Addressing Algorithmic Bottlenecks in Elastic Machine Learning with Chicle

Michael Kaufmann [1 2] **Kornilios Kourtis** [1] **Celestine Mendler-Dünner** [3] **Adrian Schüpbach** [1] **Thomas Parnell** [1]

Abstract

Distributed machine learning training is one of the most common and important workloads running on data centers today, but it is rarely executed alone. Instead, to reduce costs, computing resources are consolidated and shared by different applications. In this scenario, elasticity and proper load balancing are vital to maximize efficiency, fairness, and utilization. Currently, most distributed training frameworks do not support the aforementioned properties. A few exceptions that do support elasticity, imitate generic distributed frameworks and use micro-tasks.

In this paper we illustrate that micro-tasks are problematic for machine learning applications, because they require a high degree of parallelism which hinders the convergence of distributed training at a pure algorithmic level (i.e., ignoring overheads and scalability limitations). To address this, we propose Chicle, a new elastic distributed training framework which exploits the nature of machine learning algorithms to implement elasticity and load balancing without micro-tasks. We use Chicle to train deep neural network as well as generalized linear models, and show that Chicle achieves performance competitive with state of the art rigid frameworks, while efficiently enabling elastic execution and dynamic load balancing.

1 Introduction

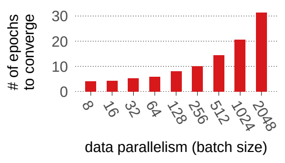
The ever-growing amounts of data are fueling impressive advances in machine learning (ML), but depend on substantial computational power to train the corresponding models. As a result, many research works focus on addressing *scalability* of distributed training across multiple machines. State of the art algorithms include Mini-batch SGD (mSGD) (Robbins & Monro, 1951; Kiefer et al., 1952; Rumelhart et al., 1988) and Local SGD (lSGD) (Lin et al., 2018) for deep neural networks (DNNs) as well as Communication-efficient distributed dual Coordinate Ascent (CoCoA) (Jaggi et al., 2014; Smith et al., 2018) for generalized linear models (GLMs).

Less work, however, has focused on *efficiency*, which is equally (if not more) important because it effectively provides more computational power at the same cost. Indeed, most works on distributed ML assume that they can operate on dedicated clusters, which is rarely the case in practice where ML applications co-inhabit common infrastructure with other applications. In these shared environments, efficiency depends on two properties: *elastic execution*: dynamically adjusting resource (e.g., CPUs, GPUs, nodes) usage as their availability changes, and *load balancing*: distributing workload across heterogeneous resources (Ou et al., 2012; Delimitrou & Kozyrakis, 2014) such that faster resources do not have to wait for slower ones. Elastic execution, specifically, enables optimization opportunities for ML applications where scaling-in or -out as training progresses can increase accuracy and reduce training time (Kaufmann et al., 2018).

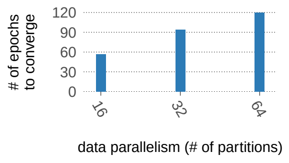
As of today, most ML distributed frameworks (e.g., Abadi et al. (2016); Paszke et al. (2017)) do not support elastic execution nor load balancing, which makes them inherently inefficient in shared environments and on heterogeneous clusters. Recently, recognizing the importance of elasticity, a number of systems attempt to address elasticity (Zhang et al., 2017; Harlap et al., 2017; Qiao et al., 2018) for ML applications using micro-tasks or similar mechanisms. Micro-tasks, where work is split up into a large number of short tasks executed as resources become available, have been extensively used in generic distributed application frameworks to address elasticity and load balancing (Zaharia et al., 2010; Ousterhout et al., 2013), so they seem a natural fit for this problem.

In this paper we argue that micro-tasks are ill-suited for ML training because they require a large number of short independent tasks for efficient scheduling. In order to support full system utilization, the number of tasks has to be chosen based on the largest possible degree of parallelism an elastic system could potentially experience. The number

[1]IBM Research, Zurich, Switzerland [2]Karlsruhe Institute of Technology, Karlsruhe, Germany [3]UC Berkeley, work conducted while at IBM Research. Correspondence to: Michael Kaufmann <kau@zurich.ibm.com>.



(a) CIFAR-10 (mSGD/CNN)



(b) Criteo (CoCoA/SVM)

Figure 1. Example of the correlation between data parallelism and the number of epochs needed to achieve a certain training goal.

of tasks, in turn, constitutes a lower bound on the data parallelism of each update which means that you need to pick the mini-batch size in mSGD[1] or the number of partitions in CoCoA accordingly. This however is not a desirable thing to do from an algorithmic point of view, since it is widely acknowledged that data parallelism comes at the cost of convergence in distributed ML applications. Note that when talking about convergence we refer to epochs to converge, where an epoch refers to one pass through the entire dataset.

Extensive studies of this impact for mSGD have, among others, been conducted by Shallue et al. (2019), Keskar et al. (2016) and Goyal et al. (2017). Figures 1a and 1b also exemplify this. The training of a simple convolutional neural network (CNN) on the CIFAR-10 dataset using mSGD requires 44% more epochs to converge when increasing the batch size from 256 to 512. Similarly, doubling the number of partitions from 16 to 32 for the training of a on the Criteo dataset using CoCoA (Jaggi et al., 2014; Smith et al., 2018) increases the number of epochs to converge by 65%. While mitigation strategies, such as warm-up (Goyal et al., 2017) and layer-wise adaptive rate scaling (You et al., 2017) exist, the fundamental problem remains. Overall, micro-tasks lead to an inherent conflict between the number of tasks to use for scheduling efficiency, where higher is better, and algorithmic ML training efficiency, where lower is better.

Fortunately, as we show in this paper, the iterative nature of ML applications allows implementing load balancing and elasticity without micro-tasks, thus eliminating the above inherent conflict. We realize our ideas in Chicle[2], an elastic, load balancing distributed framework for iterative-convergent ML training applications. Chicle combines scheduling flexibility with the efficiency of special-purpose *rigid* ML training frameworks. Chicle uses uni-tasks and schedules (stateful) data chunks instead of tasks. Each node executes only a single (multi-threaded) task that processes training samples from multiple data chunks within a single execution context. Data chunks can be moved efficiently between tasks to balance load and to scale in and out. This allows Chicle to use the optimal level of data parallelism for the currently used number of resources and combines scheduling with algorithmic efficiency. Conversely, Chicle is able to efficiently adjust the resource allocation based on feedback from the training algorithm and resource availability. The main contributions of our work are:

1) We propose *uni-tasks*, a new task model that removes the conflict between scheduling and algorithmic efficiency. We implement a prototype thereof in Chicle, a distributed ML framework that enables elastic training and dynamic load balancing in heterogeneous clusters.

2) Our evaluation illustrates that uni-tasks require significantly fewer epochs, and subsequently less time, to converge in elastic and load-balancing scenarios compared to micro-tasks.

Our paper is structured as follows: First, we provide necessary background information on the relationship between data parallelism and convergence for ML training algorithms as well as requirements for elastic execution in §2 followed by a discussion of the main ideas behind uni-tasks (§3). We continue with a detailed description of Chicle's design and implementation (§4) and present results of our experimental evaluation (§5) and conclude (§6).

2 Background & Motivation

Increasing parallelism for distributed execution of ML training workloads has well-understood tradeoffs. On one hand, ample parallelism results in less work per each independent execution unit (*task*) which leads to increased overheads (Totoni et al., 2017). On the other hand, ample parallelism allows utilizing many nodes and enables efficient scheduling (Ousterhout et al., 2013), dealing with load imbalances, and supporting elasticity. Elasticity specifically is increasingly important, since to maximize the efficiency, distributed applications are expected to scale-in and -out based on workload demands of themselves and their cohabitants. Indeed, exposing ample parallelism by dividing the problem into many micro-tasks is the standard way to implement elasticity despite the resulting execution overheads. Litz (Qiao et al., 2018), for example, a recent ML elastic framework uses micro-tasks and reports up to 23% of execution overhead.

