2018 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Ubiquitous Computing & Communications, Big Data & Cloud Computing, Social Computing & Networking, Sustainable Computing & Communications

Cymbalo: An Efficient Graph Processing Framework for Machine Learning

Xinhui Tian*†, Biwei Xie*†, Jianfeng Zhan*†
*State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences
†University of Chinese Academy of Sciences, China
{tianxinhui, xiebiwei, zhanjianfeng}@ict.ac.cn

***Abstract*—Due to the growth of data scale, distributed machine learning has become more important than ever. Some recent work, like TuX^2, show promising prospect in dealing with distributed machine learning by leveraging the power of graph computation, but still leave some key problems unsolved.**

In this paper, we propose Cymbalo, a new distributed graph processing framework for large-scale machine learning algorithms. To satisfy the specific characteristics of machine learning, Cymbalo employs a heterogeneity-aware data model, a hybrid computing model and a vector-aware programming model, to ensure small memory footprint, good computation efficiency and expressiveness. The experiment results show that Cymbalo outperforms Spark by 2.4×-3.2×, and PowerGraph by up to 5.8×. Moreover, Cymbalo can also outperform Angel, a recent parameter server system, by 1.6×-2.1×.

***Index Terms*—Large-scale machine learning, Graph computation, Distributed processing, Spark.**

I. Introduction

Machine learning is an essential component for big data analytics and artificial intelligence, and is widely applied in many domains including web search, recommendation systems, natural language processing, and computational advertising. As the scale of machine learning problems grows explosively, it has become a norm to run these problems on a cluster of machines. However, this poses great challenges on the design of distributed machine learning system, which should be both efficient and effective.

Meanwhile, many machine learning problems can naturally be modeled as graphs and leverage existing graph computation efforts. In this paper, We name the design of graph models for machine learning as *ML-Graph*. For example, matrix factorization (MF) is an algorithm widely used in recommendation systems, which can be modeled as a user-item bipartite graph. Many researches [1]–[4] on highly scalable graph computation, which have been proved to be capable of efficiently processing more than 1 trillion edges for graph analytics workloads [5], can facilitate the development of new graph processing systems for machine learning problems. However, current graph processing systems mainly focus on graph analytics workloads, such as PageRank and SSSP, and only provide simplistic data model, computing model, and programming model. Without taking the key characteristics of machine learning problems into consider, these systems lack sufficient support and could not be applied to machine learning problems effectively.

There exists some efforts aiming at solving machine learning problems with graph engines, such as BiGraph [6] and TuX^2 [7], which show promising result against classical machine learning systems. However, current ML-Graph design are quite simple and do not consider a systematic design from all the three perspectives of data model, computation model and programming model. Especially that BiGraph fails to support the varied propagation behaviors of different machine learning algorithms, and the programming models of both BiGraph and TuX^2 are not accurate and straightforward enough for machine learning algorithms. We wonder that there should be a systematic design of ML-Graph, which will consider the data model, computing model, and programming model as a unified system, and fully accommodate the characteristics of machine learning.

In this paper, we present Cymbalo, a new graph processing framework designed for machine learning problems (in other words, a new ML-Graph solution). Cymbalo adopts a heterogeneity-aware data model, a hybrid computing model, and a vector-centric programming model to efficiently support machine learning. Experimental results show that Cymbalo is typically 2.4×-3.2× faster than Spark, and outperforms a parameter server system by 1.6×-2.1×. Moreover, Cymbalo, which is written in Scala, also outperforms PowerGraph, another well-known graph computation framework written in C++, by up to 5.8×, despite the inherent language inefficiency.

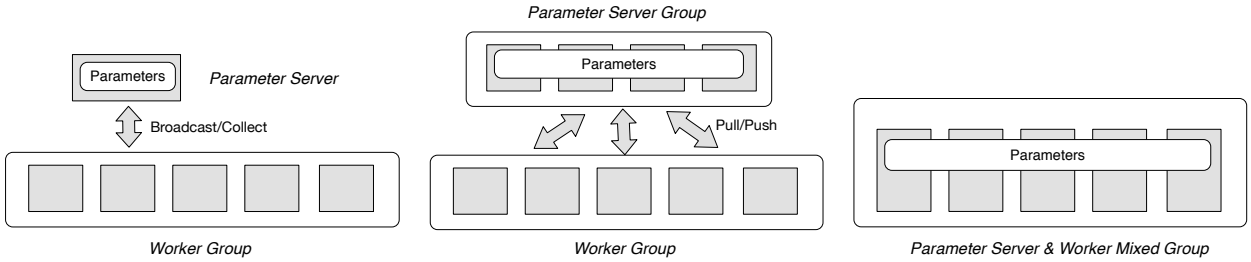
The main contributions of this paper are shown as below:

- We present Cymbalo, a new distributed graph processing framework for machine learning, which provides a heterogeneity-aware data model and a hybrid computing model to efficiently support machine learning problems.
- We design a vector-centric programming model, which is more accurate and straightforward for machine learning algorithms, than that of existing graph processing frameworks.
- We conduct comprehensive evaluation of Cymbalo on five popular graph datasets and compare it with six other well-known systems, including two machine learning systems, one parameter server system, and three graph systems.

