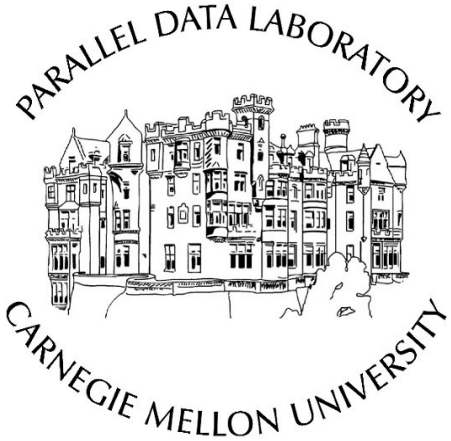

Solving the straggler problem for iterative convergent parallel ML

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

CMU-PDL-15-102
April 2015

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

Abstract

Parallel executions of iterative machine learning (ML) algorithms can suffer significant performance losses to stragglers. The regular (e.g., per iteration) barriers used in the traditional BSP approach cause every transient slowdown of any worker thread to delay all others. This paper describes a scalable, efficient solution to the straggler problem for this important class of parallel ML problems, combining a more flexible synchronization model with dynamic peer-to-peer re-assignment of work among workers. Experiments with both synthetic straggler behaviors and real straggler behavior observed on Amazon EC2 confirm the significance of the problem and the effectiveness of the solution, as implemented in a framework called FlexRR. Using FlexRR, we consistently observe near-ideal run-times (relative to no performance jitter) across all straggler patterns tested.

Acknowledgements: We thank Jim Cipar, Qirong Ho, Jin Kyu Kim, Senghyeuk Lee, and other big-learning collaborators for our many insightful discussions about system support for large-scale ML. We thank Ron Brightwell, Kurt Ferreira, Kevin Pedretti, and Lee Ward of Sandia National Laboratories for their insight about OS Jitters. We thank the members and companies of the PDL Consortium (including Actifio, APC, EMC, Emulex, Facebook, Fusion-IO, Google, Hewlett-Packard, Hitachi, Huawei, Intel, Microsoft, NEC Labs, NetApp, Oracle, Panasas, Riverbed, Samsung, Seagate, STEC, Symantec, VMWare, Western Digital) 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 and CNS-1042543 (PRObE [28]), by DARPA Grant FA87501220324, and by an AWS Education Grant award.

Keywords: Big Data infrastructure, Big Learning systems

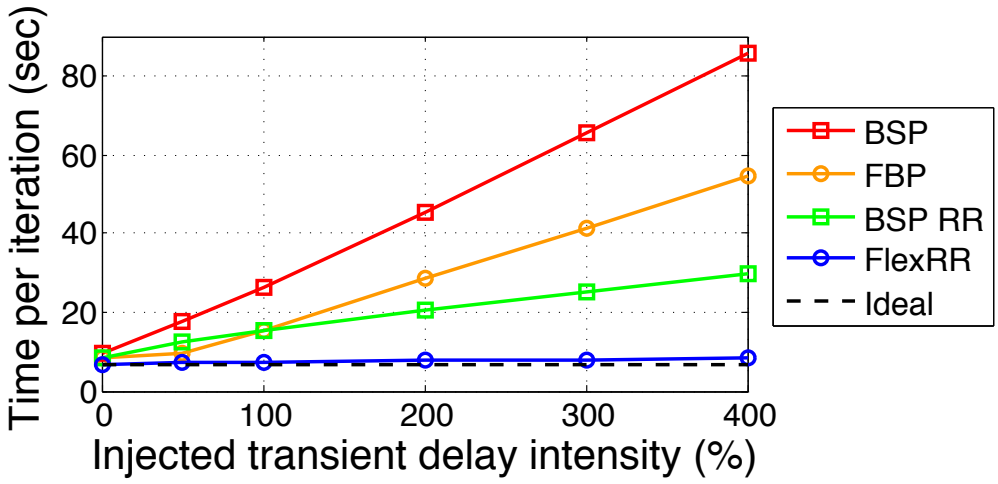

Figure 1: Performance consequences of transient stragglers. This graph shows average time-per-iteration for a collaborative filtering application (*MF*) running on 64 large EC2 instances, as a function of the scale of transient straggler effect. In this experiment, temporary slowdowns are injected into any given worker in any given iteration with a small probability. The "Ideal" line indicates the performance that would be achieved if all work were at all times perfectly balanced with no overhead. The "BSP" line shows the actual behavior when using BSP. The "FlexRR" line shows our solution, which combines *flexible consistency bounds* with *temporary work reassignments* to track Ideal closely. The "FBP" and "BSP RR" lines show use of individual ones of these two primary techniques, demonstrating that the two must be combined to solve the straggler problem for iterative convergent ML. Even with zero injected delays, FlexRR outperforms BSP by 35% due to performance jitter within EC2. Experimental details are in Section 5.

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) data, as closely as possible, such that the model can then be used to predict non-observed data points. Increasingly, parallel implementations of ML algorithms are used, as growth in the data sizes and the precision and complexity of the models creates large computation time requirements for determining parameter values.

Parallel ML computations involve synchronization and consistency challenges, similar to other parallel computations. There are diverse ML algorithms, and this paper focuses on one important subset: iterative convergent algorithms that can be solved in an input-data-parallel manner. Such algorithms begin with a guess as to the solution (i.e., the model parameter values) and proceed through multiple iterations over the input data to improve the solution. The convergence property enables such algorithms to find a good solution given any reasonable initial guess.

Most distributed implementations of such algorithms follow the Bulk Synchronous Parallel (BSP) computational model. Input data is divided among worker threads, which each iterates over its subset of the input data and determines solution adjustments based on its local view of the latest parameter values. All workers are required to 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 occurs when 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

can cause significant unproductive wait time for the others. (See Figure 1.) Unfortunately, even if the load is balanced, transient slowdowns are common in real systems 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 shared computing infrastructures rather than dedicated clusters (as is becoming increasingly common) and as the number of workers and machines increases.

This paper describes FlexRR, which combines flexible consistency bounds with temporary work reassignment to solve the straggler problem for iterative convergent ML algorithms. Flexible consistency bounds remove the barriers of BSP, allowing fast workers to proceed ahead of slowed workers by a bounded amount [16, 32, 39].[1] The bound enables convergence to be proven [32, 40], so faster workers are blocked if they are ahead of the slowest worker by more than the bound. With temporary work reassignment (which we refer to as 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 solve the straggler problem for iterative convergent ML. Flexible consistency bounds provide FlexRR with the extra time needed to detect slowed workers and address them with temporary work reassignment (RapidReassignment), before any worker reaches the bound and is blocked.

Extensive experiments demonstrate that FlexRR successfully solves the straggler problem for three real ML algorithms and various straggler behaviors. While BSP, and either technique alone, suffers large performance losses in the face of stragglers, FlexRR's solution consistently nearly matches the Ideal lower bound in which all work is at all times perfectly balanced among the available resources at no overhead (Figure 1). Experiments on Amazon EC2 instances with no injected straggler effects achieve a 35% performance improvement over the traditional BSP approach and 25% over FBP, even when executing relatively short experiments on relatively expensive EC2 instances that would be expected to have minimal resource sharing with other tenant activities. With more intense transient delays, much larger improvements (up to an order of magnitude) over BSP are observed.

This paper makes three primary contributions:

1. It describes the first runtime solution, to our knowledge, to the straggler problem for iterative convergent ML. As discussed in Section 2, previous work on stragglers focuses on impractical solutions (e.g., preventing them entirely) or other parallel computations (e.g., map-reduce jobs, which consist of idempotent tasks with no iteration or shared state).

