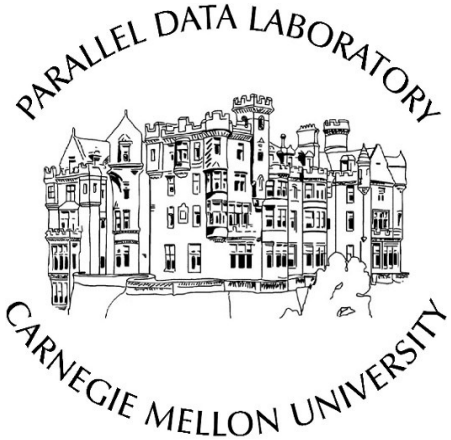

Litz: An Elastic Framework for High-Performance Distributed Machine Learning

Aurick Qiao[*,†], Abutalib Aghayev[†],
Weiren Yu[*], Haoyang Chen[*], Qirong Ho[*], Garth A. Gibson[†], and Eric P. Xing[*,†]

[*]Petuum, Inc.
[†]Carnegie Mellon University

CMU-PDL-17-103
June 2017

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

Abstract

Machine Learning (ML) is becoming an increasingly popular application in the cloud and data-centers, inspiring a growing number of distributed frameworks optimized for it. These frameworks leverage the specific properties of ML algorithms to achieve orders of magnitude performance improvements over generic data processing frameworks like Hadoop or Spark. However, they also tend to be static, unable to elastically adapt to the changing resource availability that is characteristic of the multi-tenant environments in which they run. Furthermore, the programming models provided by these frameworks tend to be restrictive, narrowing their applicability even within the sphere of ML workloads.

Motivated by these trends, we present Litz, a distributed ML framework that achieves both elasticity and generality without giving up the performance of more specialized frameworks. Litz uses a programming model based on scheduling micro-tasks with parameter server access which enables applications to implement key distributed ML techniques that have recently been introduced. Furthermore, we believe that the union of ML and elasticity presents new opportunities for job scheduling due to dynamic resource usage of ML algorithms. We give examples of ML properties which give rise to such resource usage patterns and suggest ways to exploit them to improve resource utilization in multi-tenant environments.

To evaluate Litz, we implement two popular ML applications that vary dramatically terms of their structure and run-time behavior—they are typically implemented by different ML frameworks tuned for each. We show that Litz achieves competitive performance with the state of the art while providing low-overhead elasticity and exposing the underlying dynamic resource usage of ML applications.

Acknowledgements: This research is supported in part by the National Science Foundation under awards IIS-1447676 (Big Data), CCF-1629559 (XPS Parallel), IIS-1563887, IIS-1617583, NSF CNS-1042537, and NSF CNS-1042543 (PRObE, www.nmc-probe.org). We thank the member companies of the PDL Consortium (including Broadcom, Citadel, Dell/EMC, Google, Hewlett-Packard, Hitachi, Intel Corporation, Microsoft Research, MongoDB, NetApp, Oracle, Samsung, Seagate, Tintri, Two Sigma, Toshiba, Veritas, Western Digital) for their interest, insights, feedback, and support.

Keywords: distributed systems, machine learning, parameter server, elasticity

1 Introduction

Modern clouds and data-centers are multi-tenant environments in which the set of running jobs and available resources (CPU, memory, etc.) at any given time are constantly changing [5, 44, 25]. At the same time, Machine Learning (ML) is quickly becoming a dominant application among modern distributed computing workloads. It is therefore highly desirable for ML applications executing in such an environment to be *elastic*, being able to opportunistically use additional resources when offered, and gracefully release acquired resources when requested. Elasticity is beneficial for both the individual job and for the cluster as a whole. An elastic job can make use of idle resources to complete within a shorter amount of time, and still make progress when some of its resources are removed. A cluster-wide job scheduler can dynamically re-allocate resources to speed up urgent real-time or interactive jobs, and ensure fairness by preventing jobs from holding highly contested resources for long periods of time.

Recent advancements in distributed ML frameworks, such as GraphLab [39], Petuum [51], Adam [11], Nomad [54] and various parameter servers [35] have improved the performance of distributed ML applications by an order of magnitude or more over general-purpose frameworks. They exploit unique properties of ML algorithms not always found in conventional data-processing applications, such as bounded staleness tolerance [26], uneven convergence [31], serial and parallel dependency structures in the ML model [31, 54], opportunities for bandwidth management and network message re-prioritization [49], and network message compression [50, 11]. However, there has not been as much focus on supporting elasticity, limiting the usefulness of ML frameworks in real-world computing environments. In some cases, these frameworks do not support elasticity at all [49, 31, 11], and in other cases require low-level programming that places the burden of ensuring correct execution under elasticity onto the application developer [35]. In addition, many ML frameworks require their applications to be developed with a restrictive programming abstraction upon which their optimizations can be applied, limiting their generality even within the ML domain.

On the other hand, general-purpose distributed frameworks such as Hadoop [1] and Spark [55] are well integrated with cloud and data-center environments, and are extensively used for running large-scale data processing jobs. They offer desirable features such as elastic and fault-tolerant execution, and are designed to support a wide spectrum of conventional tasks—including SQL queries, graph computations, sorting and counting algorithms, to name a few—which are typically transaction-oriented and rely on deterministic execution. Yet, as we shall discuss in depth, implementing efficient ML applications poses different needs for framework support because they are often stochastic, iterative-convergent, robust against small operational errors, and exhibit structural dependencies. Furthermore, we believe that ML programs present new opportunities for resource elasticity that arises from their aforementioned unique properties. For example, uneven convergence causes most of an ML program's parameters to converge within a few iterations [31], meaning that CPU time can be re-allocated to newly-started ML programs, whereas staleness tolerance allows for fine-grained partitioning of network bandwidth between concurrent ML programs [49].

In short, neither the general-purpose nor the ML-specialized frameworks fully meet the elasticity, generality, and efficiency needs of ML applications. While general-purpose computing systems like Spark [55] and Hadoop [1] are certainly elastic, they are not as efficient as frameworks tuned for ML programs. Their programming models hide away the necessary details needed by existing distributed ML algorithms, such as input data partitioning, computation scheduling, and consistency of shared memory access. On the other hand, existing ML-specific systems perform a rigid one-time allocation of memory, network and CPU (or GPU) resources [49, 31, 54, 12] and hold on to the resources until the end of the run. Although there are many factors that complicate building an ML framework that is elastic, general, and efficient at the same time, we summarize the ones we believe to be the most challenging to solve, and elaborate on them in Sec. 2.

First, ML applications have a wide variety of memory access patterns. A portion of the application's mutable state may be accessed when processing each and every data entry, while other portions may be coupled with, and only accessed while processing a certain data entry. When implemented in a data-parallel fashion where the entries from a large dataset are partitioned across multiple workers, it makes sense from a performance perspective to store certain mutable state on a worker co-located with a specific data entry, and other mutable state in a separate location shared by multiple workers. Thus, any elastic ML framework hoping to efficiently support such applications should also support stateful workers, giving applications the power to decide placement for mutable state.

Second, ML applications expose a wide variety of dependency structures that are exploited by efficient distributed implementations through careful scheduling of computation. For example, Gemulla *et. al.* employs a block-partitioned scheduling strategy in matrix factorization to obtain significantly faster convergence per unit of computation [17].

Furthermore, ML applications are often robust against small errors in their execution. This property has been exploited both to design systems that are able to obtain higher performance by giving up deterministic execution and consistency of memory accesses, and to create new algorithms that remain correct under asynchronous execution.

Thus, an elastic framework that is efficient and general within the ML domain should support **stateful workers**, **model**

scheduling, and **relaxed consistency**. It should provide an expressive programming model allowing the application to define a custom scheduling strategy and to specify how the consistency of memory accesses can be relaxed under it. Finally, it should correctly execute this strategy within the specified consistency requirements, while gracefully persisting and migrating application state regardless of its placement with respect to input data.

Motivated by the need for elasticity, efficiency, and generality, we present *Litz*[1], a new elastic framework for distributed machine learning that provides a programming model supporting stateful workers, model scheduling and relaxed consistency. Litz decomposes the ML application into *micro-tasks* that are agnostic to which physical machine they execute on, allowing nodes to join and leave the active computation in a transparent manner. Micro-tasks can be scheduled according to the dependencies among them, allowing the application to perform model scheduling. Every micro-task has shared access to a parameter server, with a consistency model in which updates made by a micro-task's dependencies are always visible to the micro-task itself. The Litz run-time system intelligently caches values that can be used across different micro-tasks, automatically enabling applications using relaxed consistency models to achieve higher performance. Litz achieves the following benefits:

1. **Transparent Elasticity**: Machines can be added or removed during the execution of an application in a manner that is transparent to the application, allowing ML jobs to make use of additional resources to speed up its execution, and quickly giving up resources to be used by another job.
2. **Generality within ML**: Litz's programming model provides a clear separation between the application and system, while being expressive enough to implement key ML optimizations such as model scheduling and relaxed consistency.
3. **Efficiency in ML algorithms**: Since optimizations specific to ML algorithms can be implemented on Litz, it executes them much faster than general-purpose elastic distributed systems and is comparable with other non-elastic frameworks that are specialized to particular ML optimization techniques.

