3-D Partitioning for Large-scale Graph Processing

Xue Li, Mingxing Zhang, Kang Chen, Yongwei Wu, *Senior Member, IEEE*, Xuehai Qian, and Weimin Zheng, *Senior Member, IEEE*

Abstract—Disk I/O is the major performance bottleneck of existing out-of-core graph processing systems. We found that the total I/O amount can be reduced by loading more vertices into memory every time. Although task partitioning of a graph processing system is traditionally considered equivalent to the graph partition problem, this assumption is untrue 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. By dividing each vertex into multiple sub-vertices, more vertices can be loaded into memory every time, leading to less amount of disk I/O. To explore this new opportunity, we propose a category of *3-D partitioning* algorithm that considers the hidden dimension to partition the property vector.

The 3-D partitioning algorithm provides a new tradeoff to reduce communication costs, which is adaptive to both distributed and out-of-core scenarios. Based on it, we build a distributed graph processing system CUBE and an out-of-core system SINGLECUBE. Since network traffic is significantly reduced, CUBE outperforms state-of-the-art graph-parallel system PowerLyra by up to $4.7\times$. By largely reducing the disk I/O amount, the performance of SINGLECUBE is significantly better than state-of-the-art out-of-core system GridGraph (up to $4.5\times$).

Index Terms—Graph Processing, Task Partitioning, Distributed Systems, Disk I/O, Big Data.

1 INTRODUCTION

MANY real-world problems, including MLDM problems, can be presented as graph computing tasks. Because the graph sizes are often beyond the memory capacity of a single machine, the graphs must be partitioned to distributed memory or out-of-core storage. As a result, many graph processing systems have emerged in recent years to process large-scale graphs efficiently, which can be mainly divided into two categories: distributed in-memory systems and single-machine out-of-core systems.

In distributed graph processing systems [2], [3], [4], [5], each cluster node only holds a subset of vertices/edges (i.e., a sub-graph/partition). During computation, network communications frequently happen between different nodes to exchange information. Therefore, the task partitioning algorithm plays a pivotal role because the load balancing and network cost are largely determined by it.

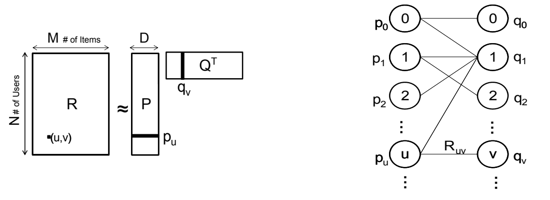
As an alternative to distributed graph processing, single-machine out-of-core systems [6], [7], [8], [9] make large-scale graph processing available on a single machine by using disks efficiently. Because of the limitation of a single machine's memory, in an out-of-core system, only a partition of the data (i.e., a sub-graph/partition) can be loaded into memory and processed every time. Besides, a vertex can update another vertex only when they are both in memory. As a result, some data will inevitably be loaded multiple times to guarantee the correctness of the algorithm. That is to say, information exchange between different partitions is implemented by disk accesses. In fact, in such systems, disk I/O is the major performance bottleneck.

In GraphChi [6], which is the first large-scale out-of-core vertex-centric graph processing system, the whole set of vertices are partitioned into disjoint intervals. It processes an interval at a time and only edges related to vertices in this interval are accessed. GraphChi uses a novel parallel sliding windows method to reduce random I/O accesses, thus provides competitive performance compared to a distributed graph system [6]. X-Stream [8] is a successor system that proposed an edge-centric programming model rather than the vertex-centric model used in GraphChi. Although accesses to vertices are random in X-Stream, edges and updates are accessed sequentially so that maximum throughput can be achieved.

Different from GraphChi/X-Stream, GirdGraph [7] groups edges into a grid representation. Vertices are partitioned into 1-D chunks, and edges are partitioned into 2-D grids. To execute a user-defined function in the edge-centric model, only edges that are related to the specific source and destination vertices are allowed to access. Through a novel dual sliding windows method, GridGraph outperforms other out-of-core systems including GraphChi and X-Stream. It is even competitive with distributed systems [7].