2. It describes FlexRR, in Sections 3 and 4, which implements this solution in a scalable and efficient manner. For example, it uses P2P communication among workers to detect slowed workers and perform work re-assignment, avoiding a central decision-making bottleneck. It also limits the number of other workers with which any worker checks progress and offloads work, in order to bound the associated communication and limit the set of machines that may have to load any given input data.

3. It demonstrates the efficacy of the combined solution, as well as the two primary techniques individually, for three real ML algorithms. Section 5 evaluates behavior under various straggler patterns, including both synthetic ones intended to represent situations that can arise and baseline performance jitter experienced on Amazon EC2.

2 Background and Related Work

This section reviews iterative convergent ML algorithms, the common approach to parallelizing these algorithms, and previous work on addressing stragglers in parallel computing and Big Data analytics.

[1] We refer to this approach as Flexible consistency Bound Parallel (FBP), which is a generic term meant to encompass both Bounded Delay consistency [39] and Stale Synchronous Parallel (SSP) [16, 32].

2.1 Iterative Convergent Algorithms

A common approach to solving many ML tasks (e.g., matrix factorization, multinomial logistic regression, and latent Dirichlet allocation) is to search a space of potential solutions looking for a solution with a large (or in the case of minimization, small) objective value, using an iterative convergent algorithm. Such algorithms start with an initial solution (assignment of model parameter values) and proceed through a sequence of iterations, each one 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 and outputs a solution. Stopping criteria may be based on surpassing a target objective value or a leveling off of objective value improvements. A key property of these algorithms is that they will converge to a good solution even if there are minor inconsistencies in the way parameter updates are calculated. This makes them amenable to parallel updates on model parameters, as well as to flexible consistency bounds.

2.2 BSP-based Parallelization and Stragglers

Most parallel implementations of iterative convergent algorithms follow the Bulk Synchronous Parallel (BSP) computational model, using an input-data-parallel approach. The input data is divided among worker threads that execute in parallel, performing the work associated with their shard of the input data, and executing barrier synchronizations at the end of each iteration. For an iterative convergent 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. This continuity provides significant efficiency benefits, relative to independently scheduling work each iteration, in addition to avoiding the scalability and latency challenges of a central scheduler. It exploits caches of input data and algorithm state populated with the information needed for the previous iteration's work assignment. This is akin to CPU cache affinity scheduling in multi-processor and multi-core systems [48, 50, 35], but with larger efficiency benefits due to input data sizes and the relatively higher costs of network transfer.

The 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 as illustrated in Figure 1.

Stragglers can occur for a number of reasons [9, 8], including heterogeneity of hardware [36], hardware failures [9], unbalanced data distribution among tasks, garbage collection in high-level languages, and various operating system effects [10, 44]. Additionally, for ML algorithms, expensive stopping criteria computations can lead to significant straggler effects, such as when the computation is performed on a different one of the machines every so many iterations. Even when it is distributed across all machines, the computation is unlikely to be balanced.

2.3 Related Work Addressing Stragglers

Stragglers have long been a problem for parallel computing, and many techniques have been developed to mitigate them.

Eliminating performance variation that causes stragglers. The scientific computing community (sometimes known as the supercomputing or High Performance Computing community)—which frequently runs applications using the BSP model—puts significant effort into identifying and removing sources of

performance jitter from the hardware and operating systems of their supercomputers [26, 44]. While this approach can be effective at reducing performance "jitter" in specialized and dedicated machines, they are not intended to solve the more general straggler problem. For instance, they are not applicable to programs written in garbage-collected languages, do not handle algorithms that inherently cause stragglers during some iterations, and do not allow for the shared computing infrastructures on which many ML applications are executed [19, 25, 31].

Blacklisting is a limited form of performance variation elimination, which attempts to mitigate stragglers by identifying workers that are falling behind and ceasing to assign work to those workers. 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 [21]. 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 has become a common approach to mitigating stragglers in data processing systems like MapReduce and Hadoop [1, 22, 9, 52, 8]. Jobs in these systems consist of stateless, idempotent map and reduce tasks, and speculative execution consists of executing some 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. Various approaches to deciding which tasks to execute redundantly, and when, have been explored [22, 9, 52, 8] for balancing the trade-off between resource efficiency and job latency.

Computation redundancy is less suitable for iterative ML algorithms than for the collections of stateless, idempotent tasks comprising map-reduce jobs. Most importantly, a worker's processing in these algorithms is generally not idempotent, and applying the same adjustments more than once could affect convergence negatively. In fact, as discussed in Section 4.4, some iterative ML algorithms involve local state at the workers that cannot tolerate redundant execution of parts of a computation. FlexRR instead 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. As described above, work assignments are generally kept intact from one iteration to the next, for efficiency reasons. This is why some workers can complete their iteration while other workers have significant work yet to do. Work stealing and work shedding are mirror approaches for adaptively rebalancing work queues among workers [12, 24, 5]. The concept is to move work from a busy worker to an idle worker. It has been studied extensively for parallel computations on shared-memory machines (and is an integral part of Intel TBB [34] and Cilk Plus [33]), and has also been applied in some distributed memory cases [23]. 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., it moves data to the thread that steals the work), FlexRR is data-centric, minimizing data movement (e.g., it limits the number of other workers to which a worker can offload work and can preload the needed input data). Third, because of its focus on *transient* stragglers, FlexRR's reassignments are only temporary. Finally, it works in conjunction with flexible consistency bounds, as discussed below.

Using less strict progress synchronization. To reduce the impact of stragglers, there have been various proposals for replacing the strict barriers of BSP with looser work coordination models, ranging from slightly looser to completely unbounded (i.e., unsynchronized workers).

FlexRR uses what we refer to in this paper as the Flexible consistency Bound Parallel (FBP) computing model, which has been recently proposed and studied under two different names: "Stale Synchronous Parallel" (SSP) [16, 32, 20] and "Bounded Delay consistency" [39]. These schemes essentially generalize BSP by allowing any worker to be no more than a bounded number of iterations ahead of the slowest worker. So,

for BSP, the bound (which we will refer to as the *FBP bound*) would be zero. With a 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 $t - 1$. Consistent with our results, such as in Figure 1, FBP has been shown to mitigate low-intensity transient straggler effects [17, 39] but not larger effects. (Increasing b only serves to push the knee of the FBP curve in Figure 1 to the right, at a cost of taking longer to converge even at 0% intensity [17].) FlexRR combines FBP with temporary work reassignment to comprehensively solve the straggler problem for input-data-parallel iterative ML algorithms.

Approaches that do not ensure that all workers coordinate progress have also been developed and explored. For example, Albrecht et al. [7] 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. In some cases, consistency and synchronization can be ignored altogether, relying on a best-effort model for updating shared data. Yahoo! LDA [6] and Project Adam [15], as well as most solutions based around NoSQL databases, rely on this model. While such approaches can work well in some cases, having no limits on how far behind some workers may get makes having confidence in ML algorithm convergence difficult. (FBP, in contrast, admits proofs of convergence [32, 39, 40]).

Another class of solutions attempts to reduce the need for synchronization by restricting the structure of the communication patterns. For example, GraphLab [42, 43] 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 know the communication pattern a priori and specify it explicitly before the computation starts.

3 FlexRR Design and 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 (i.e., the primary focus of this paper). Example applications built using FlexRR are described in Section 5.2.

FlexRR 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. The architecture is depicted in Figure 2.