The rest of the paper is organized as below. Section II gives the background of graph computation and machine learning. Section III and IV elaborates the motivation and describes the

This work is supported by the National Key Research and Development Plan of China (Grant No. 2016YFB1000600 and 2016YFB1000601).

978-1-7281-1141-4/18/$31.00 ©2018 IEEE
DOI 10.1109/BDCloud.2018.00090



(a) Iterative Batch Processing Framework (b) Parameter Server Framework (c) Graph-based Machine Learning Framework

Fig. 1. The comparison between different distributed machine learning frameworks. Comparing with other two frameworks, graph-based machine learning framework distributes parameters and training data together among the same cluster, which can achieve good computation locality and reduce the communication overhead.

design of Cymbalo. Section V shows the experimental results. Related work and conclusion are given in Section VI and VII.

II. Background

A. Graph Computation Abstraction

In recent years, many distributed graph processing systems have been proposed to solve large-scale graph processing problems. These systems often follow the *"think like a vertex"* paradigm, where a user-defined vertex program runs on the vertices of a graph $G = \{V, E\}$ in parallel. Each vertex $v \in V$ maintains an arbitrary data D_v, and communicates with its neighbors along edges. In each iteration, a vertex is activated by its neighbors or system to perform computation, and votes to halt after computation. A computation job terminates when all vertices vote to halt.

Before computation, the graph data need to be firstly partitioned among machines. Pregel [1] and other similar systems [4], [8] employ an edge-cut partitioning scheme, which evenly distributes vertices among machines. This scheme accumulates all the outgoing edges locally for each vertex to reduce the computation latency, but can not guarantee load balance for natural skewed graphs [9], where some vertices might have significantly much more neighbors than others. Systems such as PowerGraph [2] and GraphX [3] employ a vertex-cut partitioning scheme, which evenly distributes edges among machines instead of vertices. Comparing with edge-cut, vertex-cut distributes the neighboring edges of vertices with massive neighbors to guarantee load and communication balance [2] for skewed graphs.

There are also many optimized vertex-cut partitioning schemes [6], [10], [11], which considers differentiated partitioning strategies for different types of vertices. Among them, BiCut [6] is the one designed for bipartite graphs. Since many machine learning algorithms discussed in this paper can be cast as a bipartite graph, we design our partitioning scheme based on the idea of BiCut.

For graph computation, existing graph systems often provide a general but restricted computing model for diverse algorithms. For example, Pregel adopts a vertex-centric computing model, which only supports unidirectional communication from each vertex to its neighbors. PowerGraph and GraphX employ a multi-phase computing model, which decomposes the vertex program into different edge-parallel and vertex-parallel phases. Such model requires to perform massive synchronization operations even for simple algorithms such as PageRank and SSSP. These systems do not allow users to define or select computation strategies according to the specific behaviors of their applications, thus lack the flexibility to support diverse algorithms.

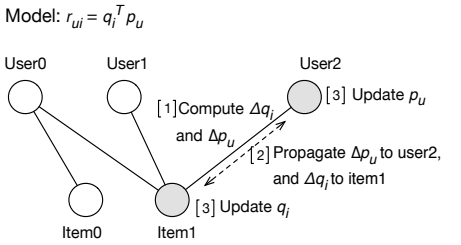
B. Distributed Frameworks For Machine Learning

There exists many other distributed machine learning frameworks, including iterative batch processing and parameter server frameworks. The main differences between them and graph-based frameworks are the strategies of model parallel. Iterative batch processing framework, such as Spark [12], does not consider model parallel. Its model parameters are stored in one single machine, broadcast to workers and collect updates from workers in each iteration. Parameter server frameworks, such as Petuum [13] and Angel [14], store training data in a group of machines, and store parameters in another group of machines distributedly. Machines for data training will pull required parameters from the parameter machines, and push updates back accordingly after computation. For graph-based frameworks, parameters are treated as different types of vertices, and distributed among the same machines which also store the training data. The comparison of different frameworks is illustrated in Fig. 1.

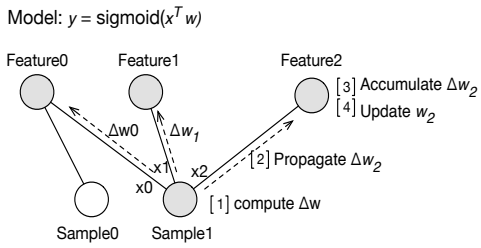
Meanwhile, graph-based system further considers balanced partitioning strategy for both training data and parameters, which can guarantee load and communication balance during the computation.

