Litz: Elastic Framework for High-Performance Distributed Machine Learning

Aurick Qiao[1,2], Abutalib Aghayev[2], Weiren Yu[1,3], Haoyang Chen[1],
Qirong Ho[1], Garth A. Gibson[2,4], Eric P. Xing[1,2],
[1]*Petuum, Inc.* [2]*Carnegie Mellon University* [3]*Beihang University* [4]*Vector Institute*

Abstract

Machine Learning (ML) is an increasingly popular application in the cloud and data-center, inspiring new algorithmic and systems techniques that leverage unique properties of ML applications to improve their distributed performance by orders of magnitude. However, applications built using these techniques tend to be static, unable to elastically adapt to the changing resource availability that is characteristic of multi-tenant environments. Existing distributed frameworks are either inelastic, or offer programming models which are incompatible with the techniques employed by high-performance ML applications.

Motivated by these trends, we present Litz, an elastic framework supporting distributed ML applications. We categorize the wide variety of techniques employed by these applications into three general themes — stateful workers, model scheduling, and relaxed consistency — which are collectively supported by Litz's programming model. Our implementation of Litz's execution system transparently enables elasticity and low-overhead execution.

We implement several popular ML applications using Litz, and show that they can scale in and out quickly to adapt to changing resource availability, as well as how a scheduler can leverage elasticity for faster job completion and more efficient resource allocation. Lastly, we show that Litz enables elasticity without compromising performance, achieving competitive performance with state-of-the-art non-elastic ML frameworks.

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, 45, 27]. 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 algorithmic and systems techniques for distributed ML applications have improved their performance by an order of magnitude or more. New algorithms such as AdaptiveRevision [39], NOMAD [42], and LightLDA [55] can better scale in distributed environments, possessing favorable properties such as staleness tolerance [39, 28], lock-free execution [42, 56], and structure-aware parallelization [20, 55]. Systems and frameworks such as GraphLab [38], Petuum [53], Adam [15], and various parameter servers [36, 28] are able to support and exploit these properties to achieve even higher performance, using techniques such as bounded-staleness consistency models [17], structure-aware scheduling [33], bandwidth management/re-prioritization [50], and network message compression [52, 15].

Although significant work is being done to push the boundaries of distributed ML in terms of performance and scalability, there has not been as much focus on elasticity, thus limiting the resource adaptability of ML applications in real-world computing environments.

General-purpose distributed frameworks such as Hadoop [1] and Spark [57] are well integrated with cloud and data-center environments, and are extensively used for running large-scale data processing jobs. They are designed to support a wide spectrum of conventional tasks—including SQL queries, graph computations, and sorting and counting—which are typically transaction-oriented and rely on deterministic execution. However, their programming models are incompatible with the algorithmic and systems techniques employed by distributed ML applications, abstracting away necessary details such as input data partitioning, computation scheduling, and consistency of shared memory access. As a result, the performance of ML applications built using these frameworks fall short of standalone implementations by two orders of magnitude or more [51].

Consequently, distributed ML applications are often implemented without support from elastic frameworks, resulting in jobs that hold a rigid one-time allocation of cluster resources from start to finish [50, 33, 56, 15]. The lack of an elastic framework, along with a suitable programming model which can support the various distributed ML techniques, is a key roadblock for implementing elastic ML applications.

Although the algorithmic and systems techniques employed by these standalone applications are diverse, they typically arise from only a few fundamental properties of ML that can be collectively supported by an elastic ML framework. This observation exposes an opportunity to design a framework that is able to support a large variety

of distributed ML techniques by satisfying a smaller set of more general requirements. We summarize these properties of ML and how they guide the design of an elastic framework below, and further elaborate on them in Sec. 2. First, ML computations exhibit a wide variety of memory access patterns. Some mutable state may be accessed when processing each and every entry of a dataset, while other state may only be accessed when processing a single data entry. To improve locality of access, ML applications explicitly co-locate mutable model parameters with immutable dataset entries [55]. Each worker machine in the computation may contain a non-trivial amount of mutable state, which needs to be properly managed under an elastic setting.

Second, ML models contain a wide variety of dependency structures. Some sets of model parameters may safely be updated in parallel, while other sets of parameters must be updated in sequence. Guided by these dependency structures, ML applications carefully schedule their model updates by coordinating tasks across physical worker machines [20]. An elastic ML framework should abstract the physical cluster away from applications while still providing enough flexibility to support this type of task scheduling.

Furthermore, ML algorithms are often iterative-convergent and robust against small errors. Inaccuracies occurring in their execution are automatically corrected during later stages of the algorithm. Distributed ML applications have been able to attain higher performance at no cost to correctness by giving up traditionally desirable properties such as deterministic execution and consistency of memory access [28]. Framework mechanisms for elasticity should not rely on a programming model that restricts this way of exploiting the error-tolerance of ML algorithms.

Thus, to efficiently support ML applications, an elastic ML framework 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. Then, 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 needs and opportunities for elasticity of ML applications, we designed and implemented *Litz*[1], an elastic framework for distributed ML that provides a programming model supporting stateful workers, model scheduling and relaxed consistency.

Litz enables low-overhead elasticity for high-performance ML applications. When physical machines are added to or removed from an active job, state and computation are automatically re-balanced across the new set of available machines without active participation by the application. Litz's programming model can express key distributed ML techniques such as stateful workers, model scheduling and relaxed consistency, allowing high-performance ML applications to be implemented. Furthermore, a cluster job scheduler can leverage Litz's elasticity to achieve faster job completion under priority scheduling, and optimize resource allocation by exploiting inherent resource variability of ML algorithms.