Improving the locality of disk I/O has been the main goal for optimizing these out-of-core systems. However, there is another way to improve overall performance, that is reducing the total I/O amount [9]. For example, in an iteration of GridGraph, only one pass over edge blocks is needed, while vertices are accessed multiple times. Moreover, the more vertices loaded every time, the fewer times

- *X. Li and M. Zhang equally contributed to this work.*
- *An earlier version of this work [1] appeared in OSDI 2016.*
- *Xue Li, Mingxing Zhang, Kang Chen, Yongwei Wu, Weimin Zheng are with the Department of Computer Science and Technology, Beijing National Research Center for Information Science and Technology (BNRist), Tsinghua University, China; Mingxing Zhang is also with Graduate School at Shenzhen, Tsinghua University and Sangfor Inc.; Xuehai Qian is now with the University of Southern California, USA.*



(a) Matrix-based View (b) Graph-based View

Fig. 1: Collaborative Filtering.

each vertex needs to be loaded on average because more vertices can exchange their information in the memory at one time. We will explain this finding furthermore by formulas in the next sections. Therefore, we can reduce the I/O amount by increasing the number of vertices loaded at one time, which is implemented by dividing each vertex to multiple smaller sub-vertices. In fact, although existing graph processing systems differ vastly in their design and implementation, they share a common assumption: the property of each vertex/edge is indivisible, thus task partitioning is equivalent to graph partitioning. In reality, for many MLDM problems, the property associated with a vertex/edge is *not* indivisible but a *vector* of data elements.

This new feature can be illustrated by a popular machine learning problem, Collaborative Filtering (CF), which is used to estimate 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 $\mathbf{R}$ with size $N \times M$, the goal is to find two low-dimensional dense matrices $\mathbf{P}$ (with size $N \times D$) and $\mathbf{Q}$ (with size $M \times D$) that are $\mathbf{R}$'s non-negative factors (i.e., $\mathbf{R} \approx \mathbf{P} \times \mathbf{Q}^{\mathbf{T}}$). Here, N and M are the numbers of users and items, respectively. D is the size of the feature vector. When formulated in a graph-centric view, the rows of $\mathbf{P}$ and $\mathbf{Q}$ correspond to vertices of a bipartite graph with edges between each vertex of $\mathbf{P}$ and each vertex of $\mathbf{Q}$. Each vertex is associated with a property vector with D features. The rating matrix $\mathbf{R}$ corresponds to edge weights. The two views are illustrated in Figure 1. The distinct nature of the graph in Figure 1 (b) is that each vertex is associated with a vector of elements, which is a common pattern when modeling MLDM algorithms as graph computing problems.

In essence, for graph problems that are formulated to solve matrix-based problems, the property of vertex or edge is usually a vector of elements, instead of a single value. During computation, the 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 partitions. Due to the common pattern of vector property, this paper considers a *new dimension* of task partitioning by assigning disjoint elements of the same property to different partitions. It is considered to be a *hidden* dimension in 1-D/2-D partitioners used in previous systems [6], [7], [8] because all of them treat the property as an indivisible component. To the best of our knowledge, we are the first to leverage the *3-D partitioning* by dividing property vectors, in addition to vertices and edges. In the out-of-core graph processing system, by dividing each vertex into L sub-vertices, more vertices can be loaded into memory every time. As a result, the times of repeatedly loading vertex data is reduced. Although this method may increase the times of loading edge data, the programmers can achieve the best performance by carefully choosing the parameter L. Our results show that by 3-D partitioning, the I/O amount reduction can up to 86.5%.

