IEEE conference on Supercomputing*, SC'03, pages 55–55. ACM, 2003.

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

[44] Power-law distribution. `http://en.wikipedia.org/wiki/Power_law`.

[45] C. Reiss, A. Tumanov, G. R. Ganger, R. H. Katz, and M. A. Kozuch. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *Proceedings of the Third ACM Symposium on Cloud Computing*, page 7. ACM, 2012.

[46] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Middleware 2001*, pages 329–350. Springer, 2001.

[47] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei. ImageNet Large Scale Visual Recognition Challenge, 2014.

[48] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *ACM SIGCOMM Computer Communication Review*, 31(4):149–160, 2001.

[49] A. Tumanov, J. Cipar, G. R. Ganger, and M. A. Kozuch. alsched: Algebraic scheduling of mixed workloads in heterogeneous clouds. In *Proceedings of the Third ACM Symposium on Cloud Computing*, page 25. ACM, 2012.

[50] J. Wang, J. Yang, K. Yu, F. Lv, T. Huang, and Y. Gong. Locality-constrained linear coding for image classification. In *Computer Vision and Pattern Recognition (CVPR), 2010 IEEE Conference on*, pages 3360–3367. IEEE, 2010.

[51] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. *HotCloud*, 10:10–10, 2010.

[52] M. Zaharia, A. Konwinski, A. D. Joseph, R. H. Katz, and I. Stoica. Improving MapReduce performance in heterogeneous environments. In *OSDI*, 2008.