III. Motivation

Many machine learning algorithms can be modeled as graphs naturally and solved by an iterative learning algorithms. For example, matrix factorization (MF) used in recommendation systems can be modeled as a user-item bipartite graph, of which each vertex is attached with a feature vector. The topic-modeling algorithm LDA can be represented as a document-word bipartite graph. Algorithms such as Logistic Regression (LR) and Support Vector Machine (SVM) can also be cast



(a) Matrix Factorization



(b) Logistic Regression

Fig. 2. Different machine learning algorithms with varied computation behaviors. MF performs a bidirectional propagation on each edge, while LR propagates values from one vertex to its neighbors along one direction.

as a sample-feature graph model with the training sample data and features as the vertices. For ease of discussion, we name the graph systems designed for machine learning as ML-Graph. Compared to some regular graph algorithms, ML-Graph is much different on data structure and computational behaviors, since it aims at supporting both machine learning and graph algorithms, instead of just graph algorithms, like existing graph systems. The design of ML-Graphs should fully consider the inherent characteristics of machine learning algorithms, which are summarized as follows:

Heterogeneous Graph Structures: Graph analytics workloads such as PageRank often run on natural graphs and assume all the vertices to be homogeneous. ML-Graph, however, often consists of heterogeneous vertices, i.e., the user and item vertices in the user-item matrix of MF. The ML-Graph is thus separated into different vertex subsets. The heterogeneous properties of these graphs can be summarized into two aspects. First, the size of different subsets might be significantly skewed. For example, Wikipedia only has ten thousands of terms, but has more than four millions of articles [6]. Second, the data types of vertices from different subsets might not be the same. Taking LR as an example, feature vertices maintain the weights of features, while the sample vertices need to maintain a sparse feature vector and a label value. Therefore, an efficient graph system should consider the heterogeneity of ML-Graphs.

Varied Computation Behaviors: The graph computation in different machine learning algorithms might involve varied propagation and update operations, as illustrated in Fig. 2. For MF, each edge performs bidirectional propagation to send messages to its two neighboring vertices in each iteration. For LR, each feature vertex first propagates its value to all its related sample vertex. Each sample vertex then computes the gradients, and performs unidirectional propagation to send them to each feature. Moreover, there are also differentiated update behaviors between different types of vertices in one single algorithm, such as the sample and feature vertices in LR. Therefore, the computation model should be flexible to support various computation strategies.

Vector-centric Computation: Many machine learning algorithms can be transformed into vector operations, such as MF and LR algorithms illustrated in Fig. 2. The programming model should take this into consider and provide vector-centric interfaces to simplify the programming of machine learning algorithms. Existing graph processing systems, however, support either vertex-centric or edge-centric programming model, but not both. When implementing a machine learning algorithm, users have to transform vector-based computations into graph computations on vertices or edges, which would be too complicated and time-consuming. Even TuX^2 [7], a latest distributed graph system designed for machine learning algorithms, provides MEGA programming model with only edge-centric interfaces for data exchange operations and vertex-centric interfaces for update operations. In contrast, a vector-centric programming model can free developers from low-level implementation details, and accelerate specific operations by leveraging the power of high-performance linear algebra libraries, such as openBLAS [15].

Considering the obvious difference between machine learning and classical graph computing, it takes delicate design to leverage the power of graph computing on machine learning problems. A new graph framework dedicated to machine learning is motivated.

IV. CYMBALO

We present Cymbalo, a new distributed graph processing framework for machine learning algorithms, which can satisfy the three requirements referred in Section II. First, Cymbalo provides a heterogeneity-aware data model, which considers the various characteristics of ML-Graphs. This data model also considers a vector-aware graph layout to eliminate unnecessary memory footprint. Second, considering the diverse propagation behaviors of different machine learning algorithms, Cymbalo provides a hybrid computation model with differentiated computation strategies. Finally, Cymbalo provides a vector-centric programming model, which is more accurate and straightforward for machine learning algorithms.

A. Heterogeneity-aware Data Model

The data model in Cymbalo considers the heterogeneous characteristics of ML-Graphs. First, to efficiently handle vertex subsets with different sizes, we implement a graph partitioning scheme based on the idea of BiCut [6], which performs differentiated partitioning strategies for vertices of different subsets. Vertices of one subset are set as the favorite vertices. This subset is usually the one with larger number of vertices. Each favorite vertex is assigned to a partition with all its

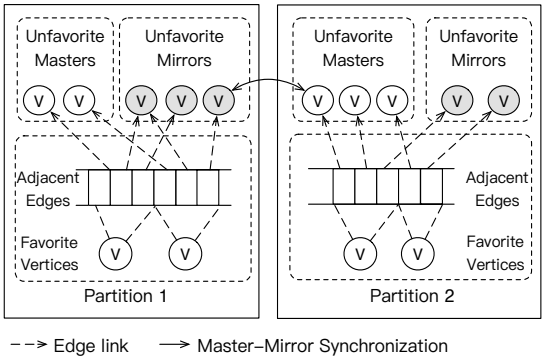


Fig. 3. The Heterogeneity-aware data layout of Cymbalo. For each favorite vertex, all the neigboring edges are assigned to the same partition of it, and organized in CSR format. For unfavorite vertices, mirrors are created and need to synchronize with their masters in each iteration.

neighboring edges. Since the adjacent list of each favorite vertex is not distributed among partitions, no remote replica needs to be created.

