Automating System Configuration of Distributed Machine Learning

Woo-Yeon Lee[1], Yunseong Lee[1], Joo Seong Jeong[1], Gyeong-In Yu[1], Joo Yeon Kim[2], Ho Jin Park[1], Beomyeol Jeon[3], Wonwook Song[1], Gunhee Kim[1], Markus Weimer[4], Brian Cho[5], Byung-Gon Chun[1*]
[1]Seoul National University, [2]Samsung Electronics,
[3]University of Illinois at Urbana-Champaign, [4]Microsoft, [5]Facebook
{wooyeonlee0, yunseong.lee0, joosjeong, gyeonginyu, jykim88, hojinpark.cs}@gmail.com,
{beomyeolj, wsong0512}@gmail.com, gunhee@snu.ac.kr, mweimer@microsoft.com, bcho@fb.com, bgchun@snu.ac.kr

***Abstract*—The performance of distributed machine learning systems is dependent on their system configuration. However, configuring the system for optimal performance is challenging and time consuming even for experts due to the diverse runtime factors such as workloads or the system environment. We present cost-based optimization to automatically find a good system configuration for parameter server (PS) machine learning (ML) frameworks. We design and implement Cruise that applies the optimization technique to tune distributed PS ML execution automatically. Evaluation results on three ML applications verify that Cruise automates the system configuration of the applications to achieve good performance with minor reconfiguration costs.**

I. Introduction

Machine learning (ML) systems are widely used to extract insights from data. Ever increasing dataset sizes and model complexity gave rise to many efforts towards efficient distributed machine learning systems. One of the popular approaches to support large-scale data and complicated models is the parameter server (PS) approach [1], [12], [16]. In this approach, training data is partitioned across *workers*, while model parameters – which compose the global model being trained – are partitioned across *servers*. During training, each of the workers computes model updates using the allocated data and sends the model updates to the corresponding servers. Workers then fetch fresh models from servers in order to work with the latest model parameter values. Servers, meanwhile, apply the model updates received from workers and send the latest model parameter values back to workers as inquired. This process occurs iteratively during the course of the ML job until the global model converges. The performance of this approach is crucially dependent on choosing the right system configuration[1] : the number of workers and servers as well as the training data and model partitioning across them [24].

Current PS implementations assume system configuration to be static: the configuration is chosen before training commences and remains unchanged until job termination [4], [5], [7], [23]. However, as we illustrate in Section II-B, choosing the best system configuration is challenging; optimal system configuration parameters vary widely for different algorithms, hyper-parameters, and environments. Furthermore, the best configuration changes during runtime as the total amount of available resource changes.

We present cost-based optimization that finds a good system configuration for PS-based frameworks. We extend a PS-based ML framework to build Cruise that automatically tunes its system configuration with the optimization technique. We would like to ideally model convergence time, but this problem is an open research question that requires modeling algorithms themselves. Instead, Cruise focuses on the system aspects and models performance of workers, as an optimization goal, analytically with the system's runtime statistics, and computes optimal configurations by solving the optimization problem. Cruise applies the new configurations efficiently during runtime by elastically changing allocated resources and migrating data. The reconfiguration allows us to make the best use of given resources and also opportunistically available resources.

Our evaluation shows that our cost model is valid and Cruise finds a good system configuration automatically to optimize the performance. With three widely-used machine learning workloads, we demonstrate that the configuration found by Cruise performs close to the optimal configuration that we find exhaustively, with the difference at most 6.5%. Cruise reduces the training time by up to 58.3% compared to static configuration within tens of seconds reconfiguration overhead.

II. Background

A. Parameter Server ML Framework

A typical machine learning (ML) process engages itself in building models from input data and such training process could be designed in various ways. In many recent ML systems, the notion of parameter servers (PS) is used to manage training models [5], [7], [16], [23].

The PS architecture shown in Figure 1 consists of workers and servers. Machine learning is an iterative convergent process, where an *epoch* is the unit of iteration. An epoch is normally defined as a full scan of the entire training data, but in asynchronous ML systems, an epoch can refer to the process of scanning the same number of training data instances as the entire dataset. Training data is partitioned across workers

*Corresponding author

[1]ML training systems have two types of configurations: system and algorithmic configurations. Algorithmic configurations include hyper-parameters such as learning rate and batch size. In this paper, we focus on system configuration parameters.

App.	NMF	MLR	LDA
(18, 14)	1.30x	2.29x	**Best**
(23, 9)	**Best**	1.12x	1.21x
(27, 5)	1.66x	**Best**	1.60x

(a) Case 1. Epoch time comparison in different algorithms.

#Topics	400	4K
(18, 14)	**Best**	1.44x
(23, 9)	1.21x	**Best**
(27, 5)	1.60x	1.20x

(b) Case 2. Epoch time comparison in different hyper-parameters in LDA.

Env.	m4.large	m4.xlarge
(34, 18)	**Best**	1.10x
(38, 14)	1.08x	1.05x
(42, 10)	1.48x	**Best**

(c) Case 3. Epoch time comparison in different VM instance types in NMF.

TABLE I: Epoch times varying numbers of workers and servers for different algorithms, hyper-parameters, and virtual machine instance types. (W, S): W denotes the number of workers and S denotes the number of servers. For each column, a cell presents a ratio between the epoch time of the configuration and the optimal epoch time.

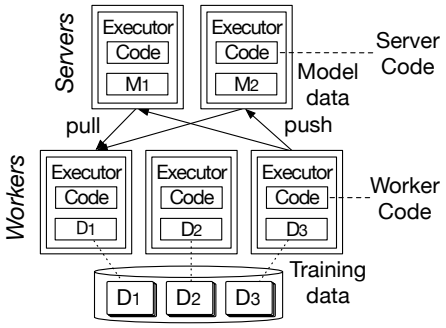

Fig. 1: Parameter server ML framework.

to achieve data parallelism. Workers execute the code to compute gradients for a model using each of its allocated data and communicate with servers to contribute towards model convergence. The model, similar to input data, is split into many partial models and distributed across servers.

Each server supports *push* and *pull* requests from workers for the partial models it is assigned to. The literature [21] contains multiple proposals for scheduling the worker-server communication. We focus on the common push/pull model below. A worker issues a push request when it wants to update a certain portion of a model. The request consists of a key, identifying the partial model to update, and the corresponding update data. When the server with the partial model receives a push request, it first searches for the model value associated with the key, and then applies the update by calling a user-defined update function defined in the server code. A worker sends a pull request when it needs to access a certain part of the model. Unlike push requests, a pull request only contains the key associated with the necessary partial model. After receiving a pull request, the server with the partial model fetches the corresponding model data and replies to the sender worker by transmitting a pull response.