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 stopping criterion may be a number of iterations, an amount of time, or a determination of convergence. The input file contains a sequence of data units in an understood format (e.g., rows that each contain one input data unit in an easy-to-process format). The most important function provided (`process-input`) is for processing an input data unit, taking the data unit 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 subset of the input data units to process each iteration. The default assignment is a contiguous range of the input data, determined based on the worker ID, number of workers, and number of data units. Each worker has an outer loop for iterating

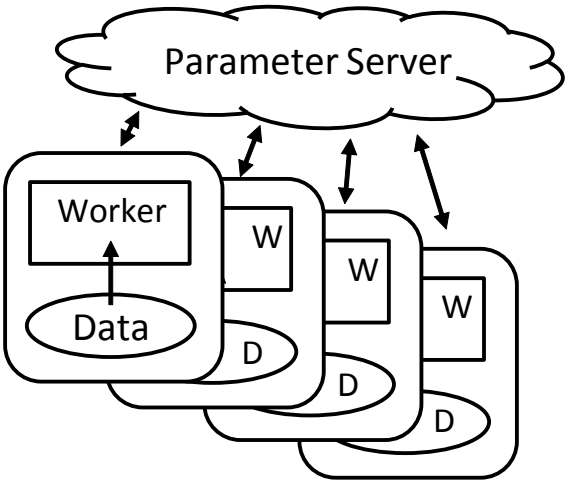

Figure 2: FlexRR architecture. This high-level picture illustrates FlexRR's logical structure, for the case of one core (so, one worker thread) per node. The worker is assigned some input data to process in each iteration, making adjustments to the model parameters stored in a parameter server. Logically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

until the stopping criterion is reached and an inner loop for each iteration[2] that reads one data unit and calls `process-input` with it as an argument.

3.2 Parameter Server for Shared State Management

FlexRR uses the increasingly popular parameter server approach [39, 32, 6, 45, 20] 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 handling the communication and consistency issues.

The FlexRR parameter server is similar to those described in the citations above. It exposes a simple key-value interface to the ML application code, allowing its functions (e.g., `process-input`) to read and update the model parameter entries (generally mathematical vectors). The two primary functions are `read-param` and `update-param`, which fetch or apply a delta to a value specified by the key. The value type is application-defined, but must be serializable and be defined with an associative aggregation function, such as plus, multiply, or union, so that updates from different worker threads can be applied in any order. For the ML applications used in this paper, this aggregation function is simply a mathematical add (+).

In order to reduce cross-node traffic, the parameter server implementation includes 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—each such process has a number of parameter server threads and maintains one shard of the shared state. Threads associated with the client-side cache communicate with the appropriate threads associated with the server shards on the other nodes. Updates are write-back cached. At each iteration, cached updates are sent to the parameter server shards, and cached entries are refreshed to handle subsequent reads.

FlexRR supports both the BSP and FBP models discussed in Section 2. Each cached value is associated with an iteration number that indicates the global iteration number it reflects (i.e., 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 FBP bound (zero, for BSP); that is, the value is up-to-date enough if it reflects all updates from iterations more than "FBP bound" before the worker's current local iteration number. Otherwise,

[2]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 be some number of iterations or some number of data units.

the read must proceed to the appropriate server shard to retrieve the value, possibly waiting there for other workers' updates.

For fault tolerance, FlexRR takes checkpoints of the parameter state to enable recovery from server failures—we use an approach similar to [17] to handle checkpointing in the presence of flexible consistency bounds. As discussed in Section 4.5, FlexRR leverages its RapidReassignment mechanism to enable fast recovery from worker failures.

3.3 Straggler Mitigation

FlexRR combines two mechanisms, flexible consistency bounds via the FBP model and temporary work reassignments via a technique we call *RapidReassignment*, to solve the straggler problem for the iterative ML applications supported by its programming model. The FBP model, described above and in Section 2, allows each worker thread to be ahead of the slowest worker by up to a specified FBP bound number of iterations. This flexibility mitigates low-intensity stragglers, as shown in Figure 1, but more importantly provides enough flexibility for RapidReassignment to be highly effective. RapidReassignment uses peer-to-peer communication to allow workers to self-identify as stragglers and temporarily offload work to workers that are ahead. FlexRR's implementation of FBP is similar to recent systems [39, 17], whereas its temporary work reassignment mechanism is new and so is described in detail in the next section.

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 indeed 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). When combined with FBP, RapidReassignment is highly effective in mitigating straggler delays of all intensities.

4.1 Worker Groups

RapidReassignment is designed with scalability in mind, using peer-to-peer coordination among workers instead of a central arbiter. Like overlay networks [49, 46], 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. 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 overlap between the helper and helpee groups is possible, 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.

Another motivation for creating worker groups is the sharing of input data. During initialization, each worker is assigned a portion of the input data to process each iteration. When work is reassigned, the 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. Toward that end, each worker under RapidReassignment prefetches the input data of its helpee group members after loading its own data.

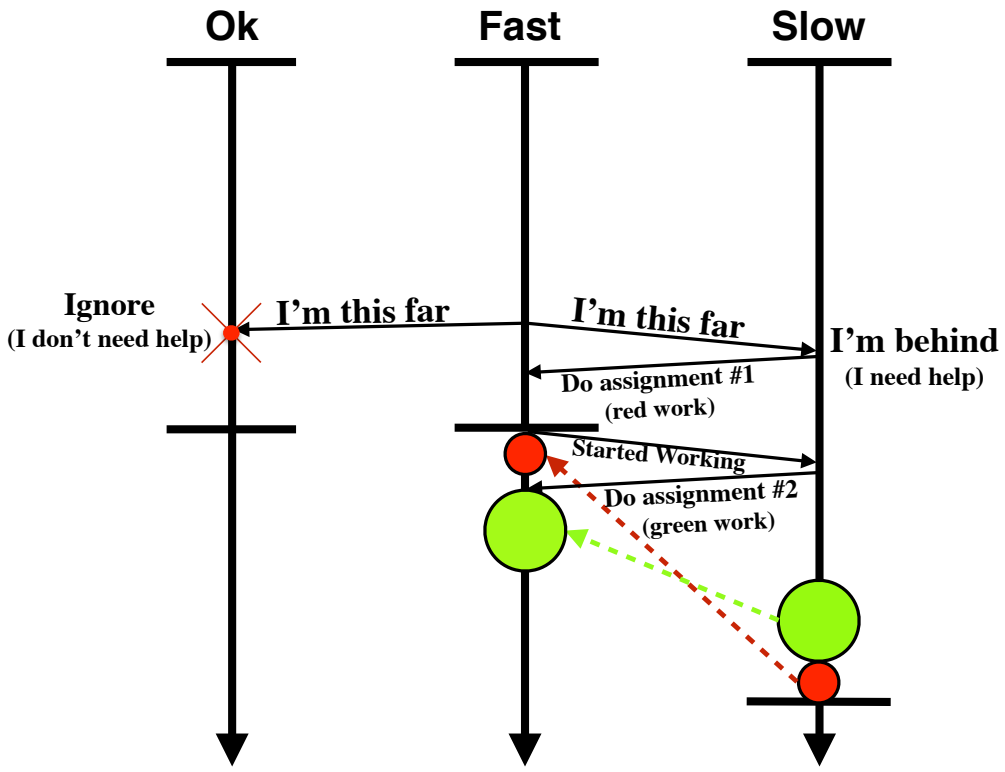

Figure 3: RapidReassignment example. The middle worker sends progress reports to the other two workers (its helper group). The worker on the left is running at a similar speed, so it ignores the message. The worker on the right is running slower, so it sends a `do-this` message to re-assign an initial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

4.2 Worker Communication

RapidReassignment uses non-blocking Message Passing Interface (MPI) for communication among workers to provide progress reports and offload work. 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 (see Section 5.6) 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 infrequently (e.g., hourly).