Our main contributions are:

1. **Event-driven Programming Model for ML:** Litz exposes an event-driven programming model that cleanly separates applications from the physical cluster they execute on, enabling stateful workers and allowing the framework to transparently manage application state and computation during elastic events. Computation is decomposed into *micro-tasks* which have shared access to a distributed parameter server.
2. **Task-driven Consistency Model for ML:** Micro-tasks can be scheduled according to dependencies between them, allowing the application to perform model scheduling. Access to the parameter server is controlled by a consistency model in which a micro-task always observes all updates made by its dependencies, while having intentionally weak guarantees between independent micro-tasks.
3. **Optimized Elastic Execution System:** Litz's execution system transparently re-balances workload during scaling events without active participation from the application. It exploits Litz's programming and consistency models to implement optimizations that reduce system overhead, allowing applications using Litz to be as efficient as those using non-elastic execution systems.

The rest of this paper is organized as follows. In Sec. 2, we review ML algorithm properties and opportunities for elasticity, while Sec. 3 and Sec. 4 describes the Litz design and optimizations. In Sec. 5, 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. Sec. 6 reviews related work, and Sec. 7 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 $\mathscr{L}(A,\mathscr{D})$, defined over a vector (or matrix) of model parameters A and collection of input data $\mathscr{D}$, and which measures how well the model parameters A fit the data $\mathscr{D}$; (2) their goal is to find a value of A that maximizes (or alternatively, minimizes) the objective $\mathscr{L}(A,\mathscr{D})$, via an *iterative-convergent* procedure that repeatedly executes a set of update equations, which

[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.

gradually move A towards an optimal value (i.e. hill-climbing). These update equations follow the generic form

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

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 $\mathscr{D}$. 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

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

where $\mathscr{D}_1, \ldots, \mathscr{D}_P$ partition the input data $\mathscr{D}$ and each Δ_i computes a partial update using $\mathscr{D}_i$ 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 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 [50, 18, 36, 28].

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 $\mathscr{D}_i$ 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 re-allocation.

2.2 Error Tolerance & 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. $A^{(t)} = A^{(t-1)} + \Delta(A^{(t-1)}, \mathscr{D}) + \varepsilon$, 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 [28, 17, 13], which allow stale model parameters to be used in update computations, i.e. $A^{(t)} = A^{(t-1)} + \Delta(A^{(t-s)}, \mathscr{D})$, 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 [16, 25]; and (2) reduce the effect of network bottlenecks caused by synchronization by allowing cached parameter values to be used. Stale-Synchronous Parallel (SSP) [28] is such a consistency model, under which a set of distributed workers may read cached values from a shared parameter server as long as their staleness do not exceed a fixed limit.

Staleness-aware ML Algorithms: Beyond simply applying bounded staleness consistency to existing algorithms, the ML community has developed new staleness-aware algorithms [39, 58, 55, 12, 29, 10, 37] 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.

2.3 Dependencies 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 $\mathscr{L}()$ 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 [44, 55, 20, 18, 35, 49, 38]. 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 $\text{dep}(A_1, A_2) > \varepsilon$. As with staleness-aware algorithms, dependency-aware algorithms converge up to an order of magnitude faster than their non-dependency-aware counterparts.

3 Litz 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 techniques in distributed ML, while exposing enough structure in the application that the underlying execution system can take control under elastic conditions. Guided by the insights presented in Sec. 2, we describe how Litz's programming model naturally arises from the properties of ML applications, and how it enables an efficient and elastic run-time implementation. For reference, a detailed summary of Litz's API can be found in Table 1.

Input Data Over-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 directly assigning input data, Litz first distributes it across a set of logical *executors*, which are in turn mapped to physical machines. Elasticity is enabled by allocating more executors than physical machines and migrating excess executors to other machines as they become available. This separation also lets Litz support stateful workers by allowing executor state to be defined and mutated by the application while being treated as a black box by the run-time system.

Micro-Tasks 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. At the end of each micro-task, control is yielded back to the run-time system, exposing frequent opportunities for executors to be migrated. During its execution, a micro-task is granted read/update access to a global parameter server via a key-value interface (`PSGet/PSUpdate` in Table 1) and applies partial updates to model parameters by modifying application state in the executor and/or updating globally-shared values in the parameter server.

Model Scheduling and Relaxed Consistency: Litz enables both model scheduling and relaxed consistency using application-defined 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 observes 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 of the other, then they may be executed in any order or in parallel, and may observe none, some, or all of the updates made by the other. This critical piece of non-determinism lets the application exploit relaxed consistency models by allowing the run-time system to cache and use stale values from the parameter server between independent micro-tasks.

Micro-Task Dispatch and Completion: A common way to specify dependencies between tasks is through a directed "dependency" graph in which each vertex corresponds to a micro-task, and an arc from vertex A to vertex B means task A is a dependency of task B. However, due to a potentially large number of micro-tasks, explicitly specifying such a graph up-front may incur significant overhead. Instead, each Litz application defines a *driver* which dynamically dispatches micro-tasks during run-time via the `DispatchTask` method. When a micro-task completes, Litz invokes the `HandleTaskCompletion` method on the driver, which can then dispatch any additional micro-tasks.

Without an explicit dependency graph, Litz needs an alternative way to decide when a micro-task should be able to observe another micro-task's updates. Otherwise, its execution system does not have enough information to know when it is safe for a micro-task to use cached parameter values, thus giving up a significant opportunity for performance optimization. To overcome this issue, Litz uses the sequence of micro-task dispatch and completion events to infer causal relationships between micro-tasks, which can then be used to generate the dependencies needed to implement its cache coherence protocol. According to the following two cases:

1. If micro-task B is dispatched *before* being informed of the completion of micro-task A, then Litz infers that the completion of A *did not cause* the dispatch of B. A *is not* a dependency of B, and B may observe some, all, or none of the updates made by A.
2. If micro-task B is dispatched *after* being informed of the completion of micro-task A, then Litz infers that A *may have caused* the dispatch of B. A *may be* a dependency of B, and B will observe all updates made by A.

This consistency model is similar to Causal Memory [11], in which causally related read/write operations are observed in the same order by all nodes. We discuss how Litz's consistency model and its cache coherence protocol can be implemented efficiently in Sec. 4.