In the figure, *Executor* is an environment on which the ML application code (e.g., worker computation code and server model update code) runs. Executor takes care of low-level system supports such as initializing and maintaining network connections between nodes. In this paper, we consider a cluster environment where each Executor runs in a container obtained from a Resource Manager such as YARN and Mesos.

B. System Configuration Challenges

The system configuration of a PS system includes the allocation of worker and server roles to available containers, as well as the partitioning of the training data across workers and the model parameters across servers. A good system configuration is essential for the performance of the machine learning system [24]. However, optimal system configurations that produce minimal training time are difficult to find, even for system experts. This is because, first, predicting how ML application code translates to actual running time - how much time each step takes - is nontrivial. Even if we were to estimate the exact running time for an algorithm, there may exist many different implementations for that particular algorithm, all having slightly different running times. Second, even for the same algorithm, using different hyper-parameters can change the application's computation or communication overhead. Finally, the capabilities of the environment on which the applications run vary from cluster to cluster.

We illustrate the challenges with experiments that vary algorithms, hyper-parameters, and machines in Table I. In each case, we fix the total number of machines, assign a fraction of the machines to run workers, and assign the rest of the machines to run servers. We experiment with all possible worker and server configurations to compare epoch time of different configurations.

All experiments in Tables Ia and Ib were run on a cluster of 32 AWS EC2 r4.xlarge instances (4 CPU vCores, 30.5GB memory, and 1.25 Gbps network bandwidth), and Table Ic shows epoch time of an ML application on either a cluster of 52 m4.large instances (2 CPU vCores, 8GB memory, and 0.5 Gbps network bandwidth) or a cluster of 52 m4.xlarge instances (4 CPU vCores, 16GB memory, and 1.0 Gbps network bandwidth). In the table, NMF denotes Non-negative Matrix Factorization, MLR denotes Multinomial Logistic Regression, and LDA denotes Latent Dirichlet Allocation. We present the details of these algorithms in Section V-A.

Case 1: ML algorithm. Different ML algorithms show different optimal configurations. From Table Ia, with (W:27, S:5) MLR achieves the smallest epoch time, whereas LDA runs 1.6 times slower with this configuration compared to LDA's optimal configuration (W:18, S:14). This is because MLR is more compute-intensive than LDA, thus requiring more workers for smaller epoch time.

Case 2: Hyper-parameter. Hyper-parameter values affect the optimal configuration of an ML application. A hyper-

parameter in LDA is the number of topics to categorize documents. Increasing the number of topics makes both computation and communication more expensive, but they are affected differently. Table Ib shows that (W:18, S:14) is the best configuration for 400 topics but is 1.44 times slower than the best configuration (W:23, S:9) for 4K topics.

Case 3: Machine environment. The specification of the cluster on which jobs run also heavily affects the best configuration due to varying computation and communication capabilities. Running NMF on different clusters, we observe that the best configuration varies drastically as shown in Table Ic. When we use AWS EC2 m4.xlarge instances, the best configuration is (W:42, S:10), which is 1.48 times slower if the same configuration of m4.large instances, compared to the m4.large best configuration (W:34, S:18).

The three factors investigated above demonstrate that discovering the optimal system configuration is challenging. Even worse, there are other factors such as algorithm implementation and dataset that can also affect the performance for different configurations. Since the problem space is too broad, it is hard to predict the performance of an ML application given a specific setting. This motivates adapting to optimal configurations automatically.

III. Finding Good System Configuration

In this section, we describe our cost formulation of the training epoch time along with our model assumptions, and how we minimize epoch time by using the cost model to find values for system configuration parameters – namely, the number of workers and servers as well as the training data and model partitioning across them.

A. Cost Model

Given the PS architecture, we define the cost C of the entire system to be the maximum of the time for each worker i to process the assigned training data in each epoch (C^i: *epoch time*). By minimizing the maximum epoch time (C), we can improve the absolute performance as well as balancing all workers' performance. Unbalanced training can slow down the learning process because training data does not contribute evenly to the global model parameters. In a general system consisting of heterogeneous containers[2] and uneven data partitions, C^i is usually different for each worker.

$$C = \max_i C^i \tag{1}$$

A worker's epoch can be further split into smaller components. Figure 2 depicts the timeline of a worker's epoch. A worker first performs computation on its training data using the current model to produce model gradients. The worker then communicates with the servers to send its gradients via push requests and fetches the updated model via pull requests. Depending on the algorithm and additional job parameters,

[2]We focus on modeling heterogeneous containers because of heterogeneous hardware or virtual machines. Transient stragglers are not part of the model. We borrow work stealing techniques from prior work [10] to handle stragglers.

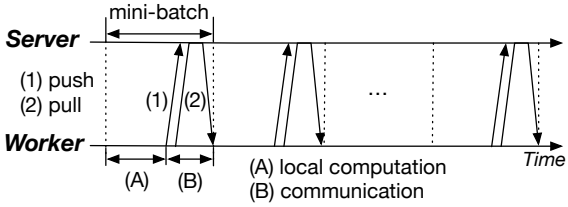

Fig. 2: A worker's epoch

workers may divide the training data into several smaller subsets and go through a computation-communication cycle for each subset. Such computation-communication cycles are called *mini-batches*. The next epoch begins once the worker has processed all of its mini-batches (i.e., all training data assigned to the worker).

To simplify the cost model, we make the following assumption on communication between workers and servers. First, push requests from workers do not block gradient computation and thus can be sent asynchronously with respect to the workers' local computation. On the other hand, a fresh pull of the whole model must always occur before local computation takes place. We assume such a model where pull requests are issued synchronously and blocks local computation [24]. The synchrony of pull requests can be partially resolved by decoupling the computation mechanism from communication threads [20]; this leads to a different cost formulation that can be understood as a variation of the one described in this section.

We define the total time spent on local computation of an epoch as *computation cost*, and the time spent on the communication as *communication cost* (denoted by (A) and (B) in Figure 2, respectively). Communication cost, to be more specific, is the sum of the elapsed times between a push request's initiation and the response for a successive pull request in each mini-batch. Using C^i_{comp} and C^i_{comm} to denote the computation and communication cost of worker i respectively, the epoch time of worker i becomes

$$C^i = C^i_{comp} + C^i_{comm} \tag{2}$$

B. Cost Formulation

a) ***Computation cost.:*** This cost depends on the size of the training dataset and the computing power of workers. The entire dataset of size D is split and distributed to w workers. C^i_{comp} depends on the size of the training dataset d_i assigned to worker i and the computing power of the worker. Depending on the time complexity f of the ML algorithm, C^i_{comp} depends on $f(d_i)$ since a worker-side computation scans all of the allocated training data during an epoch. In case an ML algorithm has linear time complexity (e.g., NMF, MLR, LDA), C^i_{comp} is proportional to d_i. In a general system consisting of heterogeneous containers (e.g., containers with different numbers of cores), each worker i takes $C^i_{w.proc}$, the time spent to perform computation on a single training data instance, which varies across workers.

$$C^i_{comp}(d_i) = C^i_{w.proc} d_i \quad (3)$$

$C^i_{w.proc}$ depends on factors such as the implementation of ML algorithm or the hardware of the worker container. In Cruise, $C^i_{w.proc}$ is measured by monitoring workers' local computation; we measure the elapsed time for workers to compute gradients and divide it by the amount of the training data instances. A larger dataset makes the computation cost more expensive, but we can reduce the cost by introducing more workers, which reduces the size of dataset d_i that each worker processes in each epoch.