4.3 RapidReassignment Actions

This section describes the five primary RapidReassignment actions in FlexRR.

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 using 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* ← *completion_diff* $+ \frac{time\_diff}{current\_avg}$

Reassigning Work. A worker compares its progress in relation to its helpers based on `progress-report` messages. If a worker finds that it has fallen behind 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. This initial work assignment is a percentage of its work for the current iteration. Section 5.6 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.

Algorithm 2 Helping Decision

1: $msg \leftarrow$ check for helping requests
2: **if** $msg.timestamp \leq last\_cancellation\_timestamp$ OR $msg.iteration > current\_iteration$ **then**
3: Discard msg
4: **else if** $msg.iteration < current\_iteration$ **then**
5: Send `begun-helping` and do the help
6: **else if** finished its own work this iteration **then**
7: Send `begun-helping` and do the help
8: **else**
9: Save msg for the end of this iteration
10: **end if**

Helping with Work. Algorithm 2 outlines the helping decision process. 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 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 (not shown in Algorithm 2).

If the iteration number in the `do-this` message equals the helper's current iteration, 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 any additional valid `do-this` messages, it will move on to its own next iteration.

Assigning Additional Work. 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 (see Figure 3). This follow-up work assignment is double the size of the initial work assignment. Subsequently, for the rest of the iteration, whenever the worker receives an additional `begun-helping` message, it will reassign an additional follow-up work assignment.

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 *FBP bound*. There is a small window in which both the helpee and a helper may do 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.

4.4 Local State

While many iterative ML algorithms (e.g., the *MF* and *MLR* applications described in Section 5.2) do not keep worker-local state regarding particular input data, some do and therefore require additional mechanism when reassigning work. Usually, this state consists of the previous decision regarding each given data item. For example, for the LDA application described in Section 5.2, a worker remembers its previous topic assignment probabilities for a document's words so that it can compute a delta to the shared model parameters when it changes those probabilities. That is, when it wants to update the model with an assignment for the current iteration, it needs to use its previous assignment to calculate the delta to be applied.

One option is to send the worker-local state along with the `do-this` message. But, because the local state can be quite large, it is desirable to avoid transporting it from worker to worker when reassigning work. We have designed a mechanism that enables a helper to not need to know the helpee's previous change, and vice versa, when using work reassignment, as illustrated in Algorithm 3. When a worker reassigns a piece of work, it will subtract its previous changes from the shared model parameters at the end of the iteration. The helper who receives this reassigned work calculates and adds the new change to the model as it normally would for all its changes (lines 8–11); this delta is automatically the appropriate delta because the helpee undid its previous change. In the next iteration, the helper subtracts the change it applied, so that the helpee's next change is likewise the appropriate delta.

While this solution slightly decreases the convergence-per-iteration, the speedup in time-per-iteration greatly outweighs this effect, as shown in Section 5. We have found this solution to be sufficient for data-parallel iterative algorithms involving worker-local state such as *LDA* and *PageRank* [14].

4.5 Fast Failure Recovery

Parameter servers often provide fault tolerance for server failures, either using checkpointing [17, 45, 6, 29] or replication [39]. Some systems also provide fault tolerance for worker failures, by starting a new worker to replace the failed one [39]. However, the newly started worker needs to load the input data, from its local disk or a file server across the network, before beginning to work at full speed, causing non-trivial recovery time. Loading the input data can be so time-consuming that some people argue that it is actually better to have the computation move on without recovering the failed worker (or its input data) [39]. Our RapidReassignment mechanism makes worker recovery much faster. Using RapidReassignment, each worker loads the input data

Algorithm 3 Model change tracking on work reallocation

```
1: Initialize all tracked_changes to 0
2: for each iteration do
3:     for each data in helped_work_last_iter do
4:         Increment model by (−tracked_changes[data])
5:         tracked_changes[data] ← 0
6:     end for
7:     for each data in own_work or helping_work do
8:         new_change ← calc_change(model, data)
9:         delta ← new_change − tracked_changes[data]
10:        Increment model by delta
11:        tracked_changes[data] ← new_change
12:    end for
13:    for each data in given_away_work do
14:        Increment model by (−tracked_changes[data])
15:        tracked_changes[data] ← 0
16:    end for
17: end for
```

of its helpee group as well as its own data.[3] As a result, upon a worker failure, members of a failed worker's helper group can take over the work of the failed worker rapidly (like it does with work reassignments).

5 Evaluation

This section evaluates the effectiveness of FlexRR. Results are reported for both an in-house cluster and a set of Amazon EC2 instances. The results support a number of important findings: (1) in the face of transient stragglers, FlexRR greatly outperforms BSP and FBP, achieving near-ideal performance; (2) to achieve ideal performance, the RapidReassignment and FBP techniques need to be combined, as is done by FlexRR, as neither along is sufficient; (3) 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 two 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, and they are connected with 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). From our testing using `iperf`, we observe a bandwidth of 1 Gbps between each pair of instances. For more carefully controlled experiments, we primarily use Cluster-A; Cluster-B is used to validate that the results on EC2 match (and even improve upon) the results on our dedicated cluster.

Injected Straggler Patterns. Our goal is to study a wide variety of transient straggler patterns that are likely to be encountered in practice. We used three distinct methodologies for generating transient straggler

[3] If space is a concern, an optimization would be to load only the portion of the helpee input data that is likely to be reassigned (i.e., the portion processed towards the end of an iteration).

patterns:

Slow Worker Pattern: Models transient worker slowdown by inserting *sleep* commands into the worker thread. At each of 10 possible delay points within an iteration, each worker is independently selected for slowdown with a 1% probability. The duration of the slowdown period for the selected worker is uniformly randomly chosen between zero and two times the duration of an iteration. We are interested in the impact of these delays as a function of how much each worker is slowed down. We denote the *transient delay intensity %* (*delay %* for short) to be the percentage by which each selected worker is slowed down (e.g., 100% delay means runs twice as slow). To simulate the effect of a worker running slower for some duration of time, 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 3 milliseconds sleep at each point for the duration of the delay.

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 it would without any delays.[4]

Power-Law Pattern: Based on a real-world straggler pattern [44], this uses a power-law distribution [4] 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 [44].

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

BSP	Classic BSP execution
FBP	Flexible consistency Bounded Parallel
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 FBP (without RapidReassignment) enables us to study the impact of RapidReassignment and flexible consistency bounds in isolation versus in combination, as in FlexRR. For flexible consistency bounds, we use a FBP bound of 1 in all experiments.

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 5.6 provides a sensitivity analysis on the parameter settings, showing that FlexRR performs well over a broad range of settings.

[4]The actual worker slowdown depends on how CPU-bound that worker is.

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%

Experimental Methodology. Every experiment was run at least thrice, and the arithmetic mean was used as the result. In every experiment, 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 different 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 [27, 38, 17, 18], 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.

