Exploring the Hidden Dimension in Graph Processing

Mingxing Zhang, Yongwei Wu, and Kang Chen, ***Tsinghua University;*** **Xuehai Qian,** ***University of Southern California;*** **Xue Li and Weimin Zheng,** ***Tsinghua University***

https://www.usenix.org/conference/osdi16/technical-sessions/presentation/zhang-mingxing

This paper is included in the Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16).

November 2–4, 2016 • Savannah, GA, USA

ISBN 978-1-931971-33-1

Open access to the Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation is sponsored by USENIX.

Exploring the Hidden Dimension in Graph Processing

Mingxing Zhang Yongwei Wu Kang Chen Xuehai Qian† Xue Li Weimin Zheng
Tsinghua University †University of Southern California*

Abstract

Task partitioning of a graph-parallel system is traditionally considered equivalent to the graph partition problem. Such equivalence exists because the properties associated with each vertex/edge are normally considered indivisible. However, this assumption is not true for many Machine Learning and Data Mining (MLDM) problems: instead of a single value, a *vector* of data elements is defined as the property for each vertex/edge. This feature opens a new dimension for task partitioning because a vertex could be divided and assigned to different nodes.

To explore this new opportunity, this paper presents *3D partitioning*, a novel category of task partition algorithms that significantly reduces network traffic for certain MLDM applications. Based on 3D partitioning, we build a distributed graph engine CUBE. Our evaluation results show that CUBE outperforms state-of-the-art graph-parallel system PowerLyra by up to 4.7× (up to 7.3× speedup against PowerGraph).

1 Introduction

Efficient graph-parallel systems require careful task partitioning. It plays a pivotal role because the load balancing and communication cost are largely determined by the partitioning strategy. All existing partitioning algorithms in current systems assume that the property of each vertex/edge is indivisible. Therefore, task partitioning is equivalent to graph partitioning. But, in reality, the property associated with a(n) vertex/edge for many Machine Learning and Data Mining (MLDM) problems is a *vector* of data elements, which is *not* indivisible.

This new feature can be illustrated by a popular machine learning problem, Collaborative Filtering (CF), which estimates the missing ratings based on a given incomplete set of (user, item) ratings. The original problem is defined in a matrix-centric view: given a sparse rating matrix **R** with size $N \times M$, the goal is to find two dense matrices **P** (with size $N \times D$) and **Q** (with size $M \times D$) that are **R**'s non-negative factors (i.e., $\mathbf{R} \approx \mathbf{P} \times \mathbf{Q^T}$). Here, N and M are the number of users and items, respectively. D is the size of feature vector. When formulated in a graph-centric view, the rows of **P** and **Q** correspond to vertices of a bipartite graph. Each vertex is associated with a property vector with D features. In contrast, the rating matrix **R** corresponds to edges. For every non-zero element (u, v) in matrix **R**, there is an edge connects vertex p_u and vertex q_v, and the weight of this edge is R_{uv}. An illustration of these two views is given in Figure 1.

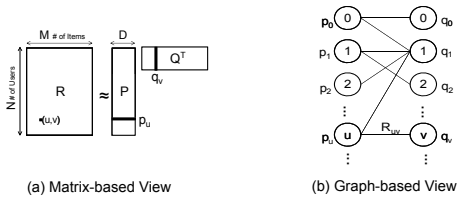


Figure 1: Collaborative Filtering.

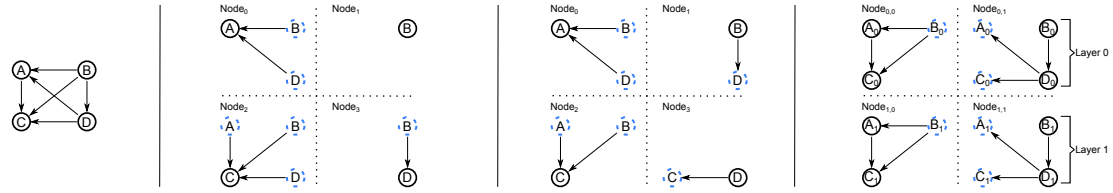
One distinct nature of the graph in Figure 1 (b) is that each vertex is associated with a divisible element vector, which is a common pattern when modelling MLDM algorithms as graph computing problems. Another good example is Sparse Matrix to Matrix Multiplication (SpMM), a prevalently used computation kernel that multiplies a dense feature matrix with a sparse parameter matrix (see Section 5.2.1 for more details). SpMM dominates the execution time of most minibatch-based neural network training algorithms.

In essence, when formulating matrix-based applications as graph problems, the property of vertex or edge is usually a vector of elements, instead of a single value. More importantly, during computation, these property vectors are mostly manipulated by *element-wise* operators, where the computations can be perfectly parallelized without any additional communication when disjoint ranges of vector elements are assigned to different nodes.

Due to the common pattern of vector property and its amenability to parallelism, this paper considers a *new dimension* of task partitioning, which is assigning disjoint elements of the same property to different nodes. It is considered to be a *hidden* dimension in existing 1D/2D partitioners used in previous systems [10, 15, 16, 24] because all of them treat the property as an indivisible component. According to our investigation, the *3D partitioning* principle could significantly reduce network traffic and improve performance.

The key intuition is that: since each node only processes a subset of elements in property vectors, it can be assigned with more edges and vertices that otherwise need to be assigned to different nodes. Therefore, on the bright side, certain communications previously happened

*M. Zhang, Y. Wu, K. Chen, X. Li and W. Zheng are with the Department of Computer Science and Technology, Tsinghua National Laboratory for Information Science and Technology (TNLIST), Tsinghua University, Beijing 100084, China; Technology Innovation Center at Yinzhou, Yangtze Delta Region Institute of Tsinghua University, Ningbo 315000, Zhejiang.



(a) Sample graph. (b) 1-D partitioning: each vertex is attached with all its incoming edges. (c) 2-D partitioning: the edges are equally partitioned. (d) 3-D partitioning: each vertex is split into two sub-vertices, and a 2-D partitioner is used for each layer.

: Vertices in blue dotted circles are replicas, while the others are masters. It is a sub-vertex that contains only a subset of properties if a subscript is attach to the vertex's ID.

Figure 2: An illustration of 1D, 2D, and 3D partitioning.

between nodes are converted to local value exchanges. But, on the other side, 3D partitioning may incur extra synchronizations between sub-vertices/edges. In either case, with 3D partitioning, programmers are given the option to carefully choose the partition strategy of this third dimension. This ability enables them to explore a new tradeoff that may lead to better performance, which is prohibited by traditional 1D/2D partitioners. Importantly, 3D partitioning does not require long property vector to be effective. Our results show that a network traffic reduction up to 90.6% can be achieved by partitioning this dimension into just *64 layers*. In other words, our algorithm works very well on property vectors with modest and reasonable size.

Based on a novel 3D partitioning algorithm, we build a distributed graph processing engine CUBE, which introduces significantly fewer communication than existing systems in many real-world cases. To achieve better performance, CUBE internally uses a matrix-based data structure for storing and processing graphs while providing a set of vertex-centric APIs for the users. The matrix-based design is inspired by a recent graph-processing system [34], which only works on a *single* machine. The design of CUBE achieves both the programming productivity of vertex programming and the high performance of a matrix-based backend.

This paper makes the following contributions.

i) We propose the *first* 3D graph partitioning algorithm (Section 3.2) for graph-parallel systems. It considers a hidden dimension that is ignored by all previous systems. Unlike traditional 1D and 2D partitioning, the new dimension allows dividing the elements of property vectors to different nodes. Our 3D partitioning offers unprecedented performance that is not achievable by traditional graph partitioning strategies in existing systems.

ii) We propose a new programming model **UPPS** (**U**pdate, **P**ush, **P**ull, **S**ink) (Section 3.3) designed for 3D partitioning. The existing graph-oriented programming models are insufficient because they implicitly assume that the entire property of a single vertex is accessed as an indivisible component.

iii) We build CUBE, a graph processing engine that adopts 3D partitioning and implements the proposed vertex-centric programming model UPPS. The system significantly reduces communication cost and memory consumption. We use matrix-based data structures in the backend which reduces the COST metric [25] of our system to as low as four (Section 4).

iv) We systematically study the effectiveness of 3D partitioning with both micro-benchmarks (Section 5.2) and real-world MLDM algorithms (Section 5.3.3). The results show that it only trades a negligible growth of graph partitioning time for a notable reduction of both communication cost and memory consumption. Overall, CUBE outperforms state-of-the-art graph-parallel system PowerLyra by up to 4.7× (up to 7.3× speedup against PowerGraph).

2 Motivation and Background

An optimal task partitioning algorithm should *1)* ensure the balance of each node's computation load; and *2)* try to minimize the communication cost across multiple nodes. As the existing schemes assume that the property of each vertex is indivisible, the partitioning of graph-processing task is originally considered equivalent to graph partitioning. More specifically, existing partitioners try to optimally place the graph-structured data, including vertices and edges, across multiple machines, so that *1)* the number of edges on each node (which is roughly proportional to computation loads) is balanced; and *2)* the number of replicas (i.e., the number of shared vertices/edges which is proportional to the communication cost) is as small as possible. Two kinds of approaches exist for solving this problem: 1D partitioning and 2D partitioning.

1D Both GraphLab [22] and Pregel [23] adopt a **1D** partitioning algorithm. It assigns each node a disjoint set of vertices and all the connected incoming/outcoming edges. This algorithm is enough for randomly generated graphs, but for real-world graphs that follow the power law, a 1D partitioner usually leads to considerable skewness [15].

2D To avoid the drawbacks of 1D partitioning, recent systems [8, 15] are based on **2D** partitioning algorithms, in which the the graph is partitioned by edge rather than vertex. With a 2D partitioner, the edges of a graph will be equally assigned to each node. The system will set

up replica of vertices to enable computation, and the automatic synchronization of these replicas requires communication. Various heuristics have been proposed to reduce communication cost by generating fewer number of replicas. For example, PowerLyra [10] uses a hybrid graph partitioning algorithm (named *Hybrid-cut*) that combines 1D partitioning and 2D partitioning with heuristics. By treating high-degree and low-degree vertices differently, Hybrid-cut achieves much lower communication cost on many real-world datasets. However, Hybrid-cut is still a special case of 2D partitioning that does not assign the same property vector to different nodes.

