Orpheus: Efficient Distributed Machine Learning via System and Algorithm Co-design

Pengtao Xie
Petuum Inc and Carnegie Mellon University
Pittsburgh, PA
pengtao.xie@petuum.com

Jin Kyu Kim
Carnegie Mellon University
Pittsburgh, PA
jinkyuk@andrew.cmu.edu

Qirong Ho
Petuum Inc
Pittsburgh, PA
qirong.ho@petuum.com

Yaoliang Yu
University of Waterloo
Waterloo, ON, Canada
yaoliang.yu@uwaterloo.ca

Eric Xing
Petuum Inc
Pittsburgh, PA
eric.xing@petuum.com

ABSTRACT

Numerous existing works have shown that, key to the efficiency of distributed machine learning (ML) is proper system and algorithm co-design: system design should be tailored to the unique mathematical properties of ML algorithms, and algorithms can be re-designed to better exploit the system architecture. While existing research has made attempts along this direction, many algorithmic and system properties that are characteristic of ML problems remain to be explored. Through an exploration of system-algorithm co-design, we build a new decentralized system Orpheus to support distributed training of a general class of ML models whose parameters are represented with large matrices. Training such models at scale is challenging: transmitting and checkpointing large matrices incur substantial network traffic and disk IO, which aggravates the inconsistency among parameter replicas. To cope with these challenges, Orpheus jointly exploits system and algorithm designs which (1) reduce the size and number of network messages for efficient communication, 2) incrementally checkpoint vectors for light-weight and fine-grained fault tolerance without blocking computation, 3) improve the consistency among parameter copies via periodic centralized synchronization and parameter-replicas rotation. As a result of these co-designs, communication and fault tolerance costs are linear to both matrix dimension and number of machines in the network, as opposed to being quadratic in existing systems. And the improved parameter consistency accelerates algorithmic convergence. Empirically, we show our system outperforms several existing baseline systems on training several representative large-scale ML models.

CCS CONCEPTS

• **Computer systems organization** → **Peer-to-peer architectures**;

SoCC'18, October 11-13, Carlsbad, CA, USA

ACM ISBN 978-1-4503-6011-1/18/10…$15.00
https://doi.org/10.1145/3267809.3267810

KEYWORDS

Distributed machine learning, system algorithm co-design, sufficient factor broadcasting, matrix-parameterized models

ACM Reference Format:
Pengtao Xie, Jin Kyu Kim, Qirong Ho, Yaoliang Yu, and Eric Xing. 2018. Orpheus: Efficient Distributed Machine Learning via System and Algorithm Co-design. In *Proceedings of ACM Symposium of Cloud Computing conference, Carlsbad, CA, USA, October 11-13 (SoCC'18),* 13 pages.
https://doi.org/10.1145/3267809.3267810

1 INTRODUCTION

Large-scale machine Learning (ML) is increasingly becoming indispensable for applications such as speech recognition, machine translation, image classification, online advertising. The growing need of analyzing big data, and thereof training sophisticated models such as deep neural networks [24], has led to the development of several distributed ML systems [1, 12, 13, 17, 27, 37].

To execute ML algorithms efficiently on a distributed system, it is important to perform *system and algorithm co-design*. On one hand, ML algorithms possess unique properties such as iterativeness and error tolerance [22], which can be leveraged to design efficient systems. On the other hand, based on the features of the system such as parallelism and (a)synchronicity, ML algorithms can be adjusted for more efficient execution.

System and algorithm co-design has been explored in existing distributed ML systems. Ho et al. [22] leveraged the error-tolerance nature of ML algorithms to propose a new consistency model called staleness synchronous parallel, which allows workers to proceed at slightly different pace but still guarantees the correctness of execution. Kim et al. [23] proposed a structure-aware dynamic scheduling approach inspired by the non-uniform convergence property of model parameters and their dependency structure. Abadi et al. [1] built a dataflow-graph based system that is tailored to the backpropagation algorithm.

Our work falls into this line of research. Through a system and algorithm co-design, we build the Orpheus system for training matrix-parameterized models (MPMs), a general class of ML models which include popular methods such as deep neural networks and topic models [9], where the parameters are represented by matrices (but as shown in [12, 38], the parameter-update matrices at each logical time

(or clock) can be computed from much lighter-weight vectors referred to as *sufficient vectors* (SVs)). In large-scale MPMs, these matrices can contain billions of elements. Existing systems [1, 11, 37] communicate and checkpoint these matrices directly, which incur substantial network traffic and disk IO, and compromise parameter-replicas' consistency.

1.1 Contributions

To address these challenges, we simultaneously exploit system design (SD) and algorithm design (AD) that leverage several mathematical properties of the MPMs and their training techniques, and thereupon develop an architecture that: (1) reduces communication cost in terms of the amount of network traffic and the number of communication messages; (2) saves disk bandwidth and reduces IO waiting for checkpointing snapshots; and (3) reduces inconsistency among parameter replicas. Most importantly, such savings are achieved without compromising the mathematical equivalence of the new ML solution to the original one.

In communication, on top of the peer-to-peer (P2P) architecture for transfer of SVs [12, 38, 42], we contribute an AD – *SV selection* and an SD – *random multicast* to further reduce the size of each network message and the number of messages. In fault tolerance, based on an AD – *using SVs to represent parameter states*, an SD – *incremental SV checkpoint* (ISVC) – is used, which continuously saves new SVs computed at each clock to stable storage. Compared with checkpointing parameter matrices, ISVC reduces disk IO, avoids compute-cycle waste and provides fine-grained (per-clock) rollbacks. In consistency models, Orpheus uses SDs including *periodic centralized synchronization* and *parameter-replicas rotation* to mitigate the incoherence among parameter replicas. In addition, we provide an easy-to-use programming model where the generation of SVs is automatically identified rather than being specified by users, which reduces users' programming efforts.

We evaluate Orpheus on three large scale applications and demonstrate that our system is efficient and scalable.

Previous works [12, 38, 42] have explored the SVs for efficient communication. Chilimbi et al. [12] proposed to reduce communication cost in parameter servers by transmitting SVs instead of matrices from workers to the server. Xie et al. [38] designed a peer-to-peer SV broadcasting framework. Two works [26, 38] proposed to reduce the number of network messages in P2P frameworks by replacing broadcasting with multicast. Our work is inspired by these existing SV-based systems and makes the following new contributions: (1) leveraging SVs to reduce the costs of fault tolerance; (2) new communication design – SV selection – for further reduction of communication cost; (3) automatic identification of SVs; (4) new protocols for alleviating the parameter-replica inconsistency resulted from decentralized synchronization; (5) GPU support.

2 MATRIX-PARAMETRIZED ML MODELS

For ease of understanding, we begin with a very brief introduction of basic ML concepts underlying our system-algorithm co-design principles. To accomplish a task (e.g., detecting faces from images), one designs a mathematical *model* (e.g., neural network (NN)). The model has value-unknown *parameters* (e.g., weights associated with the connections in an NN) and the goal is to determine (called *training*) the optimal parameter values that yield the best performance (e.g., face-detection accuracy). To achieve this goal, a set of *training data examples* (e.g., images and annotated face regions) are provided. Each example is represented by a *feature vector* (e.g., concatenating the pixel values into a vector to represent an image) and (optionally) a *label* (e.g., an annotated face region). On the training data, one defines a *loss function* to measure the discrepancy between the prediction made by the model (e.g., an image region that the model believes contains a face) and the groundtruth annotated by a human (e.g., whether this region contains a face). An *algorithm* (e.g., stochastic gradient descent (SGD)) is employed to minimize this loss to obtain the optimal parameters. During algorithm execution, the parameters are stateful and iteratively refined. Starting from an initial guess (called *initialization*) of the parameter values, the algorithm iteratively executes the following steps until these values do not change anymore (called *convergence*): (1) randomly selecting one (or a small batch of) training example(s); (2) computing an *update* (e.g., gradient in SGD) of the parameters based on their current state and the selected examples; (3) applying the update to the parameters.

In many ML models, the parameters are represented as a matrix. For instance, *multiclass logistic regression*, which classifies a K-dimensional feature vector into one of J classes, has a $J \times K$ parameter matrix where the j-th row-vector contains the weights of class j. Other examples include topic models [9], deep learning models [12, 24], distance metric learning [39], sparse coding [32], matrix factorization, and so on. These MPMs are widely used in computer vision, natural language processing, computational biology, physical sciences, advertisement, recommendation systems, to name a few.

2.1 Sufficient Vectors

For a large subset of MPMs, during training, the update matrix can be computed from a few vectors (e.g., the outer product of two vectors), which are referred to as *sufficient vectors* [12, 38] in the sense that they are sufficient to produce the update. Many ML models bear such a property, such as the ones mentioned above. Additional instances include quasi-Newton methods [5], restricted Boltzmann machine [21], group Lasso [40], and so on. The following examples present a detailed illustration.