The key intuition of 3-D partitioning is that each partition only holds a subset of elements in property vectors but can be assigned with more vertices/edges that otherwise need to be assigned to different partitions. Therefore, certain communications previously happened between different partitions are converted to local value exchanges, which are much cheaper. On the other side, 3-D partitioning may incur occasional extra synchronizations between sub-vertices/edges. In fact, the 3-D partitioning algorithm is adaptive to both distributed and out-of-core scenarios because it provides a new tradeoff, which can reduce the communication cost between different partitions (network traffic in the distributed scenario or disk I/O in the out-of-core scenario). Based on it, we build a distributed graph processing engine CUBE that introduces significantly less communication than existing distributed systems in many real-world cases. And we also build a new single-machine out-of-core graph processing system SINGLECUBE. By 3-D partitioning, it can largely reduce disk I/O amount, thus achieve better performance than other systems.

In summary, the contributions of this paper are:

- We propose the *first* 3-D graph partitioning algorithm (Section 3) for graph processing systems. It considers a hidden dimension that is ignored by all previous systems, which allows dividing the elements of property vectors. Our 3-D partitioning algorithm can be used in two scenarios: the distributed in-memory scenario and the single-machine out-of-core scenario. In both scenarios, it offers unprecedented performance not achievable by traditional graph partitioning strategies.
- We propose a new programming model **UPPS** (Section 4) designed for 3-D 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.
- We build CUBE (Section 5), a distributed graph processing engine that adopts 3-D partitioning and implements the proposed vertex-centric programming model UPPS. The system significantly reduces communication cost and memory consumption.
- We present SINGLECUBE (Section 6), which is a single-machine out-of-core graph processing system based on 3-D partitioning and the UPPS model. By carefully setting the number of layers, SINGLECUBE can largely reduce the amount of disk I/O (up to 86.5%).
- We systematically study the effectiveness of 3-D partitioning (Section 7). The results show that it leads to significantly better performance in both scenarios. Overall, CUBE outperforms state-of-the-art graph-parallel system PowerLyra by up to $4.7\times$ (up to $7.3\times$ speedup against PowerGraph) because of a notable reduction of communication cost. SINGLECUBE outperforms GridGraph by up to $4.5\times$ through reducing total disk I/O.

2 BACKGROUND

Efficient graph processing systems require cautious task partitioning. It plays a pivotal role in both distributed

TABLE 1: Partition algorithms for some systems.

	1-D	2-D	1-D/2-D	3-D
Distributed	[2], [3]	[4], [10], [11]	[5], [12]	CUBE
Out-of-core	[6], [8]	[7]	[9]	SINGLECUBE

and out-of-core systems because the network/disk-I/O cost is largely determined by the partitioning strategy. More specifically, the partitioner of a distributed graph processing system should *1)* ensure the balance of each node's computation load; and *2)* try to minimize the communication cost across multiple nodes. And for the single-machine out-of-core system, the partitioner should *1)* reduce random I/O accesses; and *2)* try to minimize the total disk I/O amount. As the existing schemes assume that the property of each vertex is indivisible, the partitioning of the graph-processing task is considered equivalent to graph partitioning. To solve this problem, there are two kinds of approaches proposed by existing systems: 1-D partitioning and 2-D partitioning. Partition algorithms used by part of existing works and our work are listed in Table 1.

Distributed Systems Some distributed systems such as GraphLab [3] and Pregel [2] adopt a **1-D** partitioning algorithm. It assigns each node/partition 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 1-D partitioner usually leads to considerable skewness [4].

To avoid the drawbacks of 1-D partitioning, distributed systems [4], [10] are based on **2-D** partitioning algorithms, in which the graph is partitioned by edge rather than the vertex. With a 2-D partitioner, the edges of a graph will be equally assigned to each partition. The system will set up the replica of vertices to enable computation, the automatic synchronization of these replicas requires communication. Various heuristics are proposed to reduce the number of replicas to reduce communication costs. For example, PowerLyra [5] uses a hybrid graph partitioning algorithm *Hybrid-cut* that combines 1-D and 2-D partitioning with heuristics. Besides, Gluon [12] is a recent distributed system that supports heterogeneous 1-D/2-D partitioning policies.