While these overheads are important, our work is motivated by another tradeoff that is specific to ML applications but not well recognized in the ML systems community: increased data parallelism hinders the convergence of ML training. In contrast to overheads, this problem exists purely at the algorithmic level. Generally, distributed training algorithms require more steps to converge in the face of high paral-

[1]The mini-batch size needs to be chosen as a multiple of the number of tasks in order to keep relative job overheads low.

[2]Chicle is the Mexican-Spanish word for latex from the sapodilla tree that is used as basis for chewing gum and a reference to Chicle's elasticity.

lelism (Shallue et al., 2019). The implication for building elastic ML frameworks is that using micro-tasks, i.e., ample parallelism to gain scheduling flexibility, leads to an inherent trade-off in terms of the number of the examples that need to be processed to converge to a solution.

In this section, we motivate our design by illustrating this issue in two different ML algorithms: Mini-batch stochastic gradient descent (SGD), extensively used to train neural networks, and CoCoA, a state-of-the art framework for distributed training of GLMs. Prior to that, we provide some necessary background on elastic scheduling and ML training.

2.1 Elasticity and load balancing

Both *load balancing* and elasticity are necessary to efficiently utilize shared infrastructure. Both are typically implemented using micro-tasks in generic analytics frameworks, such as Spark (Zaharia et al., 2010) and ML frameworks (Qiao et al., 2018; Zhang et al., 2017), where work is divided into a large number of tasks that are distributed among nodes. Tasks, i.e., self-contained, atomic entities of a function and input data, are a common abstraction of work, and represent the scheduling unit.

Under a task scheduling system, a large number of tasks are required to achieve efficiency. To allow elastic scale-out during training, the number of tasks needs to be at least as large as the maximum number of nodes that will be available at any point during training. Furthermore, common practice over-provisions nodes with many tasks per node to allow for efficient load balancing. The Spark tuning guidelines (Spark, 2019), for instance, recommend to use of up to 2–3 tasks per available CPU, while other works propose using millions of tasks (Ousterhout et al., 2013).

2.2 Distributed training algorithms

Next, we discuss training in general and introduce two training algorithms that we use in this paper. Most distributed training algorithms iteratively refine a model $\mathbf{m}$ on a training dataset D such that $\mathbf{m}$ converges towards a state that minimizes or maximizes an objective function. During each iteration i, an updated model $\mathbf{m}^{(i)}$ is computed on a randomly chosen subset $\widehat{D} \subseteq D$:

$$\mathbf{m}^{(i)} = \mathbf{m}^{(i-1)} + f_\Delta(\mathbf{m}^{(i-1)}, \widehat{D}) \qquad (1)$$

The update function f_Δ is computed in a data parallel manner across K nodes by splitting up $\widehat{D}$ into K disjoint partitions $D_k \subseteq \widehat{D}$.

$$f_\Delta(\mathbf{m}^{(i-1)}, \widehat{D}) = \frac{1}{K} \sum_{k=1}^{K} f_{\Delta,k}(\mathbf{m}^{(i-1)}, D_k) \qquad (2)$$

The computation of f_Δ is self-correcting to a certain degree, i.e., bounded errors are averaged out in subsequent iterations, and can therefore be tolerated. This property is often exploited for ML-specific optimizations, e.g., to mitigate stragglers (Cipar et al., 2013; Cui et al., 2014; Dutta et al., 2018; Ho et al., 2013). The general structure of the algorithms we are considering is depicted in Figure 2: K workers independently work on separate subproblems $f_{\Delta,k}$, each defined on a different partition D_k of the data and then combine their results to update a global model $\mathbf{m}$, which forms the basis of the next iteration. During each iteration, a worker processes $H \times L$ samples, of which H different sets of L independent samples are processed sequentially. After each set of L samples, a local model update is performed, such that learning on subsequent samples within an iteration can exploit knowledge gained so far.

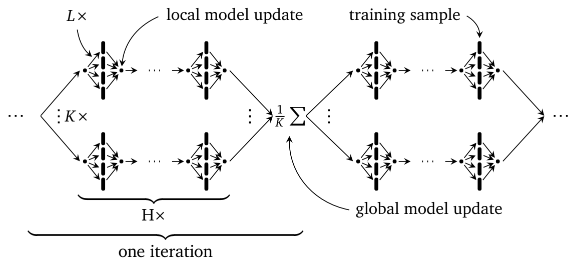


Figure 2. General structure of distributed ML algorithms we consider in this paper.

While our approach is applicable to a wide set of distributed ML training algorithms, in this paper, we focus on the following two algorithms.

Local SGD (Lin et al., 2018). A state-of-the-art algorithm and improvement upon mSGD, the de-facto standard for training of neural networks (NNs) and variants thereof. Here, $\widehat{D}$ refers to the *batch* and $|\widehat{D}| = H \times L$ refers to the batch size hyper-parameter (e.g. $|\widehat{D}| = 64$). For, $H = 1$ lSGD degrades to mSGD. The negative effect of data parallelism on the convergence of is a fundamental property of mSGD. An extensive study of this property is presented by Shallue et al. (2019).

CoCoA (Jaggi et al., 2014; Smith et al., 2018). A state-of-the-art distributed framework for the training of GLMs. It is designed to reduce communication and thus processes significantly more samples per iteration than, e.g., mSGD. We use CoCoA with a local stochastic coordinate descent (SCD) solver (Wright, 2015). The structure in Figure 2 is parameterized with $L = 1$, $H = |\widehat{D}|$, whereas $\widehat{D} = D$. The local update function $f_{\Delta,k}$ is computed by a local optimizer on partitions D_k, with $D = \bigcup_{k=1}^{K} D_k$. In a homogeneous setting each node typically processes $1/K$-th of the training dataset per iteration. Data parallelism is determined by the number of partitions K. Local optimizers detect correlations within the local dataset without global communication, i.e., the more data is randomly accessible to each optimizer instance, the less epochs are needed for CoCoA to converge. Conversely, if data access is limited in size, as would be the case when using many tasks, or if no random access is possible, convergence suffers. Kaufmann et al. (2018) empirically study the relationship between convergence rate and K and show that by starting with a large K and reducing it after a few iterations, convergence rate per epoch and time can be increased significantly.

Summary. Both algorithms exhibit an inherent trade-off between data parallelism and convergence. Intuitively, a higher degree of parallelism limits the opportunity to learn correlations across samples, and thus hurts convergence. While we focus on two particular methods in this paper, the trade-off between parallelism and convergence is fundamental in parallel stochastic algorithms.

2.3 Micro-tasks for distributed training

As exemplified in Figure 1, increasing data parallelism comes at the cost of increasing the total amount of work to achieve a certain training goal. Up until a point, the cost increase is smaller than the gain in potential parallelism, such that overall training time can be reduced by increasing the data parallelism. This, however, is only true if and only if all tasks are executed in parallel. In shared and heterogeneous environments, this is generally not true.

Consider the CIFAR-10 example from Figure 1. For simplicity, we assume perfect linear scaling and zero system overheads. If one wanted to train on up to 256 nodes, at least 256 tasks are required and thus a data parallelism of 256 or higher. According to the data in Figure 1, this requires 10 epochs to converge. Assuming that one epoch in this configuration – where all 256 tasks can run in parallel – requires one second, training completes after 10 seconds. The nature of shared systems is, however, that there are not always enough nodes available to execute all tasks in parallel. For instance, let us assume that only 128 nodes are available during the runtime of the application. Then each epoch with 256 tasks requires two seconds as two tasks have to run back to back on each node, resuling in a total training time of 20s. If one had used a data parallelism of only 128 from the beginning, instead of 256, training would only require eight epochs or 16s, instead of 20s, resulting in a training time reduction of 20%. This example illustrates the difficulty of elastic scaling of ML training using a micro-task-based system: In many cases, it is only efficient if the maximal number of nodes (resources) are actually available during most of the runtime. This, however, stands in contrast to the goals of elasticity. This problem is even more pronounced if we also consider load balancing between differently fast nodes. The number of tasks required to allow for fine-granular work redistribution is disproportionately higher than just for elastic scaling alone.