Vertices in the other subset are then set as unfavorite vertices, which are evenly distributed among partitions. For each unfavorite vertex, replicas are created in partitions having at least one neighboring edge of it. For each vertex, one of the replicas are selected as the master, while others are set as the mirrors.

This differentiated graph partitioning scheme can ensure no replicas are created for favorite vertices, thus to guarantee lower replication factor comparing with other vertex-cut partitioning schemes [6]. Meanwhile, unfavorite vertices are evenly distributed across partitions to ensure balanced communication.

After graph partitioning, Cymbalo builds a local graph structure for the subgraph structure of each partition. Considering the heterogeneity on vertex data type, the data of different types of vertices are stored separately in two arrays. Vertices and edges in each partition are then organized in a heterogeneity-aware graph layout. As discussed above, vertices in each partition can be separated into favorite and unfavorite vertices. Unfavorite vertices have mirrors on other partitions. In the graph layout of Cymbalo, vertices in each partition are sorted with the order of first favorite vertices, then unfavorite masters, and finally unfavorite mirrors. The local vertex ids are also assigned in this order. The data of edges and the ids of related neighboring unfavorite vertices are stored separately in two local arrays. Edges are indexed by favorite vertices, and organized in CSR format for fast data lookup. The data layout of Cymbalo is illustrated in Fig. 3

Furthermore, Cymbalo provides a new memory-efficient vertex layout for algorithms cast as a sample-feature graph, such as LR and SVM. For these algorithms, vertices of the favorite subset are transformed from sample data (including a label value and a sparse feature vector), while the unfavorite subset contains vertices transformed from features. Each edge between one sample vertex and one feature vertex indicates

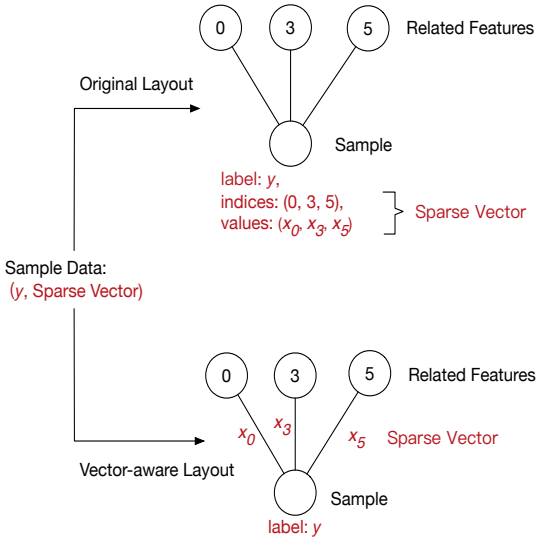


Fig. 4. Vector-aware vertex layout. The element values of the sparse feature vector in a sample vertex are stored in the data array of neighboring edges, while the element indices are stored in the local id array of feature neighbors.

that this sample vertex records a value for this feature. These relationships are also recorded in element indices of the sparse feature vector of each sample, which can result in data redundancy. To eliminate this redundant memory consumption, while retaining the efficiency of data model, we further provide a vector-aware vertex layout for such algorithms. For the sparse feature vector in a sample vertex, we store its element values in the data array of its neighboring edges and element indices in the id array of neighboring vertices. The vector-aware vertex layout is illustrated in Fig. 4. In Fig. 4, the given sample vertex maintains a sparse vector recording feature values for feature vertex 0, 3 and 5. The element values are stored as the data of neighboring edges between this sample vertex and corresponding feature vertices. In such way, the sparse vector structure do not need to be maintained by each sample vertex, thus reduce the memory footprint.

B. Hybrid Computing Model

To support varied propagation behaviors of different machine learning algorithms, Cymbalo employs a hybrid computing model, which provides two different computing strategies: a vertex-centric and an edge-centric strategy. Both computing strategies are built based on the heterogeneity-aware data model described above.

For edge-centric computation, the basic computing unit is edge. In each iteration, the neighboring vertices of each edge are first synchronized with their masters. Each edge then generates messages based on its data and the data of two neighboring vertices. The messages are then propagated to both neighboring vertices. This strategy is suitable for algorithms cast as a bipartite graph, where each vertex is attached with a feature vector, such as MF and LDA.

For vertex-centric computation, the basic computational component is vertex, along with its neighboring edges and vertices. In each iteration, neighbors of each vertex firstly synchronize with their masters. Each vertex then generate messages, which will then be propagated to all its neighbors along one edge direction. This computing strategy is suitable for algorithms that can be cast as a sample-feature graph, such as LR and SVM, where the computation is mainly performed on the sample vertices, and the feature vertices only perform update operations.

C. Vector-centric Programming Model

Cymbalo employs a vector-centric programming model called MCU (M for Mini-Batch, C for Compute, and U for Update), which provides vector-centric operators to implement machine learning algorithms. The Mini-Batch operator is provided to support mini-batch, a scheduling concept allowing users to define a batch of data attending the computation in each iteration. Mini-batch is important for many machine learning algorithms to optimize the rate of convergence. Meanwhile, the operators of MCU are flexible for users to define differentiated scheduling, computation and update strategies for vertices of different types.