Out-of-core Systems As for out-of-core systems, GraphChi [6] is a typical one that adopts 1-D partitioning. Specifically, it divides the whole set of vertices into P intervals and breaks the edge list into P shards, with each shard containing edges with destinations in corresponding intervals. It adopts a vertex-centric processing model and only processes the related sub-graph of an interval at a time. By using a novel parallel sliding windows method, GraphChi requires a smaller number of random I/O accesses and is able to process large-scale graphs in a reasonable time. However, fragmented accesses over several shards are often inevitable in GraphChi, decreasing the usage of disk bandwidth.

GridGraph [7] is an out-of-core system that adopts 2-D partitioning. It uses an edge-centric programming model in which a user-defined function is only allowed to access the data of an edge and the related source and destination vertices. Specifically, in GridGraph, vertices are also partitioned into P 1-D chunks, with each chunk containing vertices within a contiguous range. Edges are partitioned into $P \times P$ 2-D blocks according to the source and destination vertices (the source vertex of an edge determines the row of the block, and the destination vertex determines the column of the block). In each iteration, GridGraph streams every edge block by block and update instantly onto the source or destination vertex. When processing a specific block (e.g., in the i^{th} row and j^{th} column), the i^{th} and j^{th} chunks will be used. By accessing all blocks in a column-oriented or row-oriented way (as Figure 2 shows), in each iteration, edges are accessed once, and source vertex data is read P times while destination vertex data is read and written once.

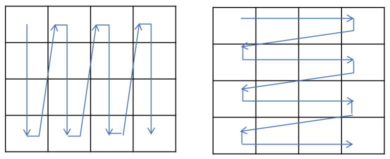


Fig. 2: Access sequence of blocks in GridGraph (P = 4).

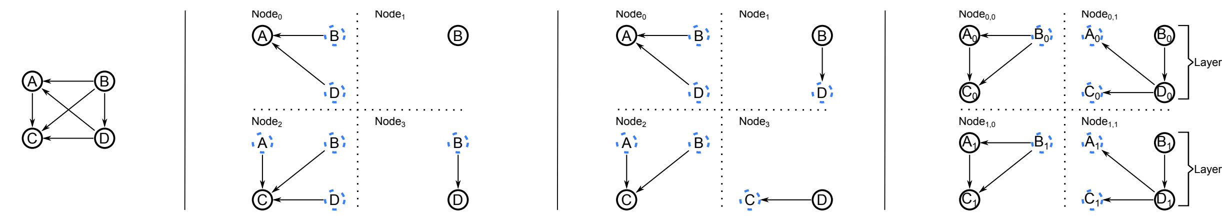
3 3-D PARTITIONING

All of the existing systems, no matter use 1-D or 2-D partitioning, treat the vertex/edge property as an indivisible component and do not assign the same property vector to different partitions. However, in many MLDM problems, a vector of data elements is associated to each vertex or edge and hence the assumption of the indivisible property is untrue. This new dimension for task partitioning naturally leads to a new category of **3-D** partitioning algorithms.

3.1 3-D Partitioning for Distributed Systems

Assuming an N-node cluster, a 3-D partitioner first selects an L, the number of layers, where N is divisible by L. Then the property vector elements associated with the vertices or edges are partitioned across L layers evenly. In this setting, each layer occupies N/L cluster 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 2-D partitioner. Therefore, the i^{th} layer maintains a copy of the graph that comprises all the i^{th} sub-vertices/edges. 3-D partitioning reduces communication cost along edges because by processing only a subset of the original vector, each node in a layer could be assigned to more vertices and edges, therefore, the graph is partitioned across fewer nodes. This essentially converts the otherwise inter-node communication to local data exchanges.