3 UNI-TASKS FOR DISTRIBUTED TRAINING

In the previous section, we showed how micro-tasks inhibit the performance of distributed training. In this section we argue that a different execution model, uni-tasks, is better suited for ML training applications. The core idea is very simple: to only use a single task per node. While this in itself is not a new concept, scientific computing has been using MPI that follows this approach for decades, the difficulty is to address the scheduling challenges that are typically addressed by micro-tasks, namely elasticity and load-balancing. Fortunately, we can exploit the iterative nature of ML training to tackle these challenges.

Core concepts. Uni-tasks consists of two main concepts: immobile tasks and mobile data chunks.

1. All training samples are stored across a large set of small fixed-sized (stateful) data chunks that can be moved between tasks by the scheduler. Data chunks can store dense and sparse training data vectors and matrices of variable size.
2. Each node only executes a single task per node (hence the name uni-tasks). Each task has full, random access to all training samples across all data chunks that are local to a task.

Additionally, a contract between the scheduler and the application is defined that regulates ownership of a data chunk.

1. During an iteration, a task owns all task-local data chunks. It can read all and make modifications to data stored in the data chunks, e.g., to update per-sample state (e.g., as needed in CoCoA). During this period, the scheduler does not add or remove data chunks.
2. In-between two iterations, the scheduler owns all data chunks. Tasks must not modify any data chunks and the scheduler is free to add or remove data chunks from any task. Tasks are notified by the scheduler of any data chunk addition or removal.

By moving data chunks between tasks in-between iterations, uni-tasks allows one to add and remove tasks for elastic scaling and to balance load across tasks on heterogeneous clusters. Uni-tasks assumes a correlation between the number of training samples in task-local data chunks and the number of samples processed by each task during each iteration. In contrast to micro-tasks, scheduling granularity is determined by the number of data chunks, not by the number of tasks. The number of data chunks does not constitute a lower bound for the level of data parallelism, as multiple data chunks are processed by the same task, hence the number level of data parallelism can be lower than the number of data chunks. In contrast to MPI, uni-tasks defines a method to shift load between tasks.

In the following paragraphs, we discuss how elasticity and load balancing are addressed for distributed training when using uni-tasks.

Elasticity. Elasticity is necessary to efficiently and fairly utilize resources in shared clusters, to reduce waiting times for job starts, and to react do varying resource demands of applications throughout their runtime. We address elasticity in the uni-tasks setting by spawning new tasks as nodes are added to the application and by terminating them if nodes need to be released. In both cases data chunks are redistributed across all available tasks. In the latter case, however, a prior notification is required such that data chunks can be transferred before the task is terminated. Elastic scaling is only possible in-between iterations.

The application is free to adjust the level of data parallelism during each iteration to any value equal or larger than the number of tasks. For both test applications, we always choose the lowest possible value.

Load balancing. Load balancing is necessary to deal with heterogeneity between cluster nodes as well as between different hardware (e.g., CPUs vs GPUs) that results in runtime differences between tasks that process the same amount of input data.

To address heterogeneity, we exploit the fact ML training algorithms are typically iterative and process a known amount of training samples during each iteration, which allows us to learn how long each task needs to process a training sample. Uni-tasks assumes that the number of training samples processed by each task is a fraction of the total number of training samples across all task-local data chunks, e.g., a task with twice as many training samples as another task also processes twice as many per iteration. This enables the scheduler to influence the runtime by moving data chunks from tasks on slower to tasks on faster nodes until their runtime aligns.

As tasks may process a different number of training samples during each iteration, their model updates need to be weighted differently as well (as proposed in Stich (2018)). We do this by multiplying the model update $f_{\Delta,k}$ of task k by $D_k/\widehat{D}$ (see Equation 2).

4 Chicle design and implementation

Here, we describe how Chicle implements an elastic distributed training framework using uni-tasks.

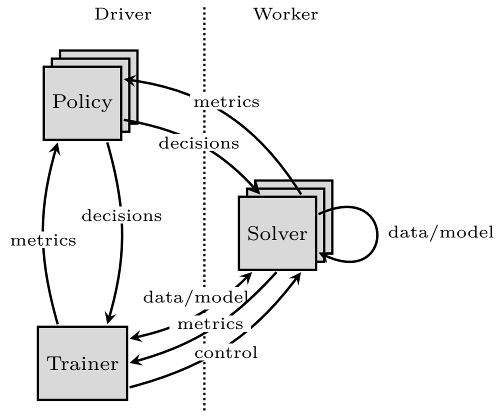


Figure 3. High-level architecture of Chicle.

4.1 Overview

Chicle, as shown in Figure 3, is based on a driver/worker design with a central driver (*trainer*) and multiple workers (*solvers*) communicating via a RDMA-based RPC mechanism (see §4.3). The driver executes the *trainer* module, which, in tandem with multiple policy modules, is responsible for coordinating training. Policy modules make scheduling decisions, such as assigning chunks, balancing load, and scaling in and out. Worker processes execute *solver* modules (uni-tasks) and implement the ML algorithms (e.g., SCD for CoCoA). Crucially, only a single (multi-threaded) worker process is executed per node. Solvers are controlled by the trainer and policy modules, which in turn receive model and state updates as well as metrics (e.g., duality-gap).

Chicle applications need to implement a trainer and solver module, and may optionally implement policy modules to control system behavior during training. For instance, our lSGD implementation uses libtorch (from PyTorch (Paszke et al., 2017)) in the solver for forward and backward propagation steps. The trainer module acts as synchronous parameter server that merges updates from solver instances. A simplified version of the lSGD code is shown in Listing 1.

In the remainder of this section, we elaborate on each module as well as the communication subsystem and in-memory data format of Chicle.

4.2 Trainer and solver

The trainer and solver modules represent application code. Trainer modules are the central controlling entity and coordinate individual solver instances in tandem with policy modules. Policy modules can implement complex (reusable) optimizations (e.g., online hyper-parameter tuning), and solver modules implement arbitrary functions for distributed execution. Only a single solver module is executed per node and application, therefore, each solver module can internally spawn threads and use all CPUs or GPUs of a node. Trainer and solver modules periodically synchronize at global barriers, e.g. in-between iterations, but can exchange additional messages at any time.

4.3 Communication subsystem

In distributed training, communication can easily become a bottleneck. For example, using CoCoA to train a model for the Criteo dataset (see Table 1), each task has to send/receive ≈16MiB in updates in-between iterations. For that reason, we built our communication subsystem on RDMA. RDMA allows low-overhead, zero-copy, one-sided operations for bulk data transfers, such as model and training (input) data as well as two-sided remote procedure calls (RPCs) using RDMA send/receive.

4.4 In-memory chunk data format

To fully exploit RDMA, data is stored in static, consecutive memory regions. The in-memory representation of training (input) data is based on fixed-sized data chunks. Chunks can store sparse or dense training data vectors and matrices. The number of training samples per data chunk can vary depending on their size. Chunks allow to easily move training data subsets between nodes. The chunk size can be tuned to an optimal value depending on dataset and system properties, e.g. to the CPU cache size.

Chicle's in-memory format is application agnostic and simply provides applications a contiguous memory space that can be moved across nodes in-between iterations. For instance, our lSGD implementation stores the backing memory of native PyTorch tensor objects in data chunks wheras for CoCoA, we simply store sparse vectors as well as per-sample state in a data chunk. Having the ability to store per-sample state in a data chunk is important as it ensures that state and the data it correlates to are always moved together.

One important limitation of Chicle's data chunk is that they must not require any serialization, as one-sided RDMA read operations are used to transfer them. Deserialization is possible. In the case of PyTorch, for instance, we restore tensor objects via the `torch::from_blob` function, which creates a new tensor object backed by the in-chunk data.

4.5 Policies

Chicle implements a flexible policy framework which we use to implement vital parts of the system. Policies make decisions based on events and metrics they receive from trainer and solver modules and return proper decisions for them. Each policy module runs in a separate thread and multiple policy modules can run at the same point in time. Policy modules coordinate with the trainer and can coordinate with each other. Next, we present the most relevant policy modules.