4 Litz Implementation and Optimizations

Litz is implemented in approximately 6500 lines of C++ code using the ZeroMQ [8] library for low latency communication and Boost's Context [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. 1. 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

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.

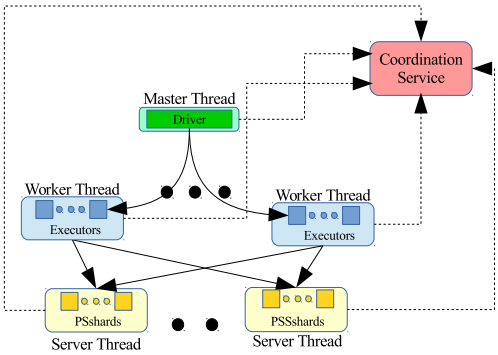

Figure 1: High-level architecture of Litz. The driver in the master thread dispatches micro-tasks to be performed by executors on the worker threads. Executors can read and update the global model parameters distributed across PSshards on the server threads.

by initially invoking `DispatchInitialTasks` and then continuously invoking `HandleTaskCompletion`, sending them to worker threads to be executed. Second, the master thread maintains the dynamic mappings between executors and 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 [31] 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) it does not actively participate in elasticity and checkpointing, but simply invoked by the execution system when needed, and (2) third-party libraries can be used to reduce programming overhead [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).When an executor needs to be moved, the worker thread first finishes any of its ongoing micro-tasks for that executor, buffering any other pending micro-tasks for that executor. The worker thread then sends the executor's 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 from shared storage. 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 redistributed 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 naïve approach is to stop all micro-tasks on every executor, then perform the transfer, then notify all worker threads of the change, and finally resume execution. This method guarantees that requests and updates are always sent to server threads that contain the target PSshard, but incurs high overhead due to suspending the entire application. Instead, the server threads perform *request and update forwarding*, and executors are never blocked from sending a parameter request or update. When a server thread receives a message for a PSshard it no longer contains, it forwards the message to the server thread it last transferred the PSshard to. Forwarding can occur multiple times until the target PSshard is found, the request/update is performed, and the response is sent back to the originating worker thread. This

way, execution of micro-tasks can proceed uninterrupted during parameter server scaling events.

Consistent Checkpoint and Recovery: To achieve fault tolerance, Litz periodically saves a checkpoint of the application to persistent storage, consisting of (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 from shared storage during recovery. 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 restored, after which normal execution is resumed.

Parameter Cache Synchronization: The consistency model outlined in Sec. 3 exposes an opportunity for the runtime 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. By only using the sequence of micro-task dispatch and completion events to infer dependencies, Litz enables both (1) and (2) to be implemented efficiently. In particular, the dependencies of micro-task B are a subset of the dependencies of micro-task A if the total number of micro-tasks that have been completed when B was dispatched is at most the total number of micro-tasks that have been completed when A was dispatched.

To implement this cache coherence protocol, the master thread maintains a single monotonically increasing *version* number that is incremented each time `HandleTaskCompletion` is invoked. Whenever the driver dispatches a micro-task, the master thread tags the micro-task with the version number at that time. After micro-task A retrieves a fresh value from the parameter server, it caches the value and tags it with A's version. When micro-task B wants to access the same parameter, it first checks if its own version is less than or equal to the version of 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 version. 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, to implement 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 version and share cached parameter values with each other. Since the micro-tasks for iteration i are dispatched before 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 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 contains an *update log* which maps parameter keys to locally aggregated updates. Whenever a micro-task invokes `PSUpdate`, the update is first aggregated with 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 being sent over the network, the results of micro-tasks are not immediately returned to the master thread after they are completed. Doing this allows the updates from many more 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 can be carefully tuned, but 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: Litz employs co-operative multitasking implemented using co-routines [2]. When one task is blocked on 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.

5 Evaluation

We start by evaluating 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 minimal performance impact. Second, a running Litz application is able to efficiently make use of additional nodes allocated to it, accelerating its time to completion. Third, a running Litz application is able to release its nodes on request, quickly freeing them to be allocated to another job.

Next, we discuss how Litz's elasticity can be leveraged by a cluster job scheduler to (1) reduce the completion time of an ML job that yields resources to a higher-priority job, and (2) improve resource allocation by exploiting the inherent decreasing memory usage of many ML algorithms.

Lastly, we evaluate Litz's performance when executing diverse applications which make use of stateful workers, model scheduling, and relaxed consistency. With the multinomial logistic regression (MLR) application, we show that our implementation on Litz is faster than the built-in implementation in Bösen [50], a non-elastic ML system for data-parallel SSP workloads. With the latent Dirichlet allocation (LDA) application, we show that our implementation on Litz is competitive with the built-in implementation in Strads [33], a non-elastic ML system for model scheduling. Furthermore, to evaluate Litz for the special case of deep learning, we implement a deep feed-forward neural network and compare its performance with Tensorflow [9].

ML Applications: MLR and LDA are popular ML applications used for multi-class classification and topic modeling, respectively. The goal of our evaluation is to show that Litz enables elasticity for these applications at little cost to performance when compared with state-of-the-art non-elastic systems. Thus, we closely follow their implementations in Bösen and Strads, using SGD and the SSP relaxed consistency model for MLR, and block-scheduled Gibbs sampling with stateful workers for LDA. For details of these implementations of MLR and LDA, we refer readers to their descriptions in Wei *et al.* [50] and Kim *et al.* [33], respectively.

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 [43] consisting of 1.2M images labeled using 1000 different object categories. The dataset is pre-processed using the LLC feature extraction algorithm [48], 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 [19] 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.