- **Multiclass Logistic Regression (MLR)** [8]. Given the parameter matrix W and a feature vector x, MLR makes a prediction $p = \gamma(W * x)$ where γ denotes the softmax operator. Based upon N training examples, W is trained by minimizing the loss function $\sum_{i=1}^{N} \ell(\gamma(W * x_i), t_i)$ where t_i is a vector representing the class label and ℓ denotes the cross-entropy loss. When MLR is trained with SGD, the update matrix generated with respect to example i is the outer product of vector $p_i - t_i$ and x_i.
- **Topic Modeling (TM)** [25] is widely utilized for discovering topics from documents. It is parameterized by a $J \times K$ *topic matrix* B, where J is the vocabulary size and K is the number of topics. To model a document which is represented with a bag-of-words vector d, TM takes a weighted sum $B * a$ of the K topics vectors where a is a weight vector and uses the squared L2 loss $\|d - B * a\|_2^2$ to measure the modeling performance. When SGD is used to train B, the update matrix generated with respect to document d is the outer product of two vectors $(B * a - d)$ and a.

- **Recurrent Neural Network (RNN)** [30] is a premier deep learning model for sequence modeling. For the data example at time t, which is represented with a feature vector x_t, RNN computes a *state vector* $h_t = \sigma(U * h_{t-1} + V * x_t)$, where h_{t-1} is the state vector at time $t-1$, U and V are two parameter matrices and $\sigma(\cdot)$ is an element-wise nonlinear transformation. The parameters are trained using SGD where the gradients are computed using a *backpropagation* procedure. The update of U computed at time t is the outer product of e_t (the *error vector* computed via backpropagation) and h_{t-1}. Likewise, the update of V is the outer product of e_t and x_t.
- **Quasi-Newton Methods** [5] are a family of optimization algorithms, including DFP, BFGS, Broyden, SR1 and so on. These methods are used to learn ML models parameterized by a vector. However, they need to maintain an (approximated) Hessian matrix in memory and update it iteratively during execution. The update of this matrix can be computed from several vectors. For instance, the update in BFGS is $u * u^\top - vv^\top$, where u and v are vectors.

By properly leveraging the sufficient vector property of these MPMs, we can greatly reduce the communication and fault tolerance costs in distributed learning of these models, as shown in the rest of the paper. It is worth noting that not all MPMs have an SV property. For example, in hidden Markov model, the update of the transition matrix cannot be computed from a few vectors.

3 SYSTEM AND ALGORITHM CO-DESIGN

Orpheus is a decentralized system that executes data-parallel[1] [12–14, 27, 37] distributed training of matrix-parameterized ML models. Its architecture is shown in Figure 1. Orpheus runs on a group of worker machines connected via a peer-to-peer (P2P) network. Unlike the client-server architectures including the parameter server [12–14, 22, 27, 37], machines in Orpheus play equal roles without server/client asymmetry and every pair of machines can communicate. The entire dataset is divided into equally-sized data partitions (denoted by D). Each machine loads one partition into the main memory from a network file system or other file systems and holds one replica (denoted by W) of the model parameters. Machines synchronize their model replicas to ensure consistency, by exchanging SVs (denoted by u and v) via network communication. Consistency refers to that the parameter values of different replicas are as close as possible. High consistency is crucial for the convergence of algorithms.

At a high-level, Orpheus leverages the SV update property of MPMs to transform matrix-based operations to vector-based operations, to achieve cost reduction in communication and fault tolerance. On one hand, since the update matrix can be computed from a few vectors, one can transmit the vectors among machines and reconstruct the update matrices at the receiver side, instead of directly transmitting full matrices. The vectors have much smaller size than the full matrix, hence communication cost can be greatly reduced. On the other hand, the parameter matrix can be represented using a set of vectors as well. This fact can be explored to improve checkpoint-based fault tolerance and recovery. Instead of saving the large-sized parameter matrix, we can choose to save the vectors representing this matrix, which substantially reduces disk IO.

[1] Model-parallelism will be left for future study.

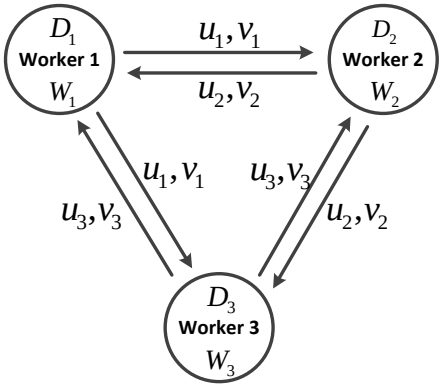

Figure 1: Architecture of Orpheus.

Around these ideas, Orpheus builds a battery of system-algorithm co-designs. For efficient communication, on top of previous works [12, 38] which transmit SVs instead of update matrices for parameter synchronization, we make two new contributions. First, we propose *SV selection*, which chooses a subset of representative SVs to communicate, to reduce the size of each message. Second, we propose *random multicast*, under which each machine sends SVs to a randomly-chosen subset of machines, to reduce the number of messages.

For light-weight fault tolerance, on the algorithm side, Orpheus represents the parameter states using a dynamically growing set of SVs. On the system side, Orpheus uses *incremental SV checkpoint*. Machines continuously save the new SVs computed in each logical time onto stable storage. To recover a parameter state, Orpheus transforms the saved SVs into a matrix. Compared with checkpointing parameter matrices [1, 41], saving vectors requires much less disk IO and does not require the application program to halt. Besides, the parameters can be rollbacked to the state at any logical time.

In programming abstraction, the SVs are explicitly exposed such that system-level optimizations based on SVs can be exploited. For a large family of applications, Orpheus is able to automatically identify the symbolic expressions representing SVs and updates, relieving users' burden of manually specifying them.

Orpheus supports two consistency models: Bulk Synchronous Parallel (BSP) [6] and Staleness Synchronous Parallel (SSP) [22]. BSP sets a global barrier at each clock. A worker cannot proceed to the next clock until all workers reach this barrier. SSP allows workers to have different paces as long as their difference in clock is no more than a user-defined *staleness* threshold.

3.1 Programming Model

In the Orpheus programming model, users specify how to generate SVs and how to reconstruct the update matrix from SVs. As we will show in Section 3.1.1, the latter can be skipped in certain applications where the SVs can be automatically identified by the system. This programming model is different from those of existing systems such as TensorFlow, MXNet, Spark, etc. In these existing programming models, users specify how to compute update matrices from data examples and how to refresh the parameter matrix using the updates.

Algorithm 1 Execution Semantics

```
Allocate an empty SVG set S = {}
Select a small batch of training examples X
for each x ∈ X
    Compute a SVG s = compute_svg (W^(old), x)
    Add s to S
end for
Send S to other machines
Receive the SVG set T from other machines
for each s ∈ S ∪ T
    Compute an update u = compute_update(s)
    Update parameters W^(new) ← W^(old) + u
end for
```

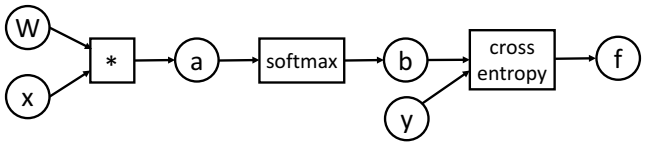

Figure 2: Expression Graph

The Orpheus programming model is SV-centric while the existing programming models are update-matrix-centric.

Orpheus programming model provides a data abstraction called Sufficient Vector Group (SVG) and two user-defined functions that generate and consume SVGs to update model parameters. Each SVG contains a set of SVs that are generated with respect to one data example and atomically produces a parameter update. The SVs are immutable and dense, and their default type is `float`. Inside an SVG, each SV has an index. To program an Orpheus application, users specify two functions: (1) `compute_svg` which takes the current parameter state and one data example as inputs and computes vectors that collectively form an SVG; (2) `compute_update` which takes an SVG and produces a parameter update. These two functions are invoked by the Orpheus engine to perform *data-parallel* distributed ML: each of the P machines holds one shard of the training data and a replica of parameters; different parameter replicas are synchronized across machines to retain consistency. Every machine executes a sequence of operations iteratively: in each clock, a small batch of training examples are randomly selected from the data shard and `compute_svg` is invoked to compute an SVG w.r.t each example; the SVGs are then sent to other machines for parameter synchronization; `compute_update` is invoked to transform locally-generated SVGs and remotely-received SVGs into updates which are subsequently added to the parameter replica. The execution semantics (per-clock) of Orpheus engine is shown in Algorithm 1. Unlike existing systems directly computing parameter updates from training data, Orpheus breaks this computation into two steps and explicitly exposes the intermediate SVs to users, enabling SV-based system-level optimizations to be exploited.

3.1.1 Automatic Identification of SVs and Updates. When ML models are trained using gradient descent or quasi-Newton algorithms, the computation of SVGs and updates can be automatically identified by the Orpheus engine, which relieves users from writing the two functions `compute_svg` and `compute_update` and eases programming. The only input required from users is a symbolic expression of the loss function, which is in general much easier to program compared with the two functions. Note that this is not an extra burden: in most ML applications, users need to specify this loss function to measure the progress of execution. By contrast, in [38], these two functions are required from users.