The rest of this paper is organized as follows. In Section 2, we review ML algorithm properties and opportunities for elasticity, while Section 3 describes the Litz design and optimizations. In Section 4, we evaluate the effectiveness of Litz's optimizations in the distributed elastic setting, as well as its performance versus two other ML frameworks that are specialized to certain ML optimization techniques. Section 5 reviews related work, and Section 6 concludes the paper with a discussion towards future work.

2 Background

While ML algorithms come in many forms (e.g. matrix factorization, topic models, factorization machines, deep neural networks), nearly all of them share the following commonalities: (1) they possess a loss or objective function $\mathcal{L}(A,\mathcal{D})$, defined over a vector (or matrix) of model parameters A and collection of input data $\mathcal{D}$, and which measures how well the model parameters A fit the data $\mathcal{D}$; (2) their goal is to find a value of A that maximizes (or alternatively, minimizes) the objective $\mathcal{L}(A,\mathcal{D})$, via an *iterative-convergent* procedure that repeatedly executes a set of update equations, which gradually move A towards an optimal value (i.e. hill-climbing). These update equations follow the generic form

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

where $A^{(t)}$ is the vector (or matrix) of model parameters at iteration t, and $\Delta()$ is a function that computes updates to A using the previous value $A^{(t-1)}$ and the input data $\mathcal{D}$. The remainder of this section provides detailed background on specific properties of ML programs, and then presents two popular ML applications (Multinomial Logistic Regression and Latent Dirichlet Allocation) which we shall use as examples throughout this paper and as the subjects of our evaluation.

2.1 Data-parallelism and Parameter Server

Arising from the iid (independent and identically distributed) assumption on input data, the update function Δ can often be decomposed as

$$\Delta(A,\mathcal{D}) = \sum_{i=1}^{P} \Delta_i(A,\mathcal{D}_i), \tag{2}$$

where $\mathcal{D}_1,...,\mathcal{D}_P$ partition the input data $\mathcal{D}$ and each Δ_i computes a partial update using $\mathcal{D}_i$ which, when aggregated, form the final update Δ. This allows each update to be calculated in a data-parallel fashion with input data and update calculations distributed across a cluster of workers.

Parameter Server: Eq. 2 shows that the model parameters A are used by the calculations of every partial update Δ_i. In a data-parallel setting it is natural to place the model parameters in a shared location accessible by every machine, known as

[1]Meant to evoke the strings of a harp, sounding out as many or as few. Litz is short for "Wurlitzer", a well-known harp maker.

a *parameter server*. Typically, implementations of this architecture consists of two types of nodes: 1) worker nodes which partition the input data and calculate partial updates and 2) parameter server nodes which partition the model parameters and aggregate/apply the partial updates sent by worker nodes. The parameter server architecture has proven to be a near-essential component of efficient distributed ML and is used in numerous applications and frameworks [49, 15, 37, 27].

Additionally, as we shall demonstrate, the parameter server architecture plays a role in exposing the inherent dynamic resource usage of ML applications. Specifically, as explained in detail in Sec. 2.4, model parameters can become more sparse (ie. mostly zeros) as run-time increases, resulting in decreasing memory usage when using a sparse in-memory representation. By separating the placement of model parameters from input data, one isolates the portion of application state that exhibits this behavior and is able to adjust its resource allocation in a fine-grained manner.

Stateful Workers: Even though the model term A appears in the calculations of each partial update, not all of it is necessarily used. In particular, there may be parts of the model which are only used when processing a single partition $\mathcal{D}_i$ of the input data. A large class of examples includes non-parametric models, whose model structures are not fixed but instead depends on the input data itself, typically resulting in model parameters being associated with each entry in the input data. In such applications, it is preferable to co-locate parts of the model on worker nodes with a particular partition of input data so they can be accessed and updated locally rather than across a network. This optimization is especially essential when the input data is large and accesses to such associated model parameters far outnumber accesses to shared model parameters. It also means that workers are *stateful*, and an elastic ML system that supports this optimization needs to preserve worker state during elastic resource adjustments.

2.2 Error Tolerance and Relaxed Consistency

ML algorithms have several well-established and unique properties, including *error-tolerance*: even if a perturbation or noise ϵ is added to the model parameters in every iteration, i.e. $A^{(t)} = A^{(t-1)} + \Delta(A^{(t-1)}, \mathcal{D}) + \epsilon$, the ML algorithm will still converge correctly provided that ϵ is limited or bounded.

Bounded Staleness Consistency: An important application of error tolerance is bounded staleness consistency models [26, 14, 9], which allow stale model parameters to be used in update computations, i.e. $A^{(t)} = A^{(t-1)} + \Delta(A^{(t-s)}, \mathcal{D})$, where $1 \leq s \leq k$ for small values of k. ML algorithms that use such consistency models are able to (1) execute in a partially asynchronous manner without sacrificing correctness, thus mitigating the effect of stragglers or slow workers [13, 23]; and (2) reduce the effect of network bottlenecks caused by synchronization by allowing cached parameter values to be used.

Staleness-aware ML Algorithms: Beyond simply applying bounded staleness consistency to existing algorithms, the ML community has developed new staleness-aware algorithms [40, 56, 53, 8, 28, 7, 36] which modify each update $\Delta()$ according to the staleness s that it experiences. The modifications usually take the form of a scaling factor $\Delta() \leftarrow c\Delta()$, which are computationally light-weight and do not create new bottlenecks. In the presence of staleness, these algorithms converge up to an order of magnitude faster than their non-staleness-aware counterparts.

Example ML Framework (Bösen): Bösen [49] is a recent framework that relaxes the consistency of access to shared parameters stored on a parameter server to achieve higher throughput and faster convergence for error-tolerant and staleness-aware ML algorithms. It implements the Stale Synchronous Parallel (SSP) consistency model [26] in which the distributed computation proceeds in a series of iterations, and stale parameter values may be used for up to a constant number of iterations that pass. Although Bösen successfully supports relaxed consistency, it restricts applications to the SSP mode of execution, limiting its support for model scheduling and the important class of dependency-aware algorithms (Sec. 2.3).

2.3 Dependency Structures and Model Scheduling

Another key property of ML algorithms is the presence of implicit *dependency structures*: supposing A_1 and A_2 are different elements of A, then updating A_1 before A_2 does not necessarily yield the same result as updating A_2 before A_1; whether this happens or not depends on the algebraic form of $\mathcal{L}()$ and $\Delta()$. As a consequence, the convergence rate and thus the running time of ML algorithms can be greatly improved through careful scheduling of parallel model parameter updates.

Dependency-aware ML Algorithms: Like the many existing staleness-aware algorithms that exploit error tolerance, there is a rich set of algorithms that use dependency structures in their models to perform better scheduling of updates [43, 53, 18, 15, 34, 48, 39]. A typical example is to partition the model into subsets, where the parameters inside a subset must be updated sequentially, but multiple subsets can be updated in parallel. Two parameters A_1 and A_2 are placed into the same subset if the strength of their dependency exceeds a threshold $\mathrm{dep}(A_1, A_2) > \epsilon$. As with staleness-aware algorithms, dependency-aware algorithms converge up to an order of magnitude faster than their non-dependency-aware counterparts.

Example ML Framework (STRADS): STRADS [31] is a recent framework that provides an interface and system architecture for model scheduling, enabling the implementation of dependency-aware algorithms. A STRADS application repeats the

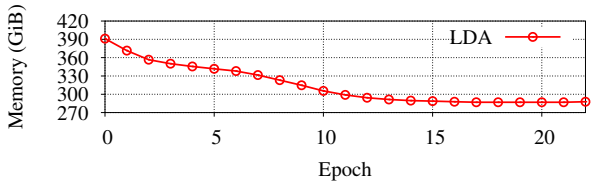

Figure 1: Aggregate memory usage on a cluster of 12 nodes during runtime of Latent Dirichlet Allocation (LDA) application (Sec. 2.6) implemented with Litz.

following until convergence: (1) partition the parameters into subsets obeying the aforementioned rules, (2) calculate partial updates in parallel according to the partitioning, and (3) collect the partial updates and apply them to the parameters. Although STRADS introduces staleness in a limited way via pipelining, it does not handle asynchronous updates to parameters, limiting its support for staleness-aware algorithms like AdaptiveRevision [41] which are designed to execute fully asynchronously.

2.4 Dynamic Resource Usage

The iterative-convergent nature of ML algorithms along with its aforementioned properties present opportunities for resource allocation not usually found in other computing tasks and open up new opportunities for multi-tenancy among concurrent ML jobs. In particular, ML programs may consume less resources as they converge to an optimal value of the model parameters A, which suggests that computing resources may be relinquished from long-running ML jobs to be spent elsewhere for better utility. We present several examples of such run-time dependent resource variability, and leave the leveraging of this phenomenon for efficient scheduling in multi-tenant clusters for future work.