Elastic scaling policy. This module interfaces with the resource manager, e.g., YARN (Vavilapalli et al., 2013), to make resource requests and get resource assignment and revocation notices. Upon receiving a new resource assignment, it registers a new worker (task) and notifies the trainer. After the current iteration, it shifts data chunks from old to new workers. It relies on the rebalancing policy (§4.5) to ensure proper load balancing. Chicle expects the resource manager to give advance notice before revoking a resource allocation. Upon receiving such a notice, it redistributes data chunks from to-be freed workers to remaining ones in a round robin fashion. As before, it relies on the rebalancing policy to ensure load balance.

Rebalancing policy. The rebalance policy observes iteration runtimes over multiple iterations to learn the per-sample runtime of each task, as described above. Between iterations, solvers are ranked according to their median performance over the last I iterations and chunks moved gradually, across multiple iterations, from slower to faster solvers until performance differences are smaller than the estimated processing time of a single chunk. This policy can also be used to address slowly changing performance of nodes, e.g. ones that are caused by the start/end of long running background jobs and restore balance after scaleing in and out. Its robustness against runtime fluctuations can be adjusted by tweaking I.

We decided against reloading data from a (shared) filesystem as data loading turned out to be more expensive than transferring loaded data between nodes, especially if input files are stored on a shared network filesystem. Moreover, our in-memory format can combine data chunks with the corresponding state, which needs to be transferred between workers anyway.

Other policies. Apart from the above described policies, we have implemented policies for straggler mitigation, global background data shuffling and others.

5 EVALUATION

Our evaluation shows how Chicle performns in an elastic setting where nodes are added and removed during training and on a heterogeneous cluster where nodes are differently fast. As no other elastic, load-balancing ML training framework is publicly available, we emulate micro-tasks with Chicle. Additionally, we compare Chicle with two state-of-the-art rigid ML training frameworks in non-elastic, non-heterogeneous scenario to establish a performance baseline.

5.1 Evaluation setup and methodology

Our test cluster consists of 16+1 nodes. Nodes are equipped with Intel Xeon E5-2630/40/50 v2/3 with 2.4 – 2.6GHz and 160 – 256GiB RAM. We execute Chicle inside Docker containers. For some heterogeneity experiments, we reduce the CPU frequency of four nodes from 2.6 to 1.2GHz. All nodes are connected by a 56GBit/s Infiniband network via a Mellanox SX6036 switch. During experiments, up to 16 nodes are used for workers and one node for the Chicle driver.

Our test applications are lSGD and CoCoA, using test accuracy as a metric for convergence for the former and the duality-gap (Jaggi et al., 2014; Smith et al., 2018) for the latter. We train on each dataset for ≈20 minutes, after which we terminate the training. Each experiment is repeated five times and average results are presented. Table 1 lists all datasets we use during the evaluation. The chunk size is set to 1MiB in CoCoA experiments and, due to the smaller dataset sizes, 200KiB for lSGD experiments.

Table 1. Number of samples (#S), features (#F) and categories (#C) of datasets used in the evaluation. Size is given for the in-memory representation.

Dataset	#S	#F	#C	Size
Higgs	11M	28	2	2.5GiB
Criteo	46M	1M	2	15GiB
CIFAR-10	60k	3072	10	162MiB
Fashion-MNIST	70k	784	10	30MiB

Synchronous local SGD. We implemented lSGD (Lin et al., 2018) for Chicle based on libtorch, the C++ backend of PyTorch (Paszke et al., 2017). We train a CNN with relu activation composing of two convolutional layers with max-pooling followed by 3 fully connected layers on the CIFAR-10 and Fashion-MNIST datasets using lSGD. We use $L = 8$ and $H = 16$, a momentum of 0.9 and a base learning rate α of 1e-4 for CIFAR-10 and 5e-4 for Fashion-MNIST. According to best practice, we scale the learning rate with the square root of the number of tasks K such that the effective learning rate $\alpha' = \alpha \times \sqrt{K}$. The global batch size (number of samples processed during each iteration across all tasks) is $K \times L \times H$. For micro-tasks, we select four values for $K = \{16, 24, 32, 64\}$. Using different values of K allows us to assess the trade-off between scheduling and algorithmic efficiency. Here, K remains constant during the training. For uni-tasks K equals number of currently used nodes. As mSGD is a special case of lSGD with $H = 1$ we trivially also support mSGD, which we use for baseline comparisons with PyTorch.

CoCoA. We implemented CoCoA with a local SCD solver for Chicle based on the original Spark implementation (Smith, 2019). We train a support vector machine (SVM) on the Higgs and Criteo datasets. We use SCD as local solver with $L = 1$ and H equal to the number of local training samples. The number of tasks K is the same as above. The algorithm parameter σ is set to the the number of tasks, and the regularization coefficient λ to the number of samples $\times$ 0.01.

Micro-tasks. As no elastic ML training framework based on micro-tasks (or any other technique) is publicly available and general-purpose frameworks such as Spark do not perform competitively (Dünner et al., 2017), we emulate micro-tasks using Chicle with a constant number of tasks K and measure the convergence rate per epoch. It is possible to do this accurately because in micro-tasks, convergence rate per epoch only depends on the number of tasks but not on the number of nodes or on which node a task is executed on. It does not, however, allow us to directly measure the convergence rate over time for micro-tasks. Instead, we project the latter by assuming an optimal schedule for the number of tasks, nodes and relative node performance. Henceforth, the number of micro-tasks is given in parentheses.

Using Chicle to emulate micro-tasks during elasticity and load balancing experiments has the additional benefit of keeping implementation-specific variables, such as the implementation of the training algorithms (lSGD and CoCoA), the communication subsystem (e.g., RDMA vs. TCP/IP), and other factors constant.

5.2 Baseline comparisons

We compare Chicle against Snap ML (Dünner et al., 2018) for CoCoA and PyTorch (Paszke et al., 2017) for mSGD in a non-elastic, non-heterogeneous scenario using the same training algorithms, hyper-parameter values and datasets on the test setup described above. None of the novel functionality of Chicle was used in this experiment. The purpose of this experiment is to show that Chicle does not impair performance in the *normal* non-elastic, non-heterogeneous case. We measure convergence rate per epoch and over time. Detailed results of this experiment are provided in §A.1 and are summarized here.

Convergence behavior per epoch for mSGD is identical on Chicle and PyTorch while Chicle requires slightly less time per epoch. Compared to Snap ML, Chicle performed virtually identically for the Higgs dataset but outperformed it for the Criteo dataset due to differences in data partitioning. This experiment confirms that Chicle's baseline performance is on par with that of highly optimized, established ML training frameworks. In contrast to those, Chicle is able to elastically scale during execution and balance load in heterogeneous clusters. Both aspects are evaluated in the following.

5.3 Elastic scaling

In this section, we evaluate Chicle with the elastic scaling policy enabled in two elastic scenarios and compare it to micro-tasks. Specifically, we consider: **i)** the effect of *data parallelism* (batch size for lSGD, and number of partitions for CoCoA) on the number of epochs to converge, and **ii)** the trade-off between scheduling efficiency and convergence under micro-tasks.

Methodology. Our test scenarios consist of gradual scale-in from 16 to 2 nodes and scale-out from 2 to 16 nodes. We add (remove) 2 nodes every 20s until the maximum (minimum) number of nodes is reached. During each run, we measure convergence per epoch and project convergence over time using an optimal schedule for uni-tasks and micro-tasks for each number of nodes. In micro-tasks, elastic scaling works by distributing a fixed number of tasks across more or fewer nodes and not by adjusting the number of tasks. Moreover, the number of nodes is typically not known by the application. Hence, we assume a fixed number of tasks independently of the nodes used. To project the time per iteration, we assume a normalized task runtime (one task, processing $1/16$th of the data takes one time unit) and compute the number of task waves necessary for each iteration.