*b) **Communication cost.**:* We model communication cost as the time a worker takes when communicating with the server. The entire model of size M is split and distributed over s servers. m_j is the size of partial models assigned to server j. We consider the following two cases to model C^i_{comm}.

1) *Server network bandwidth is the bottleneck.* The serving latency of server j is the number of bytes sent to j divided by the bandwidth b_{ij} between worker i and server j: $\frac{m_j w}{b_{ij}}$ where w is the number of workers. With a mini-batch size of B, each worker i executes $\lceil \frac{d_i}{B} \rceil$ mini-batches per epoch. Since the communication cost is determined by the slowest server, $C^i_{comm} = \lceil \frac{d_i}{B} \rceil \max_j (\frac{m_j w}{b_{ij}})$.
2) *Worker network bandwidth is the bottleneck.* In this case, the worker's network bandwidth is being fully utilized to serve push and pull requests. The cost is formed as the number of bytes sent by worker i divided by its bandwidth: $C^i_{comm} = \lceil \frac{d_i}{B} \rceil \sum_j \frac{m_j}{b_{ij}}$.

Then, the communication cost C^i_{comm} is the maximum of the above two terms.

$$C^i_{comm} = \left\lceil \frac{d_i}{B} \right\rceil \max \left(\max_j (\frac{m_j w}{b_{ij}}), \sum_j \frac{m_j}{b_{ij}} \right) \quad (4)$$

C. Optimization

*a) **Optimization problem.**:* In Figure 3, we formally define our optimization problem. The problem formulated for heterogeneous environments where some machines have higher computing power or network bandwidth. In these environments, the configuration space becomes larger, because we also need to determine the data distribution as well as deciding whether to run a worker or a server on a container.

The optimization goal is to find the parameters $\mathbf{w}, \mathbf{s}, \mathbf{d}, \mathbf{m}$ that minimize the cost function, where $\mathbf{w}$ and $\mathbf{s}$ denote the assignment of machines to workers and servers, respectively, and $\mathbf{d}$ and $\mathbf{m}$ denote the partitioning of the training dataset and partial models, respectively.

Given N machines, we adjust the configurations to meet the optimal balance between computation and communication costs. For example, using more machines as workers certainly brings down the computation cost by reducing the training data size that each worker deals with. However, this leads to high communication cost due to an increased number of push and pull requests within an epoch and fewer containers available for servers.

Given parameters
- N: the total number of machines
- D: the entire dataset size
- M: the entire model size
- B: the mini-batch size

Variables
- $\mathbf{w} = \{0,1\}^N$: $w_i = 1$ if a worker runs on machine i
- $\mathbf{s} = \{0,1\}^N$: $s_j = 1$ if a server runs on machine j
- $\mathbf{d} = (d_1, ..., d_N)$: training data partitioning for workers
- $\mathbf{m} = (m_1, ..., m_N)$: model partitioning for servers

Problem

Find $\mathbf{w}^*, \mathbf{s}^*, \mathbf{d}^*, \mathbf{m}^*$

$$= \underset{\mathbf{w},\mathbf{s},\mathbf{d},\mathbf{m}}{\operatorname{argmin}} \max_i C^i(\mathbf{w}, \mathbf{s}, \mathbf{d}, \mathbf{m})$$

$$= \underset{\mathbf{w},\mathbf{s},\mathbf{d},\mathbf{m}}{\operatorname{argmin}} \Big[\max_i \Big[C^i_{w.proc} d_i + \lceil \tfrac{d_i}{B} \rceil \max \big(\max_j (\tfrac{m_j \|\mathbf{w}\|}{b_{ij}}), \sum_j \tfrac{m_j}{b_{ij}} \big) \Big] \Big]$$

Constraints

$\|\mathbf{w}\| + \|\mathbf{s}\| = N$: workers and servers are assigned to N machines disjointly

$\sum_i d_i = D$: total number of training data samples

$\sum_j m_j = M$: total number of model partitions

Fig. 3: Optimization Problem

*b) **Solution.**:* Based on the problem definition in Figure 3, we cast the optimization problem as Mixed Integer Programming (MIP) and solves the problem using a solver library from Gurobi [9]. Since the quadratic terms affect the performance significantly, we encode integer variables d and m in binary representation, which allows the solver to multiply variables faster. As a result, the MIP program consists of $O(N^2)$ variables, $O(N)$ quadratic constraints, and objective terms. In case that we have homogeneous machines (i.e., all machines have the same computing power and network bandwidth), the optimal solution distributes d evenly across workers and m evenly across servers. Thus, we can derive an analytical solution that runs in $O(N)$. We present our analytical solution below, but due to space constraints, we omit its derivation.

$$\begin{aligned} &\mathbf{w}^* = \underset{\mathbf{w}}{\operatorname{argmin}} \Big[\frac{D}{\|\mathbf{w}\|} T(\|\mathbf{w}\|) \Big], \\ &\quad \text{where } T(\|\mathbf{w}\|) = C_{w.proc} + \frac{M}{b} \max(1, \frac{\|\mathbf{w}\|}{N - \|\mathbf{w}\|}) / B. \\ &\quad \mathbf{s}^* : s_i = 1 - \mathbf{w}_i{}^*, \quad \mathbf{d}^* : d_i = \frac{D}{\|\mathbf{w}^*\|}, \\ &\quad \mathbf{m}^* : m_j = \frac{M}{N - \|\mathbf{w}^*\|}, \quad b : \text{machines' bandwidth} \end{aligned} \quad (5)$$

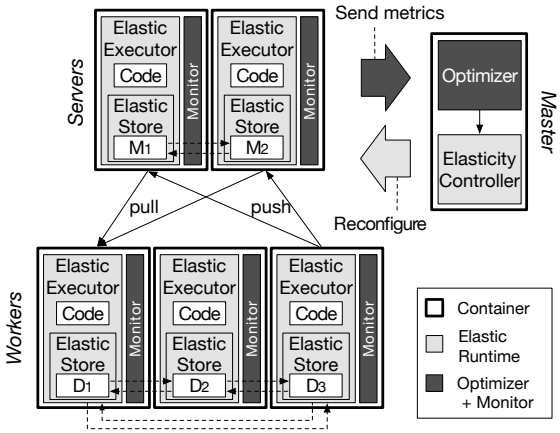

Fig. 4: Cruise Architecture.