3D In many MLDM problems, a vector of data elements is associated to each vertex or edge hence the assumption of indivisible property is untrue and not necessary. This new dimension for task partitioning naturally leads to a new category of **3D** partitioning algorithms. To be more specific, for an N-node cluster a 3 partitioner will use L copies of the graph topology, where L is the number of layers and N is divisible by L. Each of these copies is partitioned by a regular 2D partitioner among a layer of only N/L nodes. On the other hand, the vector data associated to the graph are partitioned across layers evenly. In this setting, each layer occupies N/L nodes and the same graph with only subset of elements ($1/L$ of the original property vector) in its edges/vertices are partitioned among these N/L nodes by a regular 2D partitioner. Therefore, each vertex is split into L sub-vertices and the i^{th} layer maintains a copy of the graph that comprises of all the i^{th} sub-vertices/edges. 3D partitioning reduces communication cost **along edges** (e.g., synchronizations caused by element-wise operators), because the graph is partitioned across fewer nodes in each layer, thus each node in a layer could be assigned with more vertices and edges. This essentially converts the otherwise inter-node communication to local data exchanges.

Figure 2 compares the different partition algorithms applied on the graph in Figure 2 (a). In 1D partitioning (Figure 2 (b)), each node is assigned with one vertex and the incoming edges. There are six replicas in total. In 2D partitioning (Figure 2 (c)), edges are evenly partitioned, which leads to the same number of replicas as 1D partitioning.

Figure 2 (d) illustrates the concepts of 3D partitioning, where N is 4 and L is 2. First, the total of 4 nodes are divided to two layers. We denote each node as $Node_{i,j}$, where i is the layer index and j is the node index within a layer. Second, the graph is partitioned in the same way in both layers using a 2D partitioning algorithm. Different from 1D and 2D partitioning, since the number of nodes for each layer is halved, each node is assigned with more vertices and edges. In the example, the first node in all layers ($Node_{0,0}$ and $Node_{1,0}$) are assigned with 3 edges and 3 connected vertices, in which 1 vertex is replica. The second node in all layers ($Node_{0,1}$ and $Node_{1,1}$) are also assigned with 3 edges and 4 connected vertices, but among which 2 vertices are replicas. The increased number of vertices and edges in each node (3 edges in each layer of Figure 2 (d) compared to 1 or 2 edges in Figure 2 (b),(c)) translates to the reduced number of replicas needed for each layer (3 replicas in Figure 2 (d)) compared to 6 in Figure 2 (b),(c)). Although the total number of replicas (*3 replicas* $\times$ *2 layers = 6 replicas*) in all layers stays the same, the size of each replica is halved, therefore, the network traffic needed for replica synchronization is halved[1]. In essence, a 3D partitioning algorithm reduces the number of sub-graphs in each layer and hence reduces the **intra-layer** replica synchronization overhead.

However, 3D partitioning will incur a new kind of synchronization not needed before: the **inter-layer** synchronization between sub-vertices/edges. Therefore, programmers should carefully choose the number of layers to achieve the best performance. Nevertheless, the traditional 1D and 2D partitioning do not allow programmers to explore this tradeoff. A detailed discussion of this performance tradeoff is given in Section 5.

3 Programming Model

Existing graph-oriented programming models (e.g. GAS [15], TripletView [16], Pregel [23]) are designed for 1D/2D partitioning algorithms. They are insufficient for 3D partitioning because it is assumed that all elements of a property vector are accessed as an indivisible component. Thus, we adapt the popular GAS model and incorporate it with 3D partitioning, which leads to a new model named **UPPS** (**U**pdate, **P**ush, **P**ull, **S**ink) that accommodates 3D partitioning requirements. In this section, we first introduce UPPS and describe how a graph can be partitioned in the 3D fashion using UPPS. Then we explain the operations of UPPS and demonstrate their usages with two examples.

3.1 Data

As a vertex-centric model, UPPS models the user-defined data D as a directed data graph G, which consists of a set of vertices V together with a set of edges E. Users are allowed to associate arbitrary type of data with vertices and edges. The data attached to each vertex/edge are partitioned into two classes: *1)* an indivisible property *DShare* that is represented by a single variable; and *2)* a divisible collection of property vector elements *DColle*, which is stored as a **vector** of variables. The detailed specification of UPPS is given in Table 1.

[1] In some cases, there may be a shared part of every sub-vertices. We will discuss this situation later.

Table 1: The programming model UPPS.

Data					
G	—	$\{V, E, D = \{DShare, DColle\}, S_C\}$	$G_{bipartite}$	—	$\{\mathbb{U}, \mathbb{V}, E, D = \{DShare, DColle\}, S_C\}$
$DShare_u$	—	a single variable	$DShare_{u\to v}$	—	a single variable
$DColle_u$	—	a vector of variable with size S_C	$DColle_{u\to v}$	—	a vector of variable with size S_C
$DColle_u[i]$	—	the i^{th} element of $DColle_u$	$DColle_{u\to v}[i]$	—	the i^{th} element of $DColle_{u\to v}$
$D_u[i]$	—	abbreviation of $\{DShare_u, DColle_u[i]\}$	$D_{u\to v}[i]$	—	abbreviation of $\{DShare_{u\to v}, DColle_{u\to v}[i]\}$

Computation		
UpdateVertex($\mathcal{F}$)	—	**foreach** vertex $u \in V$ **do** $D_u^{new} := \mathcal{F}(D_u)$;
UpdateEdge($\mathcal{F}$)	—	**foreach** edge $(u,v) \in E$ **do** $D_{u\to v}^{new} := \mathcal{F}(D_{u\to v})$;
Push($\mathcal{G}$, $\mathcal{A}$, $\oplus$)	—	**foreach** vertex $v \in V$, index $i \in [0, S_C)$ **do** $DColle_v^{new}[i] := \mathcal{A}(D_v[i], \bigoplus_{(u,v)\in E}(\mathcal{G}(D_u[i], D_{u\to v}[i])))$;
Pull($\mathcal{G}$, $\mathcal{A}$, $\oplus$)	—	**foreach** vertex $u \in V$, index $i \in [0, S_C)$ **do** $DColle_u^{new}[i] := \mathcal{A}(D_u[i], \bigoplus_{(u,v)\in E}(\mathcal{G}(D_v[i], D_{u\to v}[i])))$;
Sink($\mathcal{H}$)	—	**foreach** edge $(u,v) \in E$, index $i \in [0, S_C)$ **do** $DColle_{u\to v}^{new}[i] := \mathcal{H}(D_u[i], D_v[i], Du{\to}v[i])$;

Users are required to assign an integer S_C as the **collection size** that defines the size of each *DColle* vector. When only *DShare* part of the edge data is used, *DColle* of edges can be set to *NULL*. But, if *DColle* of vertices and edges are both enabled, UPPS requires that their length should be equal. This restriction avoids inter-layer communication for certain operations (see Section 3.3). Moreover, if the input graph is undirected, the typical practice is using two directed edges (in each direction) to replace each of the original undirected edge. But, for many bipartite graph based MLDM algorithms, only one direction is needed (see more details in Section 3.6).

3.2 3D Partitioning

By explicitly decoupling the divisible property vector *DColle* and the indivisible part *DShare*, UPPS allows users to divide each vertex/edge into several sub-vertices/edges so that each of them has a copy of *DShare* and a **disjoint subset** of *DColle*. Based on UPPS, a 3D partitioner could be constructed by first dividing nodes into layers based on a layer count L and then partitioning the sub-graph in each layer following a 2D partitioning algorithm $\mathcal{P}$. Thus, a 3D partitioner can be denoted as $(\mathcal{P}, L)$.

Specifically, we should first guarantee that N is divisible by L. Then, the partitioner will *1)* equally group the nodes into L layers so that each layer contains N/L nodes; *2)* partition edge set E into N/L sub-sets with the 2D partitioner $\mathcal{P}$; and *3)* randomly separate vertex set V into N/L sub-sets. In the rest of this paper, we use $Node_{i,j}$ to denote the j^{th} node of the i^{th} layer; E_j and V_j to denote the j^{th} subset of E and V, respectively.

With the above definitions, after partition, $Node_{i,j}$ contains the following data copies:

- a shared copy of $DShare_u$, if vertex $u \in V_j$;
- an exclusive copy of $DColle_u[k]$, if vertex $u \in V_j$ and $LowerBound(i) \le k < LowerBound(i+1)$;
- a shared copy of $DShare_{u\to v}$, if edge $(u,v) \in E_j$;
- an exclusive copy of $DColle_{u\to v}[k]$, if edge $(u,v) \in E_j$ and $LowerBound(i) \le k < LowerBound(i+1)$;

In the above equations, $LowerBound(i)$ equals to $i * (\lfloor S_C/L \rfloor) + min(i, S_C\%L)$. In other words, each layer of the nodes contains a shared copy of all the *DShare* data and an exclusive sub-set of the *DColle* data.

In a 3D partitioning $(\mathcal{P}, L)$, both L and $\mathcal{P}$ affect the communication cost. When $L = N$, each layer only has one node which keeps the entire graph and processes $1/L$ of *DColle* elements. In this case, no replica for *DColle* data is needed, and the intra-layer communication cost is zero. But, it could potentially incur higher inter-layer communication due to synchronization between sub-vertices/edges. When $L = 1$, there is only one layer and $(\mathcal{P}, L)$ is degenerated to the 2D partitioning $\mathcal{P}$. Therefore, the communication cost is purely determined by $\mathcal{P}$. The common practice is to choose the L between 1 and N, so that both L and $\mathcal{P}$ will affect communication cost. The programmers are responsible for investigating the tradeoff and choosing the best setting. To help users choose the appropriate L, we provide the equations to calculate communication costs for different UPPS operations which can be used as building blocks for real applications (see Section 5.2). Within a layer, one can choose any 2D partitioning $\mathcal{P}$ and it is orthogonal to L.

3.3 Computation

UPPS has four types of operations which resemble the name of the model: **U**pdate, **P**ush, **P**ull, and **S**ink. The definition of these operations are given in Table 1. All possible variant forms of computations allowed in UPPS are also encoded in these APIs.