Multinomial Logistic Regression (MLR) is a popular model for multi-way classification, such as used in the last layer of deep convolutional network for image classification [37] or text classification [41]. In *MLR*, the likelihood that each (d-dimensional) observation $\boldsymbol{x} \in \mathbb{R}^d$ belongs to each of the K classes is modeled by *softmax* transformation $p(\text{class=}k|\boldsymbol{x}) = \frac{\exp(\boldsymbol{w}_k^T \boldsymbol{x})}{\sum_j \exp(\boldsymbol{w}_j^T \boldsymbol{x})}$, 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 the stochastic gradient descent (SGD) algorithm where each gradient updates the full model [11]. We ported the MLR application from Petuum open source release [20]

Our *MLR* experiments use the *ImageNet* dataset [47] with LLC features [51], 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 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.

Similar to the other distributed ML systems [6, 29, 17, 18, 39] supporting *LDA*, our *LDA* solver implements the collapsed Gibbs sampling algorithm [30]. In every iteration, each worker goes through its assigned documents and makes adjustments to the topic assignment of the documents and the words. This algorithm requires the workers to keep track of their previous topic assignments to the documents that they are responsible for, and we handle these *local states* using the method described in Section 4.4.

The *LDA* experiments use the *Nytimes* dataset [3], 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.

5.3 Slow Worker Pattern Results

This section studies the speed and convergence rate of the ML applications under the *Slow Worker Pattern* of transient stragglers. Sections 5.4 and 5.5 present results for the other two synthetic straggler patterns.

5.3.1 Speed Tests

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

Results on Cluster-A. Figures 4–6 show the results for the *MF*, *MLR*, and *LDA* applications running on Cluster-A. BSP slows down linearly with delay intensity. FBP can mitigate delays below its FBP bound, but then too suffers linearly. FlexRR, on the other hand, nearly matches Ideal. 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. Note that at high delay %, there is some divergence from Ideal for *LDA*. That is because *LDA* uses some worker local state to track current document classifications, and as described in Section 4.4, every time FlexRR has a worker re-assign work, it will incur two extra model parameter updates. 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 the ideal run-time. 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 significantly decrease its straggler penalty (even below that of FBP), but it is the combination of flexible consistency bounds and RapidReassignment in FlexRR that nearly matches Ideal.

Results on Cluster-B. Figure 1 (on page 1) 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, FBP can mitigate stragglers only up to its FBP 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), FlexRR is 35% faster than BSP and 25% faster than FBP, because of non-injected performance jitter. This improvement on EC2 comes 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 shared clusters.

Figure 7 shows the results for *LDA* on Amazon EC2 cluster. Again, the results are qualitatively the same as on Cluster-A (compare Figure 6), with the main difference being the larger separation between FlexRR and the next best approach. FlexRR reassigns 18% of the work at 0% delay and 28% of the work at 400% delay. At 0% delay (baseline EC2 with no injected delays), FlexRR is 34% faster than BSP.