Type	Inputs	Outputs	Examples
1	vector	scalar	L2 norm
2	vector	vector	softmax
3	vector, vector	scalar	cross entropy
4	vector, vector	vector	addition, subtraction
5	matrix, vector	vector	matrix-vector product

Table 1: Different Types of Operators

The identification procedure of SVs depends on the optimization algorithm – either gradient descent or quasi-Newton – specified by the users for minimizing the loss function. For both algorithms, automatic differentiation techniques [1, 3] are needed to compute the derivative of variables. Given the symbolic expression of the loss function, such as `f=cross_entropy(softmax(W*x),y)` in MLR, the Orpheus engine first parses it into an expression graph [7] as shown in Figure 2. In the graph, circles denote variables including terminals such as `W`, `x`, `y` and intermediate ones such as `a=W*x`, `b=softmax(a)`; boxes denote operators applied to variables. According to their inputs and outputs, operators can be categorized into different types, shown in Table 1. In the sequel, we use $\partial b/\partial a$ to denote the derivative of b which is a scalar w.r.t to a which could be a scalar or a vector. Given the expression graph, Orpheus uses automatic differentiation to compute the symbolic expressions of the derivative $\partial f/\partial z$ of f w.r.t to each unknown variable z (either a terminal or an intermediate one). The computation is executed recursively in the backward direction of the graph. For example, in Figure 2, to obtain $\partial f/\partial a$, we first compute $\partial f/\partial b$, then transform it into $\partial f/\partial a$ using an operator-specific matrix A. For a type-2 operator (e.g., softmax) in Table 1, $A_{ij} = \partial b_j/\partial a_i$.

If W is involved in a type-5 operator (Table 1) which takes W and a vector x as inputs and produces a vector a and the gradient descent algorithm is used to minimize the loss function, then the SVG contains two SVs which can be automatically identified: one is $\partial f/\partial a$ and the other is x. Accordingly, the update of W can be automatically identified as the outer product of the two SVs.

If quasi-Newton methods are used to learn ML models parameterized by a vector x, Orpheus can automatically identify the SVs of the update of the approximated Hessian matrix W. First of all, automatic differentiation is applied to compute the symbolic expression of the derivative $g(x) = \partial f/\partial x$. To identify the SVs at clock k, we substitute the states x_{k+1} and x_k of the parameter vector in clock $k+1$ and k into $g(x)$ and calculate a vector $y_k = g(x_{k+1}) - g(x_k)$. We also compute another vector $s_k = x_{k+1} - x_k$. Then based on s_k, y_k and W_k (the state of W at clock k), we can identify the SVs which depend on the specific quasi-Newton algorithm instance. For BFGS, the procedures are: (1) set $y_k \leftarrow y_k/\sqrt{y_k^\top s_k}$; (2) compute $v_k = W_k s_k$; (3) set $y_k \leftarrow y_k/\sqrt{s_k^\top v_k}$. Then the SVs are identified

Algorithm 2 Joint Matrix Column Subset Selection

Input: $\{X^{(p)}\}_{p=1}^{P}$
Initialize: $\forall\, p, X_0^{(p)} = X^{(p)}, S_0^{(p)} = []$
for $t \in \{1, \ldots, C\}$ **do**
  Compute the squared L2 norm of column vectors in $\{X_{t-1}^{(p)}\}_{p=1}^{P}$
  Sample a column index i_t
  $\forall\, p, X_t^{(p)} \leftarrow X_{t-1}^{(p)} / \{x_{i_t}^{(p)}\}, S_t^{(p)} \leftarrow S_{t-1}^{(p)} \cup \{x_{i_t}^{(p)}\}$
  $\forall\, p, X_t^{(p)} \leftarrow X_t^{(p)} - S_t^{(p)} (S_t^{(p)})^{\dagger} X_t^{(p)}$
end for
Output: $\{S^{(p)}\}_{p=1}^{P}$

as y_k and v_k and the update of W_k is computed as $y_k y_k^\top - v_k v_k^\top$. For DFP, the procedures are: (1) set $s_k \leftarrow s_k / \sqrt{y_k^\top s_k}$; (2) compute $v_k = W_k y_k$; (2) set $v_k \leftarrow v_k / \sqrt{y_k^\top v_k}$. Then the SVs are identified as s_k and v_k and the update of W_k is computed as $s_k s_k^\top - v_k^\top v_k$.

3.2 Communication

Orpheus explores system-algorithm co-design to perform efficient communication. To ensure the consistency among different parameter replicas, the updates computed at different machines need to be exchanged. In parameter server architectures [12–14, 27, 37], large matrices need to be communicated among machines: (1) update matrices are sent from workers to servers and (2) parameter matrices are sent from servers to workers, which incur substantial communication overhead. The SV property is leveraged in [12] to reduce one-sided communication cost from workers to the server by transmitting SVs. But from servers to workers, the newly-updated parameters need to be sent as a matrix, which still incurs high communication overhead. Recently, several works [26, 38, 42] design a decentralized peer-to-peer (P2P) architecture where worker machines synchronize their parameter replicas by exchanging updates in the form of SVs, as shown in Figure 1. In each clock, each worker computes SVs and broadcasts them to other workers; meanwhile, each worker converts the SVs received remotely into update matrices which are subsequently added to its parameter replica. SV-transfer can reduce communication overhead from $O(JK)$ to $O(J+K)$. On top of the SV-transfer idea, we perform algorithm and system designs to further improve communication efficiency.

3.2.1 SV Selection. In ML practice, parameter updates are usually computed over a small batch (whose size typically ranges from tens to hundreds) of examples. At each clock, a batch of B training examples are selected and an update is generated with respect to each example. When represented as matrices, these B updates can be aggregated into a single matrix to communicate. Hence the communication cost is independent of B. However, this is not the case in sufficient vectors transfer: the B SVGs cannot be aggregated into one single SVG; they must be transferred individually. Therefore, communication cost grows linearly with B. To alleviate this cost, Orpheus provides *SV selection* (SVS), which chooses a subset of C SVGs (where $C < B$) – that best represent the entire batch – to communicate.

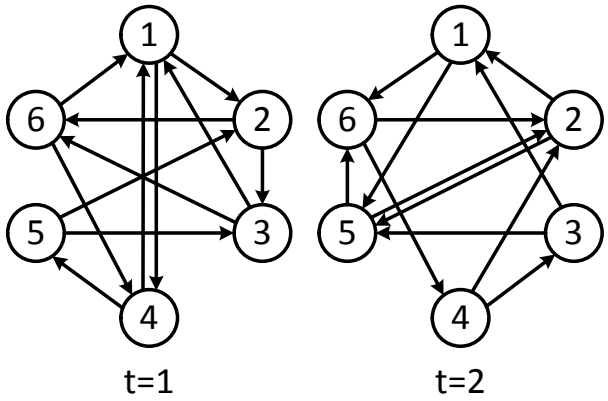

Figure 3: Random multicast.

We design an efficient sampling-based algorithm called *joint matrix column subset selection* (JMCSS) to perform SVS. Given the P matrices $X^{(1)}, \cdots, X^{(P)}$ where $X^{(p)}$ stores the p-th SV of all SVGs, JMCSS selects a subset of non-redundant column vectors from each matrix to approximate the entire matrix. The selection of columns in different matrices are tied together, i.e., if the i-th column is selected in one matrix, for all other matrices their i-th column must be selected as well to atomically form an SVG. Let $I = \{i_1, \cdots, i_C\}$ index the selected SVGs and $S_I^{(p)}$ be a matrix whose columns are from $X^{(p)}$ and indexed by I. The goal is to find out the optimal selection I such that the following approximation error is minimized: $\sum_{p=1}^{P} \|X^{(p)} - S_I^{(p)} (S_I^{(p)})^{\dagger} X^{(p)}\|_2$, where $(S_I^{(p)})^{\dagger}$ is the pseudo-inverse of $S_I^{(p)}$.

Finding the exact solution of this problem is NP-hard. To address this issue, we develop a sampling-based method (Algorithm 2), which is an adaptation of the iterative norm sampling algorithm [16]. Let $S^{(p)}$ be a dynamically growing matrix that stores the column vectors to be selected from $X^{(p)}$ and $S_t^{(p)}$ denote the state of $S^{(p)}$ at iteration t. Accordingly, $X^{(p)}$ is dynamically shrinking and its state is denoted by $X_t^{(p)}$. At the t-th iteration, an index i_t is sampled and the i_t-th column vectors are taken out from $\{X^{(p)}\}_{p=1}^{P}$ and added to $\{S^{(p)}\}_{p=1}^{P}$. i_t is sampled in the following way. First, we compute the squared L2 norm of each column vector in $\{X_{t-1}^{(p)}\}_{p=1}^{P}$. Then sample i_t ($1 \le i_t \le B + 1 - t$) with probability proportional to $\prod_{p=1}^{P} \|x_{i_t}^{(p)}\|_2^2$, where $x_{i_t}^{(p)}$ denotes the i_t-th column vector in $X_{t-1}^{(p)}$. The selected column vector is removed from $X_{t-1}^{(p)}$ and added to $S_{t-1}^{(p)}$. Then a back projection is utilized to transform $X_t^{(p)}$: $X_t^{(p)} \leftarrow X_t^{(p)} - S_t^{(p)} (S_t^{(p)})^{\dagger} X_t^{(p)}$. After C iterations, we obtain the selected SVs contained in $\{S^{(p)}\}_{p=1}^{P}$ and pack them into SVGs, which are subsequently sent to other machines.

Under JMCSS, the aggregated update generated from the C SVGs is close to that computed from the entire batch. Hence SVS does not compromise parameter-synchronization quality.