Update This operation takes **all** the information of each vertex/edge to calculate the new value. Roughly, *Update* operates on all elements of an edge or vertex in *vertical* direction. Since vertices and edges may be split into sub-vertices/edges, each node $Node_{i,j}$ needs to synchronize with nodes in other layers while updating. Note that *Update* only incurs inter-layer communicate between a node and nodes in other layers that share the same subset of vertices (V_j) or edges (E_j) (i.e., $Node_{*,j}$).

Push, Pull, Sink These three operations handles updates in *horizontal* direction: the updates follow the dependency relations determined by graph structure. For each edge $(u,v) \in E$: *Push* operation uses data of vertex u and edge (u,v) to update vertex v; *Pull* operation uses data of vertex v and edge (u,v) to update vertex u; *Sink* operation uses data of u and v to update the edge (u,v).

Push/Pull operation resembles the popular GAS (Gather, Apply, Scatter) operation. In GAS, each vertex reads data from its in-edges with the gather function $\mathcal{G}$, generates the updated value based on sum function $\oplus$, which is used to update the vertex using the apply function $\mathcal{A}$. UPPS further partitions property vertex, which is always considered as an indivisible component in GAS. To avoid inter-layer communication, UPPS restricts that the i^{th} *DColle* element of each vertex/edge will only depend on either *DShare* (which is by definition replicated in *all* layers) or the i^{th} *DColle* element of other vertices/edges (which is by definition exist in the *same* layer). Similar restriction applies to *Sink*. In other words, $Node_{i,j}$ only communicates to $Node_{i,*}$ in *Push/Pull/Sink*.

3.4 Bipartite Graph

Many MLDM problems model their input graphs as bipartite graphs, where vertices are separated into two disjoint sets $\mathbb{U}$ and $\mathbb{V}$ and edges connect pairs of vertices from $\mathbb{U}$ and $\mathbb{V}$. A recent study [11] demonstrates the unique properties of bipartite graphs and the special need of differentiated processing for vertices in $\mathbb{U}$ and $\mathbb{V}$. To capture this requirement, UPPS provides two additional APIs: *UpdateVertexU* and *UpdateVertexV*. They only update the vertices in $\mathbb{U}$ and $\mathbb{V}$, respectively. We use the bipartite-specialized 2D partitioner *bi-cut* [11] as $\mathcal{P}$ for bipartite graphs.

3.5 Compare with GAS

UPPS also follows the popular "think as a vertex" philosophy so that it is easy for programmers to use. In fact, the popular GAS model is a special case of UPPS that has $S_C \leq 1$[2]. Thus, users only need to make moderate changes to their original programs if they just want to take advantage of our efficient matrix backend.

[2]In this case, the workers can only be partitioned into one layer and hence our 3D partitioner degenerates to a traditional 1D/2D partitioner.

In contrast, if the users want to reduce the communication cost by using a 3D partitioner, the workers should be partitioned into at least two layers. As we will show, many popular algorithms can benefit from 3D partitioning without significant program change. Take the breadth-first search (BFS) as an example, in GAS, it can be implemented by: *1)* associating a boolean property to each vertex, which represents whether this vertex has been accessed or not; *2)* propagating this property in the Scatter phase; and *3)* using boolean 'OR' operation in both the Gather and Apply phase. In order to extend this application to do multi-source BFS, users of GAS model can simply *1)* replacing the original boolean variable of each vertex to a vector of boolean variables with length k, where k is the number of sources; and *2)* using element-wise 'OR' operation in the Gather and Apply phase. We see that the computation of GAS-based multi-source BFS is dominated by element-wise operations of two vectors, which shows a notable sign of optimization opportunity with 3D partitioning and UPPS. In fact, with UPPS, the multi-source BFS can be simply implemented by using the **same** program as original BFS. The only difference is that the collection size S_C is set to k rather than one.

Moreover, although it is not used in the above example, users of UPPS may want to have a complete view of the whole vector property of vertices/edges. In 3D partitioning, this intention results in a new kind of inter-layer communication, which inevitably leads to additional APIs (our Update operations). Examples of the usages of these new APIs are given in the next section.

Algorithm 1 Program for GD.

Data:
S_C :— D
$DShare_u$:— NULL; $DShare_{u\to v}$:— {*double Rate*, *double Err*}
$DColle_u$, $DColle_{u\to v}$:— *vector*<*double*>(S_C)

Functions:
$F_1(u_i, v_i, e_i)$:— {*return* $u_i.DColle[i] * v_i.DColle[i]$;}
$F_2(e)$:— {
  $e.DShare.Err := sum(e.DColle) - e.DShare.Rate$;
  return e;
}
$F_3(u_i, e_i)$:— {*return* $e_i.DShare.Err * u_i.DColle[i]$; }
$F_4(v_i, \Sigma)$:— {*return* $v_i.DColle[i] + \alpha * (\Sigma - \alpha * v_i.DColle[i])$;}

Computation for each iteration:
Sink(F_1);
UpdateEdge(F_2);
Pull(F_3, F_4, +);
Push(F_3, F_4, +);

3.6 Examples

For showcasing the usages of UPPS, we implemented two different algorithms that both solve the Collaborative filtering (CF) problem. The two algorithms together cover the usage of all operations in UPPS. In this section,

we only explain at a high-level what UPPS operations do. The detailed implementation of each UPPS operation is given in Section 4.

CF is a kind of problems that estimate the missing ratings based on a given incomplete set of (user, item) ratings. Specifically, if we use N to denote the number of users and M to denote the number of items, input of CF is $R = \{R_{u,v}\}_{N\times M}$, which is a sparse user-item matrix where each item $R_{u,v}$ represents the rating of item v given from user u. The output of CF is two matrices P and Q, which are the user feature matrix and item feature matrix, respectively. P_u and Q_v are feature vectors of user u and item v, and each of them has a size of D. If we use $Err_{u,v}$ to represent the current prediction error of user-item pair (u,v), it is calculated by subtracting the dot product of the corresponding feature vectors with the actual rate, i.e., $Err_{u,v} = <P_u,\ Q_v^T> - R_{u,v}$. The object function of CF is minimizing $\sum_{(u,v)\in R} Err_{u,v}^2$.

GD Gradient Descent (GD) algorithm [20] is a classical solution to solve CF problem, which involves randomly initializing feature vectors and improving them iteratively. The parameters of this algorithm are updated by a magnitude proportional to the learning rate α in the opposite direction of the gradient, which results in the following update rules:

$$P_i^{new} := P_i + \alpha * (Err_{i,j} * Q_j - \alpha * P_i)$$
$$Q_j^{new} := Q_j + \alpha * (Err_{i,j} * P_i - \alpha * Q_j)$$

The program of GD implemented in UPPS is given by Algorithm 1, in which + is an abbreviation of the simple "sum" function. For simplicity, we do not show regularization code used to impose non-negativity on P and Q. In Algorithm 1, the collection size S_C is set to D, hence each vertex/edge's *DColle* part is a vector of *double* with length D. For vertices, which are used for modeling the users and items, these vectors are used to store the corresponding feature vector of the user/item. For edges, these vectors are temporary buffers for reserving partial results of the dot production. As for shared data, the *DShare* part of each edge (u,v) is a pair of {*double Rate*, *double Err*}, which represents the rating given to item v from user u and the current error in predicting this rating. In contrast, *DShare* for vertices are not used.

With the data defined as above, the computation of Algorithm 1 is almost an one-to-one translation of the above equations. In the first step, Algorithm 1 calculates prediction errors $Err_{u,v}$ for every given rating (i.e., every edge) by: *1)* using a *Sink* operation to compute the production of every aligned feature elements and store the result in the edge's *DColle* vector; and *2)* using an *UpdateEdge* operation to sum up each edge's *DColle* vector (i.e., $<P_u,\ Q_v^T>$) and subtract it with the corresponding *Rate*. After calculating the current errors, the updating formulas mentioned above can be implemented in a straightforward way (the *Pull* and the *Push* operation in Algorithm 1).

As shown in Algorithm 1, programmers only need to define the *Push*, *Pull* and *Sink* operation on *one* element of the *DColle* vector (i.e., the user-defined functions operate only one index i), while the *UpdateEdge* and *UpdateVertex* reads or writes all vector elements. Importantly, programmers do *not* need to specify "which sub-vertex/edge contains which *DColle* elements". The details such as indexes of data elements for each layer are specified in a decoupled manner and automatically handled by the framework (Section 4.4).

ALS Alternating Least Squares (ALS) [38] is another algorithm to solve CF problem. It alternatively fixes one unknown feature matrix and solves another by minimizing the object function $\sum_{(u,v)\in R} Err_{u,v}^2$. This approach turns a non-convex problem into a quadratic one that can be solved optimally. A general description of ALS is as follows:

Step 1 Randomly initialize matrix P.

Step 2 Fix P, calculate the best Q that minimizes the error function. This can be implemented by setting $Q_v = (\sum_{(u,v)\in R} P_u^T P_u)^{-1} (\sum_{(u,v)\in R} R_{u,v} P_u^T)$.

Step 3 Fix Q, calculate the best P in a similar way.

Step 4 Repeat Steps 2 and 3 until convergence.

As a typical bipartite algorithm, we implement ALS with the specialized APIs described in Section 3.4. Algorithm 2 presents our program, where the regularization code is also omitted. In ALS, the collection size S_C is set to "$D + D * D$" rather than just D. Each of the *DColle* vector contains two parts: *1)* a feature vector *Vec* with size D that stores the corresponding feature vector; and *2)* a buffer *Mat* with size $D \times D$, which is used to keep the result of $Vec^T * Vec$.

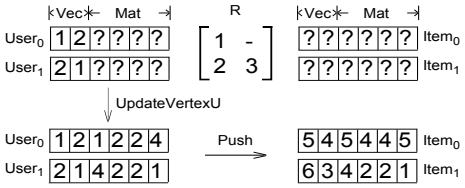


Figure 3: An illustration of ALS's Step 2. In this example, there are two users, two items, and three given ratings.