Sparsity of Model Parameters: Certain ML algorithms may find their model parameters becoming sparse (mostly zeros) as they approach convergence [31], permitting the application to use a more memory-efficient storage format (e.g. sparse vs. dense matrix) to reduce memory consumption — thus freeing up memory for new ML programs. For example, Fig. 1 shows the memory usage of a popular ML application—Latent Dirichlet Allocation (Sec. 2.6)— running on a 12 node cluster. It starts with over 390 GiB of aggregate RAM and drops by 23% to about 300 GiB within 10 epochs (passes over the input data), freeing about 90GiB that can be allocated to another job.

Non-uniform Convergence: Furthermore, model parameters may converge to their optimal values non-uniformly. For example, GraphLab showed that majority of the parameters converge in a single update in their PageRank experiments [39]; likewise, STRADS reported that over 90% of the parameters converge after ≤ 5 iterations in their Lasso experiments [31]. Current ML frameworks make use of this property to re-prioritize a fixed amount of CPU cycles onto slower-converging parameters. Yet, looking at it another way, non-uniform convergence suggests new opportunities for CPU elasticity; since computation for updating already-converged parameters can be executed less frequently (or not at all), the saved CPU cycles may be allocated to another job.

Bounded Staleness Execution: A third opportunity comes from bounded staleness consistency models [26, 49], which allows ML programs to trade off network usage for convergence speed via a tunable staleness parameter — in particular, Ho *et. al.* [26] showed that the trade off between network usage and convergence speed is non-linear and subject to diminishing returns. This second point is important for the multi-tenant setting because it implies that network allocation between different ML jobs is not a zero-sum game; rather, it is possible to intelligently allocate bandwidth to each ML job using a strategy that jointly optimizes the completion times for multiple jobs at once.

2.5 Example: Multinomial Logistic Regression (MLR)

Multinomial Logistic Regression (MLR) [32] is a multi-label classifier that is effective for large-scale text classification [38], and image classification [33]. Given training data samples with D-dimensional feature vectors $x^{(1)},...,x^{(N)}$ with corresponding labels $y^{(1)},...,y^{(N)}$ belonging to K classes, MLR learns K D-dimensional weight vectors $w^{(1)},...,w^{(K)}$ so that the predicted probability that an unlabeled data sample x belongs to class k is proportional to $\exp(w^{(k)} \cdot x)$.

In the distributed setting, MLR is commonly trained by minimizing its cross-entropy loss function using a data-parallel stochastic gradient descent (SGD) algorithm, which is error-tolerant and theoretically proven to remain correct under bounded staleness [26]. Therefore, the MLR application using SGD is a natural fit for a framework like Bösen, which employs the SSP consistency model to increase throughput while ensuring correctness. Each worker stores a subset of input data, and at each iteration processes a minibatch from the subset, computes a gradient, and updates the model parameters in the parameter server.

Method Name	Part Of	Defined By	Description
`DispatchInitialTasks()`	Driver	Application	Invoked by the framework upon start-up to dispatch the first set of micro-tasks.
`HandleTaskCompletion(result)`	Driver	Application	Invoked by the framework when a micro-task completes so that the driver can dispatch a new set of micro-tasks.
`DispatchTask(executor,args)`	Driver	Framework	Invoked by the application to dispatch a micro-task to the specified executor.
`RunTask(args)`	Executor	Application	Invoked by the framework to perform a micro-task on the executor.
`SignalTaskCompletion(result)`	Executor	Framework	Invoked by the application to indicate the completion of a micro-task.
`PSGet(key)`	Executor	Framework	Returns a specified value in the parameter server.
`PSUpdate(key,update)`	Executor	Framework	Applies an incremental update to a specified value in the parameter server.

Table 1: The programming interface for Litz, an application should define `DispatchInitialTasks` and `HandleTaskCompletion` on the driver, as well as `RunTask` on the executor.

2.6 Example: Latent Dirichlet Allocation (LDA)

Latent Dirichlet Allocation (LDA) is a widely-used Bayesian probabilistic model for topic modeling (clustering documents into different topics) [10] that is commonly trained using the popular Gibbs sampling algorithm [22]. Assuming there are D documents, K topics, V distinct words across all documents, and letting w_{di} denote the i-th word in document d, three sets of parameters are trained: (1) U, a $D \times K$ "document-topic" matrix in which U_{dk} counts the number of words in document d that are assigned to topic k, (2) W, a $V \times K$ "word-topic" matrix in which W_{vk} counts the number of times word v is assigned to topic k across all documents, and lastly (3) z_{di}, the topic assigned to each w_{di}. The algorithm repeatedly sweeps over all z_{di}, assigning each a new value randomly sampled from a distribution computed using the d-th row of U and the w_{di}-th row of W. The matrices U and W are updated to reflect this change after each new value is assigned.

In the distributed setting, processing each z_{di} is typically performed in parallel, but if done naively can hurt convergence due to the dependency structures inherent in the LDA model. In particular, processing $z_{d_1 i_1}$ and $z_{d_2 i_2}$ in parallel will concurrently modify the same row of U if $d_1 = d_2$, or the same row of W if $i_1 = i_2$. Therefore, LDA is a natural fit for a framework like STRADS, which employs a block-partitioned schedule that eliminates such write conflicts. The rows of W are divided into P blocks, each assigned to a different worker. Each worker sequentially processes the z_{di} corresponding to its local documents and currently assigned block of W. The block assignments are rotated P times so that each worker updates all of W.

Additionally, each row of U and z correspond to a particular document in the input data, and is only accessed when processing that document. They are examples of parameters which are best co-located with the input data, especially considering that z has the same size as the input data, and can be prohibitively expensive to retrieve over the network during each iteration. Therefore, LDA is an example of an application that requires stateful workers to achieve efficient distributed execution.

3 Litz Architecture and Implementation

Motivated by the ML-specific opportunities in elastic resource management, as well as the need for a framework that elastically and efficiently supports the wide array of algorithmic techniques seen in distributed ML, we turn our attention to Litz—a framework which (1) provides an expressive programming model supporting stateful workers as well as both staleness- and dependency-aware algorithms, and (2) efficiently executes the aforementioned programming model in an elastic fashion, being able to scale in or out according to changing resource demand with low overhead. We discuss Litz's programming model and execution system separately in order to clearly distinguish between the tools it provides to application developers and the system techniques it uses to optimize execution.

3.1 Programming Model and API

The main goal and challenge of designing Litz's programming model is striking a balance between being expressive enough to support the wide variety of proven algorithmic techniques in distributed ML, while exposing enough structure in the application that the underlying execution system can optimize. Guided by the insights presented in Sec. 2, we now turn to Litz's programming model and interface. In particular, we describe how it naturally arises from the aforementioned properties

of ML applications, and how it enables an efficient and elastic run-time implementation. A detailed summary of Litz's API can be found in Table 1.

Input Data Partitioning Across Executors: Eq. 2 shows that the input data and update calculations of ML applications can be partitioned and distributed across a number of workers, but it does not specify any particular partitioning scheme, nor does it require the number of partitions to be equal to the number of physical machines. Instead of making input data assignments directly to physical machines, Litz first distributes it across a set of logical *executors*, which are in turn mapped to physical machines. This separation enables elasticity by allocating more executors than physical cores and migrate executor's state and input data to other cores as they become available. It also lets Litz support stateful workers by allowing executor state to be defined and mutated by the application and treated as a black box by the run-time system.

Update Calculations and Parameter Server: Update calculations are decomposed into short-lived (typically shorter than 1 second) units of computation called *micro-tasks*, each of which calculates a partial update using the input data on a single executor. During its execution, a micro-task is granted read/update access to a global parameter server via a key-value interface (`PSGet` and `PSUpdate` in Table 1) and applies partial updates to the model parameters by modifying application-defined state in the executor and/or updating globally-shared values in the parameter server.

Model Scheduling and Bounded Staleness: Litz enables both model scheduling and bounded staleness by letting the application specify dependencies between micro-tasks. If micro-task A is a dependency of micro-task B, then (1) B is executed before A and (2) B sees all updates made by A. This strict ordering and consistency guarantee lets the application perform model scheduling by defining an ordering for when certain updates are calculated and applied. On the other hand, if neither A nor B is a dependency for the other, then they may be executed in any order or in parallel, and may see none, some, or all of the updates made by the other. This critical piece of non-determinism lets the application exploit the error-tolerance property of ML by allowing the run-time system to cache and use stale values from the parameter server between independent micro-tasks.

Micro-Task Dispatch and Inform: A generic way to specify dependencies between micro-tasks is through a dependency graph, a directed graph in which each vertex corresponds to a micro-task, and an arc from vertex A to vertex B means micro-task A is a dependency for micro-task B. Due to the potential existence of a large number of micro-tasks, however, explicitly specifying such a graph may incur a significant amount of overhead. Instead, a Litz application implicitly specifies dependencies by dynamically dispatching a micro-task whenever its dependencies are satisfied during run-time. The application defines a *driver* which is responsible for dispatching micro-tasks via the `DispatchTask` API method. Whenever a micro-task completes, the framework informs the application by invoking the `HandleTaskCompletion` method on the driver, which can then dispatch any additional micro-tasks. Upon start, the framework invokes the `DispatchInitialTasks` method on the driver so the application can dispatch an initial set of micro-tasks that do not have any dependencies. With respect to its dispatch/inform API, the consistency model Litz guarantees to the application is as follows:

- If the driver dispatches micro-task B *after* being informed of the completion of micro-task A, then Litz assumes that A is a dependency for B. In this case, B will see all updates made by A.
- If the driver dispatches micro-task B *before* being informed of the completion of micro-task A, then Litz assumes that A is not a dependency for B. In this case, B may see none, some, or all of the updates from A.