3.2.2 Random Multicast. While the P2P transfer of SVs greatly reduces the size of each message (from a matrix to a few vectors), its limitation is SVGs need to be sent from each machine to every other machine, which renders the number of messages per clock to be quadratic in the number of machines P. To address this issue, several works [26, 38] use multicast instead of broadcast: each message is only sent to a subset of rather than all machines; the correctness of execution can still be guaranteed, thanks to ML-programs' tolerance to errors [22]. In these works, the selection of neighboring machines is fixed and remains unchanged throughout. In contrast, Orpheus adopts a *random multicast* scheme: in each clock, each machine *randomly* selects $Q (Q < P - 1)$ machines to send SVs to, as shown in Figure 3.

Unlike a deterministic multicast topology [26, 38] where each machine communicates with a fixed set of machines throughout the application run, random multicast provides several benefits. First, dynamically changing the topology in each clock gives every two machines a chance to communicate directly, which facilitates more symmetric (hence faster) synchronization. Second, random multicast is more robust to network connection failures since the failure of a network connection between two machines will not affect their communication with another one. Third, random multicast makes resource elasticity simpler to implement: adding and removing machines require minimal coordination with existing ones, unlike a deterministic topology which must be modified every time a worker joins or leaves.

3.3 Fault Tolerance

In this section, further exploring the SV update property, we propose to represent the parameter matrix using SVs. Based on such a representation, a light-weight fault tolerance approach is developed.

3.3.1 SV-Based Representation of Parameters. We first show the parameter matrix can be represented as a set of SVs. At clock T, the parameter state W_T is mathematically equal to $W_0 + \sum_{t=1}^{T} \triangle W_t$ where $\triangle W_t$ is the update matrix computed at clock t and W_0 is the initialization of the parameters. As noted earlier, $\triangle W_t$ can be computed from a SVG G_t: $\triangle W_t = h(G_t)$, using a transformation h. Following the standard practice of initializing ML models using randomization, we randomly generate a SVG G_0, then let $W_0 = h(G_0)$. To this end, the parameter state can be represented as $W_T = \sum_{t=0}^{T} h(G_t)$, using a set of SVGs.

3.3.2 Incremental SV Checkpoint. Based on the SV-representation (SVR) of parameters and inspired by the asynchronous and incremental checkpointing methods [2, 31], Orpheus provides an *incremental SV checkpoint* (ISVC) mechanism for fault tolerance and recovery: each machine continuously saves the new SVGs computed in each clock to stable storage and restores the parameters from the saved SVGs when machine failure happens. Unlike existing systems [1, 41] which checkpoint large matrices, saving small vectors consume much less disk bandwidth. To reduce the frequency of disk write, the SVGs generated after each clock are not immediately written onto the disk, but staged in the host memory. When a large batch of SVGs are accumulated, Orpheus writes them together.

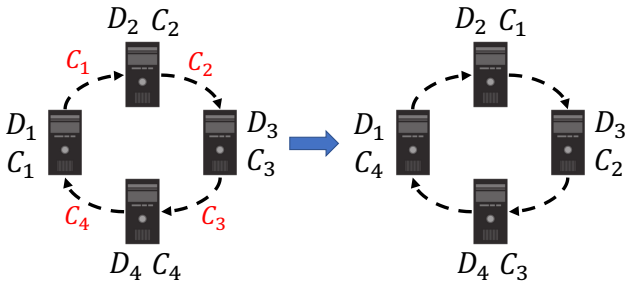

Figure 4: Parameter-replicas Rotation

ISVC does not require the application program to halt while checkpointing the SVs. The IO thread reads the SVs and the computing thread writes the parameter matrix (in the second phase of two-phase hybrid). There is no read/write conflict. In contrast, in matrix-based checkpointing, the IO thread reads the parameter matrix, which requires the computation thread to halt to ensure consistency, incurring waste of compute cycles.

ISVC is able to rollback the parameters to the state at any clock. To obtain the state at clock T, Orpheus collects the SVGs computed up to T and transforms them into a parameter matrix. This granularity is much more fine-grained than checkpointing parameter matrices. Since saving large-sized matrices to disk is time-consuming, the system can only afford to perform a checkpoint periodically and the parameter states between two checkpoints are lost. The `restore(T)` API is used for recovery where T is a user-specified clock which the parameters are to be rollbacked to. The default T is the latest clock.

3.4 Consistency

In this section, we propose two protocols to improve the consistency among parameter replicas.

3.4.1 Periodic Centralized Synchronization. In centralized architectures such as parameter servers, the shared state of parameters is maintained on the servers and local parameter caches on workers are continuously synchronized with the shared state. In this way, different parameter caches are highly consistent. In Orpheus, to avoid transmitting parameter matrices, no shared state is maintained and parameter replicas are synchronized in a decentralized manner by exchanging updates among each pair of workers. Without the orchestration of a centralized server, the workers operate autonomously and their parameter replicas have a higher risk of getting out of synchronization. To alleviate this risk, we adopt a *periodic centralized synchronization* strategy: every R iterations, the parameter replicas are averaged and reset to the average. A *centralized coordinator* sets a global barrier every R iterations. When all workers reach this barrier, the coordinator calls the `AllReduce(average)` API to average the parameter replicas and set each replica to the average. After that, workers perform decentralized synchronization until next barrier. Centralized synchronization effectively removes the parameter replicas' discrepancy accumulated during decentralized execution and it will not incur substantial communication cost since it is invoked periodically.

3.4.2 Parameter-Replicas Rotation. Orpheus adopts data parallelism, where each worker has access to one shard of the data. Since computation is usually much faster than communication, the updates computed locally are much more frequent than those received remotely. This would render imbalanced updating of parameters: a parameter replica is more frequently updated based on the local data residing in the same machine than data shards on other machines. This is another cause of out-of-synchronization. To address this issue, Orpheus performs *parameter-replica rotation*, which enables each parameter replica can explore all data shards on different machines. Logically, the machines are connected via a ring network. Parameter replicas rotate along the ring periodically (every S iterations) while each data shard sits still on the same machine during the entire execution. Figure 4 illustrates the process. Four machines are connected in a ring, each holding one data shard D. After every S iterations, the parameter replicas (denoted by C) perform a rotation clockwise: C1 moves from machine 1 to 2; C2 moves from machine 2 to 3; etc. In the next round, C1 would be moved from machine 2 to 3 and so on. We choose to rotate the parameters rather than data since the size of parameters is much smaller than data. A centralized coordinator sets a barrier every S iterations. When all workers reach the barrier, it invokes the `Rotate` API which triggers the rotation of parameter replicas.

4 IMPLEMENTATION

Orpheus is a decentralized system, where workers are symmetric, running the same software stack, which is conceptually divided into three layers (Figure 5): (1) *ML application layer* including ML programs implemented on top of Orpheus, such as multiclass logistic regression, topic models, deep learning models, etc.; (2) *Service layer* for automatic identification, SV selection, fault tolerance, etc.; (3) *P2P communication layer* for sufficient vector transfer and random multicast.

The major modules in Orpheus include: (1) an *interpreter* that automatically identifies the symbolic expressions of SVs and parameter updates; (2) a *SV generator* that selects training examples from local data shard and computes a SVG for each example using the symbolic expressions of SVs; (3) a *SV selector* that chooses a small subset of most representative SVs out of those computed by the generator; (4) a *communication manager* that transfers the SVs chosen by the selector using broadcast or random multicast and receives remote SVs; (5) an *update generator* which computes update matrices from locally-generated and remotely-received SVs and updates the parameter matrix; (6) a *central coordinator* for periodic centralized synchronization and parameter-replicas rotation.

The Orpheus programming interface exposes a rich set of *operators*, such as matrix multiplication, vector addition, and softmax, through which users write their ML programs. Each operator has a CPU implementation and a GPU implementation built upon highly optimized libraries such as Eigen[2], cuBLAS[3] and cuDNN[4]. In GPU implementation, Orpheus performs *kernel fusion* which combines a sequence of kernels into a single one, to reduce the number of kernel lunches that bear large overhead. Orpheus generates a dependency graph of operators by parsing users' program and traverses the graph to fuse consecutive operators into one CUDA kernel.

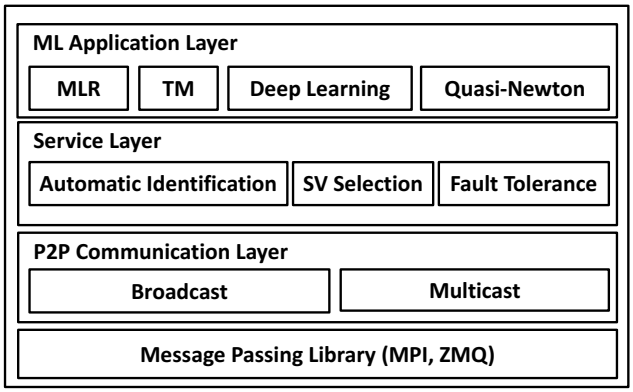

Figure 5: Software Stack

Orpheus is elastic to resource adjustment. Adding new machines and preempting existing machines do not interrupt the current execution. To add a new machine, the central coordinator executes the following steps: (1) launching the Orpheus engine and application program on the new machine; (2) averaging the parameter replicas of existing machines and placing the averaged parameters on the new machine; (3) taking a chunk of training data from each existing machine and assigning it to the new machine; (4) adding the new machine into the P2P network. When an existing machine is preempted, it is taken off from the P2P network and its data shard is re-distributed to other machines.