Figure 3 compares the different partition algorithms applied to the graph in Figure 3 (a). In 1-D partitioning (Figure 3 (b)), each cluster node is assigned with one vertex and the incoming edges. There are six replicas in total. In 2-D partitioning (Figure 3 (c)), edges are equally partitioned as much as possible and each node is also assigned with the connected vertices. The number of replicas is also six. Figure 3 (d) illustrates the concepts of 3-D partitioning, where N is 4 and L is 2. First, the total of 4 cluster nodes are divided into 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 2-D partitioning algorithm. Different from 1-D and 2-D partitioning, since the number of cluster nodes for each layer is halved (2 nodes for each layer), each node is assigned with more vertices and edges. In the example, the first node ($Node_{0,0}$ and $Node_{1,0}$) is assigned with 3 edges and 3 connected vertices, in which 1 vertex



(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 as much as possible. (d) 3-D partitioning: each vertex is split into two sub-vertices, and a 2-D partitioner is used for each layer.

A B_0 : 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.

Fig. 3: 1-D, 2-D, and 3-D partitioning in distributed systems.

is a replica. The second node ($Node_{0,1}$ and $Node_{1,1}$) is also assigned with 3 edges and 3 connected vertices but with 2 as replicas. The increased number of vertices and edges in each node (3 edges in Figure 3 (d) compared to 1 or 2 edges in Figure 3 (b),(c)) translates to the reduced number of replicas needed for each layer (3 replicas in Figure 3 (d)) compared to 6 in Figure 3 (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 3-D partitioning algorithm reduces the number of sub-graphs in each layer and hence reduces the **intra-layer** replica synchronization overhead.

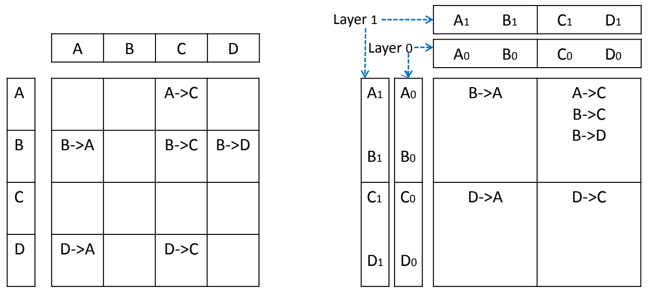
However, 3-D partitioning will incur a new kind of synchronization not needed before: the **inter-layer** synchronization between sub-vertices/edges. Therefore, programmers should carefully choose the number of layers to achieve the best performance. The traditional 1-D and 2-D partitioning do not allow programmers to explore this tradeoff. A detailed discussion of this tradeoff is given in Section 7.

3.2 3-D Partitioning for Out-of-Core Systems

Similar to distributed systems, existing out-of-core graph processing systems also assume that the property of each vertex is indivisible. This assumption is insignificant in 1-D partitioning because GraphChi reads every vertex only once for each iteration, no matter how many intervals that vertices are divided into. However, in GridGraph that uses 2-D partitioning, source vertex data will be read P times in an iteration if vertices are divided into P intervals. Thus a smaller P should be preferred to minimize the I/O amount. In fact, the smallest value of P can be calculated with the memory limit M. Although GridGraph can stream the edges during execution, it needs to cache vertices of the i^{th} and j^{th} intervals in memory when processing $grid[i][j]$ (the edge block in the i^{th} row and j^{th} column). Suppose the graph contains $|V|$ vertices and $|E|$ edges, and the size for every vertex is S_V while the size for every edge is S_E. In order to work, there needs to be $M \geq 2 * S_V * |V|/P$ because two intervals should be held in memory. That is to say, the smallest P is $2*\lceil S_V*|V|/M\rceil$. Then we give the I/O analysis of GridGraph using the method provided in [7].

Assuming edge blocks are accessed in the column-oriented order (as shown by the left figure in Figure 2), in each iteration, edges are accessed once, and source vertex data is read P times while destination vertex data is read and written once. Thus we can calculate the total disk I/O

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



(a) 2-D partitioning: edges are partitioned into 4×4 grids (P = 4). (b) 3-D partitioning: each vertex is split into two sub-vertices, and edges are partitioned into 2×2 grids (P = 2, L = 2).