Cymbalo is built on Spark. The graph model for a machine learning algorithm is organized as a RDD (Resilient Distributed Dataset) structure in Spark. Operators of the MCU programming model are then defined as the transformation operators of this RDD. The computation phase in each iteration then can be represented as a sequence of transformations on the graph RDD.

The details of different operators are described as below:

Mini-Batch The Mini-Batch operator is provided for users to configure the mini-batch scheduling strategy, including the size of batch, and the vertex subset or edge set that the strategy should run on.

Update The Update operator is provided for users to define the vertex data update operation. As discussed in previous sections, a vertex can either be attached with a vector (the case of MF and LDA), or a single weight value (the case of LR and SVM). This operator considers the difference of data type, and provides two differentiated interfaces. The first interface considers update operation on vertices maintaining a vector, and takes a user-defined function describing the update function on each vertex. The other interface considers the update operation on vertices attached with a single weight value, which treats the data of all vertices in each partition as one vector, and requires users to provide a update function based on this vector. This operator also provides an option for users to select on which vertex subset the update function runs. The formats of the user-defined update functions of two interfaces are as follows:

$$Update(Vector_{a_v}, Vector_v^i, params) \rightarrow Vector_v^{i+1}$$

$$Update(DVector_a, DVector^i, params) \rightarrow DVector^{i+1}$$

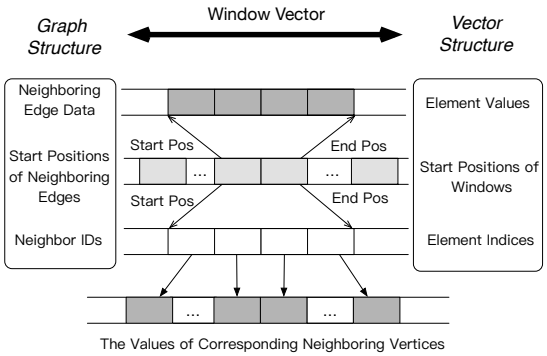


Fig. 5. The window vector abstraction.

In the first function, the $Vector_{a_v}$ indicates the accumulated values of v, i.e., the combined message received by v, such as the delta vector received by each user and item vertex during the computation of MF. The $Vector_v^i$ indicates the current vector of v in iteration i. The $params$ represents the global parameters required by this function, such as the step size and the batch size. In the second function, $DVector_a$ and $DVector^i$ indicate the accumulated values and the current values of all the vertices in each partition respectively in iteration i. For example, in LR, $DVector_a$ records all the computed gradients in each partition, and the $DVector^i$ represents the current data of all feature vertices in each partition.

Compute The Compute operator is provided for users to define the main computation phase, mainly including the propagation operations. To support machine learning algorithms with varied propagation behaviors, this operator provides two different interfaces to apply edge-centric and vertex-centric strategies of the hybrid computation model.

Moreover, for algorithms cast as a sample-feature graph, the sparse feature vector in each sample vertex is stored in a specified vector-aware graph layout to achieve small memory footprint, as described in the previous section. It worth noting that the element values and indices are stored separately in the data array of neighboring edges and the id array of neighboring vertices. Therefore, it is hard to directly build vector-centric interfaces based on the data of either vertex or edge.

To support vector-centric computation, we further propose a new vector abstraction called window vector, which hides the details of low-level graph structures and computation operations from users, and provides basic vector operations for users, such as axpy and dot product.

The window vector abstraction encapsulates the data array of neighboring edges and the id array of neighboring feature vertices of all the sample vertices in each partition, and uses two variables to record the start and end positions of current window. These two positions are used to locate the data and indices of the sparse feature vertex. When a sample vertex is scheduled to execute the user-defined compute function, the position variables of the window are fetched from the index array based on the id of this sample vertex. The user-defined

```
def SparkExec {
    /* Broadcast the weights of features */
    bcFeatures =
       ctx.broadcast(features)

    /* Compute the gradients and other params for
       features */
    (gradients, params) = samples
       .miniBatch(1000) // MiniBatch on samples
       .Aggregate(bcFeatures)
           (ComputeFunc,CombineFunc)

    /* Update features based on the UpdateFunc */
    newFeatures =
       UpdateFunc(gradients, features, params)
}
```

(a) Spark execution stages.

```
def CymbaloExec {
    /* Compute the gradients and other params for
       features */
    (gradients, params) = model
        .miniBatch(1000, "sample")
        .compute(ComputeFunc, CombineFunc)

    /* Update features based on the UpdateFunc */
    model =
       model.update(gradients, params)(UpdateFunc)
}
```

(b) Cymbalo execution stages.

```
/* Compute the gradients and other global params
for features */
ComputeFunc(featureVector, v_sample,
            gradientVector, globalParams) {
   (gradientVector, globalParams) =
        FeaturesGradient(featureVector, v_sample)
}

/* Combine the global parameters */
CombineFunc(param1, param2) {
   param = CombineParams(param1, param2)
}

/* Update features in a batch way */
UpdateFunc(gradientVector, oldFeatureVector,
           globalParams) {
   newFeatures =
       FeaturesUpdate(oldFeatureVector,
           gradientVector, globalParams)
}
```

(c) UDFs for Spark and Cymbalo.