IV. Cruise

We extend an existing PS system to automatically configure distributed ML execution. The extended system called Cruise adds Optimizer and Elastic Runtime to the PS system, as depicted in Figure 4. Optimizer estimates the optimal configuration for a running ML job using runtime metrics. Following the decision of Optimizer, Elastic Runtime applies the necessary changes dynamically to the system without stopping the running job.

A. Optimizer

Optimizer performs cost-based optimization by solving an optimal configuration problem formulated in Section III. Monitors collect runtime statistics related to the performance (e.g., the elapsed time for workers to compute gradients) and reports the metrics to Master periodically. Optimizer then estimates the performance in different system configurations based on the runtime status. By doing so, our optimizer does not require knowledge about the ML jobs (e.g., algorithms and hyper-parameters). After finding the configuration that is expected to be optimal, Optimizer maps the difference from the current configuration and generates an optimization plan, consisting of operations provided by Elastic Runtime. By executing the operations in the plan, Cruise changes the system configuration to the one with better performance. To achieve performance benefit with optimization, we need to make decisions such as when to calculate an optimization plan, whether to execute the plan or not. We describe these policies below.

1) Metric Collection: Cruise collects runtime metrics to use them as inputs to the Optimizer. Workers measure local computation time and communication time and report to Master at the end of every mini-batch. On the other hand, Servers report the metrics to Master periodically. Since the runtime metrics can fluctuate, we apply moving average to reduce noise.

2) Optimization Trigger Policy: Based on the cost model above, Cruise triggers optimization after collecting sufficient metrics to substitute the unknown variables in the cost model. We use metrics at the mini-batch granularity to be responsive to the changes of the running job. Using metrics from a configured number of subsequent mini-batches, we estimate the cost of an epoch.

In order to avoid the system from continuously reconfiguring back and forth around the estimated optimum, Optimizer predicts the performance benefit of a new configuration and skips that attempt if the gain is less than a certain threshold. A threshold number from our experience -5%- is good enough to prevent the system from "oscillating", while allowing the system to undergo moderately-sized optimizations.

When the amount of available resources (e.g., N) increases, Optimizer opportunistically tries to use the extra resources. If more resources become available, Optimizer can adjust to find an optimal configuration including the new resources. When the amount of available resources decreases, it rebalances execution accordingly.

3) Optimization Execution: Once the decision of a reconfiguration is made with the computed system configuration $(\mathbf{w}^*, \mathbf{s}^*, \mathbf{d}^*, \mathbf{m}^*)$, the new configuration is contrasted with the current configuration $(\mathbf{w}, \mathbf{s}, \mathbf{d}, \mathbf{m})$ to generate a reconfiguration plan. All plans consist of a subset of four Elastic Runtime operations, which we discuss in detail in Section IV-B. The operation `add` is for newly joining containers, while the operation `delete` deletes containers that are no longer assigned any data or partial model. The operation `switch` is to change an existing server container to worker or from worker to server. Training data and model partitioning, $(\mathbf{d}, \mathbf{m})$, can be modified by migrating data between containers to preserve the state of the running job, which we will further discuss in Section IV-B. The `move` operation migrates data between containers, starting with containers that have the largest training data or model changes in a greedy fashion, to minimize the amount of data to move and the number of movements.

Optimizer executes a plan by simply invoking Elastic Runtime API that reconfigures system transparently without stopping training. The simplest approach to execute the plan would be to invoke the operations sequentially. However, to make the reconfiguration agile, Optimizer generates the plan as directed acyclic graphs of independent operations that can be executed concurrently.

B. Elastic Runtime

Elastic Runtime is an execution environment that exposes operations which Optimizer can call to dynamically reconfigure the system. Elastic Runtime manages workers and servers in the form of containers, each integrated with an *Elastic Executor*. Elastic Executor runs application code on data encapsulated by *Elastic Store*, a distributed key-value store that constructs an effective management scheme. *Elasticity Controller* manages the distributed Elastic Executors. It is also the endpoint where Optimizer triggers reconfigurations according to the generated optimization plan.

Elastic Runtime deals with two types of reconfigurations: resource reconfiguration and workload repartitioning. Resource reconfiguration is achieved by Elastic Executor, a containerized and reconfigurable runtime which extends the existing

PS architecture's Executor. Elasticity Controller coordinates resource reconfiguration by easily adding and removing Elastic Executors. Workload repartitioning is conducted efficiently with Elastic Executor's internal component, Elastic Store. An Elastic Store encapsulates data in an in-memory storage with a management scheme that provides flexibility in the accommodated data type (e.g., training data or model data).

Transparency must be maintained in the course of a reconfiguration. Reconfiguration must occur with minimal effects to the running job by maintaining the application's access to data without any loss or significant overhead. Elastic Executor performs several additional tasks required for a transparent reconfiguration, such as adaptive data ownership management or redirection of requests to the new owner of data.

We explain the details on resource reconfiguration in Section IV-B1 and workload repartitioning in Section IV-B2 while maintaining transparency in Section IV-B3.

1) Resource Reconfiguration: Containers can be added or deleted when Optimizer determines so with operations `add` and `delete` for which the simple signatures are provided in Figure 5. When `add` is executed, Elastic Runtime simply launches an Elastic Executor on a new container. In the case of a container `delete`, Elastic Executor stops the app code and itself to release the container.

When deleting an executor, Elastic Runtime performs additional wrap up, corresponding to its role (e.g., worker or server). Before a server-side Elastic Executor shuts down, it redirects all remaining pull requests from workers to the new owning Elastic Executors to prevent workers from waiting long for a response. For worker-side, it waits until ongoing mini-batch to be finished and push requests are flushed to servers.

Elastic Runtime also provides `switch` operation that changes Elastic Executor to another type (e.g., from server to worker or from worker to server). This operation also involves the setup and cleanup procedure involved in `add` and `delete`. However, the two procedures occur in parallel in the existing container and there is no container setup or cleanup involved. This is especially beneficial in an environment with constrained resources as `add` must wait for a container to become free after a `delete` completes.

2) Workload Repartitioning: Workload repartitioning includes changing each container's ownership of training data/-partial models and migrating the data accordingly. A resource reconfiguration must occur in conjunction with workload repartitioning. When a container is added/deleted, the workload for each container must be readjusted across the new set of containers now running in the system. Workload repartitioning may occur on its own in the case of an imbalance in workload between containers.

Any runtime state of the job such as the model data across servers must be preserved, not to lose the job's progress. Thus, the states must be migrated from one container to another. In addition to such mutable data, training data across workers which remains unchanged can enjoy the benefit of migration to reduce the overhead of reloading the entire dataset in workload repartitioning. Both mutable and immutable data can be stored in Elastic Stores on which workload repartitioning occurs.