The loading of training data from CPU to GPU is overlapped with the SV generator via a *data queue*. The next batches of training examples are prefetched into the queue while the generator is processing the current one. In certain applications, each training example is associated with a *data-dependent variable* (DDV). For instance, in topic model, each document has a topic proportion vector. The states of DDVs need to be maintained throughout execution. Training examples and their DDVs are stored in consecutive host/device memory for locality and are prefetched together. At the end of a clock, GPU buffer storing examples is immediately ready for overwriting. The DDVs are swapped from GPU memory to host memory, which is pipelined using a DDV queue.

4.1 Hardware/Software-Aware SV Transfer

For efficient SV transfer, Orpheus provides a library of broadcast/multicast protocols designed for different hardware and software configurations, including (1) whether the communication is CPU-to-CPU or GPU-to-GPU; (2) whether InfiniBand[5] is available; (3) whether the consistency model is BSP or SSP.

To begin with, we introduce the protocols for broadcast, which serve as the building blocks of multicast.

- **CPU-to-CPU, BSP** In this case, we use the MPI_Allgather[6] routine to perform *all-to-all* broadcast. In each clock, it gathers the

[2] http://eigen.tuxfamily.org/index.php?title=Main_Page
[3] https://developer.nvidia.com/cublas
[4] https://developer.nvidia.com/cudnn
[5] http://www.infinibandta.com
[6] https://www.mpich.org/

SVs computed by each machine and distributes them to all machines. MPI_Allgather is a blocking operation which is in accordance with the BSP model.

- **CPU-to-CPU, SSP** Under SSP, each machine is allowed to have a different pace to compute and broadcast SVs. To enable this, the all-to-all broadcast is decomposed into multiple *one-to-all* broadcast. Each machine separately invokes the MPI_Bcast routine to broadcast its messages to others.
- **CPU-to-CPU, BSP, InfiniBand** An all-gather operation is executed by leveraging the Remote Direct Memory Access (RDMA) feature [34] provided by InfiniBand, which supports zero-copy networking by enabling the network adapter to transfer data directly to or from application memory, without going through operating system. The recursive doubling (RD) [19] algorithm is used to implement all-gather, where pairs of processes exchange their SVs via point-to-point communication. In each clock, the SVs collected during all previous clocks are included in the exchange. RDMA is used for the point-to-point transfer during the execution of RD.
- **CPU-to-CPU, SSP, InfiniBand** Each machine performs one-to-all broadcast separately, using the hardware supported broadcast (HSB) in InfiniBand. HSB is topology-aware: packets are duplicated by the switches only when necessary; therefore network traffic is reduced by eliminating the cases that multiple identical packets travel through the same physical link. The limitation of HSB is that messages can be dropped or arrive out of order, which degrades the correctness of ML execution. To retain reliability and in-order delivery, on top of HSB another layer of network protocol [28] is added, where (1) receivers send ACKs back to the root machine to confirm message delivery; (2) a message is retransmitted using point-to-point reliable communication if no ACK is received before timeout; (3) receivers use a continuous clock counter to detect out-of-order messages and put them in order.
- **GPU-to-GPU** To reduce the latency of inter-machine SV transfer between two GPUs, we use the GPUDirect RDMA[7] provided by CUDA, which allows network adapters to directly read from or write to GPU device memory, without staging through host memory. Between two network adaptors, the SVs are communicated using the methods listed above.

Similar to broadcast, several multicast protocols tailored to different hardware/software configurations are provided.

- **CPU-to-CPU** We use MPI group communication primitives. In each clock, `MPI_Comm_Split` is invoked to split the communicator `MPI_COMM_WORLD` into a target group (containing the selected machines) and a non-target group. Then the message is broadcast to the target group.
- **CPU-to-CPU, InfiniBand** Similar to broadcast, the efficient but unreliable multicast supported by InfiniBand at hardware level and a reliable point-to-point network protocol are used together. InfiniBand combines the selected machines into a single *multicast address* and sends the message to it. Point-to-point retransmission is issued if no ACK is received before timeout.
- **GPU-to-GPU** GPUDirect RDMA is used to copy buffers from GPU memory to network adaptor. Then the communication between network adaptors is handled using the two methods described above.

[7] http://docs.nvidia.com/cuda/gpudirect-rdma/#axzz4eWBuf9Rj

5 EVALUATION

We evaluate Orpheus on three ML applications: multiclass logistic regression (MLR), topic model (TM), long short-term memory network (LSTM)[8].

5.1 Experimental Setup

Cluster Setup. We used two clusters: (1) a CPU cluster having 34 machines each with 64 cores and 128 GB memory, connected by FDR10 Infiniband; (2) a GPU cluster having 40 machines each with one TitanX GPU, 16 CPU cores and 64GB memory, connected by 40Gbps Ethernet. We trained MLR and TM on the CPU cluster and LSTM on GPU cluster. Unless otherwise noted, the experiments were performed using all machines in each cluster.

Baseline Systems. We compared with (1) Spark-MLR from Spark MLlib-2.0.0 [41], which is based on an L-BFGS algorithm[9]; (2) TensorFlow-MLR and TensorFlow-LSTM: TensorFlow-1.0 implementation[10] of MLR and TensorFlow-1.7 implementation of LSTM, which are partially based on PS; (4) MXNet-MLR and MXNet-LSTM: MXNet-0.7 [11] implementation[11] of MLR and MXNet-1.1 implementation of LSTM, using PS-based data parallelism; (4) Bosen-MLR and Bosen-TM: MLR and TM implemented using a recent PS framework: Bosen [37], with additional system features (e.g., parameter-update filtering) borrowed from another PS [27]; (5) SVB-MLR and SVB-TM which are implemented on top of the sufficient vector broadcasting (SVB) [38] framework which performs P2P transfer of SVs to reduce communication cost. SVB does not support GPU programs, hence the LSTM model (which relies on GPU computation) is not compared[12]. (6) Gopal-MLR [18]: a model-parallel MLR based on Hadoop; (7) FlexiFaCT-TM [25]: a Hadoop implementation of TM. Note that Spark and Tensorflow implement a probabilistic topic model – latent Dirichlet allocation [9], which is different from the non-probabilistic TM in Orpheus. The configurations of baseline systems are tuned to achieve the best performance of each system. In LSTM experiments, all systems use cuDNN version 6.1, so they have access to the same optimized GPU kernels. In the baseline systems, matrix sparsity is leveraged to reduce costs whenever applicable. For instance, in communication, only nonzero entries are transmitted.

Datasets. Three datasets were used in the experiments: Wikipedia [33], PubMed[13] and 1B-Word [10]. The Wikipedia dataset contains ~2.4 million documents from 325K classes. Documents are represented with 20K-dimensional bag-of-words vectors. The PubMed dataset

[8] LSTM is a special type of recurrent neural network.
[9] Spark-MLR has a SGD-based implementation, which converges slower than L-BFGS.
[10] The cluster used for the MLR experiments was decommissioned in March 2017. On the MLR experiments, we were not able to compare with TensorFlow-1.7 and MXNet-1.1 which were released after March 2017.
[11] See footnote 10.
[12] The Poseidon framework [42] supports SV transfer and GPU programming, however it does not support the LSTM model, which is hence not compared.
[13] https://catalog.data.gov/dataset/pubmed

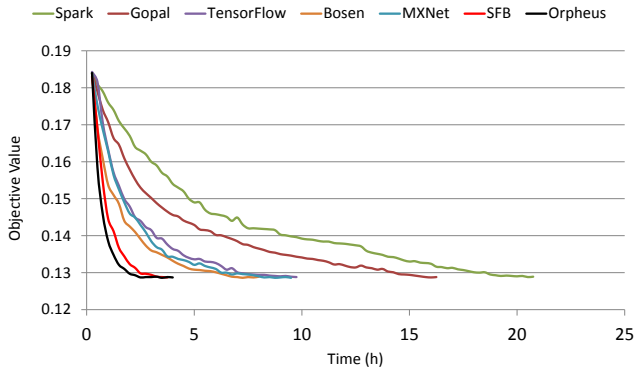

Figure 6: Convergence curves in MLR experiments. The algorithms terminate when the objectives reach 0.128 (consecutive change is less than 1e-3).

MLR			
# CPU Machines	12	34	Speedup
Spark	37.3	19.6	1.9
Gopal	31.7	15.1	2.1
TensorFlow-1.0	16.9	8.9	1.9
Bosen	15.7	7.1	2.2
MXNet-0.7	12.8	8.4	1.5
SVB	5.1	2.7	1.9
Orpheus	**4.4**	**1.9**	**2.3**
TM			
# CPU Machines	12	34	Speedup
FlexiFaCT	61.1	33.9	1.8
Bosen	49.4	23.5	2.1
SVB	20.1	9.7	2.1
Orpheus	**13.2**	**5.4**	**2.4**
LSTM			
# GPU Machines	12	40	Speedup
TensorFlow-1.7	14.2	5.9	2.4
MXNet-1.1	12.5	4.6	2.7
Orpheus	**11.9**	**4.1**	**2.9**

Table 2: The second and third columns show the convergence time (hours) of each system, under a different number of machines. The third column shows the speedup of each system when the number of machines is increased from 12 to 34 (in MLR and TM), or from 12 to 40 (in LSTM).

contains 8.2M documents and ~0.74B words. The vocabulary size is 141K. The 1B-Word dataset contains ~0.8B words with a vocabulary size of ~0.8M. The MLR, TM and LSTM experiments were conducted on the Wikipedia, PubMed and 1B-Word dataset respectively.