Fig. 6. Programming LR with the MCU model in Cymbalo and the RDD model in Spark. UDFs is short for user-defined functions. The MCU programming model can directly leverage the existing UDFs in Spark, due to its vector-centric interfaces.

compute function are as follows:

$$Compute(Vector_v, D_e, Vector_u, params) \rightarrow (Vector_{a_v}, Vector_{a_u})$$

$$Compute((y, WVector_v), DVector, params) \rightarrow DVector_a$$

The first function is used to perform edge-centric computation, which takes the data of an edge and the data of its two neighboring vertices as the parameters. D_e indicates the data of edge e, and $Vector_v$ indicates the data of vertex v. The edge-centric computation strategy is designed to handle algorithms that can be cast as a bipartite graph of which

```
def TuX2Exec {
    mbStage = new MiniBatchStage
    mbStage.SetBatchSize(1000, "sample")

    mbStage.Add(ExchangeStage0)
    mbStage.Add(ApplyStage)
    mbStage.Add(ExchangeStage1)

    ExecStages.Add(mbStage)
    ExecStages.Add(GlobalSyncStage)
}
```

(a) TuX^2 execution stages

```
/* Compute the gradient based on each edge */
Exchange0(v_feature, v_sample, edge, a_feature,
          a_sample, ctx) {
   a_feature.g +=
       SingleFeatureGradient(v_feature, v_sample)
}

/* Update features */
Apply(v_feature, a_feature, ctx) {
   v.feature.weight +=
       FeatureUpdate(v_feature, a_feature, ctx)
}

/* Update the data recorded in each sample vertex
*/
Exchange1(v_feature, v_sample, edge, a_feature,
          a_sample, ctx) {
   v_sample.margin +=
       SampleMarginCompute(v_feature, v_sample)
}

/* Compute global parameters */
Aggregate(ver, ctx) {
   ctx.param += CalcParam(ver)
}

/* Combine parameters */
Combine(ctx1, ctx2) {
   ctx.param = ctx1.param + ctx2.param;
}
```

(b) UDFs for the MEGA model

Fig. 7. Programming LR with the MEGA model in TuX^2.

each vertex maintains a feature vector. $Vector_{a_v}$ indicates the accumulated vector for vertex v, and the $params$ indicates the required global parameters.

The second function is used to perform vertex-centric computation, and handle the algorithms that can be cast as a sample-feature graph. It takes the data of a sample and the data vector of local features as parameters. The $(y, WVector_v)$ indicate the label and sparse vector of sample vertex v, where $WVector_v$ shows a window vector structure whose window is currently on vertex v.

To elaborate the efficiency of MCU, we further compare the implementation details of MCU based LR, with RDD in Spark and MEGA model in TuX^2, and show them in Fig. 6 and 7. All the implementations are based on the Stochastic Gradient Descent (SGD for short) optimization algorithm. As shown in Fig. 6, MCU can directly leverage existing ComputeFunc and UpdateFunc in Spark, avoiding unnecessary coding efforts. The vector-centric implementation in Spark and Cymbalo is more succinct than the MEGA programming model, which compels users to use edge-centric interfaces and decompose the computation into multiple sub-steps, as shown in Fig. 7.

V. Evaluation

We implement Cymbalo on Spark 2.1.2, and evaluate Cymbalo against two state-of-the-art machine learning systems including Spark 2.1.2, a parameter server system Angel 1.5.1, and three distributed graph systems including PowerGraph 2.2 [2], PowerLyra [10] (based on PowerGraph 2.2) and GraphX [3] (based on Spark 2.1.2). Both GraphX and PowerGraph use a 2D edge partitioning scheme, while PowerLyra uses the BiCut for graph partitioning.

The cluster used for evaluation contains nine Huawei RH2285 servers connected with a 1GigE network. Each server is equipped with two Intel Xeon E5645 processors(each has 12 physical cores), 32GB memory, and 1TB disk.

TABLE I
Dataset Details(K: thousand, M: million, B: billion)

Dataset	Description	# of users / samples	# of items / features	# of edges
KDD10 (LR, SVM)	Educational data from KDD Cup 2010 [16]	19 M	29 M	566 M
KDD12 (LR, SVM)	The dataset of Tecent Weibo from KDD Cup 2012 [17]	150 M	54 M	1.6 B
WeSpam (LR, SVM)	The dataset of web spam pages from Webb Spam Corpus 2011 [18]	350 K	16 M	1.3 B
Netflix (MF)	The dataset of netflix prize [19]	480.2 K	17.8 K	100.5 M
Synthesized Data (MF)	Synthesized data based on netflix dataset	6.4 M	17.8 K	1.3 B

We use three workloads for evaluation: LR, SVM, and MF. All workloads are learned by SGD. The training data we used in evaluation is shown in Table I. KDD10, KDD12 and WebSpam are used in LR and SVM related experiments. The Netflix dataset and another synthesized dataset based on it are used in MF related experiments.