Data access interface	**Description**
`put(Key, Value)`	Puts (`Key`,`Value`) to Elastic Store
`get(Key)`	Gets the value associated with `Key`
`update(Key, Func, Delta)`	Updates the value for `Key` with the result of `Func(Value, Delta)`
Reconfiguration interface	**Description**
`add(ResourceConf, RuntimeConf)`	Adds new containers and starts runtime on them
`delete(Containers)`	Deletes existing containers
`switch(Container, RuntimeConf)`	Switches a container to run a specified runtime
`move(Blocks, SrcContainer, DstContainer)`	Moves blocks from one container to another

Fig. 5: Elastic Runtime Interfaces.

Data Storage and Ownership Management: Data management in Elastic Runtime involves a collection of Elastic Stores where the actual key-value tuples are stored. The actual ownership of each data instance is maintained by the respective Elastic Store, but Elasticity Controller also maintains a global ownership view to orchestrate migration between Elastic Stores. Ownership tables are updated during the migration process, which we will discuss the details below in this section.

Elastic Stores are composed of *blocks* containing data and a block is owned by exactly one Elastic Store. The entire key-space of data is partitioned and each block contains data for a range in the key-space. For an even partitioning of keys over blocks, each block stores data for a hashed key range. Clients of Elastic Stores - worker and server code in our paper - use a *key* which is mapped to a value to access each key-value tuple. For each client access, the only Elastic Store owning the block where the key-value tuple is stored processes the request according to the *ownership table*.

Data Access: Elastic Runtime allows values to be stored to and retrieved from Elastic Stores through simple operations, similar to what can be done in distributed hash tables (DHTs) [8]. The difference of Elastic Stores over such key-value stores is that Elastic Runtime exposes options to migrate data. Elastic Store provides simple and standard operations for clients to access and update each data instance with a key as shown in Figure 5. Elastic Runtime guarantees that operations are served exactly once by maintaining a single owner of the block containing the key-value tuple on which the operation is conducted across all Elastic Stores. When an operation is requested to the Elastic Store that does not own the block, the request is processed by remotely accessing the owner according to the ownership table in each Elastic Store.

In addition to the `put/get` operations, we provide `update` operation, which atomically executes `Func`, a user-defined function that should be commutative and accumulative

to guarantee atomic incremental updates.

In Cruise, when starting a job, a worker Elastic Executor loads its assigned set of training data using `put` into its local Elastic Store. While running the job, Elastic Executor fetches the data to process for each mini-batch from the local Elastic Store using `get`. Servers, however, must use `update` when processing a push request to guarantee atomicity. To process a pull request, servers simply `get` model data from the local Elastic Store.

Data Migration: `move` operation changes ownership and migrates data between Elastic Stores. This should be done carefully to prevent loss or duplicated processing of an operation, while changing block owner. It is also the most critical factor that determines reconfiguration performance and thus Elastic Runtime executes multiple `moves` concurrently, each `move` parallelized in block units.

We implement the following protocol in Elastic Runtime to provide an efficient migration process. Elasticity Controller initiates a migration for a set of blocks by sending a message to the source container. The source container migrates blocks concurrently to the destination container and reports Elasticity Controller about the completion of the migration for every block, upon each ACK message from the destination container. Finally, Elasticity Controller broadcasts ownership change of the block to all other containers. Specifically, the block migration is done in two distinct steps: ownership handover and actual data transfer. In the source container, when starting migration for a block it hands over ownership first, so access operations for the block in this container are redirected to the destination container. In the destination container, when it takes an ownership it starts queueing access operations for the block and starts processing them after receiving actual block data.

The key point in the migration process is that block ownership is transferred atomically such that there is always a single owner for a block. Another key point is that even if multiple blocks are requested for migration, client access to a key is blocked only during the actual migration of the block containing the key.

3) Transparency during Reconfiguration: Dynamic reconfiguration must occur without any extra work for Elastic Runtime's clients, but also must refrain from any performance degradation. Such *transparent* reconfigurations include the following requirements. First, client access APIs must be supported during reconfiguration, maintaining read-my-write consistency. Second, in effort to serve client access, overheads such as increased number of remote data accesses are inevitable due to resource reconfigurations. Such inefficiency must be minimized. Finally, the reconfiguration must guarantee that the accuracy of the model being learned is unaffected. Elastic Runtime reinforces the key requirements in achieving transparent reconfigurations with the following features.

Data accessibility: Data must be accessible any time during and after data migration for client access. Elastic Store enables remote access with the ownership table maintained atomically during the migration process.

Data locality: Though data is remotely accessible through local Elastic Stores, remote access is expensive. Elastic Executor aligns its workload partitioning with the actual data in its Elastic Store to maximize locality with the migration protocol. During a migration, when worker code running on Elastic Executor asks for a batch of data to process, a local set of data is guaranteed to be returned by keeping track of the keys for local training data.

Dynamic ownership table: In Cruise, workers send requests to specific servers containing the key of the partial model according to each worker's local ownership table. When the ownership update is immediately broadcasted to all worker Elastic Executors during migration, workers can immediately request to the new owner server. For requests that arrive at the old owner server prior to the worker-side ownership update, Elastic Executor of the old owner server refers to its ownership table and redirects the requests to the new owning server.

V. Evaluation

We implemented Cruise with around 20K lines of code in Java 1.8. We built Cruise on Apache REEF [22], a library for application development on cluster resource managers such as Apache YARN and Apache Mesos. REEF provides a control plane for data processing frameworks including the negotiation with the cluster resource manager and the control channel between containers.

We evaluate Cruise with three machine learning applications. Our evaluation mainly consists of the following four sections: (1) We compare the performance of our expected optimal configuration to that of the actual optimal configuration (Section V-B). (2) We demonstrate that Cruise reduces epoch time, speeding up training (Section V-C). (3) We show how Cruise optimizes the system configuration when resource availability changes (Section V-D) and in heterogeneous environments (Section V-E). (4) We investigate the overhead incurred while optimizing the system. (Section V-F).

A. Experimental Setup

Default cluster setup: We run experiments on AWS EC2 instances with YARN running on Ubuntu 14.04. Unless explicitly mentioned, we use 32 r4.xlarge instances, each of which has 4 virtual cores, 30.5GB RAM, and 1.25 Gbps network connection [3]. We launch one Elastic Executor per machine to run a worker or a server.

Workloads: We choose three popular ML workloads in different categories: recommendation, classification, and topic modeling as summarized in Table II.

Non-negative Matrix Factorization (NMF) is commonly used in recommendation systems. The main idea is to find undetermined entries in a given matrix. NMF factorizes a matrix M $(m \times n)$ into factor matrices L $(m \times r)$ and R $(r \times n)$, where $M \approx LR$. We implement NMF via the stochastic gradient descent (SGD) algorithm, similar to the one described in [21]. Matrix R is partitioned across servers, while L is

[3]AWS specifies that the r4.xlarge type provides *up to 10 Gbps* network bandwidth. We measured the actual bandwidth with iperf tool.