Figure 3 presents a typical example of ALS' Step 2. First, only the users' feature matrix P is initialized, so that only the feature vector of every vertex in $\mathbb{U}$ contains valid data; and the others are '?'. Then, an *UpdateVertexU* operation is used to calculate $Vec^T * Vec$ for every vertex in $\mathbb{U}$, and the results are stored in the corresponding *Mat* area. After that, a *Push* operation is used to aggregate the corresponding values. For each $v \in \mathbb{V}$, $\sum_{(u,v)\in R} R_{u,v} P_u^T$ and $\sum_{(u,v)\in R} P_u^T P_u$ are calculated and stored in v's *Vec* (i.e., *DColle[0:D-1]*) and *Mat* (i.e., *DColle[D:D+D^2-1]*) area, respectively. Finally, the optimal value of Q_v is calculated by solving a linear equation,

which is implemented by calling the DSYSV function in LAPACK [2] (not illustrated in Figure 3). Similarly to Step 2, Step 3 of ALS can be implemented with the symmetrical call of *UpdateVertexV*, *Pull*, and *UpdateVertexU*.

Algorithm 2 Program for ALS.

```
Data:
    S_C :— D + D * D
    DShare_u :— NULL;    DShare_{u→v} :— {double Rate}
    DColle_u :— vector<double>(S_C);    DColle_{u→v} :— NULL
Functions:
    F_1(v) :— {
        foreach (i, j) from (0,0) to (D−1, D−1) do
            v.DColle[D + i * D + j] := v.DColle[i] * v.DColle[j];
        return v;
    }
    F_2(u_i, e_i) :— {
        if i < D do return e_i.DShare.Rate * u_i.DColle[i];
        else return u_i.DColle[i];
    }
    F_3(v) :— { DSYSV(D, &v.DColle[0], &v.DColle[D]); return v;}
Computation for each iteration:
    UpdateVertexU(F_1);
    Push(F_2, +, +);
    UpdateVertexV(F_3);
    UpdateVertexV(F_1);
    Pull(F_2, +, +);
    UpdateVertexU(F_3);
```

4 CUBE

To implement UPPS model, we build a new graph computing engine, CUBE. It is written in C++ and based on MPICH2.

4.1 Graph Loading and Partitioning

In CUBE, each node starts by loading a separate subset of the graph. The 3D partitioning algorithm in CUBE consists of a 2D partitioning algorithm $\mathcal{P}$ and a layer count L, in which L is assigned by users. Thus, after loading, the 2D partitioner $\mathcal{P}$ is used to calculate an assignment of edges (i.e., E_j defined in Section 3.2); and, similarly, a random partitioner is used to partitioning the vertices (i.e., V_j). With these assignments, a global shuffling phase is followed to dispatch the loaded data to where they should be according to the partition policy given in Section 3.2. After shuffling, each $Node_{i,j}$ contains a copy of $D_a[k]$ and $D_{b\to c}[k]$, if vertex $a \in V_j$, edge $(b \to c) \in E_j$, and $LowerBound(i) \le k < LowerBound(i+1)$. Moreover, we use *Hybrid-cut* [10] as the default 2D partitioner and *Bi-cut* [11] is used for bipartite graphs, as they work well on real-world graphs.

4.2 Update

In an *Update*, all the elements of *DColle* properties are needed. To implement this kind of operation, each vertex or edge is assigned a node as the master to perform the *Update*, which needs to gather all the required data before execution. The master node then iterates all data elements it collected, applies the user-defined function and finally scatters the updated values. For bipartite graph oriented operations, *UpdateVertexU* and *UpdateVertexV*, only a subset of vertex data is gathered.

As defined before, E_j and V_j are the subset of edges and vertices in j^{th} partition determined by a 2D partitioning algorithm, and $Node_{*,j}$ is the set of nodes in all layers to process E_j and V_j. In *Update*, each edge or vertex in E_j (or V_j) should have *one* master node $Node_{i,j}$, $i \in [0, L)$ among $Node_{*,j}$ that needs to gather all data elements for the edge or vertex to perform update operation. We define the set of edges or vertices of which the master node is $Node_{i,j}$ as $E_{i,j}$ or $V_{i,j}$. So we have $\bigcup_{i=0}^{L-1} E_{i,j} = E_j$ and $\bigcup_{i=0}^{L-1} V_{i,j} = V_j$. For simplicity, we randomly select a node from $Node_{*,j}$ for each edge and vertex in E_j and V_j. The inter-layer communications are incurred in *Update* by gathering and scattering, which are implemented by two rounds of *AllToAll* communication among the same nodes in different layers (i.e. $Node_{*,j}$).

For certain associative operations (e.g. sum), only the aggregation of the elements in a node is needed. For example, GD algorithm (Algorithm 1) only requires the sum of each node's local *DColle* elements. We allow users to define a **local combiner** for *Update* operations. With the local combiner, each node reduces its local *DColle* elements before sending the single value to its master. Local combiner further reduces communication because the master node only needs to gather one rather than S_C/L elements from each node in all other layers. For operations that can be specified by a custom *MPI_OP*, we leverage the existing *MPI_AllReduce* operation instead of gather and scatter to further reduce network traffic.

4.3 Push, Pull, Sink

A replica for $D_u[i]$ exists at node $Node_{i,j}$ if $\exists v: (u,v) \in E_j$ or $\exists v: (v,u) \in E_j$. The execution of each operation starts with replica synchronization within each layer. It could be implemented by executing L *AllToAll* communications among $Node_{i,*}$ concurrently in each layers.

After synchronization, for *Push* and *Pull*, the user-defined gather function $\mathcal{G}$ is used to calculate the gather result for each vertex; for *Sink*, the user defined function $\mathcal{H}$ is applied to each edge. After that, for *Push* or *Pull*, another L *AllToAll* communications among $Node_{i,*}$ are used to gather the results reduced by the user defined sum function $\oplus$ and then the user defined function $\mathcal{A}$ updates the vertex data. Similar to the *Update*, the sum function $\oplus$ is also used as a local combiner, so that the gather results are locally aggregated before sending. In bipartite mode, only a subset of vertex data is synchronized in *Push* and *Pull*.

Table 2: A collection of real-world graphs.

Dataset	$\|\mathbb{U}\|$	$\|\mathbb{V}\|$	$\|E\|$	Best 2D Partitioner	Description
Libimseti	135,359	168,791	17,359,346	Hybrid-cut	Dating data from libimseti.cz. [7]
Last.fm	359,349	211,067	17,559,530	Bi-cut	Music data from Last.fm. [9]
Netflix	17,770	480,189	100,480,507	Bi-cut	Movie review data from Netflix. [38]

4.4 Matrix-based data structure

Vertex-centric programming models are productive for developing graph programs. But, according to recent investigations, the performance of a naive vertex-cenric implementation can be $2\times - 6\times$ lower than matrix-based execution engines [19, 34, 37]. Therefore, CUBE uses matrix-based backend data structures.

In CUBE, both edge and vertex data are stored continuously. The edges are modeled as a sparse matrix and stored in coordinate list (COO) format, which is a list of (source vertex ID, destination vertex ID, edge value) tuples. The vertex data are simply stored in a continuous array. Since each worker only maintains a subset of graph data, the global ID of its vertices may not be continuous (e.g., vertex C is missing in sub-graph 1 of Figure 4). Thus we need to implement an efficient mechanism for index conversion. Many traditional graph engines use the inefficient hash map based data structure for indexing. Instead, CUBE maps the non-continuous global ID to continuous local ID for each worker. The mechanism is shown in Figure 4. In each worker, the vertex data are stored in a dense vector indexed by its local ID; the row/column ID of its sub-graph is substituted by local ID to ensure quick and straightforward location of the corresponding vertex data for each edge. Moreover, rather than using the simple dictionary order, we sort the edge data in **Hilbert order** [6], which is akin to ordered edges (a,b) by the interleaving of the bits of a and b. It has been shown that Hilbert order exhibits locality in both dimensions rather than one dimension as in the dictionary order, and hence incurs much fewer cache misses. Note that all the mapping and sorting procedures are performed in the initial preparing stage before the following many computing iterations. Therefore, the cost of the preparing procedure is amortized. The system records all data exchanging information at the preparing phase, so there is no need for global/local ID converting during the computation.

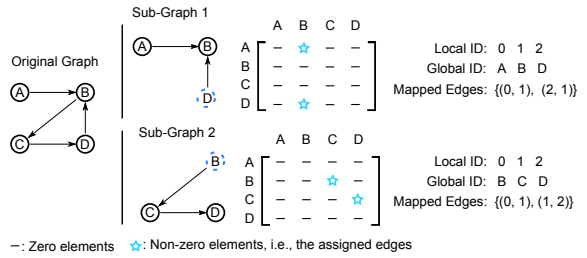


Figure 4: An illustration of the mapping between local and global IDs.

5 Evaluation

This section presents evaluation results of CUBE and compares it with two existing frameworks, PowerGraph [15] and PowerLyra [10]. For each case, we provide: *1)*. Mathematical equations that calculate the communication traffic; *2)*. Experimental performance results that validate the prediction based on communication traffic. To get a thorough understanding of CUBE, we also discuss other aspects such as scalability, memory consumption, partitioning cost, and COST metric.

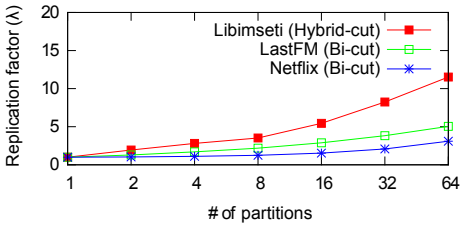
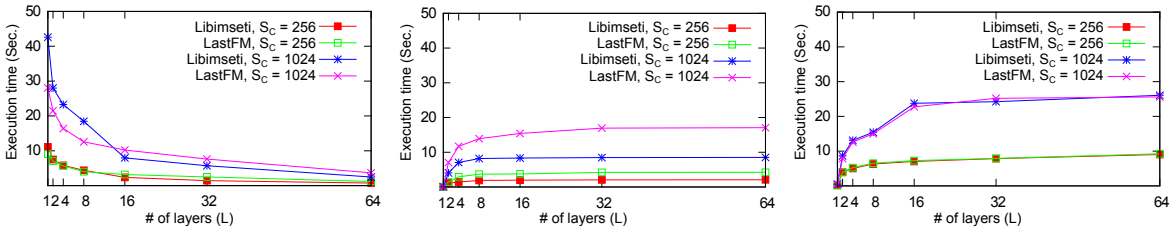


Figure 5: The best replication factor of each dataset.