TABLE II
Spark Configuration

Property	Value
spark.executor.memory	27 GB
spark.executor.cores	12
spark.memory.storageFraction	0.5
spark.default.parallelism	96
spark.reducer.maxSizeInFlight	100 MB

First of all, we evaluate the efficiency of the vector-aware vertex layout. As discussed above, this layout is used for algorithms modeled as sample-feature graphs. We calculate the reduction fraction of memory footprint for LR on three datasets when adopting the vector-aware vertex layout, comparing with the case of normal graph layout. As shown in Fig. 8, the memory footprint is reduced by 19%-28% for all three datasets.

We then evaluate the execution efficiency of Cymbalo. For LR and SVM, we compare Cymbalo with Spark and a parameter server system called Angel, which is built on Apache Yarn, and runs each worker and parameter server in a yarn container. The configurations of Spark is shown in Table II. For Angel, we configure each machine to run a worker container and a parameter server container, and set the memory of the worker container the same size as that of the Spark executor. We use the execution time of one-iteration to indicate the performance, and run each workload 100 iterations and take the average to be the final execution time for comparison. As shown in Fig. 9, for LR and SVM workload, Cymbalo outperforms Spark by 2.4×-3.2×, mainly due to the efficient graph computation model. Cymbalo is faster than Angel by 1.6×-2.1×, due to the efficient data model and partitioning scheme, which can explore the locality of parameters, and thus reduce communication overhead.

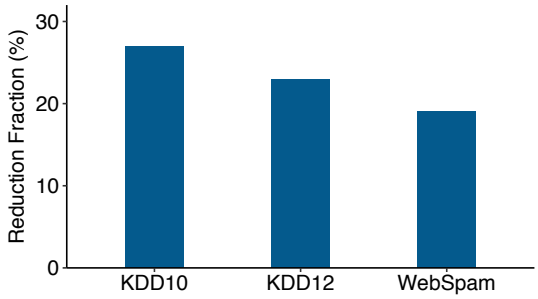


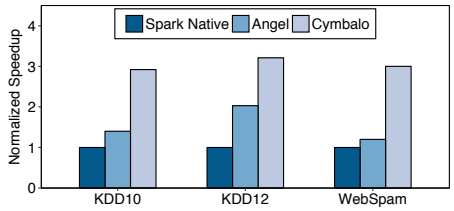
Fig. 8. The reduce fraction of memory footprint of the vector-aware vertex layout on three datasets for LR workload, when comparing with a layout without the awareness of vector structure.

For MF workload, we compare Cymbalo with Angel, GraphX, PowerGraph and PowerLyra. As shown in Fig. 9, Cymbalo achieves the best performance in this case, due to its heterogeneity-aware data model and efficient edge-centric computing strategy. PowerGraph achieves the worst performance, because it needs to perform two GAS phases in each iteration, which compute the accumulated values for user vertices and item vertices respectively [7].

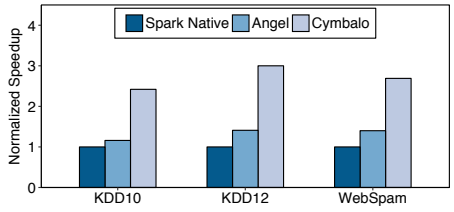
VI. Related Work

In recent years, many distributed graph-parallel processing systems have been proposed to deal with massive graph data. Gemini [20] adopts a sparse-dense signal-slot abstraction, a chunk-based partitioning scheme, a dual representation scheme to build a scalable graph processing system on top of efficiency. LazyGraph [21] proposes a lazy data coherency approach which allows vertex replicas to share the global view only at some selected coherency points to reduce the number of synchronization. Meanwhile, many graph processing systems [22], [23] support out-of-core mechanisms, which consider how to efficiently utilize disks to accelerate graph computation.

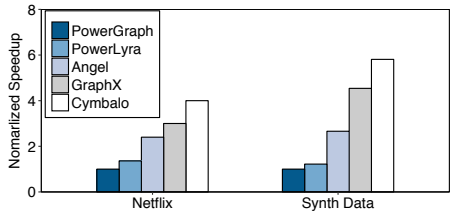
Many other distributed systems have also been proposed in recent years for large-scale machine learning problems. The Google Parameter Server [24] distributes data and parameters among workers and servers, and provides efficient communi-



(a) LR



(b) SVM



(c) MF

Fig. 9. The performance comparison between different systems on LR, SVM and MF workloads. The higher is the better.

cation model and flexible consistency models to achieve high performance and scalability. STRADS [25] further consider optimizations on model parallelism, task scheduling and uneven convergence.

VII. Conclusion

We propose Cymbalo, a new distributed graph processing framework with efficient support for machine learning algorithms from multiple perspectives, including data model, computation model, and programming model. Experimental results show that Cymbalo outperforms many state-of-the-art distributed machine learning systems, like Spark, PowerGraph, and Angel, obtaining a speedup ranging from 1.6× up to 5.8×.