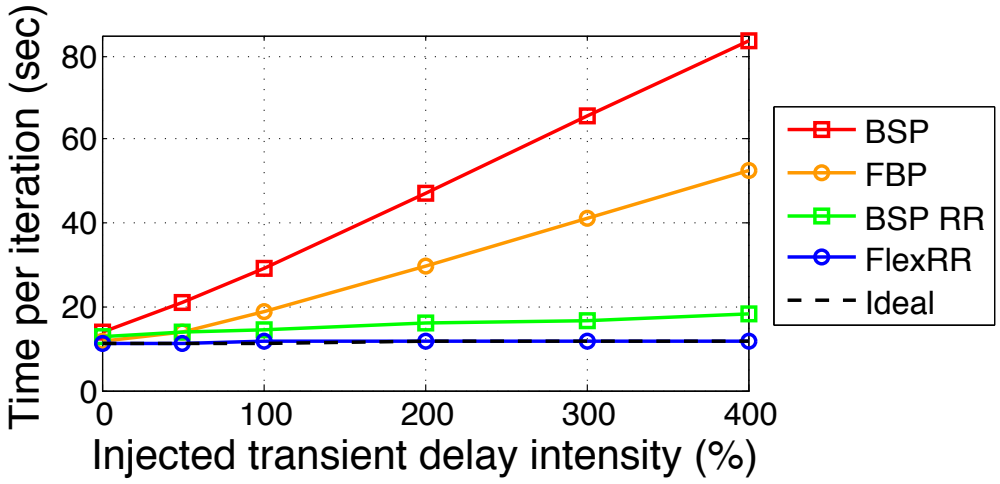

Figure 4: MF Speed Test on Cluster-A, Slow Worker Pattern

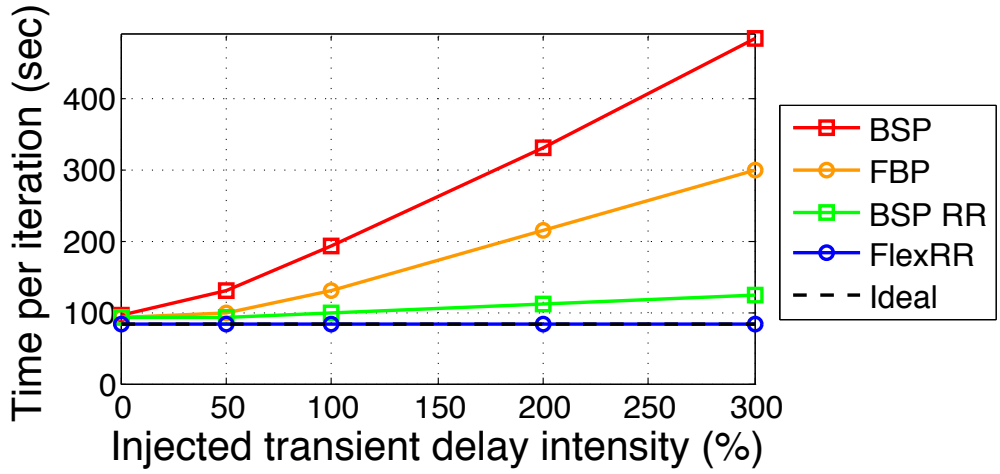

Figure 5: MLR Speed Test on Cluster-A,Slow Worker Pattern

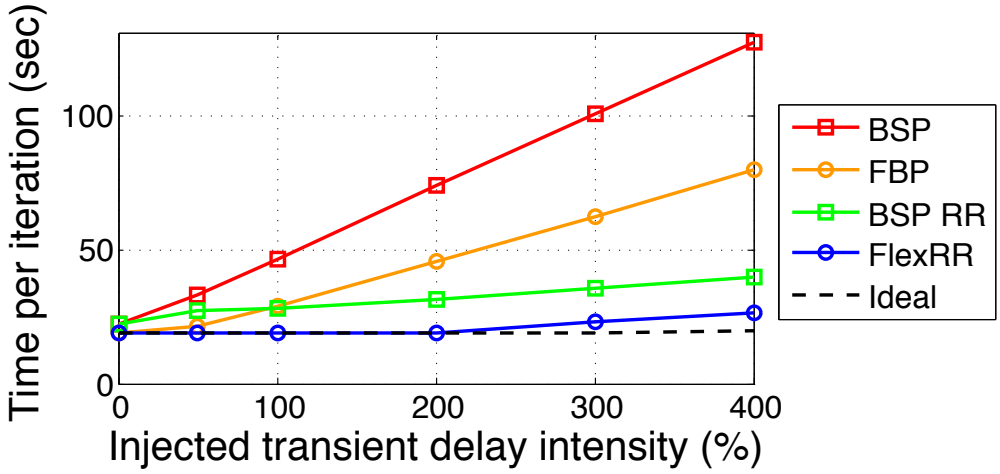

Figure 6: LDA Speed Test on Cluster-A, Slow Worker Pattern

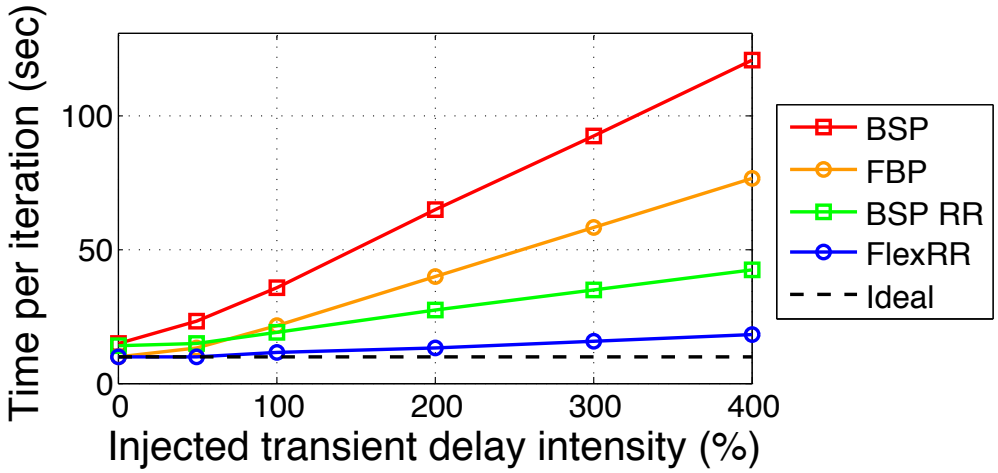

Figure 7: LDA Speed Test on Cluster-B, Slow Worker Pattern

5.3.2 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 from Section 5.3.1 by the number of iterations needed to reach convergence by the BSP experiment.[5] Given limited space, we show here only our experiments on Cluster-A for both *MF* and *LDA*.

Criteria for Convergence. We use the following stopping criterion, based on guidance from machine learning 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 8 shows the results for *MF*. For all delay %, BSP (and BSP RR) required 112 iterations to reach convergence and FBP 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 at 400% delay. As expected, adding RapidReassignment to BSP improves its convergence times, to faster than FBP but still much slower than FlexRR.

Figure 9 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 5.3.1.

5.4 Disrupted Machine Pattern Results

This section explores the *Disrupted Machine Pattern* described in Section 5.1, comparing the average time-per-iteration (measured from 20 iterations) of FlexRR to the alternative modes. Figure 10 shows results for

[5] We use BSP iterations in this lower bound because the flexible consistency bounds of FlexRR and FBP can lead to a (modest) increase in the number of iterations needed [16, 32]. Our experimental results confirm this.

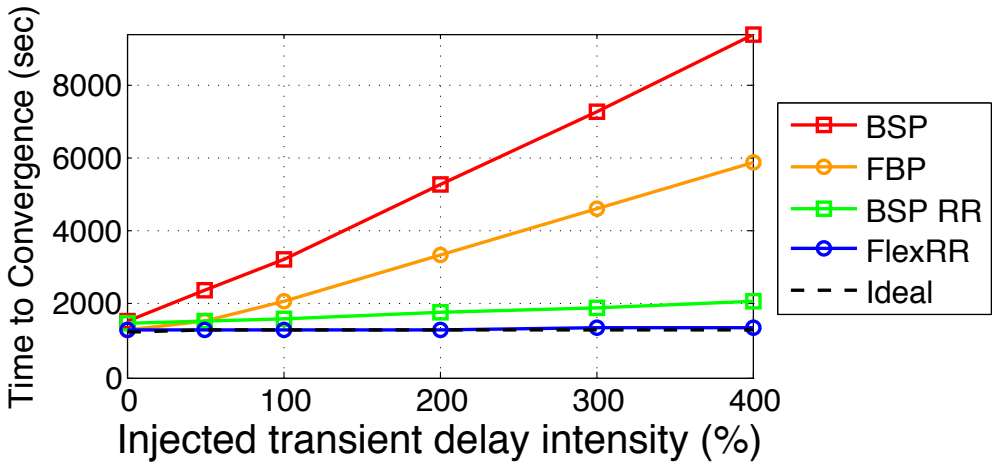

Figure 8: MF Convergence Test, Slow Worker Pattern

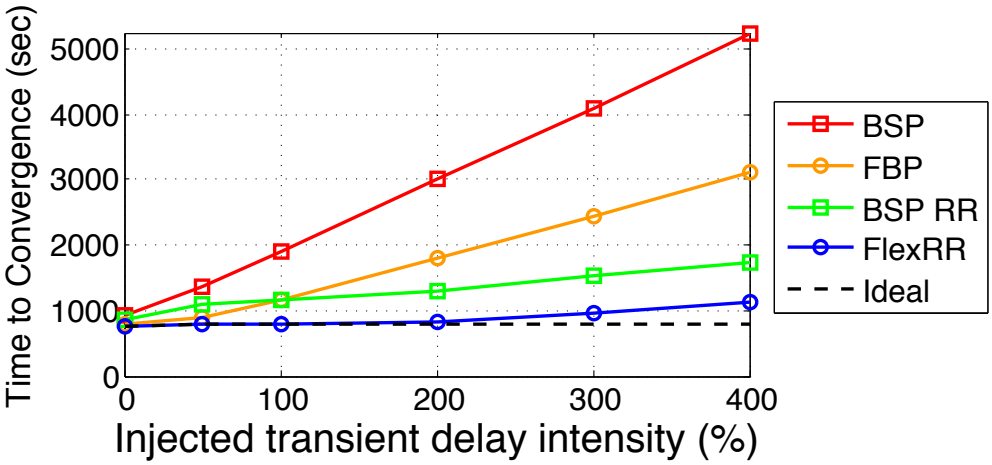

Figure 9: LDA Convergence Test, Slow Worker Pattern

MF on Cluster-A—other results are qualitatively similar. Both FBP and BSP RR were able to individually reduce the delay experienced by BSP, by up to 49% and 42%, respectively. The combination of the two techniques in FlexRR was able to match Ideal speed by improving the BSP run-time by 63%.

5.5 Power-Law Pattern Results

This section considers the third transient straggler pattern: the *Power-Law Pattern* described in Section 5.1. 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 [44], and the configurations with smaller α values yield more severe delay. We compared the average time-per-iteration (measured from 20 iterations) when running four different modes: BSP, FBP, BSP RR, and FlexRR.

Figure 11 shows the results for *MF*. For $\alpha = 11$, FBP 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* (Figure 12) and *LDA* (Figure 13) show similar trends. For $\alpha = 11$, FBP and BSP RR were 36% and 31% respectively faster than BSP for *MLR* and 37% and 42% respectively faster than BSP

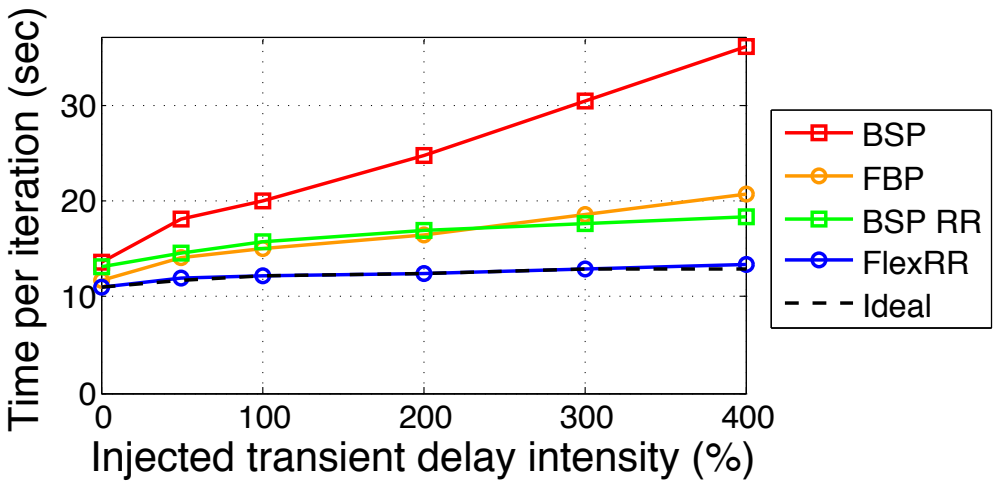

Figure 10: MF Speed Test, Disrupted Machine Pattern

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*. But, in all cases, FlexRR significantly outperforms the other three modes.