5.1 Evaluation setup

We conduct the experiments on an 8-node Intel® Xeon® CPU E5-2640 based system. All nodes are connected with a 1Gb Ethernet and each node has 8 cores running at 2.50 GHz. We use a collection of real-world bipartite graphs gathered by the Stanford Network Analysis Project [1]. Table 2 shows the basic characteristics of each dataset.

Since our 3D partitioning algorithm relies on a 2D partitioner within each layer, we first select the *best* 2D partitioner for each dataset. To do so, we evaluated **all** existing 2D partitioning algorithms in PowerGraph and PowerLyra. This includes the heuristic-based Hybrid-cut [10], the bipartite-graph-oriented algorithm Bi-cut [11] and many other random/hash partitioning algorithms. We calculated the average number of replicas for a vertex (i.e., replication factor, λ) for each algorithm. λ includes *both* original vertices and the replicas. We consider the best partitioner as the one that has the smallest λ. To capture the number of partitions, we use λ_x to denote the average number of replicas for a vertex when a graph is partitioned into x sub-graphs (e.g., $\lambda_1 = 1$). Table 2 also shows the best 2D partitioner for each data set: Hybrid-cut is the best for Libimseti, while Bi-cut is the best for LastFM and Netflix. For LastFM, source set (i.e., $\mathbb{U}$) should be used as the favorite subset, while for Netflix, target set (i.e., $\mathbb{V}$) should be used as the favorite subset. Here, "favorite subset" is an input parameter defined by Bi-cut that usually should be set to the larger vertex set of the bipartite graph.



(a) SpMM (b) SumV (c) SumE

Figure 6: The impact of layer count on average execution time for running the micro benchmarks with 64 workers.

Figure 5 shows the replication factor of each dataset for the best 2D partitioning algorithm. We see that Bi-cut is effective if the size of two vertex subsets in a bipartite graph is significantly skewed. It is indeed the case for Netflix: the size of its target subset (i.e., $|\mathbb{V}|$) is 27 times more than its source subset (i.e., $|\mathbb{U}|$). Therefore, the replication factor grows moderately with the number of partitions (e.g., λ_{64} of Netflix is only 3.09). On the other side, LastFM is more balanced and its replication factor grows faster. We show in later sections that, the faster the replication factor grows the better speedup our 3D partitioning algorithm can achieve. Therefore, the improvement is the most significant for Libimseti and the least for Netflix.

5.2 Micro Benchmarks

CUBE allows users to specify the layer count L which is a key factor determining the tradeoff between the amount of intra-layer and inter-layer communication. Two extreme values for L are: 1, where the inter-layer communication is zero and 3D partition degenerates to 2D partitioning; and N (the number of workers), where the intra-layer communication is zero. In general, as L becomes larger, the intra-layer communication decreases and inter-layer communication increases.

We present the equations to calculate communication traffic for three micro-benchmarks and show the performance results as L changes from 1 to 64. The reason why we use micro-benchmarks first before discussing full applications is two-fold. First, each micro-benchmark only requires a *single* operation in UPPS so that we can isolate it from other impacts. Second, the equations obtained for each case can be used as building blocks to construct communication traffic equations for real applications. We will show that the performance results can be indeed explained by the traffic equations.

5.2.1 SpMM

The Sparse Matrix to Matrix Multiplication (SpMM) multiplies a dense and small matrix $\mathbf{A}$ (size $D \times H$) with a big but sparse matrix $\mathbf{B}$ (size $H \times W$), where $D \ll H$, $D \ll W$. This computation kernel is prevalently used in many MLDM algorithms. For example, in training phase of some certain kinds of Deep Learning algorithms [14], the big sparse matrix $\mathbf{B}$ is used to represent the network parameters and the small dense $\mathbf{A}$ is a minibatch of training data, in which D is the batch size (usually ranges from 20 to 1000).

In UPPS, this problem could be modeled by a bipartite graph with $|V| = H + W$, where $|\mathbb{U}| = H$ and $|\mathbb{V}| = W$. The non-zero elements in the big sparse matrix are represented by an edge $i \rightarrow j$ (from a vertex in $\mathbb{U}$ to a vertex in $\mathbb{V}$) with $DShare_{i\rightarrow j} = b_{i,j}$ and $DColle_{i\rightarrow j} = NULL$. On the other side, the dense matrix $\mathbf{A}$ is modeled by vertices: the i^{th} column of $\mathbf{A}$ is represented as the *DColle* vector associated with vertex i in $\mathbb{U}$, where $S_C = D$ and $DShare = NULL$. The computation of a SpMM operation is implemented by a single *Push* (or *Pull*) operation.

Figure 6a shows the execution time of SpMM on 64 workers with L from 1 to 64. Since the computation of SpMM is always equally partitioned into each node, the reduction on execution time is mainly caused by the reduction on network traffic. Formally, if a 3D partitioner $(\mathcal{P}, L)$ is used for partitioning the graph into N nodes, a total of $\lambda_{N/L} * |V|$ replicas will be used in each layer. Since the communication of each Push/Pull operation only involves intra-layer communication and only the *DColle* elements of vertices are needed to be synchronized, the total network traffic can be calculated by summing the number of *DColle* elements sent in each layer, which is $(S_C/L) * (\lambda_{N/L} - 1) * |V|$.

For the bipartite graph in SpMM, synchronization is only needed among replicas in the sub-graph where the vertices are updated ($\mathbb{U}$ or $\mathbb{V}$). If SpMM is implemented as a *Push*, the network traffic is $(S_C/L) * (\lambda^{\mathbb{V}}_{N/L} - 1) * |\mathbb{V}|$; if it is implemented as a *Pull*, the network traffic is $(S_C/L) * (\lambda^{\mathbb{U}}_{N/L} - 1) * |\mathbb{U}|$. Here $\lambda^{\mathbb{U}}_{N/L}$ and $\lambda^{\mathbb{V}}_{N/L}$ are replication factor for $\mathbb{U}$ and $\mathbb{V}$, respectively.

As a result, the amount of network traffic in a SpMM operation can be calculated by the following equations, in which S denotes the size of each $DColle_u[i]$. The traffic is doubled because two rounds of communications (gather and scatter) are needed in replica synchronization.

$$\text{Traffic}(\text{SpMM}_{\text{Push}}) = 2 * \text{S} * S_C * (\lambda^{\mathbb{V}}_{N/L} - 1) * |\mathbb{V}| \quad (1)$$

$$\text{Traffic}(\text{SpMM}_{\text{Pull}}) = 2 * \text{S} * S_C * (\lambda^{\mathbb{U}}_{N/L} - 1) * |\mathbb{U}| \quad (2)$$

For a general graph, $|V|$ is the total number of synchronized vertices. Thus, we have:

$$\text{Traffic}(\text{Push/Pull}) = 2 * \text{S} * S_C * (\lambda_{N/L} - 1) * |V| \quad (3)$$

Our results show that, with Hybrid-cut used as $\mathcal{P}$ in partitioning the Libimseti dataset, λ_2 equals to 1.93 and λ_{64} equals to 11.52. Hence $1-(0.93/10.52)=91\%$ of the network traffic is reduced by partitioning the graph into 32 layers (so that in each layer just has 2 partitions) rather than 1. Figure 6a shows that the reduction on network traffic incurs a 7.78× and 7.45× speedup on average execution time when S_C is set to 256 and 1024, respectively.

5.2.2 SumV

In SpMM, the best performance is always achieved by having as many layers as possible (i.e. best L is the number of workers). This is because SpMM incurs only intra-layer communications. In contrast, for operations that require inter-layer communications, the network traffic and execution time will increase with large L. To understand this aspect, we consider a micro benchmark SumV, which computes the sum of all elements in *DColle* vector for each vertex and stores the result in the corresponding *DShare* of each vertex (i.e., $DShare_u := sum(DColle_u)$). SumV can be implemented by a single *UpdateVertex*. As we have mentioned in Section 4.2, a local combiner can be used to reduce the network traffic of SumV. However, this optimization is not used in our experiments since we intend to measure the overhead of general cases.

Figure 6b provides the execution time of SumV on 64 workers with L from 1 to 64. We see that as L increases, the execution time becomes longer, this validates our previous analysis. We also see that the slope of this curve is decreasing when L becomes larger. To explain this phenomenon, we calculate the exact amount of network traffic during the execution of one SumV. Specifically, for enabling an *UpdateVertex* operation, each master node $Node_{i,j}$ needs to gather all elements of *DColle* of v, if $v \in V_{i,j}$. Since $V_{i,j} \subseteq V_j$, the total amount of data that $Node_{i,j}$ should gather is $S_C * |V_{i,j}| - \frac{S_C}{L} * |V_{i,j}| = \frac{L-1}{L} * S_C * |V_{i,j}|$. Then, all master nodes perform the update and scatter a total amount of $(L-1)*|V|$ DShare data. As a result, the total communication cost of a SumV operation is

$$\begin{aligned}\text{Traffic(SumV)} &= \text{Traffic(UpdateVertex)} \\ &= 2*\text{S}*\frac{L-1}{L}*S_C*|V|+\text{S}*(L-1)*|V|\end{aligned} \tag{4}$$

We see that if S_C is large enough, the communication cost will be dominated by the first term, which has an upper bound and the slope of its increase becomes smaller as L becomes larger. Since the execution time is roughly decided by network traffic, we see the very similar trend in Figure 6b.

5.2.3 SumE

SumE is a similar micro benchmark to SumV, it does the same operations for all edges. Figure 6c presents the average execution time for executing a single *UpdateEdge*, which performs the equation "$DShare_{u\to v} := sum(DColle_{u\to v})$". The communication cost of SumE is almost the same as SumV, except that *DColle* of edges rather than vertices are gathered and scattered. As a result, the communication cost of a SumE operation is:

$$\begin{aligned}\text{Traffic(SumE)} &= \text{Traffic(UpdateEdge)} \\ &= 2*\text{S}*\frac{L-1}{L}*S_C*|E|+\text{S}*(L-1)*|E|\end{aligned} \tag{5}$$

As we can infer from the equation, data lines in Figure 6c share the same tendency of the lines in Figure 6b.