Application	Dataset	Hyper-parameter	Num. of model parameters
NMF	16x Netflix (1.9M users, 71K movies)	1K rank	1K * 71K
MLR	Synthetic sparse (100K samples, 160K features)	4K classes	4K * 160K
LDA	PubMed (8.2M documents, 141K words)	400 topics	400 * 141K

TABLE II: Description of datasets used in evaluation. The dataset is partitioned to workers. Num. of model parameters refers to the total number of parameters in model, which is partitioned to servers.

App.	Initial (W, S)	Cruise's Choice	Optimal (W, S)	Rel. Perf
NMF	(27, 5)	(24, 8)	(23, 9)	98.8%
	(18, 14)	(22, 10)		93.9%
MLR	(18, 14)	(26, 6)	(27, 5)	97.8%
	(23, 9)			
LDA	(27, 5)	(18, 14)	(18, 14)	100%
	(23, 9)			

TABLE III: Comparison between the configurations found by Cruise's Optimizer and the ground truth optimum found by the grid search for the cases in Table I. Relative performance is the epoch time in the optimal (W, S), divided by the epoch time in each configuration.

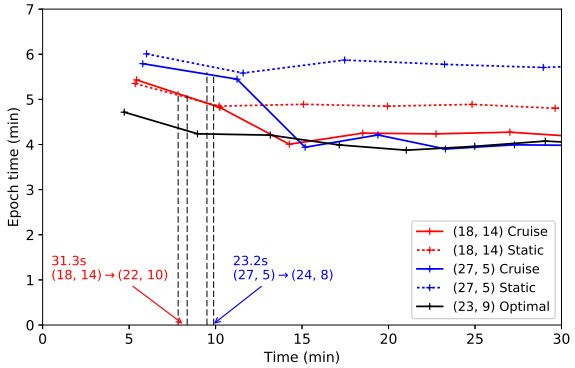

Fig. 6: Epoch time of an NMF job starting at 3 different configurations. The black line shows the global optimum and the blue and red lines show optimizations from the other initial configurations. The dotted lines show the performance without optimization to the corresponding colors. The vertical lines represent the reconfiguration of each cases.

partitioned across workers, where the smallest unit of training data is a single user's rating matrix ($1 \times n$). NMF experiments use the 16x Netflix dataset whose size is around 40 times greater than the one in the evaluation of [21]. We set mini-batch size to be 10K.

Multinomial Logistic Regression (MLR) is an algorithm for classification. Each d-dimensional observation $x \in R^d$ belongs to one of the M classes, with the model parameter size of $M \times d$. We also implement MLR using SGD. Our experiments use a synthetic dataset generated by a public script from the Petuum framework [3]. The dataset is around 46 times greater than the one used in the evaluation of [21]. We process 1K observations in a mini-batch for our experiments.

LDA is an algorithm to discover hidden properties (topic) from a group of documents. Each document consists of a bag of words, where LDA associates latent topic assignments. Our LDA implementation uses an efficient variant of the collapsed Gibbs sampling algorithm [25], which is widely used [17], [21]. We run LDA experiments using the PubMed dataset. Our dataset is 15 times larger than one of the datasets used in [21]. We process 1K documents in a mini-batch.

Optimizer setup: We observe that the performance of the initial mini-batches fluctuate until the system stabilizes. To prevent Optimizer from computing the cost inaccurately, we configure it to wait until all workers finish a set of mini-batches. In addition, Optimizer does not trigger reconfiguration if the estimated performance gain (in terms of the cost) is below 5% in order to avoid oscillation. As mentioned in Section III-C, we use the $O(N)$ analytical solver for the homogeneous environment and use the ILP solver for the heterogeneous environment.

B. Finding Baselines with Grid Search

Before evaluating Cruise's optimization, we find the baseline for all experiments. We simply perform a grid search that runs all possible configurations $(\mathbf{w}, \mathbf{s})$, to find the ground truth optimal configurations for the various experiments. Since such a grid search including $(\mathbf{d}, \mathbf{m})$ is quite complicated, we use heuristics to eliminate these variables. In the homogeneous environment, an even partitioning is intuitively optimal. The result of this grid search yields the 'Optimal (W, S)' column in Table III. In the heterogeneous environment, we distribute blocks proportional to each machine's power based on metrics including worker's local computation time and server's network bandwidth.

C. Optimization in the Homogeneous Environment

After performing the grid search, we run NMF, MLR, and LDA each starting with its optimal configuration and the optimal configurations for the other two applications (as the two configurations are reasonable starting points for users running the applications in the same cluster).

Cruise finds configurations close to the ground-truth optimum found in Section V-B for the various cases mentioned in Section II-B in the homogeneous environment. Table III shows the comparisons: In NMF and MLR, Cruise chooses near-optimal configurations where the number of machines for each role differs by one node compared to the optimum found by grid search. The resulting performance in terms of epoch time is slightly inferior to the optimum but the difference is smaller than 6.1%. Cruise finds the optimal configuration in LDA with the same performance as the optimum.

Figure 6 depicts how Cruise decreases the epoch times of NMF. Starting at (W:27, S:5), Cruise moves to (W:24, S:8), with the relative performance of 1.1% slower than the optimum at (W:23, S:9). With the initial (W:18, S:14) configuration,

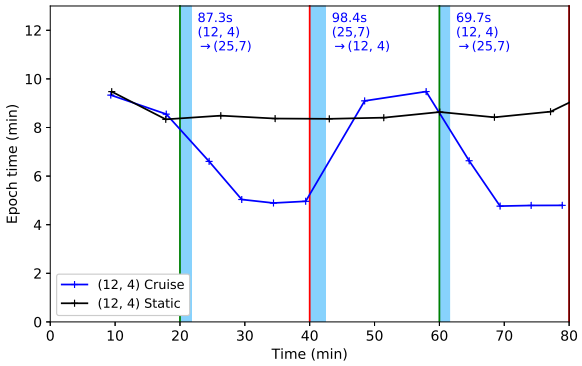

Fig. 7: Utilizing opportunistic resources in the NMF job. The blue line shows Cruise's ability to adapt to resource availability compared to the baseline drawn in black line, where the configuration is fixed to the initial 16 resources. Vertical lines represent the event of resource addition (green) and reclamation (red). The areas filled in sky-blue denote the reconfigurations.