5.6 Sensitivity Study

This section shows results of sensitivity tests used to determine good settings for FlexRR parameters, and their impact on performance We vary each parameter across its Table 1 range while using the default values for the remaining parameters. For brevity, we show sensitivity results for *MF*'s average time-per-iteration (for 20 iterations) when running on Cluster-A, although similar results hold for the other two applications, for convergence time, and for Cluster-B.

Helper Group Size Test. Recall that the *helper group* is the set of workers to whom a worker is eligible to provide assistance. Figure 14 shows the results of varying the *helper group size* from zero helpers, which is equivalent to running in FBP 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 15 shows the results of varying the *follow-up work assignment size* from 0% (equivalent to FBP) 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

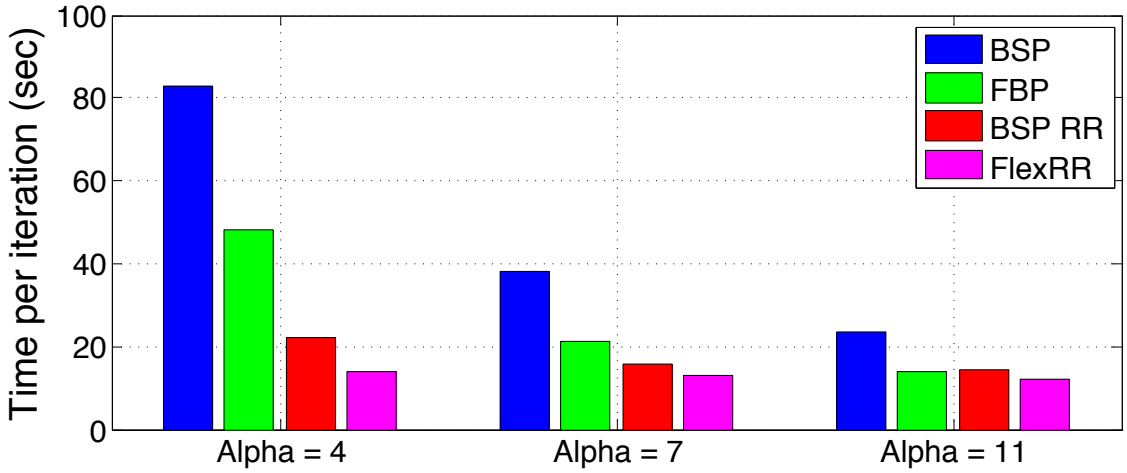

Figure 11: MF Speed Test, Power-Law Pattern

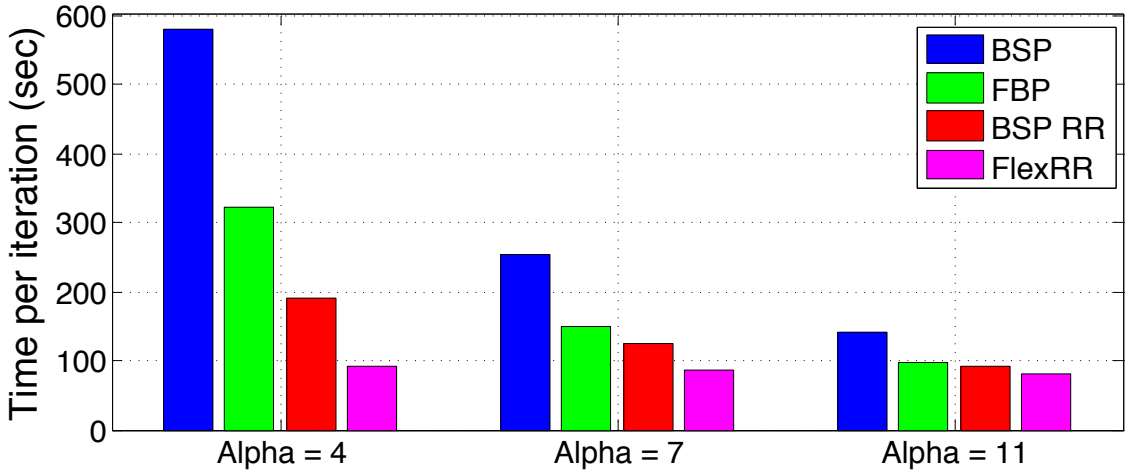

Figure 12: MLR Speed Test, Power-Law Pattern

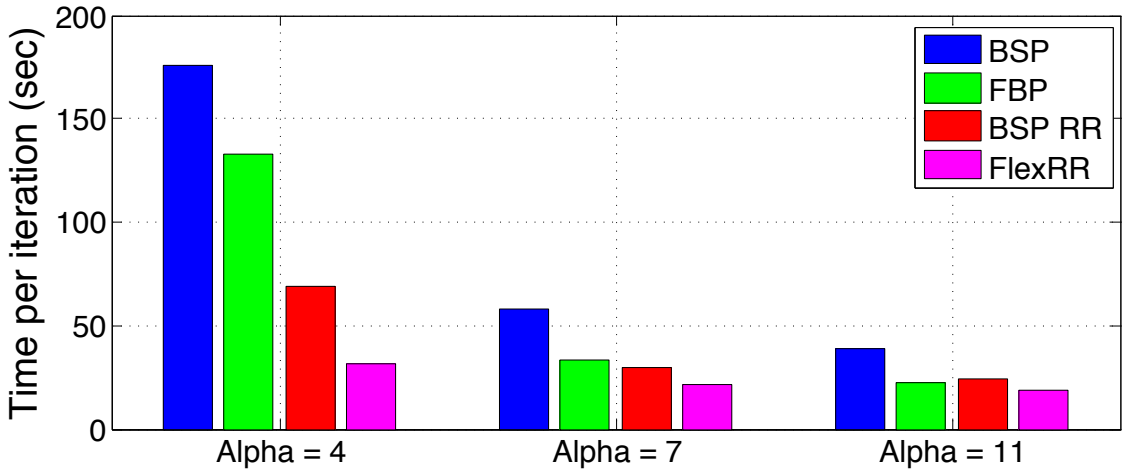

Figure 13: LDA Speed Test, Power-Law Pattern

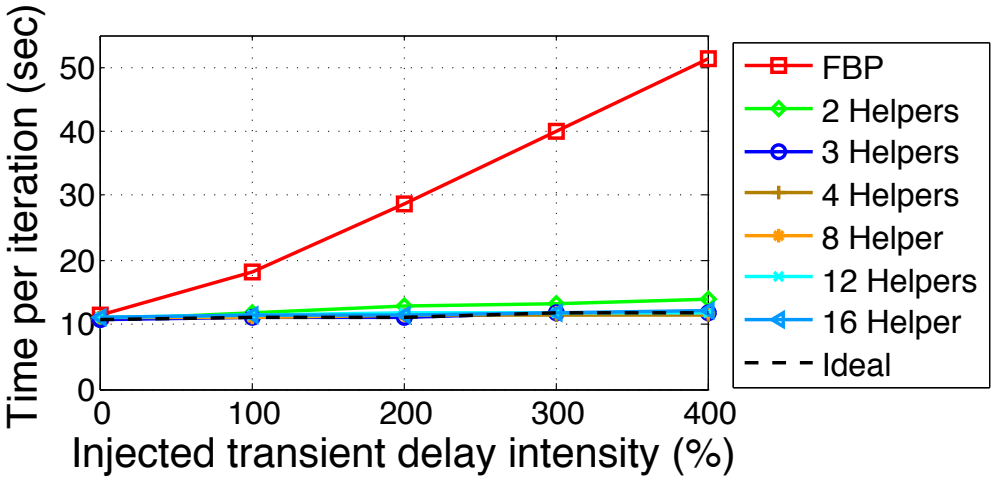

Figure 14: Sensitivity to Helper Group Size

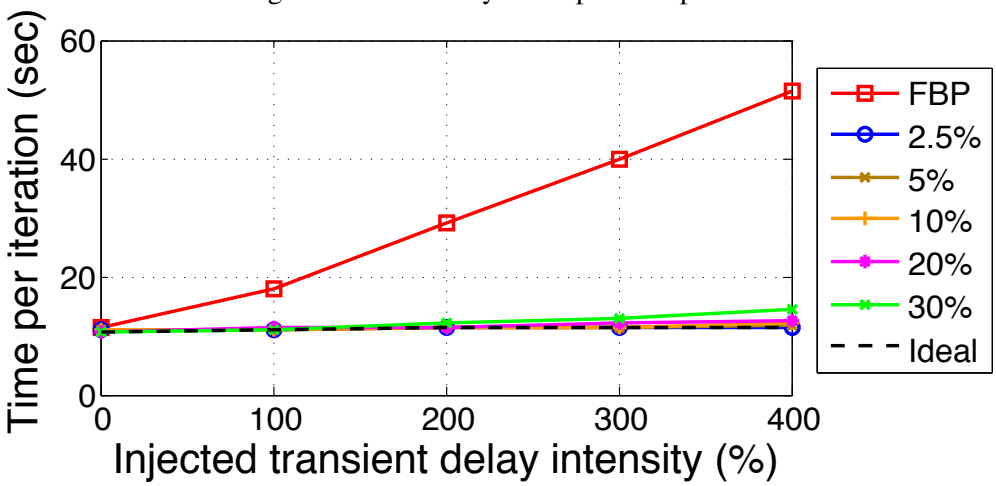

Figure 15: Sensitivity to Work Assignment Size

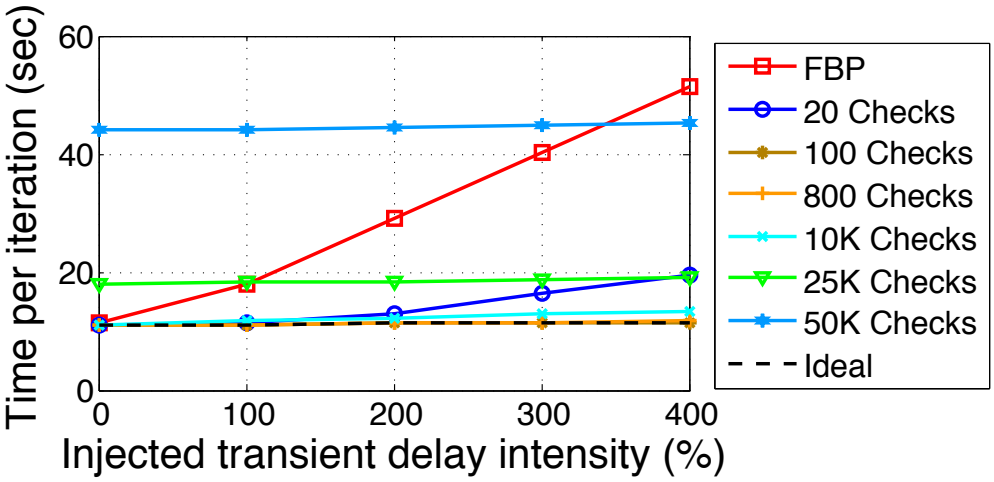

Figure 16: Sensitivity to Message Check Frequency

risk of not reacting fast enough, while checking too often can cause an unnecessary overhead. Figure 16 shows that any frequency between 100 and 10K performs well. Once the frequency is greater than 10K, the overhead of checking messages negatively impacts performance.

6 Conclusion

FlexRR solves the straggler problem for iterative convergent data-parallel ML. By integrating flexible consistency bounds (FBP) with temporary peer-to-peer work reassignment (RapidReassignment), FlexRR successfully avoids having unhindered worker threads wait for workers experiencing transient slowdowns. Experiments with real ML applications under a variety of real and synthetic straggler behaviors confirm that FlexRR achieves near-Ideal performance. On Amazon EC2, with no injected delays, this results in 35% reduction in runtimes. Under various synthetic straggler patterns, the improvement is much larger, up to an order of magnitude.

References

[1] Apache Hadoop. `http://hadoop.apache.org/`.

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

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

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

[5] 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.

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

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

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

[9] 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.

[10] 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.

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

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