5.2.4 Summary

We see from the micro benchmarks that, *Update* becomes slower as L increases while *Push/Pull/Sink* becomes faster. Given a real-world algorithm which uses the basic operations in UPPS as building blocks, programmers should first obtain the replication factor of the graphs and plug it into the equations to estimate the best L that achieves lowest communication cost.

5.3 Real Applications

Besides the micro-benchmarks described above, we also implemented the GD and ALS algorithm that we explained in Section 3.6. ALS involves intra-layer communications due to *Push/Pull* and inter-layer communications due to *UpdateVertex*. GD combines the intra-layer operation *Sink* with the inter-layer operation *UpdateEdge*. The *UpdateEdge* of GD can be optimized by the local combiner while ALS cannot. ALS explores the specialized APIs for bipartite graphs while GD uses the normal ones. As a conclusion, the implementation of these two algorithms covers all common patterns of CUBE, and hence many other algorithms can be considered as some weighted combinations of GD and ALS. For example, the back-propagation algorithm for training neural networks can be implemented by combining an ALS-like round (for calculating the loss function) and a GD-like round (that updates parameters).

In the following sections, we first demonstrate the performance improvements of CUBE over the existing systems PowerGraph and PowerLyra. Then, we present a piecewise breakdown of our performance gain by calculating the network traffic reductions as in Section 5.2.

5.3.1 Implementation

Both PowerGraph and PowerLyra have provided their implementation of GD and ALS, we use *oblivious* [15] for PowerGraph and the corresponding best 2D partitioners (as listed in Table 2) for PowerLyra.

In CUBE, the implementation of GD and ALS are similar to those given in Section 3.6. However, some optimizations for further reducing network traffic are applied. For GD, we enable a local combiner for the *UpdateEdge* operation. For ALS, we merge successive *UpdateVertexU* and *UpdateVertexV* operations into one (e.g., the two *UpdateVertexV* operations at line 3 and line

Table 3: Results on execution time. Each of the cell gives data in the format of "PowerGraph / PowerLyra / CUBE" (in Second/Iteration). The number in parenthesis is the chosen L.

D	# of workers	Libimseti GD	Libimseti ALS
64	8	9.78 / 9.56 / 2.04 (2)	70.8 / 70.4 / 46.7 (8)
	16	8.04 / 8.16 / 1.95 (4)	72.6 / 71.5 / 37.6 (16)
	64	6.82 / 6.89 / 2.59 (4)	87.0 / 86.8 / 28.7 (64)
128	8	14.99 / 14.94 / 3.87 (2)	261 / 258 / 193 (8)
	16	12.81 / 12.91 / 2.62 (4)	270 / 270 / 135 (16)
	64	11.64 / 11.62 / 3.33 (8)	331 / 331 / 109 (64)
D	# of workers	LastFM GD	LastFM ALS
64	8	12.0 / 8.98 / 3.45 (2)	124 / 73.5 / 70.9 (8)
	16	10.5 / 8.22 / 2.59 (2)	128 / 69.5 / 61.6 (16)
	64	10.4 / 9.86 / 2.48 (4)	158 / 111 / 57.6 (64)
128	8	19.0 / 13.8 / 4.74 (2)	465 / 263 / 270 (4)
	16	17.6 / 13.5 / 3.35 (4)	490 / 253 / 200 (16)
	64	18.6 / 17.8 / 3.47 (8)	Failed / Failed / 230 (64)
D	# of workers	Netflix GD	Netflix ALS
64	8	34.4 / 27.7 / 6.03 (1)	256 / 204 / 110 (2)
	16	26.7 / 17.3 / 3.97 (1)	186 / 107 / 60.4 (2)
	64	18.3 / 7.42 / 4.16 (1)	179 / 66.0 / 42.5 (8)
128	8	51.8 / 38.6 / 9.65 (1)	865 / 657 / 463 (1)
	16	41.9 / 23.0 / 6.59 (1)	669 / 340 / 258 (2)
	64	30.6 / 11.3 / 6.55 (2)	Failed / 239 / 118 (8)

4 of Algorithm 2 is actually implemented as one *UpdateVertexV* operation whose input function successively execute F_3 and F_1). The 2D partitioning algorithm $\mathcal{P}$ used for consisting our 3D partitioner is the listed in Table 2, and hence is the same as PowerLyra.

5.3.2 Execution Time

Table 3 shows execution time results. D is the size of the latent dimension, which gives opportunities that were not exploited in previous systems. In general, a higher D produces higher accuracy of prediction with higher both memory consumption and computational cost. We report the execution time of GD and ALS on three datasets (Libimseti, LastFM and Netflix) with three different number of workers (8, 16 and 64). For each case, we conduct the evaluation on three systems: PowerGraph [15], PowerLyra [10] and CUBE, the results are shown in the same order in the table. The number in parenthesis for CUBE indicates the chosen L for the reported execution time, which is the one with best performance. "Failed" means that the execution in this case failed due to exhausted memory.

As a summary of the results, CUBE outperforms PowerLyra by up to 4.7× and 3.1× on the GD and ALS algorithm respectively. The speedup over PowerGraph is even higher (about $7.3\times - 1.5\times$). According to our analysis, the speedup on ALS is mainly caused by the reduction on network traffic, while the speedup on GD is caused by both the reduction on network traffic and the increasing of data locality. This is because that the computation part of the ALS algorithm is dominated by the DSYSV kernel, which is a CPU-bounded algorithm that has an $O(N^3)$ complexity. In contrast, the GD algorithm is mainly memory bandwidth bounded and hence is sensitive to memory locality. Next, we quantitatively discuss the network traffic of the two applications.

5.3.3 Communication Cost

As we have mentioned above, the improvement of CUBE is mainly from two aspects: *1)* reduction on network communications; and *2)* the adoption of a matrix backend. Thus, in order to further understand the performance gain, we performed a detailed analysis on the effect of network reductions, and the rests are resulted from the matrix backend.

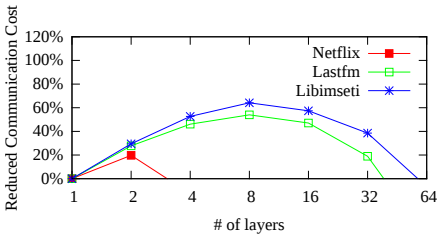


Figure 7: Reduction on GD (64 workers, $D = 128$).

GD The network traffic of a CUBE program can be calculated with the equations given in Section 5.2. But, since a local combiner is used for *UpdateEdge*, its communication cost is only $2 * 8\text{byte} * (L-1) * |E|$[3]. The network traffic for a *Sink* is half of *Push/Pull*. As a result, communication cost of each GD iteration is:

$$\begin{aligned}\text{Traffic(GD)} = {} & (2+2+1) * 8\text{byte} * (\lambda_{N/L} - 1) * S_C * |V| \\ & + 2 * 8\text{byte} * (L-1) * |E|\end{aligned} \quad (6)$$

The reduced network traffic is plotted in Figure 7, which are both the results of mathematical derivation and experimental evaluation. This is because that, as the metadata exchanged by workers account for only a negligible part of the whole communication cost, the measured results are almost identical to the number calculated by formulas. As we can see from the figure, the network traffic reduction for GD is related to replication factor, density of graph (i.e. $|E|/|V|$) and S_C. If the density large enough ($|E|/|V| \gg S_C$), the best choice is to group all nodes into one layer. It happens to be the case for Netflix dataset, which has a density of more than 200. Therefore, the best L is almost always 1 for a small D (except when D=128 and worker count is 64, i.e., the illustrated case in Figure 7). In contrast, for Libimesti, whose density is only 57, our 3D algorithm can reduce about 64% network traffic.

In order to further understand the effectiveness of our 3D partitioner, we have also performed a piecewise

[3]This result is based on Equation 5, in which $S = 8$. The first term is divided by D/L because we use a local combiner, and the second term is zero because *DShare* is *NULL*

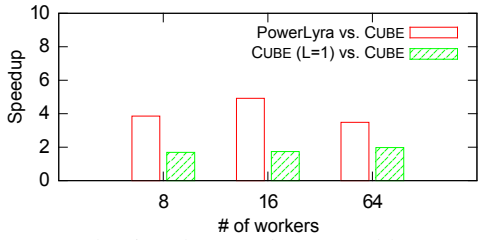


Figure 8: Evaluating the speedup caused by network reduction only for running GD on Libimesti with $D = 128$.

breakdown analysis of the speedup achieved by CUBE. This is possible, because we can estimate the performance improvements gained by 3D partitioning **only** through comparing CUBE with itself that has layer count L be fixed to 1. Figure 8 illustrates the results on Libimesti. As we can see, 3D partitioner accounts for about half of the whole speedup (up to about $2\times$). The results on Lastfm are quite similar to Libimesti but, as we can also infer from Figure 7, most of the speedup for Netflix is resulted from our matrix backend. This is why, in Table 3, the best layer count for running GD on Netflix is usually set to 1. However, if D is set to 2048, even for Netflix, the best L becomes 8 with 64 workers, which achieves a $2.5\times$ speedup compared to $L = 1$.

Moreover, since we use a matrix-based backend that is more efficient than the graph engine used in PowerGraph and PowerLyra, the **total** speedup on memory-bounded algorithms, such as GD, is still up to $4.7\times$. A similar speedup ($1.2\times - 7\times$) is reported by the single-machine graph engine GraphMat [34], which also maps a vertex program to matrix backend.

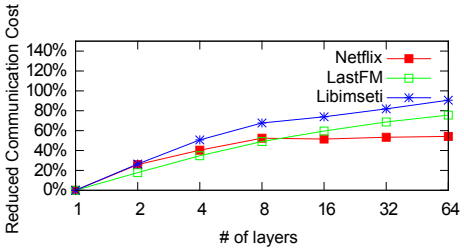


Figure 9: Reduction on ALS (64 workers, $D = 128$).