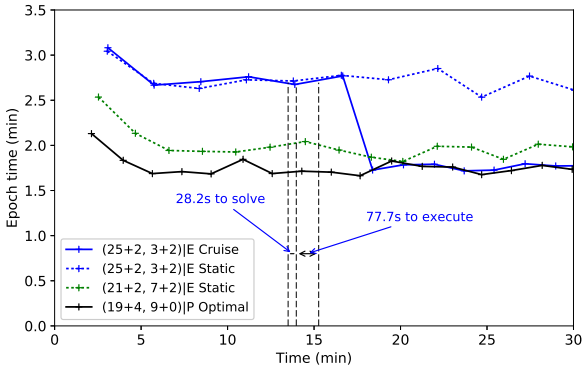

Fig. 8: Epoch time of NMF at different starting points in the heterogeneous environment. 4 machines are r4.4xlarge ("Faster") and the rest 28 are r4.xlarge ("Slower") EC2 instances. Notations of configuration in legend is defined in Section V-E.

Cruise reconfigures to (W:22, S:10), with 6.5% slower performance than the optimum. We observe that Cruise optimizes the misconfigured NMF jobs with significant drops in epoch time of 35.8% and 22.3% in each case, soon stabilizing to that of the new configuration. Our experiments with other applications in the same environment also decrease epoch time by 55.3% and 8.7% in MLR, and 37.5% and 17.4% in LDA.

D. Utilizing Opportunistic Resources

The previous experiments show how Cruise optimizes system configuration when available resources do not change. Cruise's capability to optimize system configuration during runtime, however, is more powerful when available resources change over time. Cruise keeps track of available resources in the cluster and updates the system configuration if there are changes in resource availability. In this experiment, we assume that the cluster has 16 extra containers that are available opportunistically; starting with 16 containers, we add/reclaim 16 containers at every 20 minutes. We show how Cruise's runtime optimization utilizes opportunistic resources by comparing cases with and without optimization. In both cases, we run ML jobs with the initial configuration of (W:14, S:2), the actual optimum found for 16 containers from experiments.

Figure 7 demonstrates that the average epoch time approximately halves when 16 more resources are available. At the 20th minute, the configuration moves toward (W:25, S:7), taking advantage of the added resources. The reconfiguration takes around 87.3 seconds, most of the overhead caused by the data migration of the half of total training data and model data, to the new containers. At the 40th minute, Cruise returns to the previous configuration (W:12, S:4) with 98.4 seconds of reconfiguration overhead for data migration and state cleanup. The additional resources become available again at the 60th minute and Cruise goes to (W:25, S:7), same as the previous optimization at the [20, 40] minutes time interval.

E. Optimization in the Heterogeneous Environment

There are two different aspects to the setup in the heterogeneous environment from the homogeneous environment. Workload should be partitioned differently corresponding to machine types and each type of machine should be assigned a more proper role (e.g., worker or server). The heterogeneous environment uses two types of machines: in addition to the 28 instances of r4.xlarge, we use 4 faster machines (r4.4xlarge) that have 16 virtual cores, 122 GB RAM, and 5.00 Gbps network connection. [4] In our experiments, we allocate the faster instances evenly to begin with, 2 for workers and 2 for servers. For block partitioning, we start all experiments with *even* partitioning denoted as 'E', whereas the optimal configuration distributes blocks *proportially* to machine's capability, which is denoted as 'P'. For example, we denote a configuration of 20 workers with 3 strong machines and 12 servers with 1 strong machine with even block partitioning as (W:17+3, S:11+1)|E. We run the same three applications with the same starting points as the homogeneous environment, and we show how differently Cruise optimizes the configuration in the heterogeneous environment.

Figure 8 shows the results of running NMF starting at (W:25+2, S:3+2)|E. Cruise reconfigures the job to (W:20+4, S:8+0)|P close to the ground truth optimum in the heterogeneous environment (W:19+4, S:9+0)|P. Data is repartitioned so that the faster instances have about 2 times more blocks than the slower instances, reflecting the heterogeneity. Epoch time decreases by 35.2% (from 162 s to 105 s). Our experiments with other applications in the same environment also reduce the epoch times by 58.3% in MLR starting at (W:16+2, S:12+2)|E and 41.3% in LDA starting at (W:25+2, S:3+2)|E.

To focus on the benefit of role reassignment and workload repartitioning, we run the job again at (W:21+2, S:7+2)|E, without any optimization (the green line). Here, we observe

[4] AWS specifies that the r4.4xlarge type provides *up to 10 Gbps* network bandwidth. We measured the actual bandwidth with iperf tool.

that this static configuration is 12.4% slower (118 s vs. 105 s) than the epoch time for the configuration chosen by Cruise.

F. Reconfiguration Speed

The optimization procedure is composed of cost calculation and plan execution. Cost calculation takes 30 ms for the homogeneous environment and 37.4 s for the heterogeneous environment on average. More time is spent on plan execution, especially for `move`: the overhead includes (de)serialization time, network transfer time, and the time to acquire the lock on the block to migrate. The data size also affects the time to execute the operation. The overhead of `add` and `delete` is relatively small, which takes around 2~3 s for resource initialization and cleanup. The time for `switch` is more negligible since there is no cost for resource setup.

Here we break down the plan execution of an NMF experiment that starts from (W:18, S:14) in Section V-C. The plan changes the configuration to (W:22, S:10), composed of 4 `switch`es from server to worker and 30 `move`s that repartition data and model blocks. It takes 31.3 s in total for our plan executor to execute these operations in parallel. Most of this time is taken during worker-side `move`s, due to the huge size of data being migrated. Input data is divided into 200 worker blocks. A worker block is 100 MB, each of which contains 10K items. 18 `move`s migrate 36 data blocks in total between worker executors. The longest `move` takes 25.8 s, migrating 4 blocks at once, serving as the bottleneck to this plan execution time. On the other hand, model data is divided into 128 server blocks. Block size is 2 MB and each contains only 280 items. The plan migrates 36 model blocks with 12 `move`s. Server-side `move`s take at most 1.4s.

VI. Related Work

An early version of our work appeared in a workshop paper that sketched a high-level approach to optimizing PS system configurations [6]. This paper presents the complete problem formulation, design, implementation, and evaluation of the system. Cruise differs from other systems in that it automatically tunes PS system configuration by solving the configuration optimization problem with runtime PS system metrics and applying the solution to the running system efficiently. Below we summarize the works that are most relevant to Cruise.

TuPAQ [18] is a system for identifying ML model configurations (e.g., support vector machine vs. logistic regression, hyper-parameter values) that lead to high performance in terms of model accuracy, built on Apache Spark [14], [26]. TuPAQ casts ML model identification as a query planning problem and applies a bandit allocation strategy as well as various optimizations such as batching, optimal cluster sizing, and advanced hyper-parameter tuning techniques to solve the problem efficiently. This study focuses on supervised ML models. In contrast, Cruise addresses the problem of tuning system configuration in the PS architecture.

SystemML [13] is a hybrid runtime system that uses in-memory Control Program (CP) and MapReduce (MR) jobs to run declarative ML programs. There also is another version of the work [2] that applied the same concept to Spark [26], regarding Spark Driver as the CP and Executors as the worker jobs. The system focuses on optimizing memory configurations during runtime when there is a change in available resources. On the other hand, Cruise optimizes the running time of ML applications on the PS architecture by automatically tuning system configuration.