In other words, Litz lets the application dynamically dispatch micro-tasks, *infers* dependencies between them from the sequence of `DispatchTask` and `HandleTaskCompletion` calls, and enforces its consistency model based on those inferred dependencies. We discuss how this consistency model, along with a corresponding cache coherence protocol, can be implemented efficiently in Sec. 3.2.

3.2 Implementation and Optimizations

Litz is implemented in approximately 6500 lines of C++ code using the ZeroMQ [6] library for low latency communication and Boost's Coroutine2 [2] library for low overhead context-switching between micro-tasks. The run-time system is comprised of a single *master thread* along with a collection of *worker threads* and *server threads*, as shown in Fig. 2. The application's driver exists in the master thread and its executors exist in the worker threads. The key/value pairs comprising the parameter server are distributed across a set of logical *PSshards* stored in the server threads. Additional worker and server threads may join at any time during the computation, and the run-time system can re-distribute its load to make use of them. They may also gracefully leave the computation after signaling to the master thread and allowing their load to be transferred to other threads.

The master thread coordinates the execution of the application. First, it obtains micro-tasks from the driver by initially invoking the `DispatchInitialTasks` and then continuously calling `HandleTaskCompletion` methods, sending them to worker threads to be executed. Second, the master thread maintains the dynamic mappings between executors and

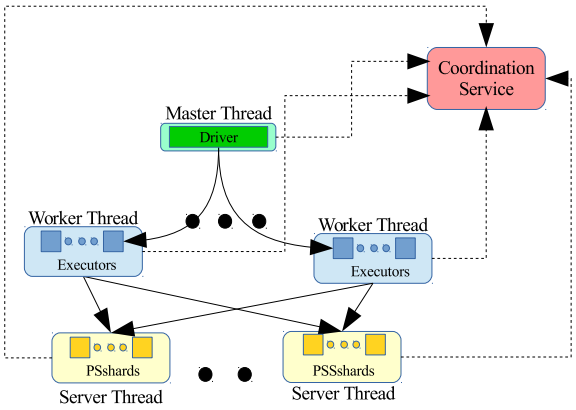

Figure 2: High-level architecture of Litz. The driver in the master thread dispatches micro-tasks to be performed by executors on the worker threads. Executors can read and update the global model parameters distributed across PSshards on the server threads.

worker threads, as well as between PSshards and server threads. When worker or server threads join or leave the computation, it initiates load re-distribution by sending commands to move executors between worker threads or PSshards between server threads. Third, the master thread periodically triggers a consistent checkpoint to be taken of the entire application state, and automatically restores it when a failure is detected. Each thread registers with an external coordination service such as ZooKeeper [29] or etcd [4] in order to determine cluster membership and detect failures. In order to transfer and checkpoint the driver and executors, Litz requires the application to provide serialization and de-serialization code. The programming burden on the developer is low since (1) this code does not actively participate in elasticity and checkpointing, but simply invoked by the run-time system whenever needed, and (2) various third-party libraries can be used to reduce programming overhead, such as Boost Serialization [3].

Worker Thread Elasticity: Each worker thread maintains the state of and runs the micro-tasks for a subset of all executors. After any worker threads join the active computation, executors are moved to them from the existing worker threads (scaling out). Similarly, before any worker threads leave the active computation, executors are moved from them to the remaining worker threads (scaling in). Currently Litz aims to have roughly the same number of executors residing on each worker thread, but can be modified to support load balancing using other measures of load.

When an executor needs to be moved, the master thread first sends a command to its worker thread instructing it to suspend execution of micro-tasks for that executor. After receiving the command, the worker thread finishes any ongoing micro-tasks for that executor while buffering any pending micro-tasks dispatched by the driver that have not yet started. It then sends the executor's application state and its queue of buffered micro-tasks over the network to the receiving worker thread.

The transfer of the executor's input data is treated differently in the scale-in and scale-out cases. When scaling in, Litz aims to free the requested resources as quickly as possible. The input data is discarded on the originating worker thread to avoid incurring extra network transfer time, and re-loaded on the target worker thread. When scaling out, Litz aims to make use of the new worker thread as quickly as possible. The input data is sent directly from the memory of the originating worker thread to avoid incurring extra disk read time on the target worker thread.

Parameter Server Elasticity: Similar to worker threads and executors, each server thread stores and handles the requests and updates for a subset of all PSshards, which are re-distributed before scaling in and after scaling out. However, since requests and updates are continuously being sent to each PSshard and can originate from any executor, their transfer requires a special care. In particular, a worker thread may send requests or updates to a server thread that no longer contains the target PSshard, which can occur if the PSshard has been moved but the worker thread has not yet been notified. A naive solution to this problem is to suspend micro-tasks on every executor, then perform the transfer, notify all worker threads of the change, and finally resume execution. This method guarantees that worker threads always send requests and updates to server threads that contain the target PSshard, but requires suspending the execution of the entire system. Instead, the server threads perform *request and update forwarding* amongst each other. Whenever a server thread receives a request for or update to a value on a PSshard it no longer contains, it forwards the message to the server thread it last transferred the PSshard to. If the PSshard

is currently being transferred away, the server thread buffers the requests and/or updates and forwards them after the transfer is completed. This can happen multiple times until the target PSshard is found, the request/update is performed, and the response is sent back to the originating worker thread. The actual coordination of the transfer is handled between the master thread and the server threads independent of the worker threads until they are notified of the transfer. This way, execution of micro-tasks can proceed uninterrupted during parameter server scaling events.

Consistent Checkpoint and Recovery: To achieve fault tolerance, Litz is able to periodically create and save a consistent checkpoint of the application's entire execution state. When a checkpoint is triggered, the master thread suspends the execution of the application by waiting for all the executors to finish their current micro-tasks, and buffer any further micro-tasks. A checkpoint will then be taken, writing to persistent storage (1) the state of the driver, (2) the buffered micro-tasks for each executor, (3) the state of each executor, and (4) the key-value pairs stored in each PSshard. Input data is not saved, but is re-loaded again during the recovery process. When a failure is detected through the external coordination service, Litz triggers an automatic recovery from the latest checkpoint. The saved driver, executors, buffered micro-tasks, and parameter server values are loaded from persistent storage, after which normal execution is resumed.

Parameter Cache Synchronization: The consistency model outlined in Sec. 3.1 exposes an opportunity for the run-time system to optimize execution by caching and re-using values from the parameter server instead of retrieving them over the network for each access. Specifically, a micro-task A is allowed to use a cached parameter if its value reflects all updates made by all micro-tasks that A depends on. This means that (1) multiple accesses of the same parameter by micro-task A can use the same cached value and (2) a micro-task B whose dependencies are a subset of A's can use the same cached values that were used by A. The following discussion focuses on supporting (2) since (1) is just the specific case when $A = B$, thus the same mechanism that supports (2) will work for both cases.

Suppose a cached parameter value was retrieved by micro-task A. In order to determine if it can be re-used by micro-task B, a method is needed to quickly check if the dependencies of B are a subset of the dependencies of A. In the general case when dependencies are explicitly specified, performing this check for each access of a parameter value can incur a significant overhead. However, by only using the *sequence* of `DispatchTask` and `HandleTaskCompletion` calls to infer dependencies, Litz effectively reduces the number of possible combinations of micro-tasks that can occur as dependencies. When a micro-task is dispatched, *all* other micro-tasks whose completion the driver has been informed of are considered to be dependencies. Thus, the dependencies of micro-task B are a subset of the dependencies of micro-task A if the total number of `HandleTaskCompletion` calls made when B was dispatched is at most the total number of `HandleTaskCompletion` calls made when A was dispatched.

This cache coherence protocol can be implemented with low overhead. The master thread maintains a single logical clock that is incremented each time `HandleTaskCompletion` is invoked. When the driver dispatches a micro-task by invoking `DispatchTask`, the master thread tags the micro-task with the clock at that time, which is called its *parent clock*. After micro-task A retrieves a fresh value from the parameter server, it caches the value tagged with its parent clock. When micro-task B wants to access the same parameter, it first checks if its parent clock is less than or equal to the clock tagged to the cached value. If so, then the cached value is used; otherwise a fresh copy of the parameter is retrieved from the parameter server and tagged with B's parent clock. A cache exists on each Litz process running at least one worker thread, so that it can be shared between different worker threads in the same process.

This cache coherence protocol allows Litz to automatically take advantage of parameter caching for applications that use bounded staleness. For example, in SSP (Sec. 2.2) with staleness s, all micro-tasks for iteration i are dispatched when the last micro-task for iteration $i-s-1$ is completed. Thus, every micro-task for the same iteration has the same parent clock and share cached parameter values with each other. Since the micro-tasks for iteration i are dispatched before the those for iterations between $i-s$ and $i-1$ finish (when $s \geq 1$), the values they retrieve from the parameter server may not reflect all updates made in those prior iterations, allowing staleness in the parameter values being accessed.