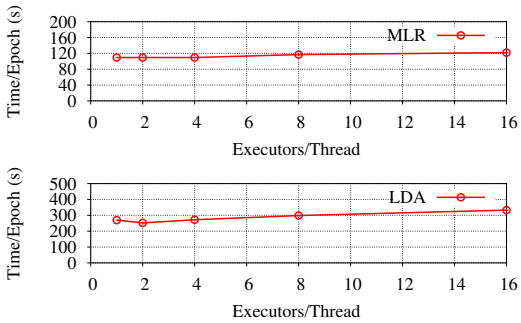

Figure 2: Average time per epoch for MLR and LDA when running with various numbers of executors per worker thread. In both cases the overhead of increasing the number of executors is insignificant. We define one epoch as performing a single pass over all input data.

5.1 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. 2 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× the number of executors, and gradually increases to 1.11× the baseline when using 16× the number of executors. For LDA, the execution time initially decreases to 0.94× the baseline when using 2× the number of executors, and thereafter gradually increases to 1.23× the baseline when using 16× the number of executors. We believe the overhead introduced by increasing the number of executors is quite an acceptable trade-off for elasticity and can still be reduced with further optimizations.

5.1.1 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. Each node runs both worker threads and server threads, so both executors and PSshards are rebalanced during scaling. The experiments for LDA in

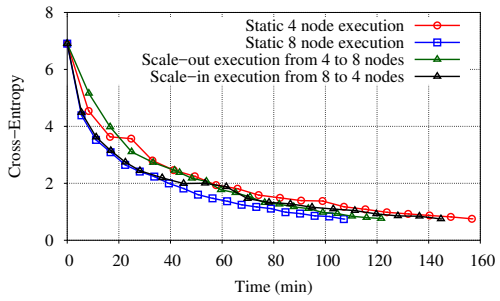

Figure 3: MLR execution on Litz with 4 nodes, with 8 nodes, with an elastic execution that scales out from 4 nodes to 8 nodes, and with an elastic execution that scales in from 8 nodes to 4 nodes. For the scale-out execution, the nodes are added at about 40 minutes into execution. For the scale-in execution, the nodes are removed at about 30 minutes into execution.

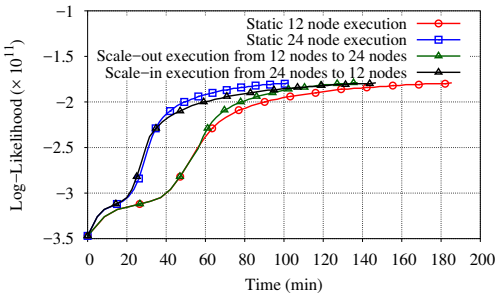

Figure 4: LDA execution on Litz with 12 nodes, with 24 nodes, and with an elastic execution that scales out from 12 nodes to 24 nodes. For the scale-out execution, the nodes are added at about 55 minutes into execution. For the scale-in execution, the nodes are removed at about 33 minutes into execution.

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. 3 shows the convergence plots for MLR, 4 new nodes 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. 4 shows the convergence plots for LDA, 12 new nodes added after ≈55min of execution. The static 12 node execution completes in ≈183min while the scale-out execution completes in ≈134min, resulting in a 27% shorter total run-time.

5.1.2 Ideal Scale Out

Next, we evaluate the amount of room for improvement still achievable over Litz's current scale-out performance. Following a similar construction as Pundir et al. [41], 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

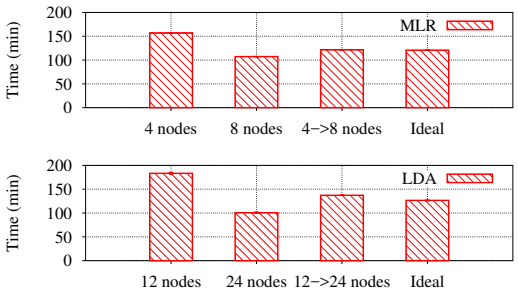

Figure 5: Static, scale-out, and ideal scale-out (See Sec. 5.1.1) execution times for MLR and LDA implemented on Litz. We scale out MLR from 4 nodes to 8 nodes, and LDA from 12 nodes to 24 nodes. Each experiment was performed several times, error bars are omitted due to their negligible size.

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. 5 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 actual scale-out 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 actual scale-out execution time is ≈134min, 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.

5.1.3 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 Litz jobs can release resources with low latency, quickly transferring executors and PSshards away from requested nodes so that they can be used by another job. 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. As with the scale-out experiments,

these experiments were run using m4.4xlarge EC2 instances.

We run each experiment at least three times and report the average. Fig. 3 shows the convergence plots for MLR with the scale-in event. We start the job with 8 nodes, and remove 4 nodes ≈30 minutes into execution. The convergence plot closely follows the plot of 8-node static execution until the scale-in event, and the plot of 4-node static execution after that. Similarly, Fig. 4 shows the convergence plots for LDA with the scale-in event. We start the job with 24 nodes, and remove nodes ≈33 minutes into execution. The convergence plot closely follows the plot of 24-node static execution until the scale-in event, and the plot of 12-node static execution after that.

For MLR, the scale-in event takese 2.5 seconds on average, while for LDA the average 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 ≈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 ≈4.6GiB of executor state that need to be transferred away. A benchmark of cluster network showed that it can sustain a bandwidth of 2.0Gbps between pairs of machines, meaning that the 4.6GiB of LDA executor state can ideally be transfered within 20s. Nevertheless, the current transfer times are reasonable for an external scheduler to wait for. For comparison, even a pre-emptive environment like the AWS Spot Market gives users a warning time of 120s before forcefully evicting their nodes.

5.2 Elastic Scheduling

Elasticity has many potential applications in both the cloud and data-center. In the cloud, elasticity can be leveraged to take advantage of transient nodes in spot markets [26] and drastically reduce the monetary cost of renting computation resources. In the data-center, a cluster-wide scheduler can optimize resource utilization by adaptively consolidating applications into fewer physical machines [30].