- K micro-tasks on N nodes require $\lceil K/N \rceil$ task waves, as only N tasks can be executed at the same time. In consequence, each iteration requires $16/K \times \lceil K/N \rceil$ time units. For instance, $K = 32$ tasks on $N = 14$ nodes require $\lceil 32/14 \rceil = 3$ task waves and $16/32 \times 3 = 1.5$ time units per iteration.
- For CoCoA on uni-tasks, load is redistributed such that a single iteration takes $16/N$ time units. For instance, on 14 nodes, one iteration requires $16/14 = 1.14$ time units. For lSGD on uni-tasks, the batch size is adjusted such that each iteration still only requires one time unit. Instead, the number of iterations per epoch increases by $16/N$.

Our time projections do not include data transfer overheads. As each task needs to communicate model updates, the total communication volume of micro-tasks is as least as high as that of uni-tasks, hence by ignoring data transfer overheads, we favor micro-tasks.

Results. Figure 4 shows detailed convergenve over time plots for elastic scale-in and out for different data parallelism values. Convergence per epoch results are provided in the appendix (§A.2). Generally, the higher the data parallelism, the more epochs are needed to converge for micro-tasks, which is consistent with our initial problem statement and previous studies (Shallue et al., 2019). As Figure 4 shows, the increased scheduling efficiency of using more micro-tasks cannot compensate for the reduced convergence rate per epoch and micro-tasks (16) consistently outperforms other micro-tasks configurations.

Moreover, the convergence rate over time with uni-tasks is equal or higher during scale-in and -out, showing that the ability to adjust the level of data parallelism across a wide range can improve convergence per epoch and over time.[3] This ability is not only beneficial in shared environments but can also be exploited to accelerate the training process in general. Kaufmann et al. (2018) show for CoCoA, that scaling in training at specific points in time can accelerate training by up to $6\times$. Smith et al. (2017) report that increasing the batch size as alternative to reducing the learning rate once convergence slows down is beneficial for mini-batch SGD. Both cases could be implemented with Chicle.

However, results differ across algorithms and datasets. For lSGD, scale-in as well as scale-out on uni-tasks improves convergence over time compared to the best micro-tasks configuration. In the scale-out case, the global batch size for uni-tasks is smaller in the beginning but equalizes with micro-tasks (16) quickly as nodes are added. In the scale-in case, the global batch size for uni-tasks is the same as for micro-tasks (16) in the beginning but is quickly reduced. As it is smaller for longer, compared to the scale-out case, the convergence benefits over micro-tasks (16) are higher in the scale-in case.

The average maximal test accuracy for uni-tasks is virtually identical to that of micro-tasks (16), which is the best micro-tasks configuration in all but one case: In the scale-in case for CIFAR-10, uni-tasks achieves an average maximal test accuracy of 65.6% compared to 65.0% for micro-tasks (16).

Results for CoCoA are similar. Scaling in reduces the number of epochs as well as time to converge, as suggested in Kaufmann et al. (2018). After each scale-in step (which can be identified in Figure 4c and Figure 4d) convergence rate improves. The reason for this behavior is that the local SCD solver has access to additional training data and can therefore identify new correlations across training samples locally. Scaling out behaves similarly which is, at first sight, counter intuituve as every task gets to see fewer and fewer training samples as training scales out. However, during scale-out the data chunks that are moved to newly added tasks are picked randomly from each old task which effec-

[3] Applications are free to choose any level of data parallelism equal or larger the number of tasks if it benefits convergence.

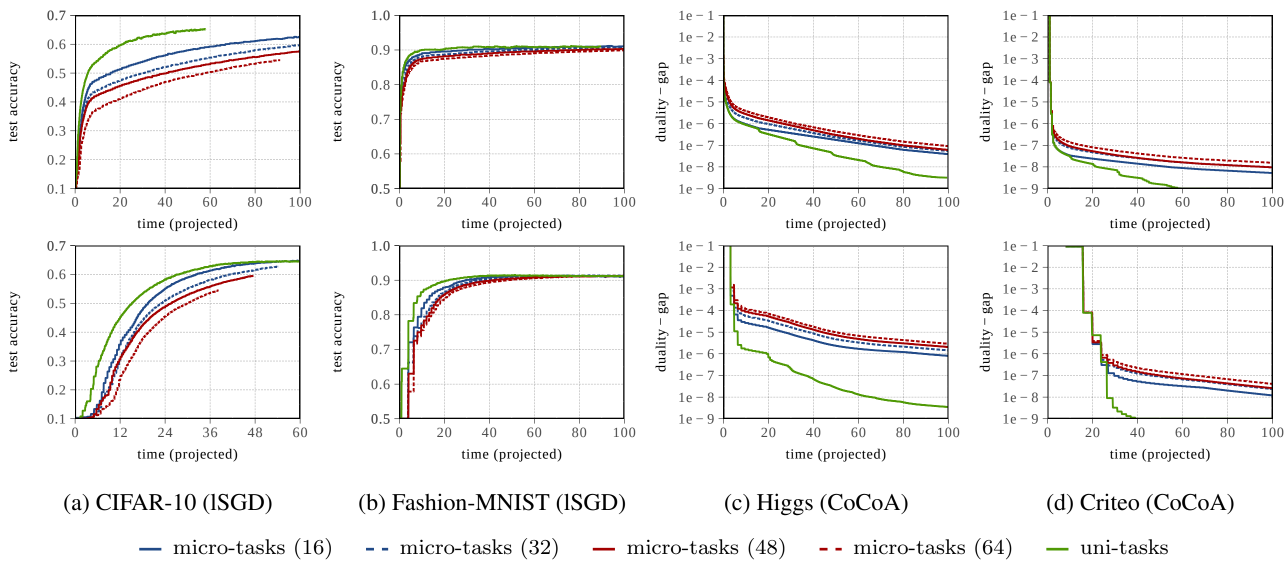


Figure 4. Convergence over time (projected) vs. data parallelism for elastic **scale-in (top)** and **scale-out (bottom)** experiments. The number of micro-tasks is given in parentheses. Time is normalized to 100 time units.

tively shuffles training samples. This also allows the solver to identify new correlations locally while also decreasing the duration of each iteration.

5.4 Load balancing

In this section, we compare Chicle, with the load balancing policy enabled, to micro-tasks in a heterogeneous scenario with nodes of different speed. Such a scenario can occur in practice, as compute clusters are often not replaced completely but extended and partially replaced over time using multiple generations of hardware (e.g., CPUs, GPUs) (Delimitrou & Kozyrakis, 2014). Even the same cloud instance type can be backed by different models and generations of hardware (Ou et al., 2012).

In a heterogeneous scenario, faster nodes should perform more of the overall work than slower nodes, such that all nodes finish at the same time for each iteration. In a micro-task based system, this is achieved by scheduling more tasks on fast nodes than slow nodes. This, however, requires multiple tasks to be executed per node so that one or more of them can be moved to other nodes. In consequence, no load balancing is possible with micro-tasks (16) on our 16 node test cluster. Chicle balances load by shifting data chunks, of which there are typically hundreds or thousands, from slow nodes to fast nodes and by adjusting the number of samples that individual uni-tasks process in each iteration, such that all tasks finish at the same time, independently of the node performance.

Methodology. We evaluate heterogeneous load balancing in two scenarios: 1) We configure the load balancing policy of Chicle to assume eight fast and eight slow nodes, with the latter being $1.5\times$ slower than the former and measure the number of epochs to converge. This simple scenario allows us to project time to convergence. 2) We execute Chicle with the load balancing policy enabled on our test cluster where the CPU frequency of four nodes has been reduced to increase the level of heterogeneity. We measure the task and iteration runtimes as well as the number of data chunks of each task across the load balancing process to show how Chicle can correctly learn task runtime and balance load in response.

In micro-tasks, load balancing works by balancing fixes-size tasks across all nodes and not by adjusting the number training samples per task. Hence, we assume that each task processes the same number of training samples per iteration. To project the time per iteration, we assume a normalized task runtime: One task, processing $1/16$th of the data takes one time unit on the fast nodes and 1.5 time units on the slow nodes. We use this to compute the optimal (shortest) schedule for each iteration.