ALS As discussed in Section 5.3.1, we merged the successive *UpdateVertexU* and *UpdateVertexV* in ALS for reducing synchronizations. After the merge, each iteration of the ALS algorithm only needs to execute each of the four operations (i.e., *UpdateVertexU*, *Push*, *UpdateVertexV* and *Pull*) in bipartite mode once. Thus, based on the estimating formulas given in Section 5.2 (i.e, Equation 1, Equation 2 and Equation 4), the network traffic needed in each iteration is:

$$\begin{aligned}\text{Traffic(ALS)} =& 2 * 8\text{byte} * (\lambda_{N/L} - 1 + \frac{L-1}{L}) * S_C * (|\mathbb{U}| + |\mathbb{V}|) \\ &+ 8\text{byte} * (L-1) * (|\mathbb{U}| + |\mathbb{V}|)\end{aligned} \quad (7)$$

According to Equation 7, we can infer that our 3D partitioner can achieve more significant network traffic reduction on a graph if it is hard to reduce replicas (i.e. λ_N is large). Figure 9 shows the relation between layer count L and the proportion of reduced network traffics when executing ALS with 64 workers and $D = 128$. For example, $\lambda_{64} = 11.52$ for Libimseti (Figure 5), thus network traffic is drastically reduced by 90.6% by partitioning the graph into 64 layers. Table 3 shows that such reduction leads to about $3\times$ speedup on the average execution time. In contrast, the replication factor for the other two datasets is relatively small and hence the speedup is also not as significant as the speedup on Libimseti.

Similar to GD, we have also performed the piecewise breakdown analysis for ALS, the results show that almost **all** ($> 90\%$) of the performance improvements are from 3D partitioning. As we have mentioned in Section 5.3.2, this is because that the computation part of the ALS algorithm is dominated by the DSYSV kernel. DSYSV is a CPU-bounded algorithm that computes the solution to a real system of linear equations, which has an $O(N^3)$ complexity and its state-of-the-art implementation has already efficiently explored its inner-operation locality. As a result, there is not much help of adopting a matrix backend.

5.4 Scalability

For many graph algorithms, the communication cost grows with the number of nodes used. Therefore, the scalability could be limited for those algorithms on small graphs. This is because that the network time may soon dominate the whole execution time, and the reduction of computation time could not offset the increase of network time.

While the potential scalability limitations exist, since our 3D partitioning algorithm reduces the network traffic, CUBE scales better than PowerGraph and PowerLyra. As we can see from Table 3, for Libimseti and LastFM, the execution time of PowerLyra actually *increases* after the number of workers reaches 16, while CUBE with lower network traffic can scale to 64 workers in most cases. Although the scalability of CUBE also becomes limited for more than 16 workers, we believe that it is mainly because that the graph size is not large enough. We expect that for those billion/trillion-edge graphs used in industry [13], our system will be able to scale to hundreds of nodes. To partially validate our hypothesis, we tested CUBE on a random generated synthetic graph, which also follows the power law and contains around one billion edges. The results show that CUBE can scale to 128 workers easily (a further $2.2\times$ speedup is achieved with 128 vs. 32 workers.). Moreover, existing techniques [3, 21] that could improve Pregel/PowerGraph's scalability can also be used to improve our system.

5.5 Memory Consumption

Table 3 shows that, L for the best performance of ALS is almost always equal to the number of workers on Libimesti and LastFM dataset. However, L affects the total memory consumption in different ways. On one side, when L increases, the size of memory for replicas of *DColle* is reduced by the partition of property vector. On the other side, the memory consumption could increase because *DShare* needs to be replicated on each layer. Specifically, since each edge has *DShare* data with type *double* in our case, the total memory needed in ALS is $(\lambda_{N/L} * S_C * |V| + L * |E|) * 8$ bytes, where $S_C = D^2 + D$. For example, Figure 10 shows the total memory consumption (the sum of the memory needed on all nodes) with different L when running ALS on Libmesti with 64 workers.

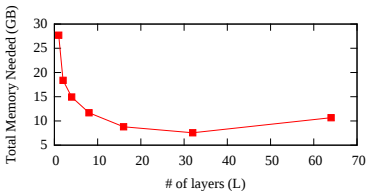


Figure 10: Total memory needed for running ALS with 64 workers and $D = 32$, $S_C = 1056$.

We see that the total memory consumption first decreases, but after a point (roughly $L = 32$) it slightly increases. The memory consumption with $L = 64$ is larger than $L = 32$, because the reduction on replicas of *DColle* data cannot offset the increase of shared *DShare* data. Therefore, $L = 64$ is the parameter for the best performance at the cost of a slightly increased memory consumption. Nevertheless, we see that the total memory consumption at $L = 1$ is *much* larger than cases when $L > 1$. Therefore, CUBE using a 3D partitioning algorithm usually consume less memory than PowerGraph and PowerLyra.

5.6 Partitioning Time

Some works [18] indicated that intelligent graph partitioning algorithms may have a dominating time and hence actually increase the total execution time. However, according to Chen et al. [10], this is only partially true for simple heuristic-based partitioning algorithms. As we can deduce from the definitions given in section 3.2, the partitioning complexity of a 3D partitioner is almost the same as the 2D partitioning algorithm. Thus it only trades a negligible growth of graph partitioning time for a notable speedup during graph computation. Moreover, for those sophisticated MLDM applications that CUBE focuses on, the ingress time typically only counts for a small partition of the overall computation time. As a result, we believe that the partitioning time of CUBE is negligible.

Specifically, the whole setup procedure of CUBE can be split into three phases namely loading, assigning and re-dispatching. *1).* In the loading phase, each node reads an exclusive part of graph data, which is the same as most existing systems. *2).* In the assigning phase, the assignment of each edge is calculated by the 2D partitioner $\mathcal{P}$. Since both hybrid-cut and bi-cut can calculate the assignment for each edge independently, this phase is also very fast (at least its cost is not larger than PowerLyra). Finally, *3).* in the re-dispatching phase, each node sends its loaded data to other nodes if it is necessary (according to the data partition policy detailed in Section 3.2). Typically, the cost of sending edge data is proportional to L, while the cost of sending vertex data is negatively related to L as there are fewer replicas. If there are initial data for vertexes, the total sending cost is approximately equal to the total memory consumption. As illustrated by Figure 10, this means that setting $L > 1$ may actually reduce the cost. In contrast, if vertexes data are randomly initialized, we do have a larger cost with larger L. But, as mentioned in above, typically this cost will not exceed the communication cost of one computing iteration, and hence is acceptable.

5.7 Discussion

COST A recent study [25] shows that some distributed systems may only scale well when its single-threaded implementation has a high cost. The paper proposes a new metric COST (i.e. **C**onfiguration that **O**utperforms a **S**ingle **T**hread) to capture this type of inefficiency. If the COST of a system is c, it means that it takes c workers for this system to outperform a single-threaded implementation of the same algorithm. We also conduct COST analysis for CUBE. To do so, we built single-threaded implementations of both GD and ALS, which are just straightforward transformations of the algorithms described in Section 3.6 to BLAS operators. Based on them, we evaluate the COST of CUBE and find that it is only up to 4, which is moderate. In comparison, McSherry et al. [25] indicates that the data-parallel systems reported in recent SOSP and OSDI either have "a surprisingly *large COST, often hundreds of cores*, or simply *underperform one thread* for all of their reported configurations". Our results align with recent investigations [31, 34], which shows that matrix-centric systems usually have a much better COST than vertex-centric systems.

Faster Network The interconnect of our platform is using 1Gb Ethernet, which is a common configuration used in several recent papers [10]. Readers may wonder that whether the speedups presented is reproducible on a faster experimental setup, which is becoming more and more popular. However, according to our evaluation, when we use 10Gb network, the execution time of PowerGraph/PowerLyra is only reduced by up to 30%, such

reduction is smaller than our *MPI_Alltoallv* based system. For example, when running ALS on Netflix dataset with 64 nodes and $D = 128$, the execution time of PowerLyra only reduces from 239s/iter (1Gb) to 187s/iter (10Gb). In contrast, our CUBE is accelerated from 118s/iter (1Gb) to 63.9s/iter (10Gb) (in which the time consumed by *MPI_AllToAllv* is reduced from 73.1s/iter to 17.8s/iter). Although counter-intuitive, it seems that PowerGraph/PowerLyra cannot fully utilize the network optimization and hence the speedup of CUBE over PowerGraph/PowerLyra is even bigger over a 10Gb network.

Impact of Graph Structure As mentioned in Section 5.1, structure of the input graph does have a great impact of the speedup that can be achieved by our 3D partitioning algorithm. Essentially, less skewness in graphs means that there are fewer opportunities for existing 2D partitioner (e.g., hybrid-cut, bi-cut) to explore[4]. Thus, the replica factor λ_x increases more faster with the number of sub-graphs x, which leads to a better speedup of using 3D partitioning. This is the reason that why we call a 2D partitioner "better" and use it in the evaluation if it produces fewer replicas. We want to make sure that the speedup we achieved is not based on a poor $\mathcal{P}$.

Comparison with Other Systems There are currently a variety of graph-parallel systems. Here we only concentrate on comparing with PowerLyra because its partitioning algorithm produces significant fewer replicas than the others and hence it incurs the lowest network traffic. Moreover, according to Satish et al. [31], Giraph and SociaLite [32] is slower than GraphLab, and hence will be much slower than PowerLyra. As for CombBLAS [8], due to the restriction of its programming model, both the ALS and GD algorithm can only be implemented by S_C times of SpMV in ComBLAS [31], which is extremely slow when S_C is large.

Scope of Application In general, our method is applicable to algorithms analyzing relations among divisible properties. The algorithms presented in this paper are only examples but not all we can support. As an illustration, the matrix to matrix multiplication and matrix factorization examples presented above are building blocks of many other MLDM algorithms. Thus, these problems (e.g., neural network training, mini-batched SGD, etc.) can also benefit from our method. Moreover, some algorithms, whose basic version have only indivisible properties, have advanced versions that involve divisible properties (e.g., Topic-sensitive PageRank [17], Multi-source BFS [35], etc), which obviously can also take advantage of a 3D partitioner.

[4]Many state-of-the-art 2D partitioning algorithms take advantage from the fact that most real-world graphs follow power law, hence they may not work well if the data is not that skewed.