We present two specific instances where the elasticity enabled by Litz can benefit job scheduling. First, when a high-priority job needs to be scheduled, an elastic ML application can avoid preemption by cooperatively releasing resources. Second, the inherent resource variability of many ML applications allow Litz to automatically release memory throughout the lifetime of an ML job, freeing resources to be used by other jobs. Serious design and implementation of such a scheduler and its policies is deserving of thorough investigation, which we leave for future work.

5.2.1 Priority Scheduling

In multi-tenant computing environments, users frequently submit jobs (both ML and non-ML) which can have differing priorities. To meet the stricter SLA requirements of high-priority jobs, a scheduler must sometimes re-allocate some resources used by a lower-priority job. If the lower-priority job is inelastic, then it may be killed or suspended, leaving the rest of its resources under-utilized and delaying its completion time. For long-running jobs such as training ML models, their resources may need to be re-allocated several times during their lifetimes.

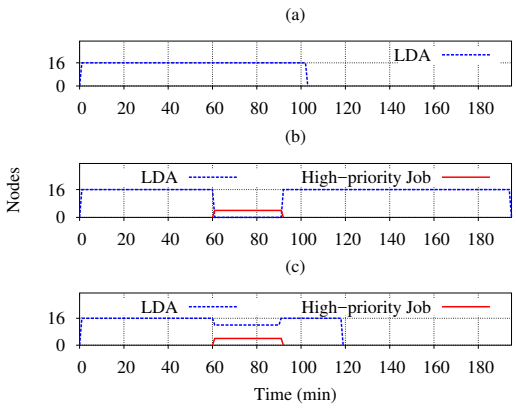

Figure 6: Priority scheduling experiments as described in Sec. 5.2.1. The graphs show the resource allocation over time in the cases of (a) LDA job which is uninterrupted, (b) LDA job which is killed when a higher-priority job is submitted, and (c) LDA job which elastically scales in when a higher-priority job is scheduled. We ran each experiment three times and saw negligible variation between each instance.

However, with the elasticity mechanism enabled by Litz, a long-running ML application can simply scale-in to use a fewer amount of resources, while the higher-priority job uses the released resources. After the higher-priority job completes, it can scale-out again, uninterrupted. We implemented this priority scheduling policy on a cluster of 16 m4.4xlarge nodes, and launched an LDA job on all 16 machines that runs for ≈100min if left uninterrupted (Fig. 6(a)). A higher-priority job is launched 60min into its runtime, requiring 4 nodes for 30min. Without elasticity, the LDA job is killed and re-started after the higher-priority job ends, requiring a total of ≈190min to complete (Fig. 6(b)). However, by leveraging elasticity to scale-in the LDA job, it can continue to run using 12 nodes and completes in ≈120min (Fig. 6(c)). At the same time, waiting for LDA to scale-in only increased the completion time of the high-priority job from 30min to 31min.

5.2.2 ML Resource Variability

The iterative-convergent nature of ML algorithms presents opportunities for resource scheduling not usually found in other computing tasks. One advantage of elasticity in an ML framework is that in addition to scaling in and out based on the directions from a cluster scheduler, an elastic ML framework can leverage resource variability that is inherent in ML applications to autonomously give up resources.

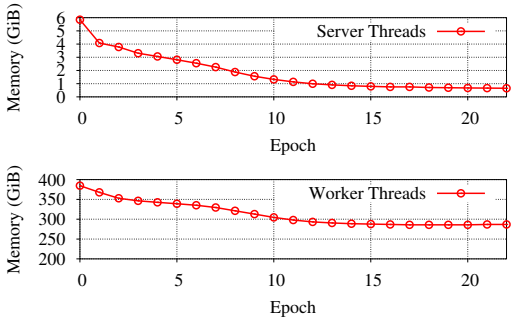

Figure 7: Memory usage on a cluster of 12 m4.4xlarge nodes during runtime of LDA implemented using Litz, broken down by server threads and worker threads. During the first 10 epochs, memory usage of server threads decrease by 5GiB, while memory usage of worker threads decrease by 70GiB.

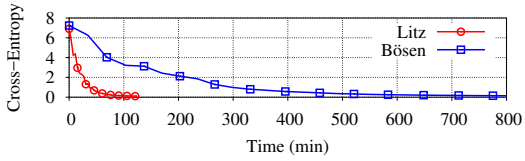

Figure 8: Multinomial Logistic Regression (MLR) running on 8 nodes using 25% of the ImageNet ILSVRC2012 dataset. Litz achieves convergence about 8× faster than Bösen.

In particular, many ML algorithms, including LDA, may find their model parameters becoming sparse (ie. mostly zeros) as they approach convergence [33], allowing memory usage to be reduced by using a more memory-efficient storage format (ie. sparse vector). 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. 7 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 reduced by 20% and respective resources can be released. This dynamic resource usage of ML jobs, when exposed through an elastic framework like Litz, can inform the policies of a cluster scheduler that allocates resources between many jobs.

5.3 Performance Experiments

We compare our Litz implementations of MLR and LDA with those built-in with the open-source versions of Bösen and Strads, respectively. 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 [21] memory allocator. These settings are the default for both Bösen and Strads.

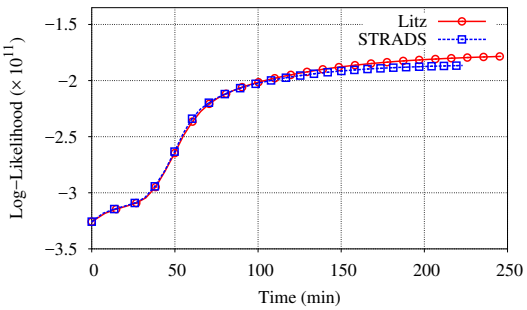

Figure 9: Latent Dirichlet Allocation (LDA) training algorithm running on Strads and Litz with the subsampled ClueWeb12 dataset. Litz completes all 34 epochs roughly 6% slower than Strads, but achieves a better objective value.

MLR 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.* [50] 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 such as step size and minibatch size. As Fig. 8 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.