- For micro-tasks, K tasks on eight fast and eight slow nodes, the optimal schedule is $\max(i \times 1.5s, j \times 1.0s) \times 16/K$ long with i (j) being the number of tasks on each slow (fast) node such that the schedule length is minimal. For instance, with $K = 64$ tasks, the optimal schedule is $\max(3 \times 1.5s, 5 \times 1.0s) \times 16/64 = 1.25s$ per iteration.

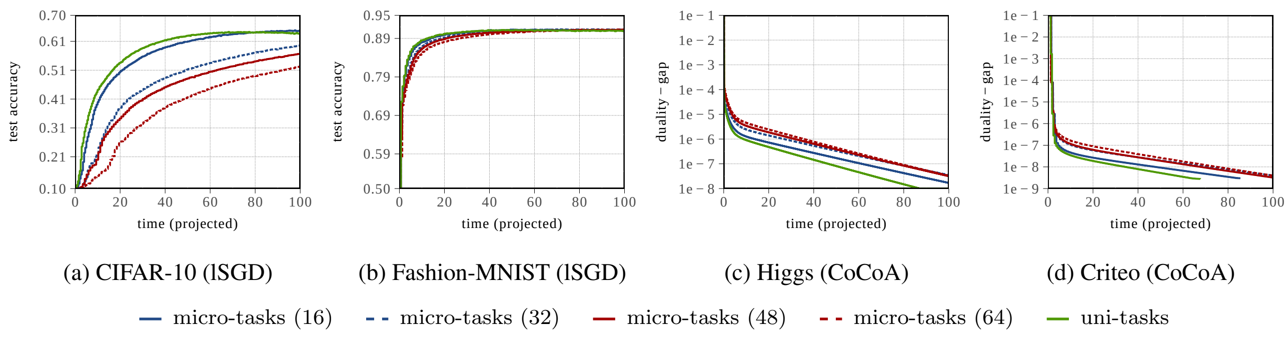


Figure 5. Convergence over time (projected) when balancing load balancing in a heterogeneous cluster. The number of micro-tasks is given in parentheses. Time is normalized to 100 time units.

- For uni-tasks, load is redistributed such that fast nodes process $1.5\times$ as many training samples as slow nodes, resulting in an iteration duration of $1.2s$.

As before, our time projections do not include data transfer overheads, which favors micro-tasks.

Results. Figure 5 shows detailed convergenve over time plots for different data parallelism values. Convergence per epoch results are shown in §A.3. Per epoch, Chicle converges as fast as micro-tasks (16). Over time, however, Chicle converges faster than any micro-tasks configuration as it requires as few epochs to converge as micro-tasks (16) but can balance load more effectively than micro-tasks (64), which reduces iteration duration and thus combines algorithmic and scheduling efficiency. For lSGD, the average maximal test accuracy is ≈0.5% lower with uni-tasks than with micro-tasks (16). However, no load balancing is actually possible with the latter. Compared to other micro-task configurations, uni-tasks achieves a similar average maximal test accuracies. For CoCoA, uni-tasks converges virtually identical to micro-tasks (16) per epoch but outperforms it over time due to its ability to balance load more effectively.

Swimlane diagrams in Figure 6 visualize the load balancing process for the Criteo dataset on our test cluster where the CPU frequency of four nodes has been reduced to 1.2GHz to improve the visibility of this process. Results for the other datasets are similar and provided in the appendix (§A.3).

The top diagram shows task runtimes per node and iteration without load balancing. Here, iteration duration is determined by the four slow nodes. Task runtimes are visualized by horizontal black bars. Bars that start at the same time represent tasks of the same iteration. Space in-between bars represents time during which tasks are inactive, i.e., communicating or waiting for the latest model update from the trainer. The middle diagram shows task runtimes with load balancing enabled. During the first iteration, task runtimes are the same as without load balancing. As load is

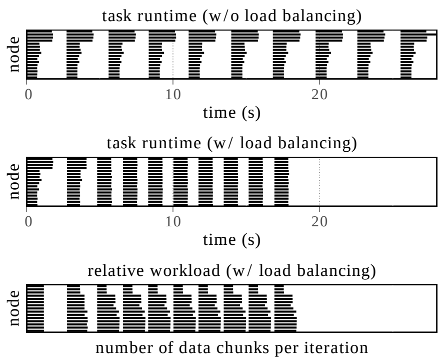


Figure 6. Visualization of the load balancing process on a real heterogeneous cluster.

shifted during subsequent iterations, task runtimes align and iteration durations reduce. The bottom diagram shows the relative workload (not time) of tasks in the middle diagram. It shows how the workload is shifted from slow to fast nodes. The length of the bars represent the number of data chunks for each task and iteration, relative to all other tasks and iterations. After a few iterations, workload and task runtimes stabilize as Chicle has learned the performance of each node and balance load accordingly.

6 Conclusion and future work

We presented Chicle, a distributed ML training framework based on uni-tasks. Chicle enables efficient elastic scaling and load balancing without incurring overheads that are typical for micro-task systems and can thereby accelerate time to convergence by orders of magnitude in some cases. Our work touches many issues that distinguish distributed ML training from regular distributed applications, such as their sensitivity to data parallelism. Still, many aspects of ML workloads remain unexplored, and we believe there is a lot of potential to further exploit the unique properties of ML algorithms to build more efficient systems.

References

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. In *OSDI*, volume 16, pp. 265–283, 2016.

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*, Santa Ana Pueblo, NM, 2013. USENIX. URL https://www.usenix.org/conference/hotos13/solving-straggler-problem-bounded-staleness.

Cui, H., Cipar, J., Ho, Q., Kim, J. K., Lee, S., Kumar, A., Wei, J., Dai, W., Ganger, G. R., Gibbons, P. B., Gibson, G. A., and Xing, E. P. Exploiting bounded staleness to speed up big data analytics. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pp. 37–48, Philadelphia, PA, 2014. USENIX Association. ISBN 978-1-931971-10-2. URL https://www.usenix.org/conference/atc14/technical-sessions/presentation/cui.

Delimitrou, C. and Kozyrakis, C. Quasar: resource-efficient and qos-aware cluster management. *ACM SIGPLAN Notices*, 49(4):127–144, 2014.

Dünner, C., Parnell, T., Atasu, K., Sifalakis, M., and Pozidis, H. Understanding and optimizing the performance of distributed machine learning applications on apache spark. In *2017 IEEE International Conference on Big Data (Big Data)*, pp. 331–338. IEEE, 2017.

Dünner, C., Parnell, T., Sarigiannis, D., Ioannou, N., Anghel, A., Ravi, G., Kandasamy, M., and Pozidis, H. Snap ml: A hierarchical framework for machine learning. In *Advances in Neural Information Processing Systems*, pp. 250–260, 2018.

Dutta, S., Joshi, G., Ghosh, S., Dube, P., and Nagpurkar, P. Slow and stale gradients can win the race: Error-runtime trade-offs in distributed sgd. In Storkey, A. and Perez-Cruz, F. (eds.), *Proceedings of the Twenty-First International Conference on Artificial Intelligence and Statistics*, volume 84 of *Proceedings of Machine Learning Research*, pp. 803–812, Playa Blanca, Lanzarote, Canary Islands, 09–11 Apr 2018. PMLR. URL http://proceedings.mlr.press/v84/dutta18a.html.

Goyal, P., Dollár, P., Girshick, R., Noordhuis, P., Wesolowski, L., Kyrola, A., Tulloch, A., Jia, Y., and He, K. Accurate, large minibatch sgd: Training imagenet in 1 hour. *arXiv preprint arXiv:1706.02677*, 2017.

Harlap, A., Tumanov, A., Chung, A., Ganger, G. R., and Gibbons, P. B. Proteus: agile ml elasticity through tiered reliability in dynamic resource markets. In *Proceedings of the Twelfth European Conference on Computer Systems*, pp. 589–604. ACM, 2017.

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*, pp. 1223–1231, 2013.

Jaggi, M., Smith, V., Takác, M., Terhorst, J., Krishnan, S., Hofmann, T., and Jordan, M. I. Communication-efficient distributed dual coordinate ascent. In *Advances in neural information processing systems*, pp. 3068–3076, 2014.