6 Other Related Work

Several graph parallel systems [8, 10, 12, 15, 16, 26, 27, 28, 29, 30, 36, 39] have been proposed for processing the large graphs and sparse matrices. Although these systems are different from each other in terms of programming models and backend implementations, our system, CUBE, is fundamentally different from all of them by adopting a novel 3D partitioning strategy. As shown in Section 2, this 3D partitioning reduces network traffic by up to 90.6%. Besides graph partitioning, there are also many algorithms have been proposed for partitioning large matrices [4, 5]. Our 3D partitioning algorithm is inspired by the 2.5D matrix multiplication algorithm presented by Solomonik et al. [33]. However, the 2.5D algorithm is designed for multiplying two dense matrices and hence cannot be used in graph processing. Regarding the backend execution engine, GraphMat [34] provides a vertex programming frontend and maps it to a matrix backend. However, it is based on a single-machine system that cannot scale out by adding more nodes. Our system adopts the same strategy as GraphMat while extending it to a distributed environment.

7 Conclusion

We argue that the popular "task partitioning == graph partitioning" assumption is untrue for many MLDM algorithms and may result in suboptimal performance. For those MLDM algorithms, instead of a single value, a *vector* of data elements is defined as the property for each vertex/edge. We explore this feature and propose a category of *3D partitioning* algorithm that considers the hidden dimension to partition the property vector to different nodes. Based on 3D partitioning, we built CUBE, a new graph computation engine that *1)* adopts the novel 3D partitioning for reducing communication cost; *2)* provides the users with a new vertex-centric programming model UPPS; and *3)* leverages a matrix-based data structure in the backend to achieve high performance. Our evaluation results show that CUBE outperforms the existing 2D and vertex-based frameworks PowerLyra by up to 4.7× (up to 7.3× speedup over PowerGraph).

8 Acknowledgments

We want to thank Chuntao Hong and Pin Gao for their advises, all the anonymous reviewers and our shepherd for their detailed and useful reviews. This work is supported by Natural Science Foundation of China (61433008, 61373145, 61572280, 61073011, 61133004, U1435216), National Key Research & Development Program (2016YFB1000500), National Basic Research (973) Program of China (2014CB340402). Spanish Gov. European ERDF under TIN2010-21291-C02-01 and Consolider CSD2007-00050.

References

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

[2] ANDERSON, E., BAI, Z., BISCHOF, C., BLACKFORD, S., DEMMEL, J., DONGARRA, J., DU CROZ, J., GREENBAUM, A., HAMMARLING, S., MCKENNEY, A., AND SORENSEN, D. *LAPACK Users' Guide*, third ed. 1999.

[3] AWARA, K., JAMJOOM, H., AND KANLIS, P. To 4,000 compute nodes and beyond: Network-aware vertex placement in large-scale graph processing systems. SIGCOMM '13, pp. 501–502.

[4] BALLARD, G., BULUC, A., DEMMEL, J., GRIGORI, L., LIPSHITZ, B., SCHWARTZ, O., AND TOLEDO, S. Communication optimal parallel multiplication of sparse random matrices. SPAA '13, pp. 222–231.

[5] BALLARD, G., DEMMEL, J., HOLTZ, O., AND SCHWARTZ, O. Graph expansion and communication costs of fast matrix multiplication: Regular submission. SPAA '11, pp. 1–12.

[6] BENDER, M. A., BRODAL, G. S., FAGERBERG, R., JACOB, R., AND VICARI, E. Optimal Sparse Matrix Dense Vector Multiplication in the I/O-Model. SPAA '07, pp. 61–70.

[7] BROZOVSKY, L., AND PETRICEK, V. Recommender system for online dating service. Znalosti '07.

[8] BULUÇ, A., AND GILBERT, J. R. The combinatorial BLAS: design, implementation, and applications. *IJHPCA 25* (2011), 496–509.

[9] CELMA, O. *Music Recommendation and Discovery in the Long Tail*. Springer, 2010.

[10] CHEN, R., SHI, J., CHEN, Y., AND CHEN, H. Powerlyra: Differentiated graph computation and partitioning on skewed graphs. EuroSys '15, pp. 1:1–1:15.

[11] CHEN, R., SHI, J., ZANG, B., AND GUAN, H. Bipartite-oriented distributed graph partitioning for big learning. APSys '14.

[12] CHENG, R., HONG, J., KYROLA, A., MIAO, Y., WENG, X., WU, M., YANG, F., ZHOU, L., ZHAO, F., AND CHEN, E. Kineograph: Taking the pulse of a fast-changing and connected world. EuroSys '12, pp. 85–98.

[13] CHING, A., EDUNOV, S., KABILJO, M., LOGOTHETIS, D., AND MUTHUKRISHNAN, S. One trillion edges: Graph processing at facebook-scale. *Proc. VLDB Endow. 8*, 12 (2015), 1804–1815.

[14] COATES, A., HUVAL, B., WANG, T., WU, D. J., CATANZARO, B. C., AND NG, A. Y. Deep learning with COTS HPC systems. ICML '13, pp. 1337–1345.

[15] GONZALEZ, J. E., LOW, Y., GU, H., BICKSON, D., AND GUESTRIN, C. Powergraph: Distributed graph-parallel computation on natural graphs. OSDI '12, pp. 17–30.

[16] GONZALEZ, J. E., XIN, R. S., DAVE, A., CRANKSHAW, D., FRANKLIN, M. J., AND STOICA, I. Graphx: Graph processing in a distributed dataflow framework. OSDI '14, pp. 599–613.

[17] HAVELIWALA, T. H. Topic-sensitive pagerank. In *Proceedings of the 11th International Conference on World Wide Web* (New York, NY, USA, 2002), WWW '02, ACM, pp. 517–526.

[18] HOQUE, I., AND GUPTA, I. Lfgraph: Simple and fast distributed graph analytics. In *Proceedings of the First ACM SIGOPS Conference on Timely Results in Operating Systems* (New York, NY, USA, 2013), TRIOS '13, ACM, pp. 9:1–9:17.

[19] HUANG, C.-C., CHEN, Q., WANG, Z., POWER, R., ORTIZ, J., LI, J., AND XIAO, Z. Spartan: A distributed array framework with smart tiling. USENIX ATC '15, pp. 1–15.

[20] JANNACH, D., ZANKER, M., FELFERNIG, A., AND FRIEDRICH, G. *Recommender systems: an introduction*. Cambridge University Press, 2010.

[21] KHAYYAT, Z., AWARA, K., ALONAZI, A., JAMJOOM, H., WILLIAMS, D., AND KALNIS, P. Mizan: A system for dynamic load balancing in large-scale graph processing. EuroSys '13, pp. 169–182.

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

[23] MALEWICZ, G., AUSTERN, M. H., BIK, A. J., DEHNERT, J. C., HORN, I., LEISER, N., AND CZAJKOWSKI, G. Pregel: A system for large-scale graph processing. SIGMOD '10, pp. 135–146.

[24] MCSHERRY, F. Spectral partitioning of random graphs. FOCS '01, pp. 529–.

[25] MCSHERRY, F., ISARD, M., AND MURRAY, D. G. Scalability! But at What Cost? In *Proceedings of the 15th USENIX Conference on Hot Topics in Operating Systems* (Berkeley, CA, USA, 2015), HOTOS'15, USENIX Association, pp. 14–14.

[26] MURRAY, D. G., MCSHERRY, F., ISAACS, R., ISARD, M., BARHAM, P., AND ABADI, M. Naiad: A timely dataflow system. SOSP '13, pp. 439–455.

[27] PRABHAKARAN, V., WU, M., WENG, X., MCSHERRY, F., ZHOU, L., AND HARIDASAN, M. Managing large graphs on multi-cores with graph awareness. USENIX ATC'12, pp. 4–4.

[28] PUNDIR, M., LESLIE, L. M., GUPTA, I., AND CAMPBELL, R. H. Zorro: Zero-cost reactive failure recovery in distributed graph processing. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, SoCC '15, pp. 195–208.

[29] ROY, A., BINDSCHAEDLER, L., MALICEVIC, J., AND ZWAENEPOEL, W. Chaos: Scale-out graph processing from secondary storage. SOSP '15, pp. 410–424.

[30] ROY, A., MIHAILOVIC, I., AND ZWAENEPOEL, W. X-stream: Edge-centric graph processing using streaming partitions. SOSP '13, pp. 472–488.

[31] SATISH, N., SUNDARAM, N., PATWARY, M. M. A., SEO, J., PARK, J., HASSAAN, M. A., SENGUPTA, S., YIN, Z., AND DUBEY, P. Navigating the maze of graph analytics frameworks using massive graph datasets. SIGMOD '14, pp. 979–990.

[32] SEO, J., PARK, J., SHIN, J., AND LAM, M. S. Distributed socialite: A datalog-based language for large-scale graph analysis. *Proc. VLDB Endow. 6*, 14 (2013), 1906–1917.

[33] SOLOMONIK, E., AND DEMMEL, J. Communication-optimal Parallel 2.5D Matrix Multiplication and LU Factorization Algorithms. Euro-Par '11, pp. 90–109.

[34] SUNDARAM, N., SATISH, N., PATWARY, M. M. A., DULLOOR, S. R., ANDERSON, M. J., VADLAMUDI, S. G., DAS, D., AND DUBEY, P. GraphMat: High performance graph analytics made productive. *Proc. VLDB Endow. 8*, 11 (2015), 1214–1225.

[35] THEN, M., KAUFMANN, M., CHIRIGATI, F., HOANG-VU, T.-A., PHAM, K., KEMPER, A., NEUMANN, T., AND VO, H. T. The more the merrier: Efficient multi-source graph traversal. *Proc. VLDB Endow. 8*, 4 (Dec. 2014), 449–460.

[36] WU, M., YANG, F., XUE, J., XIAO, W., MIAO, Y., WEI, L., LIN, H., DAI, Y., AND ZHOU, L. Gram: Scaling graph computation to the trillions. SoCC '15, pp. 408–421.

[37] ZHANG, M., WU, Y., CHEN, K., MA, T., AND ZHENG, W. Measuring and optimizing distributed array programs. *Proc. VLDB Endow. 9*, 12 (Aug. 2016), 912–923.

[38] ZHOU, Y., WILKINSON, D., SCHREIBER, R., AND PAN, R. Large-scale parallel collaborative filtering for the netflix prize. AAIM '08, pp. 337–348.

[39] ZHU, X., CHEN, W., ZHENG, W., AND MA, X. Gemini: A computation-centric distributed graph processing system. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)* (Savannah, GA, Nov. 2016), USENIX Association.