LDA 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.* [33]. Both LDA instances were configured to use the same number of block partitions as well as the same LDA hyper-parameters α and β. We ran each application until 34 epochs have been completed, where an *epoch* is equivalent to a full pass over the input data. As Fig. 9 shows, our LDA implementation on Litz completes all epochs roughly 6% slower than that on Strads. However, it also achieves a better objective value (measured in log-likelihood), resulting in faster convergence than Strads overall. Even though more investigation into the per-epoch convergence difference is needed, we can attribute the throughput difference to the optimizations built into Strads, which employs a ring-topology specifically optimized for the block-partitioned model scheduling strategy used by LDA.

Deep Neural Networks (DNNs): To evaluate Litz with DNNs, we implemented a particular deep learning model called a deep feed-forward network [22], which forms the

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

basis of many deep learning applications. We used a network with two hidden layers with ReLU activation and one output layer with Softmax activation. We trained this model using both Litz and TensorFlow [9] on 4 m4.4xlarge EC2 instances, with the CIFAR-10 [34] dataset. This dataset consists of 60K images, which are pre-processed into vectors of ≈98K features, labeled using 10 classes. Both systems used the same data-parallel SGD algorithm, and were configured with the same tuning parameters such as a learning rate of 0.0001 and mini-batch size of 64. The training using Tensorflow progressed at a pace of ≈79s per batch, while the training using Litz progressed 3.4× faster at a pace of ≈23s per batch.

6 Discussion and Related Work

Recently, there has been a growing interest in utilizing *transient* nodes in the cloud spot markets for big-data analytics. 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 [47] 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 [46] 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 [54] 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 [26] 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.

7 Conclusion and Future Work

We present the design and implementation of Litz, an evolutionary step in the elastic execution of ML applications in clouds and data-centers. We identify three important classes of distributed ML techniques—stateful workers, model scheduling, and relaxed consistency—and designed Litz's programming model to collectively support each of them. By adopting an event-driven API, Litz is able to control the execution of its applications, transparently migrating their state and computation between physical machines. Litz achieves elasticity—the ability to scale out and in based on changing resource availability—without compromising the state-of-the-art efficiency of non-elastic ML systems.

Furthermore, we describe the inherent dynamic memory usage of ML applications. We show that Litz is able to expose these patterns and significantly decrease its demand for memory across the lifetimes of ML jobs. 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 [32, 24, 23]. However, no previous work exists that exploit the specific resource usage patterns of ML applications. In future work, we plan to further investigate and identify the resource usage patterns of distributed ML applications, and then leverage their resource variability together with the elasticity of Litz for more efficient scheduling of ML jobs.

Lastly, we identify deep learning and elastic GPU computing as another interesting direction for future work. In particular, how does the relatively low-level event-driven API of Litz fit together with the higher-level symbolic programming models of deep learning frameworks like TensorFlow, MXNet [14], and DyNet [40]? With the current trend towards using compiler techniques to separate deep learning programming and execution [6, 7], we believe that frameworks like Litz will play an important role in the elastic and efficient execution of many future deep learning applications. The answers to these problems deserve thorough investigation.

Acknowledgements

We thank the anonymous reviewers for their valuable feedback. We thank the members and companies of the PDL Consortium: Alibaba Group, Broadcom, Dell EMC, Facebook, Google, HP Enterprise, Hitachi, IBM Research, Intel, Micron, Microsoft Research, MongoDB, NetApp, Oracle, Salesforce, Samsung, Seagate Technology, Two Sigma, Toshiba, Veritas and Western Digital for their interest, insights, feedback, and support. Our work was supported by the U.S. National Science Foundation awards IIS1447676 and CCF1629559, the Natural Sciences and Engineering Research Council of Canada award PGSD-471301-2015, as well as the Beijing Advanced Innovation Center for Big Data and Brain Computing at Beihang University.

References

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

[2] Boost Context. www.boost.org/doc/libs/1_63_0/libs/context/.

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

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

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

[6] NNVM. http://nnvm.tvmlang.org/.

[7] XLA. https://www.tensorflow.org/performance/xla/.

[8] ZeroMQ. http://zeromq.org.

[9] ABADI, M., BARHAM, P., CHEN, J., CHEN, Z., DAVIS, A., DEAN, J., DEVIN, M., GHEMAWAT, S., IRVING, G., ISARD, M., ET AL. Tensorflow: A system for large-scale machine learning.

[10] AGARWAL, A., AND DUCHI, J. C. Distributed delayed stochastic optimization. In *Advances in Neural Information Processing Systems* (2011), pp. 873–881.

[11] AHAMAD, M., NEIGER, G., BURNS, J. E., KOHLI, P., AND HUTTO, P. W. Causal memory: definitions, implementation, and programming. *Distributed Computing 9*, 1 (Mar 1995), 37–49.

[12] AHN, S., SHAHBABA, B., WELLING, M., ET AL. Distributed stochastic gradient mcmc. In *ICML* (2014), pp. 1044–1052.

[13] BAILIS, P., VENKATARAMAN, S., FRANKLIN, M. J., HELLERSTEIN, J. M., AND STOICA, I. Probabilistically bounded staleness for practical partial quorums. *Proc. VLDB Endow. 5*, 8 (Apr. 2012), 776–787.

[14] CHEN, T., LI, M., LI, Y., LIN, M., WANG, N., WANG, M., XIAO, T., XU, B., ZHANG, C., AND ZHANG, Z. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274* (2015).

[15] CHILIMBI, T., SUZUE, Y., APACIBLE, J., AND KALYANARAMAN, K. Project adam: Building an efficient and scalable deep learning training system. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)* (Broomfield, CO, Oct. 2014), USENIX Association, pp. 571–582.

[16] CIPAR, J., HO, Q., KIM, J. K., LEE, S., GANGER, G. R., GIBSON, G., KEETON, K., AND XING, E. Solving the straggler problem with bounded staleness. In *Presented as part of the 14th Workshop on Hot Topics in Operating Systems* (Berkeley, CA, 2013), USENIX.