ML and System Hyper-parameter Setup. The topic number in TM and the state-vector dimension in LSTM was set to 50K and 40K respectively. As a result, the parameter matrix in MLR, TM and LSTM has a size of 325K×20K, 50K×141K, 40K×40K, containing ~6.5B, ~7.1B, ~1.6B entries respectively. The parameters in all applications were trained using the SGD algorithm with a mini-batch size of 100. In SV selection, the number of selected SVs was set to 25. In random multicast, the number of destinations each machine sends a message to was set to 4. The consistency model was set to SSP with a staleness value of 4.

5.2 Overall Results

Comparison with Other Systems. We first compare the convergence speed of these systems. In this comparison, no system uses fault tolerance. By *convergence*, it means the loss function value (e.g., cross-entropy loss in MLR, negative log-likelihood in TM and LSTM) on the training set levels off. A better system takes less time to converge. In each of our experiments, different systems converged to the same loss value. Figure 6 shows the convergence curves for MLR (34 machines). The second and third columns of Table 2 shows the convergence time of each system, under a different number of machines. On all three models, Orpheus converges faster than the baseline systems.

We first compare Orpheus with parameter server (PS) based systems including Bosen [37], TensorFlow [1], MXNet [11] and with a peer-to-peer architecture SVB [38] (which has leveraged SVs for efficient communication). On MLR, with 34 CPU machines, the speedup of Orpheus over Bosen, TensorFlow-1.0, MXNet-0.7 and SVB is 3.7x, 4.7x, 4.4x and 1.4x respectively. On TM, with 34 CPU machines, Orpheus is 4.4x and 1.8x faster than Bosen and SVB respectively. On LSTM, with 40 GPU machines, Orpheus is 1.4x faster than TensorFlow-1.7 and 1.1x faster than MXNet-1.1. As we will further show later, these speedups are achieved mainly because Orpheus is more efficient in communication. Compared with SVB, Orpheus reduces the number of sent SVs using SV selection and provides a random multicast scheme which works better than the deterministic multicast scheme utilized in SVB. Similar to SVB, Orpheus transmits small-sized vectors instead of large-sized matrices, which greatly reduces network traffic, compared with PS-based systems.

Next, we compare Orpheus with the rest of baseline systems. Using 34 CPU machines, Orpheus is 10.3x and 7.9x faster than Spark and Gopal on MLR, and 6.3x faster than FlexiFaCT on TM. Gopal outperforms Spark possibly because it uses a better distributed-algorithm which is based on model-parallelism. However, it is slower than PS systems due to the disk IO overhead of Hadoop. So is FlexiFaCT, which is a Hadoop-based system. Spark is at least two times slower than PS systems (implemented with C++) due to the overhead incurred by Resilient Distributed Datasets and Java Virtual Machine.

Scalability. We evaluated how Orpheus scales up as the number of machines increases. The results on MLR and LSTM are shown in Figure 7. With 34 CPU machines, Orpheus achieved 27.3x speedup on MLR. With 40 GPU machines, a 32.2x speedup is achieved on LSTM. The scalability of Orpheus is better than baseline systems, as shown in the fourth column of Table 2, where we measured the speedups for MLR and TM when the number of CPU/GPU machines increases from 12 to 34 and for LSTM when the number of GPU machines increases from 12 to 40. For example, in LSTM

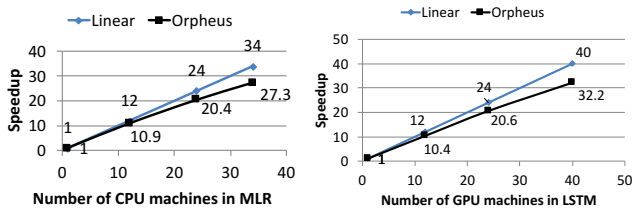

Figure 7: Scalability with more machines: (left) Orpheus-MLR; (middle) Orpheus-LSTM.

experiments with 40 GPU machines, Orpheus achieves a speedup of 2.9 compared with using 12 machines.

5.3 Evaluation of Individual Components

In this section, we evaluate the impact of each individual component. We compare the following systems: (1) Matrix+PS: synchronizing parameter copies by transmitting full matrices using a parameter server (PS) architecture; (2) SVB: SV broadcasting (SVB); (3) SVB+SVS: adding SV selection (SVS) to SVB; (4) SVB+SVS+RM: adding random multicast (RM); (5) SVB+SVS+RM+PCS: adding periodic centralized synchronization (PCS); (6) SVB+SVS+RM+PRR: adding parameter-replicas rotation (PRR). In these systems, no checkpointing is used. The number of machines in MLR, TM, and LSTM is set to 34, 34 and 40 respectively. Table 3 shows the convergence time of these systems, where we make the following observations. First, SVB is much more efficient than Matrix+PS. SVB is 2.6x, 2.4x and 2x faster than Matrix+PS on MLR, TM and LSTM respectively. The reason is: SVB transmits small-sized vectors while Matrix+PS transmits large matrices; the communication cost of SVB is much smaller. Second, adding SVS to SVB further reduces the runtime – by 15%, 19%, and 20% – on MLR, TM and LSTM respectively. SVS selects a subset of SVGs for communication, which reduces network traffic. Third, incorporating RM reduces the runtime of SVB+SVS by 9%, 16%, and 20% on the three applications. Under RM, in each clock, each machine selects a subset of machines to send SVs to, which reduces the number of network messages. Fourth, adding PCS further speeds up convergence. Via PCS, the incoherence among different parameter copies is alleviated, which reduces noise and improves convergence quality. Fifth, comparing the last two rows of this table, we confirm the effectiveness of PRR in reducing convergence time. Under PRR, each parameter replica has the chance to explore all data shards on different machines, which facilitates symmetric update of parameters.

Figure 8 shows the breakdown of network waiting time and computation time for four configurations. Compared with Matrix+PS, SVB greatly reduces network waiting time by avoiding transmitting matrices. It slightly increases the computation time since the same SVG needs to be converted into an update matrix at each worker. Adding SVS further decreases network time since it reduces the number of transmitted SVs. SVS causes the increase of computation time because of the overhead of executing the JMCSS algorithm (Algorithm 2). The network time is further reduced by using RM, which decreases the number of network messages. RM has little impact on the computation time.

	MLR	TM	LSTM
Matrix+PS	7.1	23.5	16.8
SVB	2.7	9.7	8.6
SVB+SVS	2.3	7.9	6.9
SVB+SVS+RM	2.1	6.6	5.5
SVB+SVS+RM+PCS	2.0	5.9	4.9
SVB+SVS+RM+PCS+PRR	1.9	5.4	4.1

Table 3: Convergence time (hours) of different system configurations

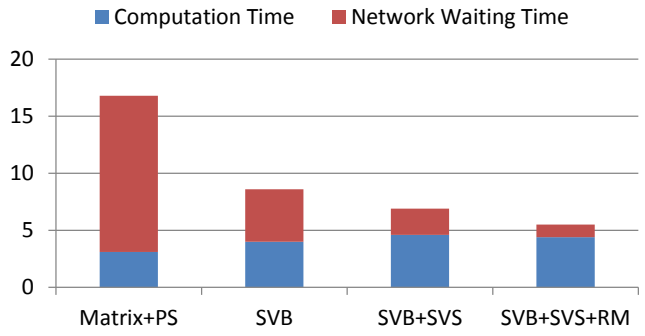

Figure 8: Breakdown of network waiting time (hours) and computation time (hours).

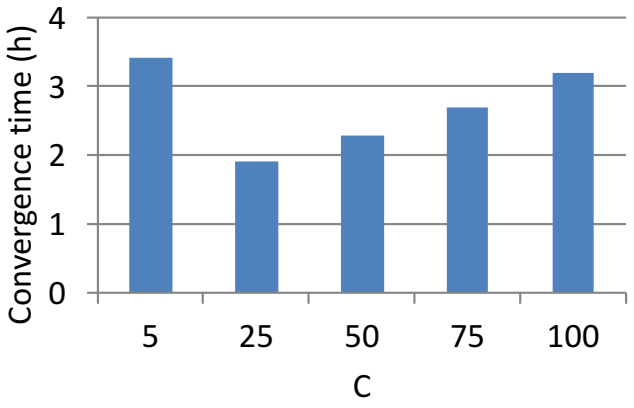

Figure 9: How the system parameter C in SV selection affects the running time of Orpheus for MLR.

In the sequel, we present more detailed evaluations of several key components.

SV Selection. Figure 9 shows the convergence time for MLR under varying C – the number of selected SVs. Compared with using the entire batch ($C = 100$), selecting a subset (e.g., $C = 25$) of SVs to communicate significantly speeds up convergence, thanks to the reduced network traffic. On the other hand, C cannot be too small, which otherwise incurs large approximation errors in parameter updates. For example, the convergence time under $C = 5$ is worse than that under $C = 100$.