Parameter Update Aggregation: Updates for the same parameter value may be generated many times by the same micro-task, and by many different micro-tasks. Since the parameter updates in ML applications are incremental and almost always additive, they can be aggregated locally before sending to the parameter server in order to reduce network usage. To facilitate the aggregation of updates, each Litz process that runs at least one worker thread also contains an *update log* which is stored as a mapping from parameter keys to aggregated updates. Whenever a micro-task updates a parameter value by invoking `PSUpdate`, the given update is aggregated into the corresponding entry in the update log, or is inserted into the update log if the corresponding entry does not exist. Therefore, an update sent to the parameter server can be a combination of many updates generated by different micro-tasks on the same Litz process.

In order to maximize the number of updates that are locally aggregated before sending them over the network, the results of micro-tasks are not immediately returned to the master thread after they are completed. Instead, when a micro-task completes, its updates remain in the update log and the result of the micro-task is buffered to be returned to the master thread

at a later time. Doing this allows the updates from multiple micro-tasks to be sent in aggregated form to the server threads, reducing total network usage. The update log is periodically flushed by sending all updates it contains to the server threads to be applied. After each flush, all buffered micro-task results are returned to the master thread, which then informs the driver of their completion. The period of flushing is a parameter that can be carefully tuned, but is not a main subject of study in Litz. We find that the simple strategy of flushing only when all micro-tasks on a worker thread are finished works well in practice.
Co-operative Multitasking: To efficiently execute many micro-tasks on each worker thread, Litz employs co-operative multitasking implemented with a co-routine library [2]. When one task is blocked in an invocation of `PSGet` waiting for a value to be returned from a server thread, the worker thread will switch to executing another micro-task that is not blocked so that useful work is still performed. Each micro-task is executed within a co-routine so that switching between them can be done with low-latency, entirely in user-space. Using co-routines provides the benefit of overlapping communication with computation, while retaining a simple-to-use, synchronous interface for accessing the parameter server from micro-tasks.

4 Evaluation

To evaluate Litz, we start by showing its high performance when executing diverse applications including both data- and model-parallel workloads. We implement MLR in the data-parallel fashion described in Sec. 2.5 using SGD running under SSP, and demonstrate superior performance to the built-in MLR application in Bösen [49], which is specialized to such data-parallel SSP workloads. We also implement LDA in the model-parallel fashion described in Sec. 2.6 using Gibbs sampling running under the block-partitioned schedule, and demonstrate superior performance to the built-in LDA application in STRADS [31], which is specialized to such model-parallel workloads.

We then evaluate the Litz's elasticity mechanism and demonstrate its efficacy along several directions. First, with its parameter caching, update aggregation, and co-operative multi-tasking, Litz is able to sustain increasing numbers of executors and micro-tasks with insignificant performance impact. Second, a running Litz application is able to efficiently make use of additional nodes allocated to it, accelerating its rate of completion. Lastly, a running Litz application is able to release its nodes on request, quickly freeing them to be re-allocated by an external resource scheduler, while continuing to make progress.

Finally, as an instance of resource variability in ML applications, we demonstrate how the memory usage of an LDA application varies during the runtime, and how the architecture of Litz combined with its elasticity can leverage this phenomenon for a more efficient scheduling.
Cluster Setup: Unless otherwise mentioned, the experiments described in this section are conducted on nodes with the following specifications: 16 cores with 2 hardware threads each (Intel Xeon E5-2698Bv3), 64GiB DDR4-2133 memory, 40GbE NIC (Mellanox MCX314A-BCCT), Ubuntu 16.04 Linux kernel 4.4. The nodes are connected with each other through a 40GbE switch (Cisco Nexus 3264-Q), and access data stored on an NFS cluster connected to the same switch. Each machine runs one Litz process which contains both worker threads and server threads; the master thread is co-located with one of these processes.
Input Datasets: Unless otherwise mentioned, we run MLR on the full ImageNet ILSVRC2012 dataset [42] consisting of 1.2M images labeled using 1000 different object categories. The dataset is pre-processed using the LLC feature extraction algorithm [47], producing 21K features for each image, resulting in a post-processed dataset size of 81GB. We run LDA on a subsample of the ClueWeb12 dataset [16] consisting of 50M English web pages. The dataset is pre-processed by removing stop words and words that rarely occur, resulting in a post-processed dataset with 10B tokens, 2M distinct words, and total size of 88GB.

4.1 MLR and LDA Performance Comparisons

We compare our Litz implementations of MLR and LDA with those that are distributed with the open-source versions of Bösen and STRADS, respectively. Since we wish to demonstrate that Litz by itself is able to efficiently support the same applications supported by these more specialized frameworks, we closely follow their implementations in Bösen and STRADS, sharing a significant portion of the core algorithm code. All three systems along with their applications are written using C++, and to further ensure fairness, we compiled all three using the `-O2 -g` flags and linked with the TCMalloc [19] memory allocator. These settings are the default for both Bösen and STRADS. The following experiments show that Litz can efficiently support multiple paradigms of distributed ML algorithms which were previously supported by specialized frameworks.
Comparison with Bösen: We compare Litz With Bösen running the MLR application on 25% of the ImageNet ILSVRC2012 dataset[2] using 8 nodes. The open-source version of Bösen differs from the system described by Wei *et. al.* [49] in that it does not implement early communication nor update prioritization, but is otherwise the same and fully supports SSP execution. Both MLR instances were configured to use the same SSP staleness bound of 2 as well as the same SGD tuning parameters

[2]With the full dataset, the Bösen baseline does not complete within a reasonable amount of time.

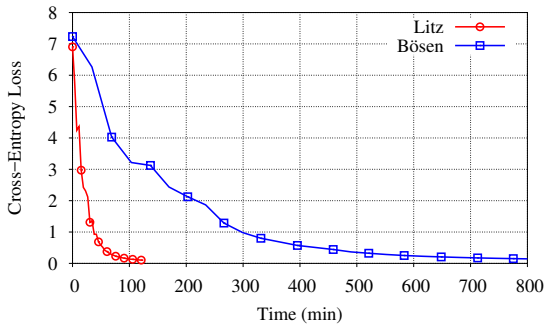

Figure 3: Multinomial Logistic Regression (MLR) running on 8 nodes using 25% of the ImageNet ILSVRC2012 dataset. Litz achieves convergence about 8× faster than Bösen.

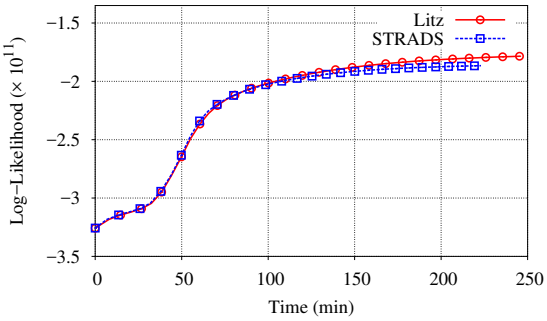

Figure 4: Latent Dirichlet Allocation (LDA) training algorithm running on STRADS and Litz with the sub-sampled ClueWeb12 dataset. Litz achieves convergence about 1.06× slower than STRADS.

such as step size and minibatch size. As Fig. 3 shows, our MLR implementation on Litz converges about 8× faster than that on Bösen. Our profiling of Bösen and cursory examination of its code shows that it does not fully utilize CPUs due to lock contention. We believe the wide gap in performance is not due to fundamental architectural reasons, and that Bösen should be able to narrow the gap on such SSP applications given a more optimized implementation.

Comparison with STRADS: We next compare Litz with STRADS running the LDA application using 12 nodes. The open-source version of STRADS is the same implementation used in Kim *et. al.* [31]. Both LDA instances were configured to use the same number of block partitions as well as the same LDA hyperparameters α and β. As Fig. 4 shows, our LDA implementation on Litz achieves similar performance, converging only 1.06× slower than that on STRADS.

4.2 Elasticity Experiments

Before discussing elastic scaling, we evaluate Litz's performance characteristics over increasing numbers of executors. The worker threads achieve elasticity by re-distributing executors amongst themselves when their numbers change, and by over-partitioning the application's state and computation across larger numbers of executors, Litz is able to scale out to larger numbers of physical cores and achieve a more balanced work assignment. Thus it is critical for Litz applications to still perform well in such configurations. We run the MLR application on 4 nodes and the LDA application on 12 nodes, varying the number of executors from 1 to 16 per worker thread. Fig. 5 shows how the throughput of each application changes when the number of executors increases. Using a single executor per worker thread as the baseline, the execution time for MLR does not noticeably change when using $4\times$ the number of executors, and gradually increases to $1.11\times$ the baseline when using $16\times$ the number of executors. For LDA, the execution time initially decreases to $0.94\times$ the baseline when using $2\times$ the number of executors, and thereafter gradually increases to $1.23\times$ the baseline when using $16\times$ the number of executors. We believe the overhead introduced by increasing the number of executors is quite acceptable and is insignificant when compared with Litz's performance improvements over Bösen and STRADS.