[17] DAI, W., KUMAR, A., WEI, J., HO, Q., GIBSON, G., AND XING, E. P. High-performance distributed ml at scale through parameter server consistency models. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence* (2015), AAAI'15, AAAI Press, pp. 79–87.

[18] DEAN, J., CORRADO, G., MONGA, R., CHEN, K., DEVIN, M., MAO, M., SENIOR, A., TUCKER, P., YANG, K., LE, Q. V., ET AL. Large scale distributed deep networks. In *Advances in neural information processing systems* (2012), pp. 1223–1231.

[19] GABRILOVICH, E., RINGGAARD, M., AND SUBRAMANYA, A. Facc1: Freebase annotation of clueweb corpora, version 1 (release date 2013-06-26, format version 1, correction level 0). http://lemurproject.org/clueweb12/, 2013.

[20] GEMULLA, R., NIJKAMP, E., HAAS, P. J., AND SISMANIS, Y. Large-scale matrix factorization with distributed stochastic gradient descent. In *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (New York, NY, USA, 2011), KDD '11, ACM, pp. 69–77.

[21] GHEMAWAT, S., AND MENAGE, P. TCMalloc : Thread-Caching Malloc. http://goog-perftools.sourceforge.net/doc/tcmalloc.html.

[22] GOODFELLOW, I., BENGIO, Y., AND COURVILLE, A. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[23] GRANDL, R., CHOWDHURY, M., AKELLA, A., AND ANANTHANARAYANAN, G. Altruistic scheduling in multi-resource clusters. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)* (GA, 2016), USENIX Association, pp. 65–80.

[24] GRANDL, R., KANDULA, S., RAO, S., AKELLA, A., AND KULKARNI, J. Graphene: Packing and dependency-aware scheduling for data-parallel clusters. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)* (GA, 2016), USENIX Association, pp. 81–97.

[25] HARLAP, A., CUI, H., DAI, W., WEI, J., GANGER, G. R., GIBBONS, P. B., GIBSON, G. A., AND XING, E. P. Addressing the straggler problem for iterative convergent parallel ml. In *Proceedings of the Seventh ACM Symposium on Cloud Computing* (New York, NY, USA, 2016), SoCC '16, ACM, pp. 98–111.

[26] HARLAP, A., TUMANOV, A., CHUNG, A., GANGER, G., AND GIBBONS, P. Proteus: agile ml elasticity through tiered reliability in dynamic resource markets. In *Proceedings of the Eleventh European Conference on Computer Systems* (New York, NY, USA, 2017), EuroSys '17, ACM.

[27] HINDMAN, B., KONWINSKI, A., ZAHARIA, M., GHODSI, A., JOSEPH, A. D., KATZ, R., SHENKER, S., AND STOICA, I. 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* (Berkeley, CA, USA, 2011), NSDI'11, USENIX Association, pp. 295–308.

[28] HO, Q., CIPAR, J., CUI, H., LEE, S., KIM, J. K., GIBBONS, P. B., GIBSON, G. A., GANGER, G., AND XING, E. P. More effective distributed ml via a stale synchronous parallel parameter server. In *Advances in Neural Information Processing Systems 26*, C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Weinberger, Eds. Curran Associates, Inc., 2013, pp. 1223–1231.

[29] HONG, M. A distributed, asynchronous and incremental algorithm for nonconvex optimization: An admm based approach. *arXiv preprint arXiv:1412.6058* (2014).

[30] HUANG, Q., SU, S., XU, S., LI, J., XU, P., AND SHUANG, K. Migration-based elastic consolidation scheduling in cloud data center. In *2013 IEEE 33rd International Conference on Distributed Computing Systems Workshops* (July 2013), pp. 93–97.

[31] HUNT, P., KONAR, M., JUNQUEIRA, F. P., AND REED, B. Zookeeper: Wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference* (Berkeley, CA, USA, 2010), USENIXATC'10, USENIX Association, pp. 11–11.

[32] JYOTHI, S. A., CURINO, C., MENACHE, I., NARAYANAMURTHY, S. M., TUMANOV, A., YANIV, J., MAVLYUTOV, R., GOIRI, I., KRISHNAN, S., KULKARNI, J., AND RAO, S. Morpheus: Towards automated slos for enterprise clusters. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)* (GA, 2016), USENIX Association, pp. 117–134.

[33] KIM, J. K., HO, Q., LEE, S., ZHENG, X., DAI, W., GIBSON, G. A., AND XING, E. P. Strads: A distributed framework for scheduled model parallel machine learning. In *Proceedings of the Eleventh European Conference on Computer Systems* (New York, NY, USA, 2016), EuroSys '16, ACM, pp. 5:1–5:16.

[34] KRIZHEVSKY, A. Learning multiple layers of features from tiny images.

[35] KUMAR, A., BEUTEL, A., HO, Q., AND XING, E. P. Fugue: Slow-worker-agnostic distributed learning for big models on big data. In *AISTATS* (2014), pp. 531–539.

[36] LI, M., ANDERSEN, D. G., PARK, J. W., SMOLA, A. J., AHMED, A., JOSIFOVSKI, V., LONG, J., SHEKITA, E. J., AND SU, B.-Y. Scaling distributed machine learning with the parameter server. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)* (Broomfield, CO, Oct. 2014), USENIX Association, pp. 583–598.

[37] LI, M., ANDERSEN, D. G., AND SMOLA, A. Distributed delayed proximal gradient methods. In *NIPS Workshop on Optimization for Machine Learning* (2013).

[38] LOW, Y., BICKSON, D., GONZALEZ, J., GUESTRIN, C., KYROLA, A., AND HELLERSTEIN, J. M. Distributed graphlab: A framework for machine learning and data mining in the cloud. *Proc. VLDB Endow. 5*, 8 (Apr. 2012), 716–727.