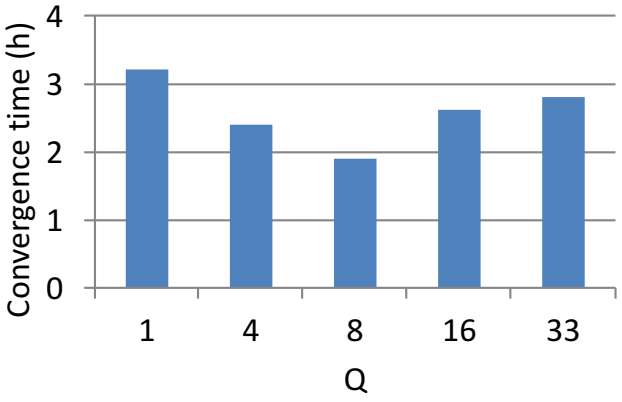

Figure 10: How the system parameter Q in random multicast affects the running time of Orpheus for MLR.

Random Multicast (RM). Figure 10 shows the convergence time of MLR under varying Q – the number of destinations each machine sends messages to. As can be seen, communicating with all machines (i.e., $Q = 33$) incurs much more running time than using random multicast ($Q = 4$), which demonstrates the effectiveness of RM in improving communication efficiency. The efficiency results from RM's ability to reduce the number of network messages. However, Q cannot be too small. Otherwise, the running time increases (e.g., when $Q = 1$), due to the severe synchronization delays.

We compared RM with two multicast schemes: (1) deterministic multicast (DM) [26, 38] where each sending machine sends messages to a fixed set of 4 receiving machines; the set of 4 receiving machines is different for each sending machine and is chosen to balance network load across the cluster and prevent network hot spots; (2) round-robin multicast (RRM) [26, 36] where every two workers communicate with each other periodically according to a deterministic circular order. Figure 11 shows the convergence time of MLR and LSTM under DM, RRM and RM. In both applications, RM takes less time to converge. Compared with DM, RM uses a randomly changing multicast topology to enable each pair of machines to have some chance to communicate directly, thus facilitating more symmetric (hence faster) synchronization of parameter replicas. Compared with RRM, the randomness of RM facilitates faster "mixing" [20] of parameter copies and hence speeds up convergence.

To examine the robustness of RM against network connection failures, we simulated the effect that in each clock the connections between 10% of machine pairs are "broken" randomly. Figure 12 shows the relative increase of convergence time when failure happens. As can be seen, the relative increase under RM is much smaller than that under DM and RRM, confirming that RM is more robust, due to its random nature.

Fault Tolerance and Recovery. We compare the following configurations: (1) no checkpoint; (2) matrix-based checkpoint: every 100[14] clocks, Orpheus saves the parameter matrix onto the disk; when saving matrices, the computation halts to ensure consistency; (3) SV-based checkpoint; (4) matrix-based recovery: we simulated

[14]Choosing a number smaller than 100 would entail more disk-IO waiting.

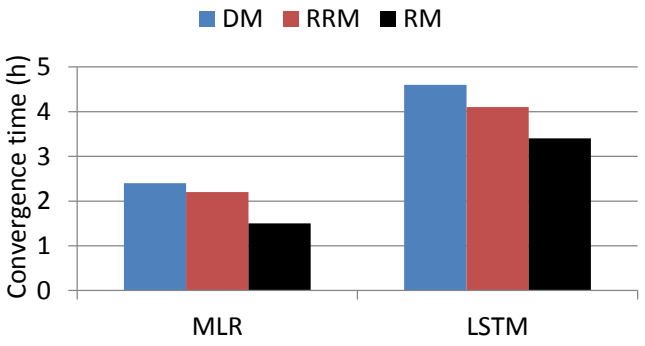

Figure 11: Convergence time in Orpheus under deterministic, round-robin and random broadcast for MLR and LSTM.

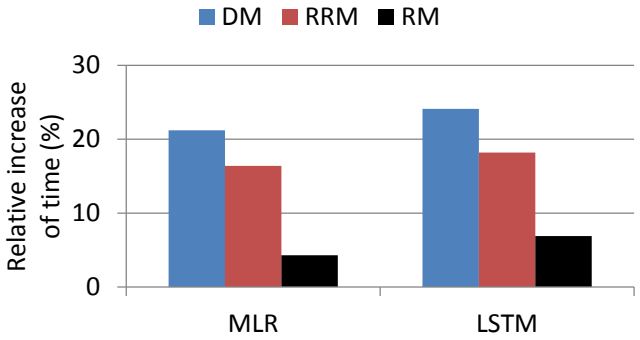

Figure 12: Relative increase of convergence time when network connection fails.

the effect that in each clock, each machine fails with a probability of 0.01; when failure happens, a recovery is performed based on the checkpointed matrices; (5) SV-based recovery: machine failure is simulated in the same way as (4) and recovery is based on the saved SVs. For (2) and (3), no machine failure happens.

Table 4 shows the convergence time of MLR (34 machines) and LSTM (40 machines) under different configurations. From this table, we observe the following. First, compared with no-checkpoint, matrix-based checkpoint method substantially increases the convergence time while our SV-based checkpointing incurs very little increase. The reasons are twofold: (1) saving vectors consumes much less disk bandwidth than saving matrices; (2) checkpointing SVs does not halt computation, wasting no compute cycles, which is not the case in checkpointing matrices. Second, in the case where machine failure happens, matrix-based recovery causes more slowing-down of convergence than SV-based recovery. This is because in matrix-based recovery, the parameters can only be rolled back to the state that is saved every 100 clocks. If the failure happens at clock 199, the parameters are rolled back to the state saved at clock 100. The computation from clock 101 to 198 are wasted. However, in SV-based recovery, the parameter state after every clock is preserved. It can always roll back to the latest parameter state (e.g., the state after clock 198) and no computation is wasted.

	MLR	LSTM
No checkpoint	1.9	4.1
Matrix-based checkpoint	2.4	5.2
SV-based checkpoint	1.9	4.2
Matrix-based recovery	2.9	6.7
SV-based recovery	2.3	4.8

Table 4: Convergence time (hours) under different configurations of fault tolerance and recovery.

6 RELATED WORKS

Many distributed ML systems have been built recently, including (1) dataflow systems such as Hadoop-based Mahout [15] and Spark-based MLlib [41]; (2) graph computation frameworks such as Pregel [29] and GraphLab [17]; (3) parameter server (PS) architectures such as DistBelief [15], Project Adam [12], ParameterServer [27], Bosen PS [37] and GeePS [13]; (4) hybrid systems such as TensorFlow [1] and MXNet [11]. These systems explore the tradeoffs among correctness of computation, ease of programmability and efficiency of execution. Though not designed specifically for ML models, Hadoop and Spark support the embarrassingly-parallel execution of ML training. They provide easy-to-use programming interface and can be applied to almost any ML application. However, the bulk synchronous parallel consistency imposed by these systems inhibits the training efficiency. Parameter server architectures are arguably the most widely used in distributed ML due to three merits: (1) they adopt asynchronous or bounded asynchronous parallelism that improves efficiency without sacrificing correctness; (2) they have wide applicability to nearly all ML models; (3) they provide unified programming interface that facilitates the agile development of ML programs. When used to train MPMs, these systems operate on large-sized matrices, which incur high overhead in all system aspects. It is worth noting that these systems are more general than Orpheus in the sense that they can support the training of more ML models. Orpheus is only applicable when the ML model is parameterized by a matrix and this matrix satisfies the sufficient vector property.

Peer-to-peer (P2P) architectures have been investigated in distributed ML [26, 36, 38, 42]. Li et al. [26] propose to synchronize parameter replicas by exchanging parameter updates in a P2P manner to simplify fault-tolerance. Watcharapichat et al. [36] design a decentralized architecture to exchange partial gradient of deep neural networks among machines, for the sake of saturating cluster resources. These two works transmit matrices in the network and do not leverage the SVs to reduce communication cost. Two recent works [38, 42] explore the idea of broadcasting SVs. They focus on reducing network traffic and do not leverage the SVs to improve efficiency in fault tolerance, nor support the automatic discovery of SVs. Bellet et al. [4] designed a decentralized solution where agents operate asynchronously and communicate over a network in a peer-to-peer fashion, to learn personalized models under strong privacy requirements. Vanhaesebrouck et al. [35] proposed two asynchronous gossip algorithms running in a fully decentralized manner to learn agents in a collaborative peer-to-peer network, where each agent learns a personalized model according to its own learning objective and improves upon its locally trained model by communicating with other agents that have similar objectives.

7 CONCLUSIONS AND DISCUSSIONS

We have described the Orpheus system, which thoroughly exploits system and algorithm co-design to support the efficient training of MPMs. Orpheus transforms matrix-based communication and fault tolerance into vector-level execution, which substantially reduces the costs from quadratic in matrix dimensions down to linear. Evaluation on three applications demonstrates the low communication and fault-tolerance costs, high execution speed and scalability of Orpheus, which greatly improves over well-established baseline systems.