4.2.1 Elastic Scale Out

As jobs finish in a multi-tenant setting and previously used resources are freed up, additional allocations can be made to a currently running job. It is therefore important for the job to be capable of effectively using the additional resources to speed up its execution. In this section, we evaluate Litz's performance characteristics when scaling a running application out to a larger number of physical nodes. We run experiments scaling MLR jobs from 4 to 8 nodes, and LDA jobs from 12 to 24 nodes. The experiments for LDA in this section were performed using m4.4xlarge instances on AWS EC2, each with 16 vCPUs and 64GiB of memory.

To evaluate the speed-up achieved, we compare our scale-out experiments with static executions of the applications using both the pre-scaling number of nodes and the post-scaling number of nodes. Fig. 7 shows the convergence plots for MLR, 4 new nodes are added after ≈40min of execution. The static 4 node execution completes in ≈157min while the scale-out execution completes in ≈122min, resulting in a 22% shorter total run-time. Fig. 8 shows the convergence plots for LDA,

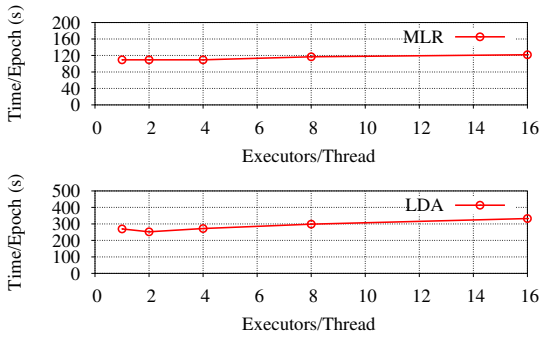

Figure 5: Average time per epoch for MLR and LDA when running with various numbers of executors per worker thread. In both cases the overhead of increasing the number of executors is insignificant. We define one epoch as performing a single pass over all input data.

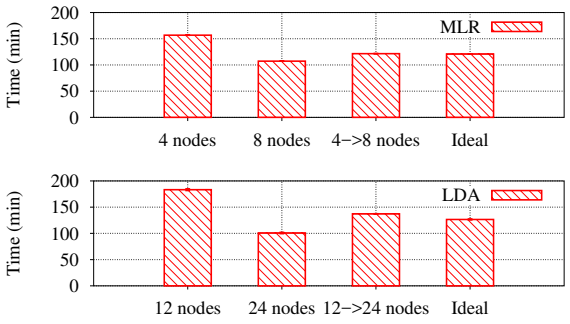

Figure 6: Static, scale-out, and ideal scale-out execution times for MLR and LDA implemented on Litz. We scale out MLR from 4 nodes to 8 nodes, and LDA from 12 nodes to 24 nodes. (Sec. 4.2.1 describes how ideal is computed.)

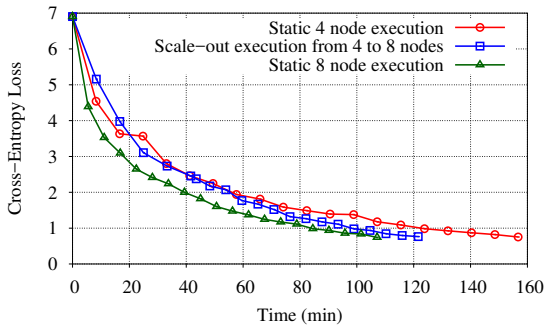

Figure 7: MLR execution on Litz with 4 nodes, with 8 nodes, and with an elastic execution that scales out from 4 nodes to 8 nodes. The nodes are added at about 40 minutes into execution. The elastic execution completes about 22% faster than the static 4 node execution.

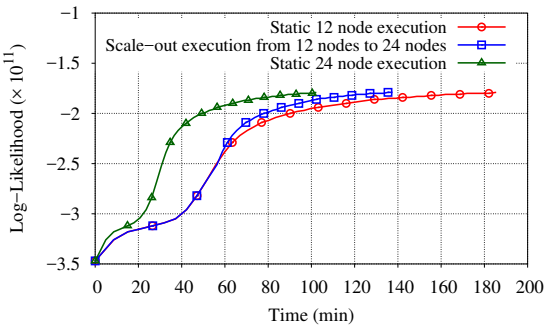

Figure 8: LDA execution on Litz with 12 nodes, with 24 nodes, and with an elastic execution that scales out from 12 nodes to 24 nodes. The nodes are added at about 55 minutes into execution. The elastic execution completes about 25% faster than the static 12 node execution.

12 new nodes are added after ≈ 55min of execution. The static 12 node execution completes in ≈ 183min while the scale-out execution completes in ≈ 137min, resulting in a 25% shorter total run-time.

Next, we evaluate the amount of room for improvement still achievable over Litz's current scale-out performance. We define and compare with a simple *ideal* scale-out execution time which intuitively measures the total run-time of a job that instantly scales out and adapts to use the additional nodes. For example, consider a job that scales out from 4 to 8 nodes after completing 30% of its iterations, its ideal scale-out execution time is the sum of the time at which the scale-out was triggered and the time it takes a static 8 node execution to run the last 70% of its iterations. Fig. 6 compares the static pre-scaling, static post-scaling, scaling, and ideal execution times for both MLR and LDA. For MLR, the static 8 node execution completes in ≈ 107min, giving an ideal scale-out execution time of ≈ 121min. The scale-up execution time is ≈ 122min, indicating a less than 1% difference from the ideal. Similarly for LDA, the static 24 node execution completes in ≈ 101min, giving an ideal scale-out execution time of ≈ 127min. The scale-up execution time is ≈ 137min, indicating a 5% difference from the ideal. LDA's higher overhead stems from the large worker state that is inherent to the algorithm, which need to be serialized and sent over the network before the transferred executors can be resumed. We believe this overhead can be reduced further through careful optimization of the serialization process, by minimizing the number of times data is copied in memory and compressing the data sent over the network.

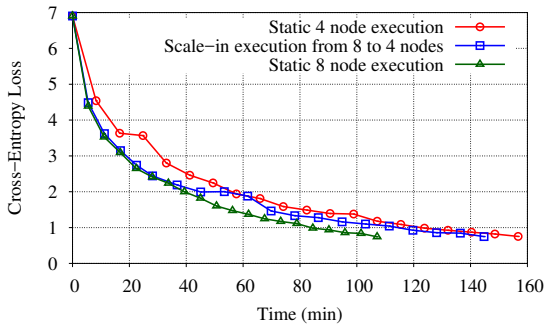

Figure 9: MLR execution on Litz with 4 nodes, with 8 nodes, and with an elastic execution that scales in from 8 nodes to 4 nodes. The nodes are removed at about 30 minutes into execution.

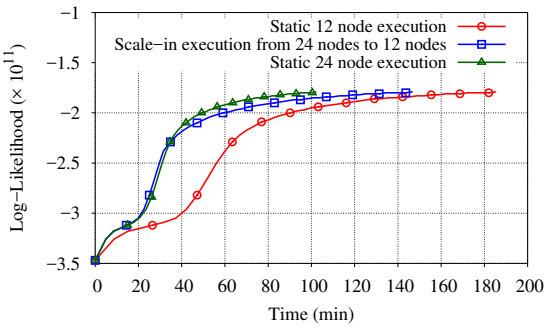

Figure 10: LDA execution on Litz with 12 nodes, with 24 nodes, and with an elastic execution that scales in from 24 nodes to 12 nodes. The nodes are removed at about 33 minutes into execution.

4.2.2 Elastic Scale In

As new and higher priority jobs are submitted in a multi-tenant environment, the resource allocation for a currently running job may be reduced and given to another job. In this section, we evaluate Litz's scale-in performance based on two key factors. First, we show that Litz applications continue to make progress after scaling in, with performance comparable to the static execution on the fewer nodes. Second, we show that running jobs can release nodes with relatively low latency, quickly transferring executors and PSshards away from requested nodes so that they can be used by another job. As with the scale-out experiments, LDA was run using m4.4xlarge instances on AWS EC2.

Fig. 9 shows the convergence plot for an execution of MLR that scales in from 8 to 4 nodes after $\approx$30min, with the same graphs for the static executions reproduced from Fig. 7 for comparison. Similarly, Fig. 10 shows the convergence plot for an execution of LDA that scales in from 24 to 12 nodes after $\approx$33min, with the same graphs for the static executions reproduced from Fig. 8 for comparison. In both cases, the elastic execution progresses the same as the static pre-scaling execution until the nodes are removed, after which it progresses with similar performance as the static post-scaling execution.

To show that the release of nodes is fast, we measure the time between when the scale-in event is triggered and when the last Litz process running on a requested node exits. This represents the time an external job scheduler needs to wait before all requested resources are free to be used by another job. We run each experiment at least three times and report the average. For MLR, the last process takes on average 2.5s to exit, while the average time for LDA is 43s. The low latency for MLR is due to a combination of its stateless workers and Litz's default behavior of discarding input data upon scaling in. As a result, the only state that needs to be transferred are the PSshards residing on the server threads of each requested node, which total $\approx$10MiB when split between 8 nodes. The executors in LDA, on the other hand, are stateful and contain a portion of its model parameters. When distributed across all nodes, each node contains $\approx$4.6GiB of executor state that need to be transferred away. We believe these times are reasonable for an external scheduler to wait for. As a comparison, even a pre-emptive environment like the AWS Spot Market gives the application a warning time of 120s before forcefully evicting its nodes.