[39] MCMAHAN, B., AND STREETER, M. Delay-tolerant algorithms for asynchronous distributed online learning. In *Advances in Neural Information Processing Systems* (2014), pp. 2915–2923.

[40] NEUBIG, G., DYER, C., GOLDBERG, Y., MATTHEWS, A., AMMAR, W., ANASTASOPOULOS, A., BALLESTEROS, M., CHIANG, D., CLOTHIAUX, D., COHN, T., ET AL. Dynet: The dynamic neural network toolkit. *arXiv preprint arXiv:1701.03980* (2017).

[41] PUNDIR, M., KUMAR, M., LESLIE, L. M., GUPTA, I., AND CAMPBELL, R. H. Supporting on-demand elasticity in distributed graph processing. In *Cloud Engineering (IC2E), 2016 IEEE International Conference on* (2016), IEEE, pp. 12–21.

[42] RECHT, B., RE, C., WRIGHT, S., AND NIU, F. Hogwild!: A lock-free approach to parallelizing stochastic gradient descent. In *Advances in Neural Information Processing Systems 24*, J. Shawe-Taylor, R. Zemel, P. Bartlett, F. Pereira, and K. Weinberger, Eds. Curran Associates, Inc., 2011, pp. 693–701.

[43] RUSSAKOVSKY, O., DENG, J., SU, H., KRAUSE, J., SATHEESH, S., MA, S., HUANG, Z., KARPATHY, A., KHOSLA, A., BERNSTEIN, M., BERG, A. C., AND FEI-FEI, L. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV) 115*, 3 (2015), 211–252.

[44] SCHERRER, C., TEWARI, A., HALAPPANAVAR, M., AND HAGLIN, D. Feature clustering for accelerating parallel coordinate descent. In *Advances in Neural Information Processing Systems* (2012), pp. 28–36.

[45] SCHWARZKOPF, M., KONWINSKI, A., ABD-EL-MALEK, M., AND WILKES, J. Omega: flexible, scalable schedulers for large compute clusters. In *SIGOPS European Conference on Computer Systems (EuroSys)* (Prague, Czech Republic, 2013), pp. 351–364.

[46] SHARMA, P., GUO, T., HE, X., IRWIN, D., AND SHENOY, P. Flint: Batch-interactive data-intensive processing on transient servers. In *Proceedings of the Eleventh European Conference on Computer Systems* (New York, NY, USA, 2016), EuroSys '16, ACM, pp. 6:1–6:15.

[47] SUBRAMANYA, S., GUO, T., SHARMA, P., IRWIN, D., AND SHENOY, P. Spoton: A batch computing service for the spot market. In *Proceedings of the Sixth ACM Symposium on Cloud Computing* (New York, NY, USA, 2015), SoCC '15, ACM, pp. 329–341.

[48] WANG, J., YANG, J., YU, K., LV, F., HUANG, T., AND GONG, Y. Locality-constrained linear coding for image classification. In *IN: IEEE CONFERENCE ON COMPUTER VISION AND PATTERN CLASSIFICATOIN* (2010).

[49] WANG, M., XIAO, T., LI, J., ZHANG, J., HONG, C., AND ZHANG, Z. Minerva: A scalable and highly efficient training platform for deep learning. In *NIPS Workshop, Distributed Machine Learning and Matrix Computations* (2014).

[50] WEI, J., DAI, W., QIAO, A., HO, Q., CUI, H., GANGER, G. R., GIBBONS, P. B., GIBSON, G. A., AND XING, E. P. Managed communication and consistency for fast data-parallel iterative analytics. In *Proceedings of the Sixth ACM Symposium on Cloud Computing* (New York, NY, USA, 2015), SoCC '15, ACM, pp. 381–394.

[51] WEI, J., KIM, J. K., AND GIBSON, G. A. Benchmarking Apache Spark with Machine Learning Applications. In *Carnegie Mellon University Parallel Data Lab Technical Report CMU-PDL-16-107, Oct. 2016*.

[52] XIE, P., KIM, J. K., ZHOU, Y., HO, Q., KUMAR, A., YU, Y., AND XING, E. P. Distributed machine learning via sufficient factor broadcasting. *CoRR abs/1511.08486* (2015).

[53] XING, E. P., HO, Q., DAI, W., KIM, J. K., WEI, J., LEE, S., ZHENG, X., XIE, P., KUMAR, A., AND YU, Y. Petuum: A new platform for distributed machine learning on big data. *IEEE Trans. Big Data 1*, 2 (2015), 49–67.

[54] YAN, Y., GAO, Y., CHEN, Y., GUO, Z., CHEN, B., AND MOSCIBRODA, T. Tr-spark: Transient computing for big data analytics. In *Proceedings of the Seventh ACM Symposium on Cloud Computing* (New York, NY, USA, 2016), SoCC '16, ACM, pp. 484–496.

[55] YUAN, J., GAO, F., HO, Q., DAI, W., WEI, J., ZHENG, X., XING, E. P., LIU, T.-Y., AND MA, W.-Y. Lightlda: Big topic models on modest computer clusters. In *Proceedings of the 24th International Conference on World Wide Web* (2015), ACM, pp. 1351–1361.

[56] YUN, H., YU, H.-F., HSIEH, C.-J., VISHWANATHAN, S., AND DHILLON, I. Nomad: Non-locking, stochastic multi-machine algorithm for asynchronous and decentralized matrix completion. *Proceedings of the VLDB Endowment 7*, 11 (2014), 975–986.

[57] ZAHARIA, M., CHOWDHURY, M., FRANKLIN, M. J., SHENKER, S., AND STOICA, I. Spark: Cluster Computing with Working Sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing* (Berkeley, CA, USA, 2010), HotCloud'10, USENIX Association, pp. 10–10.

[58] ZHANG, R., AND KWOK, J. T. Asynchronous distributed admm for consensus optimization. In *ICML* (2014), pp. 1701–1709.