Kaufmann, M., Parnell, T., and Kourtis, K. Elastic CoCoA: Scaling in to improve convergence. *NeurIPS 2018 Systems for ML workshop*, December 2018.

Keskar, N. S., Mudigere, D., Nocedal, J., Smelyanskiy, M., and Tang, P. T. P. On large-batch training for deep learning: Generalization gap and sharp minima. *arXiv preprint arXiv:1609.04836*, 2016.

Kiefer, J., Wolfowitz, J., et al. Stochastic estimation of the maximum of a regression function. *The Annals of Mathematical Statistics*, 23(3):462–466, 1952.

Lin, T., Stich, S. U., and Jaggi, M. Don't use large mini-batches, use local sgd. *arXiv preprint arXiv:1808.07217*, 2018.

Ou, Z., Zhuang, H., Nurminen, J. K., Ylä-Jääski, A., and Hui, P. Exploiting hardware heterogeneity within the same instance type of amazon ec2. In *Presented as part of the*, 2012.

Ousterhout, K., Panda, A., Rosen, J., Venkataraman, S., Xin, R., Ratnasamy, S., Shenker, S., and Stoica, I. The case for tiny tasks in compute clusters. In *Proceedings of the 14th Workshop on Hot Topics in Operating Systems*, volume 13 of *HotOS '16*, Santa Ana Pueblo, NM, 2013. USENIX. URL https://www.usenix.org/conference/hotos13/case-tiny-tasks-compute-clusters.

Paszke, A., Gross, S., Chintala, S., Chanan, G., Yang, E., DeVito, Z., Lin, Z., Desmaison, A., Antiga, L., and Lerer, A. Automatic differentiation in pytorch. In *NIPS-W*, 2017.

Qiao, A., Aghayev, A., Yu, W., Chen, H., Ho, Q., Gibson, G. A., and Xing, E. P. Litz: Elastic framework for high-performance distributed machine learning. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pp. 631–644, Boston, MA,

2018. USENIX Association. ISBN 978-1-931971-44-7. URL https://www.usenix.org/conference/atc18/presentation/qiao.

Robbins, H. and Monro, S. A stochastic approximation method. *The annals of mathematical statistics*, pp. 400–407, 1951.

Rumelhart, D. E., Hinton, G. E., Williams, R. J., et al. Learning representations by back-propagating errors. *Cognitive modeling*, 5(3):1, 1988.

Shallue, C. J., Lee, J., Antognini, J., Sohl-Dickstein, J., Frostig, R., and Dahl, G. E. Measuring the effects of data parallelism on neural network training. *Journal of Machine Learning Research*, 20(112):1–49, 2019.

Smith, S. L., Kindermans, P.-J., Ying, C., and Le, Q. V. Don't decay the learning rate, increase the batch size. *arXiv preprint arXiv:1711.00489*, 2017.

Smith, V. Cocoa for spark, 2019. URL https://github.com/gingsmith/cocoa. accessed 2019/05/10.

Smith, V., Forte, S., Chenxin, M., Takáč, M., Jordan, M. I., and Jaggi, M. Cocoa: A general framework for communication-efficient distributed optimization. *Journal of Machine Learning Research*, 18:230, 2018.

Spark. http://spark.apache.org/docs/latest/tuning.html#level-of-parallelism, 2019. URL http://spark.apache.org/docs/latest/tuning.html#level-of-parallelism.

Stich, S. U. Local sgd converges fast and communicates little. *arXiv preprint arXiv:1805.09767*, 2018.

Totoni, E., Dulloor, S. R., and Roy, A. A case against tiny tasks in iterative analytics. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, HotOS '17, pp. 144–149, New York, NY, USA, 2017. ACM, ACM. ISBN 978-1-4503-5068-6. doi: 10.1145/3102980.3103004. URL http://doi.acm.org/10.1145/3102980.3103004.

Vavilapalli, V. K., Murthy, A. C., Douglas, C., Agarwal, S., Konar, M., Evans, R., Graves, T., Lowe, J., Shah, H., Seth, S., Saha, B., Curino, C., O'Malley, O., Radia, S., Reed, B., and Baldeschwieler, E. Apache Hadoop YARN: Yet another resource negotiator. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, pp. 5:1–5:16, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2428-1. doi: 10.1145/2523616.2523633. URL http://doi.acm.org/10.1145/2523616.2523633.

Wright, S. J. Coordinate descent algorithms. *Math. Program.*, 151(1):3–34, 2015. ISSN 0025-5610.

You, Y., Gitman, I., and Ginsburg, B. Large batch training of convolutional networks. *arXiv preprint arXiv:1708.03888*, 2017.

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*, HotCloud'10, Berkeley, CA, USA, 2010. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1863103.1863113.

Zhang, H., Stafman, L., Or, A., and Freedman, M. J. Slaq: Quality-driven scheduling for distributed machine learning. In *Proceedings of the 2017 Symposium on Cloud Computing*, SoCC '17, pp. 390–404, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-5028-0. doi: 10.1145/3127479.3127490. URL http://doi.acm.org/10.1145/3127479.3127490.

A Appendix

A.1 Baseline comparisons

We compare Chicle against Snap ML (Dünner et al., 2018) for CoCoA and PyTorch (Paszke et al., 2017) for mSGD in a non-elastic, non-heterogeneous scenario. Neither compared-to framework is able to elastically scale nor balance load. The purpose of this comparison is to show that the elasticity and load balancing capabilities of Chicle and uni-tasks do not come at a cost of performance in the *normal* case. In consequence, Chicle's elasticity and load balancing policies are also not used during these experiments. Both frameworks are executed with RDMA-enabled MPI communication backends. We measure the convergence per epoch and over time. Each experiment is repeated 5×.

PyTorch. As no lSGD implementation for PyTorch exists, we compared Chicle to PyTorch using mSGD. mSGD is a special case of lSGD with $H = 1$. Chicle's mSGD training algorithm uses libtorch, the C++ backend of PyTorch, which allows us to rule out the implementation of the training algorithm as source for any potential differences. For both datasets, a learning rate of 0.002 and a momentum of 0.9 is used.

Convergence per epochs is virtually identical to PyTorch. This is expected as both are based on libtorch and therefore use the same training algorithm implementations, CNN and hyper-parameters. Per time, Chicle is slightly faster, which is likely due to overheads introduced by Python, which do not afflict Chicle, at it is natively implemented in C++. The maximal test accuracy that was achieved within the test duration is 65.2% for CIFAR-10 with both frameworks. For Fashion-MNIST, Chicle has a 0.2% lead over PyTorch with 91.4%. Note that we did not tune hyper-parameters for each dataset dataset nor adjust them online, which is why the test accuracy for CIFAR-10 degrades slightly after reaching a peak.

Snap ML. Chicle's CoCoA/SCD implementation for the training of a SVM is based on the original Spark implementation (Smith, 2019). The algorithm parameter σ is set to the the number of tasks, and the regularization coefficient λ to the number of samples $\times$ 0.01. Compared to Snap ML, Chicle shows similar convergence and runtime behavior for the Higgs dataset. For Criteo, however, Chicle converges much faster. This is due to the sensitivity of Criteo to data partitioning. Chicle randomly assigns data chunks to tasks, whereas Snap ML splits the data into 16 contiguous partitions. Per iteration, Snap ML is slightly faster as Chicle's reduce and broadcast primitives are less optimized than their MPI counterparts used by Snap ML.

With one exception (Criteo), Chicle performs similarly to both rigid frameworks in a baseline scenario, showing that neither Chicle not uni-tasks impair baseline performance.