4.3 Resource Variability in ML Applications

One advantage of elasticity in an ML framework is that in addition to scaling in and out based on the directions from the cluster scheduler, an elastic parameter server can leverage resource variability that is inherent in ML applications to autonomously give up resources. For example, Fig. 1 shows how the aggregate memory usage of LDA drops during its run-time on Litz. Although LDA running on STRADS has a similar decreasing memory usage, the lack of elasticity in STRADS does not allow it to leverage this phenomenon for efficient scheduling.

Litz, on the other hand, can detect variability in the resource usage and reduce the number of worker and server threads accordingly. Fig. 11 shows the breakdown of memory usage during LDA. Server threads that store the model start with 6 GiB and drop to around 1 GiB by the 10th epoch, suggesting that the server threads can be reduced by 80%. Similarly, the worker threads start with 370 GiB of memory and reduce to about 300 GiB by the 10th epoch, suggesting that their count can be halved and respective resources can be released. We leave adding autonomous scale-in to Litz as future work.

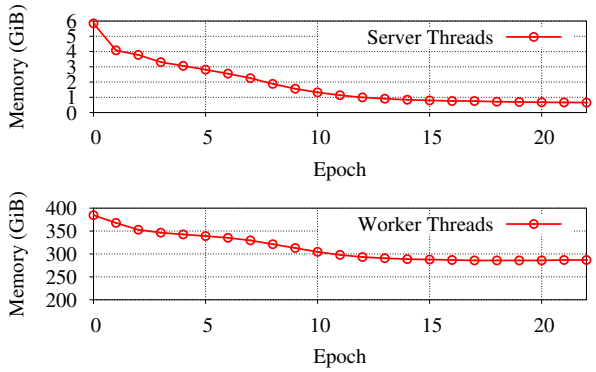

Figure 11: Breakdown of the aggregate memory usage on a cluster of 12 nodes during runtime of Latent Dirichlet Allocation (LDA) application (Sec. 2.6) implemented with Litz.

5 Related Work and Discussion

Hadoop [1] and Spark [55] are examples of elastic frameworks that are heavily used for large-scale data analytics on the cloud and data centers. As with Litz, a key enabler of elasticity in these frameworks is the quantization of a large job into short tasks. However, their programming models do not fit the requirements of distributed ML applications, and lack the flexibility to directly manage consistency and scheduling that is needed by staleness- and dependency-aware algorithm. On the other hand, specialized ML frameworks like Bösen [49], Adam [12], and Graphlab [39], to name a few, lack the quantization of jobs. Their approach is similar to the traditional HPC applications written in MPI, which launch OS processes that hold the state of the whole computation on a fixed number of pre-allocated nodes, running until the end of the job. Strads [31] and the parameter server framework of Li *et. al.* [35], while having the concepts of tasks and a task scheduler, do not perform any automatic management of application state under elastic conditions. The latter, which seems to allow for application elasticity in principle, places the onus on the application developer to re-balance worker state upon a scaling event, and to ensure that correctness is not violated. Compared with the aforementioned frameworks, Litz provides a task-based programming model that exposes the necessary details needed by distributed ML applications, while abstracting away the low-level system details that arise when executing under elastic conditions, allowing the application developer to focus on the algorithmic details of distributed ML.

Recently, there has been a growing interest in utilizing *transient* nodes in the cloud spot markets for big-data analytics, which has started to affect ML-specialized systems. The systems developed for this setting try to execute jobs with the performance of *on-demand* nodes at a significantly cheaper cost, using transient nodes. The challenge for these systems is to deal with the bulk revocations efficiently by choosing right fault-tolerance mechanism. For example, SpotOn [46] dynamically determines the fault-tolerance mechanism that best balances the risk of revocation with the overhead of the mechanism. While SpotOn applies these fault-tolerance mechanisms at the systems level—using virtual machines or containers—Flint [45] argues that application-aware approach is preferable and can improve efficiency by adapting the fault-tolerance policy. Flint, which is based on Spark, proposes automated and selective checkpointing policies for RDDs, to bound the time Spark spends recomputing lost in-memory data after a bulk revocation of transient nodes. TR-Spark [52] argues that RDDs—the checkpointing unit in Spark—are too coarse-grained, making Spark unfit to run on transient resources, and takes Flint's approach further by providing fine-grained task-level checkpointing.

Unlike Flint and TR-Spark that adapt a general-purpose Spark framework to achieve cost-effective analytics with transient resources, Proteus [24] adapts a specialized ML framework to achieve significantly faster and cheaper execution, while introducing elasticity optimizations tuned for the setting. Specifically, Proteus stores the ML model on parameter servers that run on reliable on-demand nodes, and makes the workers stateless so that they can be run on transient node, effectively pushing workers' states to parameter servers, along with the model. This is a reasonable approach for the spot market setting where bulk revocations can take offline a large number of workers without notice. Although it works well for applications with small worker state, with an increasing data and model size, the approach may run into performance problems due to the communication overhead between workers and their state stored on the parameter servers. Litz, on the other hand, keeps the worker state in the workers and assumes a cooperative cluster scheduler that will ask the running application to give up nodes and wait for state to be transferred away. This approach results in high performance while still providing elasticity.

6 Conclusion and Future Work

We present Litz—an evolutionary step in the development of distributed frameworks specialized for ML. By adopting a micro-task model, Litz shows that parameter server systems do not have to be monolithic and inflexible applications running on a fixed amount of preallocated resources: Litz achieves **elasticity**—the ability to scale out and in based on the resource availability—without compromising the efficiency of the monolithic specialized ML frameworks.

With the adoption of micro-task model also comes the **generality** that enables Litz to express model-parallel algorithms. So far, parameter server architectures have been predominantly data-parallel systems. Litz can express both, data- and model-parallel algorithms while achieving on par or better performance with the state of the art, in addition to being elastic.

Resource variability during the runtime of large data-analytics jobs is well known, and many schedulers have been introduced to exploit this variability for an efficient scheduling of jobs [30, 21, 20]. However, no previous work exist on the resource usage of specialized ML frameworks. As a future work, we plan to investigate the resource usage patterns of large ML jobs and leverage the resource variability together with the elasticity of Litz for a more efficient scheduling of ML jobs.

References

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

[2] Boost Coroutine2. `www.boost.org/doc/libs/1_63_0/libs/coroutine2/`.

[3] Boost Coroutine2. `http://www.boost.org/doc/libs/1_64_0/libs/serialization/`.

[4] etcd. `http://coreos.com/etcd/`.

[5] Kubernetes. `http://kubernetes.io`.

[6] ZeroMQ. `http://zeromq.org`.

[7] Alekh Agarwal and John C Duchi. Distributed delayed stochastic optimization. In *Advances in Neural Information Processing Systems*, pages 873–881, 2011.

[8] Sungjin Ahn, Babak Shahbaba, Max Welling, et al. Distributed stochastic gradient mcmc. In *ICML*, pages 1044–1052, 2014.

[9] Peter Bailis, Shivaram Venkataraman, Michael J. Franklin, Joseph M. Hellerstein, and Ion Stoica. Probabilistically bounded staleness for practical partial quorums. *Proc. VLDB Endow.*, 5(8):776–787, April 2012.

[10] David M. Blei, Andrew Y. Ng, and Michael I. Jordan. Latent dirichlet allocation. *J. Mach. Learn. Res.*, 3:993–1022, March 2003.

[11] Trishul Chilimbi, Yutaka Suzue, Johnson Apacible, and Karthik Kalyanaraman. Project adam: Building an efficient and scalable deep learning training system. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 571–582, Broomfield, CO, October 2014. USENIX Association.

[12] Trishul Chilimbi, Yutaka Suzue, Johnson Apacible, and Karthik Kalyanaraman. Project Adam: Building an Efficient and Scalable Deep Learning Training System. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 571–582, Broomfield, CO, October 2014. USENIX Association.

[13] James Cipar, Qirong Ho, Jin Kyu Kim, Seunghak Lee, Gregory R. Ganger, Garth Gibson, Kimberly Keeton, and Eric Xing. Solving the straggler problem with bounded staleness. In *Presented as part of the 14th Workshop on Hot Topics in Operating Systems*, Berkeley, CA, 2013. USENIX.

[14] Wei Dai, Abhimanu Kumar, Jinliang Wei, Qirong Ho, Garth Gibson, and Eric P. Xing. High-performance distributed ml at scale through parameter server consistency models. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*, AAAI'15, pages 79–87. AAAI Press, 2015.

[15] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Andrew Senior, Paul Tucker, Ke Yang, Quoc V Le, et al. Large scale distributed deep networks. In *Advances in neural information processing systems*, pages 1223–1231, 2012.

[16] Evgeniy Gabrilovich, Michael Ringgaard, and Amarnag Subramanya. Facc1: Freebase annotation of clueweb corpora, version 1 (release date 2013-06-26, format version 1, correction level 0). http://lemurproject.org/clueweb12/, 2013.

[17] Rainer Gemulla, Erik Nijkamp, Peter J. Haas, and Yannis Sismanis. Large-scale matrix factorization with distributed stochastic gradient descent. In *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '11, pages 69–77, New York, NY, USA, 2011. ACM.