Compared with existing systems, the applicability of Orpheus is narrower. It is mainly suitable for learning large-sized MPMs that have the SV update property while existing systems can be applied to a broader range of MPMs and vector-parameterized models. When the size of a parameter matrix is small, existing systems are better choices than Orpheus. Orpheus is based on a peer-to-peer architecture, where the number of network messages grows quadratically with the number of machines. This makes it a less-ideal system when tens of thousands of machines are used. Partial broadcasting approaches have been investigated in [26, 38] to reduce the number of network messages. In these approaches, each machine sends messages to a subset of rather than all machines. However, their effectiveness is yet to be verified in data-center-scale computing environments. Under circumstances where a very large mini-batch size is preferred, Orpheus may be less favorable. When the parameter matrix or update matrix is sparse, the advantage of Orpheus over PS systems diminishes since the latter can leverage the sparsity to reduce communication cost.

While system and algorithm co-design effectively boosts system efficiency, there is a caveat. The tight coupling between the system and algorithms may break abstractions and make the system less modular as a whole, thus harder to maintain in the long term.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers for providing insightful feedback that helps to improve this work a lot. The authors are also grateful for the valuable comments from Yi Zhou, Abhimanu Kumar, Hao Zhang, Shizhen Xu. The work is supported by National Science Foundation IIS1447676 and CCF1629559.

REFERENCES

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. TensorFlow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI). Savannah, Georgia, USA*.

[2] Saurabh Agarwal, Rahul Garg, Meeta S Gupta, and Jose E Moreira. 2004. Adaptive incremental checkpointing for massively parallel systems. In *Proceedings of the 18th annual international conference on Supercomputing*. ACM, 277–286.

[3] Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, James Bergstra, Ian Goodfellow, Arnaud Bergeron, Nicolas Bouchard, David Warde-Farley, and Yoshua Bengio. 2012. Theano: new features and speed improvements. *arXiv preprint arXiv:1211.5590* (2012).

[4] Aurélien Bellet, Rachid Guerraoui, Mahsa Taziki, and Marc Tommasi. 2017. Personalized and Private Peer-to-Peer Machine Learning. *arXiv preprint arXiv:1705.08435* (2017).

[5] Dimitri P Bertsekas. 1999. *Nonlinear programming*. Athena scientific Belmont.

[6] Kanishka Bhaduri, Ran Wolff, Chris Giannella, and Hillol Kargupta. 2008. Distributed decision-tree induction in peer-to-peer systems. *Statistical Analysis and Data Mining: The ASA Data Science Journal* (2008).

[7] Christian H Bischof, Paul D Hovland, and Boyana Norris. 2008. On the implementation of automatic differentiation tools. *Higher-Order and Symbolic Computation* 21, 3 (2008), 311–331.

[8] Christopher M Bishop et al. 2006. *Pattern recognition and machine learning*. springer New York.

[9] David M Blei, Andrew Y Ng, and Michael I Jordan. 2003. Latent dirichlet allocation. *Journal of machine Learning research* 3, Jan (2003), 993–1022.

[10] Ciprian Chelba, Tomas Mikolov, Mike Schuster, Qi Ge, Thorsten Brants, Phillipp Koehn, and Tony Robinson. 2013. One billion word benchmark for measuring progress in statistical language modeling. *arXiv preprint arXiv:1312.3005* (2013).

[11] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. 2015. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274* (2015).

[12] Trishul Chilimbi, Yutaka Suzue, Johnson Apacible, and Karthik Kalyanaraman. 2014. Project Adam: building an efficient and scalable deep learning training system. In *USENIX Symposium on Operating Systems Design and Implementation*.

[13] Henggang Cui, Hao Zhang, Gregory R Ganger, Phillip B Gibbons, and Eric P Xing. 2016. Geeps: Scalable deep learning on distributed gpus with a gpu-specialized parameter server. In *Proceedings of the Eleventh European Conference on Computer Systems*. ACM, 4.

[14] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Andrew Senior, Paul Tucker, Ke Yang, Quoc V Le, et al. 2012. Large scale distributed deep networks. In *NIPS*.

[15] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: simplified data processing on large clusters. *CACM* (2008).

[16] Amit Deshpande and Santosh Vempala. 2006. Adaptive sampling and fast low-rank matrix approximation. In *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques*. Springer, 292–303.

[17] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. Powergraph: Distributed graph-parallel computation on natural graphs. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*. 17–30.

[18] Siddharth Gopal and Yiming Yang. 2013. Distributed training of Large-scale Logistic models. In *ICML*.

[19] William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. 1996. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel computing* 22, 6 (1996), 789–828.

[20] Bernhard Haeupler. 2015. Simple, fast and deterministic gossip and rumor spreading. *Journal of the ACM (JACM)* 62, 6 (2015), 47.

[21] Geoffrey E Hinton and Ruslan R Salakhutdinov. 2006. Reducing the dimensionality of data with neural networks. *Science* 313, 5786 (2006), 504–507.

[22] Qirong Ho, James Cipar, Henggang Cui, Seunghak Lee, Jin Kyu Kim, Phillip B Gibbons, Garth A Gibson, Greg Ganger, and Eric Xing. 2013. More effective distributed ml via a stale synchronous parallel parameter server. In *NIPS*.

[23] Jin Kyu Kim, Qirong Ho, Seunghak Lee, Xun Zheng, Wei Dai, Garth A Gibson, and Eric P Xing. 2016. STRADS: a distributed framework for scheduled model parallel machine learning. In *Proceedings of the Eleventh European Conference on Computer Systems*. ACM, 5.

[24] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*. 1097–1105.

[25] Abhimanu Kumar, Alex Beutel, Qirong Ho, and Eric P Xing. 2014. Fugue: Slow-Worker-Agnostic Distributed Learning for Big Models on Big Data.. In *AISTATS*. 531–539.

[26] Hao Li, Asim Kadav, Erik Kruus, and Cristian Ungureanu. 2015. MALT: distributed data-parallelism for existing ML applications. In *Proceedings of the Tenth European Conference on Computer Systems*.

[27] Mu Li, David G Andersen, Jun Woo Park, Alexander J Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J Shekita, and Bor-Yiing Su. 2014. Scaling distributed machine learning with the parameter server. In *USENIX Symposium on Operating Systems Design and Implementation*.

[28] Jiuxing Liu, Amith R Mamidala, and Dhabaleswar K Panda. 2004. Fast and scalable MPI-level broadcast using InfiniBand's hardware multicast support. In *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*. IEEE, 10.

[29] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: a system for large-scale graph processing. In *ACM SIGMOD International Conference on Management of data*.

[30] Tomas Mikolov, Martin Karafiát, Lukas Burget, Jan Cernockỳ, and Sanjeev Khudanpur. 2010. Recurrent neural network based language model.. In *Interspeech*, Vol. 2. 3.

[31] Bogdan Nicolae and Franck Cappello. 2013. AI-Ckpt: leveraging memory access patterns for adaptive asynchronous incremental checkpointing. In *Proceedings of the 22nd international symposium on High-performance parallel and distributed computing*. ACM, 155–166.

[32] Bruno A Olshausen and David J Field. 1997. Sparse coding with an overcomplete basis set: A strategy employed by V1? *Vision research* (1997).

[33] Ioannis Partalas, Aris Kosmopoulos, Nicolas Baskiotis, Thierry Artieres, George Paliouras, Eric Gaussier, Ion Androutsopoulos, Massih-Reza Amini, and Patrick Galinari. 2015. LSHTC: A Benchmark for Large-Scale Text Classification. *arXiv:1503.08581 [cs.IR]* (2015).

[34] Sayantan Sur, Uday Kumar Reddy Bondhugula, Amith Mamidala, H-W Jin, and Dhabaleswar K Panda. 2005. High performance rdma based all-to-all broadcast for infiniband clusters. In *International Conference on High-Performance Computing*. Springer, 148–157.

[35] Paul Vanhaesebrouck, Aurélien Bellet, and Marc Tommasi. 2017. Decentralized collaborative learning of personalized models over networks. In *International Conference on Artificial Intelligence and Statistics (AISTATS)*.

[36] Pijika Watcharapichat, Victoria Lopez Morales, Raul Castro Fernandez, and Peter Pietzuch. 2016. Ako: Decentralised Deep Learning with Partial Gradient Exchange. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*. ACM, 84–97.

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

[38] Pengtao Xie, Jin Kyu Kim, Yi Zhou, Qirong Ho, Abhimanu Kumar, Yaoliang Yu, and Eric Xing. 2016. Distributed Machine Learning via Sufficient Factor Broadcasting. *Conference on Uncertainty in Artificial Intelligence* (2016).

[39] Eric P Xing, Michael I Jordan, Stuart Russell, and Andrew Y Ng. 2002. Distance metric learning with application to clustering with side-information. In *Conference on Neural Information Processing Systems*.

[40] Ming Yuan and Yi Lin. 2006. Model selection and estimation in regression with grouped variables. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)* (2006).

[41] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *USENIX Symposium on Networked Systems Design and Implementation*.

[42] Hao Zhang, Zeyu Zheng, Shizhen Xu, Wei Dai, Qirong Ho, Xiaodan Liang, Zhiting Hu, Jinliang Wei, Pengtao Xie, and Eric P Xing. 2017. Poseidon: An Efficient Communication Architecture for Distributed Deep Learning on GPU Clusters. *USENIX Annual Technical Conference* (2017).