References

[1] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. ACM, 2010, pp. 135–146.

[2] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "Powergraph: Distributed graph-parallel computation on natural graphs." in *OSDI*, vol. 12, no. 1, 2012, p. 2.

[3] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica, "Graphx: Graph processing in a distributed dataflow framework." in *OSDI*, vol. 14, 2014, pp. 599–613.

[4] X. Tian, Y. Guo, J. Zhan, and L. Wang, "Towards memory and computation efficient graph processing on spark," in *2017 IEEE International Conference on Big Data (Big Data)*, Dec 2017, pp. 375–382.

[5] A. Ching, S. Edunov, M. Kabiljo, D. Logothetis, and S. Muthukrishnan, "One trillion edges: Graph processing at facebook-scale," *Proceedings of the VLDB Endowment*, vol. 8, no. 12, pp. 1804–1815, 2015.

[6] R. Chen, J. Shi, B. Zang, and H. Guan, "Bipartite-oriented distributed graph partitioning for big learning," in *Proceedings of 5th Asia-Pacific Workshop on Systems*. ACM, 2014, p. 14.

[7] W. Xiao, J. Xue, Y. Miao, Z. Li, C. Chen, M. Wu, W. Li, and L. Zhou, "Tux2: Distributed graph computation for machine learning." in *NSDI*, 2017, pp. 669–682.

[8] C. Avery, "Giraph: Large-scale graph processing infrastructure on hadoop," *Proceedings of the Hadoop Summit. Santa Clara*, vol. 11, no. 3, pp. 5–9, 2011.

[9] S. N. A. Project. Stanford large network dataset collection. [Online]. Available: http://snap.stanford.edu/data/

[10] R. Chen, J. Shi, Y. Chen, and H. Chen, "Powerlyra: Differentiated graph computation and partitioning on skewed graphs," in *Proceedings of the Tenth European Conference on Computer Systems*. ACM, 2015, p. 1.

[11] F. Petroni, L. Querzoni, K. Daudjee, S. Kamali, and G. Iacoboni, "Hdrf: Stream-based partitioning for power-law graphs," in *Proceedings of the 24th ACM International on Conference on Information and Knowledge Management*. ACM, 2015, pp. 243–252.

[12] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *NSDI*. USENIX Association, 2012, pp. 2–2.

[13] 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," *IEEE Transactions on Big Data*, vol. 1, no. 2, pp. 49–67, 2015.

[14] J. Jiang, L. Yu, J. Jiang, Y. Liu, and B. Cui, "Angel: a new large-scale machine learning system," *National Science Review*, vol. 5, no. 2, pp. 216–236, 2017.

[15] Z. Xianyi, W. Qian, and W. Saar, "Openblas: an optimized blas library.(2016)," 2016.

[16] H.-F. Yu, H.-Y. Lo, H.-P. Hsieh, J.-K. Lou, T. G. McKenzie, J.-W. Chou, P.-H. Chung, C.-H. Ho, C.-F. Chang *et al.*, "Feature engineering and classifier ensemble for kdd cup 2010," in *KDD Cup*, 2010.

[17] Y. Juan, Y. Zhuang, W.-S. Chin, and C.-J. Lin, "Field-aware factorization machines for ctr prediction," in *Proceedings of the 10th ACM Conference on Recommender Systems*. ACM, 2016, pp. 43–50.

[18] S. Webb, J. Caverlee, and C. Pu, "Introducing the webb spam corpus: Using email spam to identify web spam automatically." in *CEAS*, 2006.

[19] J. Bennett, S. Lanning *et al.*, "The netflix prize," in *Proceedings of KDD cup and workshop*, vol. 2007. New York, NY, USA, 2007, p. 35.

[20] X. Zhu, W. Chen, W. Zheng, and X. Ma, "Gemini: A computation-centric distributed graph processing system." in *OSDI*, 2016, pp. 301–316.

[21] L. Wang, L. Zhuang, J. Chen, H. Cui, F. Lv, Y. Liu, and X. Feng, "Lazygraph: lazy data coherency for replicas in distributed graph-parallel computation," in *PPoPP '18*, 2018, pp. 276–289.

[22] D. Yan, Y. Huang, M. Liu, H. Chen, J. Cheng, H. Wu, and C. Zhang, "Graphd: Distributed vertex-centric graph processing beyond the memory limit," *IEEE Transactions on Parallel and Distributed Systems*, vol. PP, no. 99, pp. 1–1, 2018.

[23] M. Zhang, Y. Wu, Y. Zhuo, X. Qian, C. Huan, and K. Chen, "Wonderland: A novel abstraction-based out-of-core graph processing system," in *ASPLOS '18*. New York, NY, USA: ACM, 2018, pp. 608–621.

[24] 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*, vol. 14, 2014, pp. 583–598.

[25] J. K. Kim, Q. Ho, S. Lee, X. Zheng, W. Dai, G. A. Gibson, and E. P. Xing, "Strads: a distributed framework for scheduled model parallel machine learning," in *Proceedings of the Eleventh European Conference on Computer Systems*. ACM, 2016, p. 5.