[18] Rainer Gemulla, Erik Nijkamp, Peter J Haas, and Yannis Sismanis. Large-scale matrix factorization with distributed stochastic gradient descent. In *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 69–77. ACM, 2011.

[19] Sanjay Ghemawat and Paul Menage. TCMalloc : Thread-Caching Malloc. `http://goog-perftools.sourceforge.net/doc/tcmalloc.html`.

[20] Robert Grandl, Mosharaf Chowdhury, Aditya Akella, and Ganesh Ananthanarayanan. Altruistic scheduling in multi-resource clusters. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 65–80, GA, 2016. USENIX Association.

[21] Robert Grandl, Srikanth Kandula, Sriram Rao, Aditya Akella, and Janardhan Kulkarni. Graphene: Packing and dependency-aware scheduling for data-parallel clusters. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 81–97, GA, 2016. USENIX Association.

[22] Thomas L. Griffiths and Mark Steyvers. Finding scientific topics. *PNAS*, 101(suppl. 1):5228–5235, 2004.

[23] Aaron Harlap, Henggang Cui, Wei Dai, Jinliang Wei, Gregory R. Ganger, Phillip B. Gibbons, Garth A. Gibson, and Eric P. Xing. Addressing the straggler problem for iterative convergent parallel ml. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*, SoCC '16, pages 98–111, New York, NY, USA, 2016. ACM.

[24] Aaron Harlap, Alexey Tumanov, Andrew Chung, Greg Ganger, and Phil Gibbons. Proteus: agile ml elasticity through tiered reliability in dynamic resource markets. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '17, New York, NY, USA, 2017. ACM.

[25] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. Mesos: A Platform for Fine-grained Resource Sharing in the Data Center. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, pages 295–308, Berkeley, CA, USA, 2011. USENIX Association.

[26] Qirong Ho, James Cipar, Henggang Cui, Seunghak Lee, Jin K. Kim, Phillip B. Gibbons, Garth A. Gibson, Greg Ganger, and Eric Xing. More effective distributed ml via a stale synchronous parallel parameter server. In C.j.c. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K.q. Weinberger, editors, *Advances in Neural Information Processing Systems 26*, pages 1223–1231. 2013.

[27] Qirong Ho, James Cipar, Henggang Cui, Seunghak Lee, Jin Kyu Kim, Phillip B. Gibbons, Garth A Gibson, Greg Ganger, and Eric P Xing. More effective distributed ml via a stale synchronous parallel parameter server. In C.J.C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K.Q. Weinberger, editors, *Advances in Neural Information Processing Systems 26*, pages 1223–1231. Curran Associates, Inc., 2013.

[28] Mingyi Hong. A distributed, asynchronous and incremental algorithm for nonconvex optimization: An admm based approach. *arXiv preprint arXiv:1412.6058*, 2014.

[29] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'10, pages 11–11, Berkeley, CA, USA, 2010. USENIX Association.

[30] Sangeetha Abdu Jyothi, Carlo Curino, Ishai Menache, Shravan Matthur Narayanamurthy, Alexey Tumanov, Jonathan Yaniv, Ruslan Mavlyutov, Inigo Goiri, Subru Krishnan, Janardhan Kulkarni, and Sriram Rao. Morpheus: Towards automated slos for enterprise clusters. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 117–134, GA, 2016. USENIX Association.

[31] Jin Kyu Kim, Qirong Ho, Seunghak Lee, Xun Zheng, Wei Dai, Garth A. Gibson, and Eric P. Xing. Strads: A distributed framework for scheduled model parallel machine learning. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, pages 5:1–5:16, New York, NY, USA, 2016. ACM.

[32] Balaji Krishnapuram, Lawrence Carin, Mario A. T. Figueiredo, and Alexander J. Hartemink. Sparse multinomial logistic regression: Fast algorithms and generalization bounds. *IEEE Trans. Pattern Anal. Mach. Intell.*, 27(6):957–968, June 2005.

[33] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012.

[34] Abhimanu Kumar, Alex Beutel, Qirong Ho, and Eric P Xing. Fugue: Slow-worker-agnostic distributed learning for big models on big data. In *AISTATS*, pages 531–539, 2014.

[35] Mu Li, David G. Andersen, Jun Woo Park, Alexander J. Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J. Shekita, and Bor-Yiing Su. Scaling distributed machine learning with the parameter server. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 583–598, Broomfield, CO, October 2014. USENIX Association.

[36] Mu Li, David G Andersen, and Alexander Smola. Distributed delayed proximal gradient methods. In *NIPS Workshop on Optimization for Machine Learning*, 2013.

[37] Mu Li, Li Zhou Zichao Yang, Aaron Li Fei Xia, David G. Andersen, and Alexander Smola. Parameter server for distributed machine learning. *NIPS workshop*, 2013.

[38] Jun Liu, Jianhui Chen, and Jieping Ye. Large-scale sparse logistic regression. In *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '09, pages 547–556, New York, NY, USA, 2009. ACM.

[39] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M. Hellerstein. Distributed graphlab: A framework for machine learning and data mining in the cloud. *Proc. VLDB Endow.*, 5(8):716–727, April 2012.

[40] Brendan McMahan and Matthew Streeter. Delay-tolerant algorithms for asynchronous distributed online learning. In *Advances in Neural Information Processing Systems*, pages 2915–2923, 2014.

[41] H. Brendan McMahan and Matthew Streeter. Delay-tolerant algorithms for asynchronous distributed online learning. *Advances in Neural Information Processing Systems (NIPS)*, 2014.

[42] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252, 2015.

[43] Chad Scherrer, Ambuj Tewari, Mahantesh Halappanavar, and David Haglin. Feature clustering for accelerating parallel coordinate descent. In *Advances in Neural Information Processing Systems*, pages 28–36, 2012.

[44] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. Omega: flexible, scalable schedulers for large compute clusters. In *SIGOPS European Conference on Computer Systems (EuroSys)*, pages 351–364, Prague, Czech Republic, 2013.

[45] Prateek Sharma, Tian Guo, Xin He, David Irwin, and Prashant Shenoy. Flint: Batch-interactive data-intensive processing on transient servers. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, pages 6:1–6:15, New York, NY, USA, 2016. ACM.

[46] Supreeth Subramanya, Tian Guo, Prateek Sharma, David Irwin, and Prashant Shenoy. Spoton: A batch computing service for the spot market. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, SoCC '15, pages 329–341, New York, NY, USA, 2015. ACM.

[47] Jinjun Wang, Jianchao Yang, Kai Yu, Fengjun Lv, Thomas Huang, and Yihong Gong. Locality-constrained linear coding for image classification. In *IN: IEEE CONFERENCE ON COMPUTER VISION AND PATTERN CLASSIFICATOIN*, 2010.

[48] Minjie Wang, Tianjun Xiao, Jianpeng Li, Jiaxing Zhang, Chuntao Hong, and Zheng Zhang. Minerva: A scalable and highly efficient training platform for deep learning. In *NIPS Workshop, Distributed Machine Learning and Matrix Computations*, 2014.

[49] Jinliang Wei, Wei Dai, Aurick Qiao, Qirong Ho, Henggang Cui, Gregory R. Ganger, Phillip B. Gibbons, Garth A. Gibson, and Eric P. Xing. Managed communication and consistency for fast data-parallel iterative analytics. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, SoCC '15, pages 381–394, New York, NY, USA, 2015. ACM.

[50] Pengtao Xie, Jin Kyu Kim, Yi Zhou, Qirong Ho, Abhimanu Kumar, Yaoliang Yu, and Eric P. Xing. Distributed machine learning via sufficient factor broadcasting. *CoRR*, abs/1511.08486, 2015.

[51] Eric P. Xing, Qirong Ho, Wei Dai, Jin Kyu Kim, Jinliang Wei, Seunghak Lee, Xun Zheng, Pengtao Xie, Abhimanu Kumar, and Yaoliang Yu. Petuum: A new platform for distributed machine learning on big data. *IEEE Trans. Big Data*, 1(2):49–67, 2015.

[52] Ying Yan, Yanjie Gao, Yang Chen, Zhongxin Guo, Bole Chen, and Thomas Moscibroda. Tr-spark: Transient computing for big data analytics. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*, SoCC '16, pages 484–496, New York, NY, USA, 2016. ACM.

[53] Jinhui Yuan, Fei Gao, Qirong Ho, Wei Dai, Jinliang Wei, Xun Zheng, Eric Po Xing, Tie-Yan Liu, and Wei-Ying Ma. Lightlda: Big topic models on modest computer clusters. In *Proceedings of the 24th International Conference on World Wide Web*, pages 1351–1361. ACM, 2015.

[54] Hyokun Yun, Hsiang-Fu Yu, Cho-Jui Hsieh, SVN Vishwanathan, and Inderjit Dhillon. Nomad: Non-locking, stochastic multi-machine algorithm for asynchronous and decentralized matrix completion. *Proceedings of the VLDB Endowment*, 7(11):975–986, 2014.

[55] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster Computing with Working Sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association.

[56] Ruiliang Zhang and James T Kwok. Asynchronous distributed admm for consensus optimization. In *ICML*, pages 1701–1709, 2014.