[13] J. Bradley, A. Kyrola, D. Bickson, and C. Guestrin. Parallel coordinate descent for l1-regularized loss minimization. In *Proceedings of the 28th International Conference on Machine Learning*, ICML'11, pages 321–328. ACM, 2011.

[14] S. Brin and L. Page. The anatomy of a large-scale hypertextual Web search engine. *Computer Networks*, 1998.

[15] 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.

[16] 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.

[17] 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.

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

[19] 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.

[20] W. Dai, J. Wei, X. Zheng, J. K. Kim, S. Lee, J. Yin, Q. Ho, and E. P. Xing. Petuum: A framework for iterative-convergent distributed ml. *arXiv preprint arXiv:1312.7651*, 2013.

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

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

[23] 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.

[24] 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.

[25] 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.

[26] 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.

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

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

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

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

[31] 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.

[32] 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.

[33] Intel. Intel Cilk Plus. `https://www.cilkplus.org/`.

[34] Intel. Intel Threading Building Blocks. `https://www.threadingbuildingblocks.org/`.

[35] V. Kazempour, A. Fedorova, and P. Alagheband. Performance implications of cache affinity on multicore processors. In *Euro-Par 2008–Parallel Processing*, pages 151–161. Springer, 2008.

[36] 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.

[37] 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.

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

[39] 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.

[40] 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.

[41] 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.

[42] 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.

[43] 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.

[44] 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.

[45] 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.

[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] M. S. Squillante and E. D. Lazowska. Using processor-cache affinity information in shared-memory multiprocessor scheduling. *Parallel and Distributed Systems, IEEE Transactions on*, 4(2):131–143, 1993.

[49] 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.

[50] J. Torrellas, A. Tucker, and A. Gupta. Benefits of cache-affinity scheduling in shared-memory multiprocessors: A summary. In *ACM SIGMETRICS Performance Evaluation Review*, volume 21, pages 272–274. ACM, 1993.

[51] 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.

[52] M. Zaharia, A. Konwinski, A. D. Joseph, R. H. Katz, and I. Stoica. Improving mapreduce performance in heterogeneous environments. In *OSDI*, volume 8, page 7, 2008.