Yan et al. [24] propose a cost formulization that predicts the computation and communication overheads of Deep Neural Network (DNN) applications by modeling the internals of the algorithm. In contrast, Cruise measures computation and communication time at runtime instead of modeling the internals of the algorithm, uses the runtime measurement for cost-based optimization, and applies the estimated optimal system configuration by reconfiguring a running job, which allows us to take advantage of resource elasticity.

Starfish [11], built on Apache Hadoop [19], performs optimization on MapReduce. It gathers job profiles from runtime statistics via dynamic instrumentation for job-level tuning, guaranteeing shorter execution times. Many of Starfish's design considerations come from the MapReduce programming model, while Cruise targets ML applications running on the PS architecture.

Recent works like Bösen [21], Ako [20], and MALT [15] do not decide on the number of workers and servers since each node runs both worker and server. They have different styles of model synchronization. Bösen is a PS implementation, which requires all-to-all communication. Ako is a peer-to-peer DNN training system. Ako exchanges partial gradients across multiple rounds to adjust the hardware and statistical efficiency. Similarly, MALT is a peer-to-peer ML training system where each node exchanges parameter updates with $\log n$ nodes deterministically. In contrast, Cruise employs cost-based optimization, allows flexible worker and server allocation, handles elastically changing resources, and considers heterogeneous environments.

VII. Conclusion

In this paper, we present a methodology to automatically tune system configuration of PS-based ML systems. We build Cruise by extending an existing PS-based system to optimize system configuration based on the methodology. Cruise Optimizer estimates the optimal system configuration - resource configuration and workload partitioning - using a cost-based model with runtime metrics. Elastic Runtime enables efficient runtime reconfigurations according to the computed optimal configuration. Our evaluation shows that Cruise frees ML application developers of choosing right system configuration by tuning system configuration automatically. Cruise is publicly available at https://github.com/snuspl/cruise.

Acknowledgement

We thank all reviewers for their comments. This research was supported by Samsung Research Funding Center of Samsung Electronics under Project Number SRFC-TC1603-01.

REFERENCES

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

[2] M. Boehm, M. W. Dusenberry, D. Eriksson, A. V. Evfimievski, F. M. Manshadi, N. Pansare, B. Reinwald, F. R. Reiss, P. Sen, A. C. Surve, et al. SystemML: Declarative machine learning on spark. *Proceedings of the VLDB Endowment*, 9(13):1425–1436, 2016.

[3] Carnegie Mellon University. Petuum Bösen, 2016. https://github.com/petuum/bosen.

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

[5] T. Chilimbi, Y. Suzue, J. Apacible, and K. Kalyanaraman. Project adam: Building an efficient and scalable deep learning training system. In *OSDI*, 2014.

[6] B.-G. Chun, B. Cho, B. Jeon, J. S. Jeong, G. Kim, J. Y. Kim, W.-Y. Lee, Y. S. Lee, M. Weimer, Y. Yang, and G.-I. Yu. Dolphin: Runtime optimization for distributed machine learning. In *NIPS ML Sys Workshop*, 2016.

[7] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, A. Senior, P. Tucker, K. Yang, Q. V. Le, et al. Large scale distributed deep networks. In *NIPS*, 2012.

[8] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. *SIGOPS*, 2007.

[9] I. Gurobi Optimization. Gurobi optimizer, 2017.

[10] A. Harlap, H. Cui, W. Dai, J. Wei, G. R. Ganger, P. B. Gibbons, G. A. Gibson, and E. P. Xing. Addressing the straggler problem for iterative convergent parallel ml. In *SoCC*, 2016.

[11] H. Herodotou, H. Lim, G. Luo, N. Borisov, L. Dong, F. B. Cetin, and S. Babu. Starfish: A self-tuning system for big data analytics. In *CIDR*, volume 11, pages 261–272, 2011.

[12] Q. Ho, J. Cipar, H. Cui, S. Lee, J. K. Kim, P. B. Gibbons, G. A. Gibson, G. Ganger, and E. P. Xing. More effective distributed ml via a stale synchronous parallel parameter server. In *NIPS*, 2013.

[13] B. Huang, M. Boehm, Y. Tian, B. Reinwald, S. Tatikonda, and F. R. Reiss. Resource elasticity for large-scale machine learning. In *SIGMOD*, 2015.

[14] T. Kraska, A. Talwalkar, J. C. Duchi, R. Griffith, M. J. Franklin, and M. I. Jordan. MLbase: A distributed machine-learning system. In *CIDR*, volume 1, pages 2–1, 2013.

[15] H. Li, A. Kadav, E. Kruus, and C. Ungureanu. MALT: Distributed data-parallelism for existing ml applications. In *EuroSys*, 2015.

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

[17] A. Smola and S. Narayanamurthy. An architecture for parallel topic models. In *Proceedings of the VLDB Endowment (PVLDB)*, volume 3, pages 703–710, Sept. 2010.

[18] E. R. Sparks, A. Talwalkar, D. Haas, M. J. Franklin, M. I. Jordan, and T. Kraska. Automating model search for large scale machine learning. In *SoCC*, 2015.

[19] The Apache Software Foundation. Apache Hadoop, 2015. http://hadoop.apache.org.

[20] P. Watcharapichat, V. L. Morales, R. C. Fernandez, and P. Pietzuch. Ako: Decentralised deep learning with partial gradient exchange. In *SoCC*, 2016.

[21] J. Wei, W. Dai, A. Qiao, Q. Ho, H. Cui, G. R. Ganger, P. B. Gibbons, G. A. Gibson, and E. P. Xing. Managed communication and consistency for fast data-parallel iterative analytics. In *SoCC*, 2015.

[22] M. Weimer, Y. Chen, B.-G. Chun, T. Condie, C. Curino, C. Douglas, Y. Lee, T. Majestro, D. Malkhi, S. Matusevych, B. Myers, S. Narayanamurthy, R. Ramakrishnan, S. Rao, R. Sears, B. Sezgin, and J. Wang. Reef: Retainable evaluator execution framework. In *SIGMOD*, 2015.

[23] E. P. Xing, Q. Ho, W. Dai, J. K. Kim, J. Wei, S. Lee, X. Zheng, P. Xie, A. Kumar, and Y. Yu. Petuum: A new platform for distributed machine learning on big data. In *SIGKDD*, 2015.

[24] F. Yan, O. Ruwase, Y. He, and T. Chilimbi. Performance modeling and scalability optimization of distributed deep learning systems. In *SIGKDD*, 2015.

[25] L. Yao, D. Mimno, and A. McCallum. Efficient methods for topic model inference on streaming document collections. In *SIGKDD*, 2009.

[26] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, 2012.