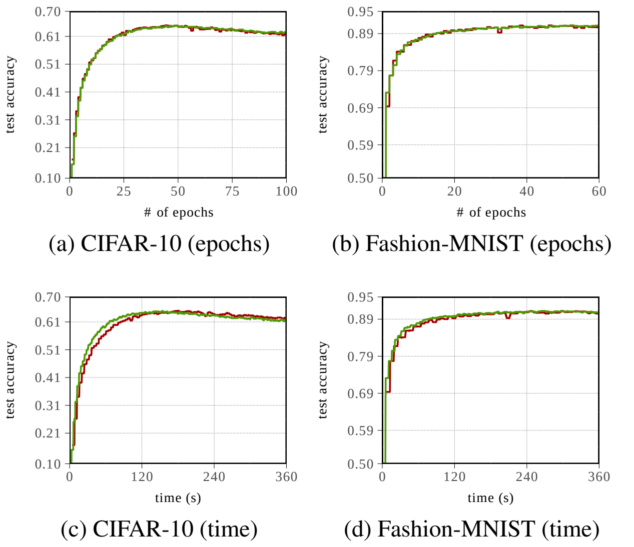


Figure 7. Comparison with PyTorch w.r.t. convergence over epochs (top) and time (bottom).

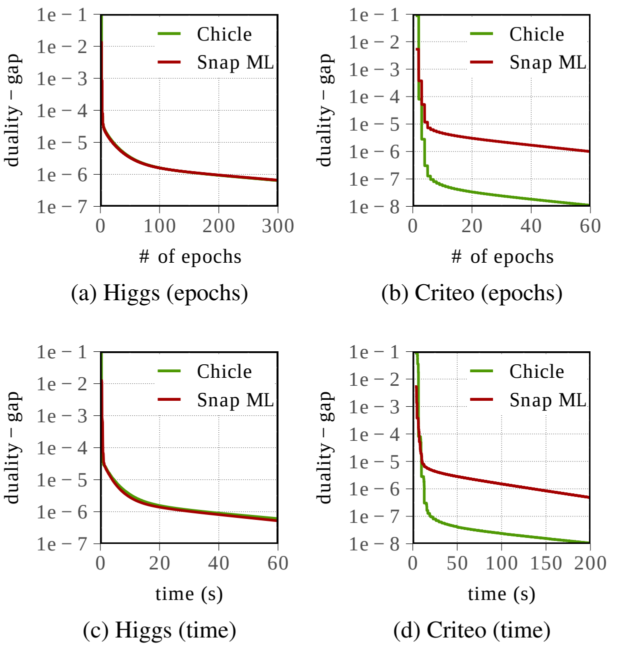


Figure 8. Comparison with Snap ML w.r.t. convergence over epochs (top) and time (bottom).

A.2 Elastic scaling

Figure 9 shows per-epoch convergence results for the elastic scaling experiments.

A.3 Load balancing

Figure 10 shows per-epoch convergence results for the load balancing experiments. Figure 11 shows the load balancing process during the first 10 (CoCoA) and 50 (lSGD) iterations.

A.4 Example application

Listing 1 shows a simplified trained and solver module for mSGD on Chicle.

```
Trainer::run() {
  while (!done) {
    signal(StartIteration);
    wait(IterationFinished);
    model = merge_updates(); // merge and
    broadcast(model);        // broadcast updates
  }
}

Solver::run() {
  while (!done) {
    wait(IterationStarted);
    model = get_model() // fetch model
    // perform training
    sample = get_next_sample();
    output = model->forward(sample->data);
    loss = compute_loss(output, sample->label);
    loss->backward();
    sgd_optimizer.step();
    send(model) // post updates
    signal(IterationFinished);
  }
}
```

Listing 1. A minimal Chicle application example

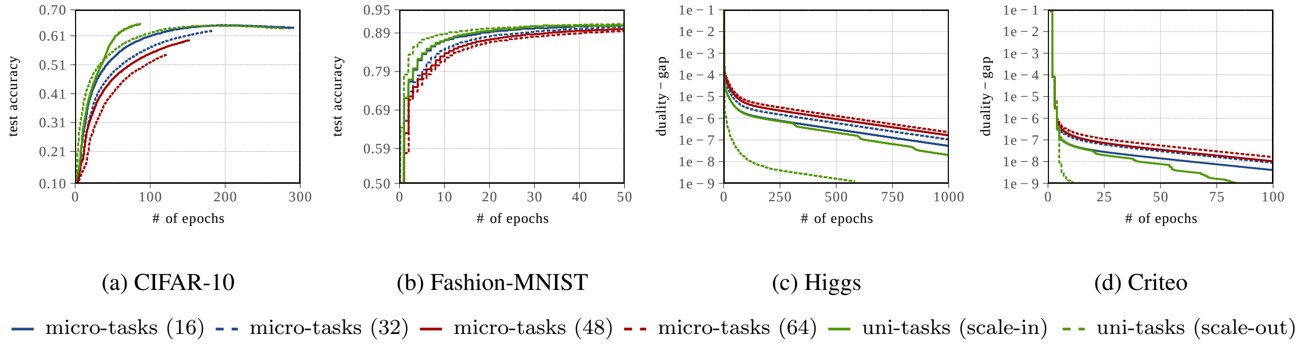


Figure 9. Convergence per epoch vs. data parallelism for elastic scale-in/out experiments. The number of micro-tasks is given in parentheses.

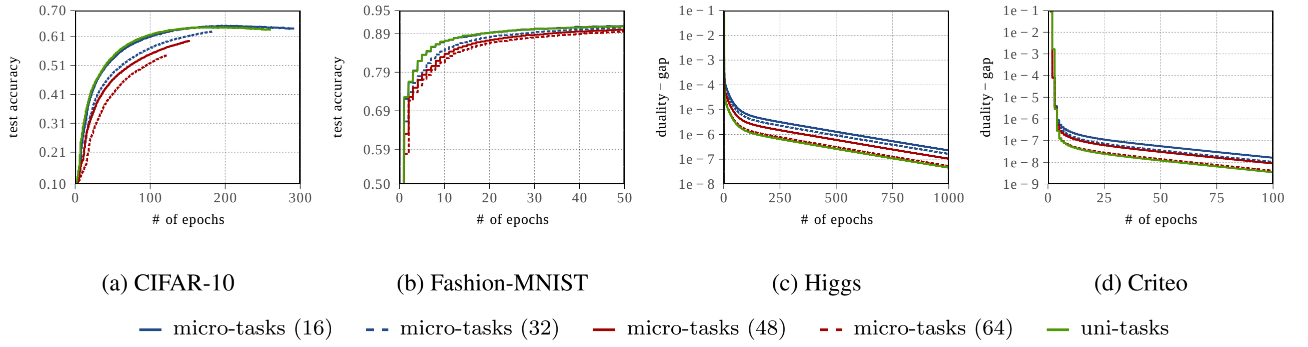


Figure 10. Convergence per epoch when balancing load balancing in a heterogeneous cluster. The number of micro-tasks is given in parentheses.

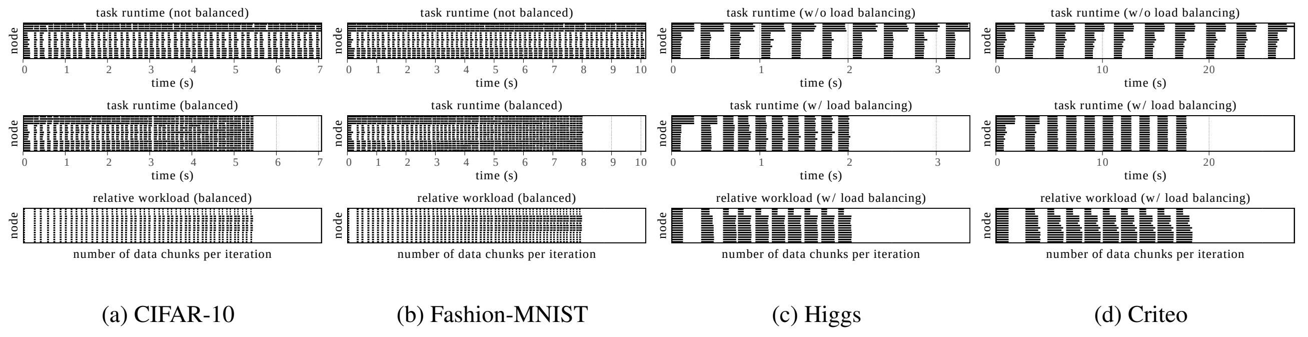


Figure 11. Task execution duration and per worker workload for the load balancing in a heterogeneous cluster experiments.