Fig. 4: 2-D and 3-D partitioning in out-of-core systems.

amount for an iteration, which is $S_E*|E|+(P+2)*S_V*|V|$. Given the memory limit M, we can get the minimum value:

$$\text{Traffic(M)} = S_E * |E| + (2 * \lceil S_V * |V|/M\rceil + 2) * S_V * |V| \quad (1)$$

As we have mentioned, in many MLDM problems, the vertex property is a vector of data elements and hence can be divided. In out-of-core systems, every vertex can also be divided into L sub-vertices evenly, whose size is $\lceil S_V/L\rceil$ at most. Since the property vectors are mostly manipulated by element-wise operators, sub-vertices with the same elements are positioned in the same layer. That is to say, We divide the whole $|V|$ vertices into L layers, with all the i^{th} sub-vertices in the i^{th} layer. As a result, the smallest value of P will be $2*\lceil\lceil S_V/L\rceil * |V|/M\rceil$. Figure 4 illustrates the 2-D partitioning and 3-D partitioning algorithms applied to the sample graph in Figure 3 (a). In 2-D partitioning (Figure 4 (a)), vertices are partitioned into 4 chunks ($P = 4$), and edges are partitioned into 4×4 blocks. In 3-D partitioning (Figure 4 (b)), vertices are divided into 2 layers ($L = 2$). As a result, the new value of P will be 2 to maintain the same memory consumed. Because every vertex is divided into 2 sub-vertices, in each layer, half of the total vertex data is contained. At the same time, all edge data is needed during computation for every layer.

To implement an element-wise operator, all layers will be processed one by one, with each layer comprising of corresponding sub-vertices as well as all of the edges. Although all sub-vertices are still read P times as source vertex data for each iteration, since P is reduced, the total read amount for vertex data is reduced. However, 3-D partitioning will incur another overhead. For calculating each layer, the edge data will be accessed once, i.e., edges are read L times totally instead of only once. Formally, given L and the memory limit M, the minimum I/O amount for an iteration is:

$$\text{Traffic(M)} = L * S_E * |E| + (2 * \lceil\lceil S_V/L\rceil * |V|/M\rceil + 2) * S_V * |V| \quad (2)$$

TABLE 2: 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** $DColle_{u\to v}^{new}[i] := \mathcal{H}(D_u[i],\ D_v[i],\ Du{\to}v[i])$;			

Obviously, the first part of this formula is in proportion to L, while the second part is in negative correlation with L. For many real-world MLDM algorithms, S_V is far larger than S_E, thus increasing L will reduce the total I/O amount and lead to better performance. We should also note that this assumption is not true for some other applications such as PageRank and BFS (where S_V is the size of a single value and can not be divided). As a result, the layer count needs to be set as one, and then the disk I/O amount is completely as same as that of GridGraph. In other words, the 2-D partitioning strategy adopted by GridGraph is a special case of our 3-D partitioning. In general, the programmers should carefully choose the number of layers to achieve the best performance, just as in the distributed scenario. A detailed discussion of this tradeoff is presented in Section 7.

4 UPPS

Graph-oriented programming models of existing works are designed for 1-D/2-D partitioning, thus insufficient for 3-D partitioning because it is assumed that all elements of property vector are accessed as an indivisible component. Therefore, we propose a new model, **UPPS** (**U**pdate, **P**ush, **P**ull, **S**ink) that accommodates 3-D partitioning requirements. In this section, we first introduce UPPS in the distributed scenario. We will describe the operations of UPPS and showcase their usages with two examples. The UPPS model for out-of-core systems is a simplified version of that described in this section and will be discussed in Section 6.

4.1 Data

UPPS is a vertex-centric model. The user defined data D is modeled 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 the 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 2. 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 is set to $NULL$. If $DColle$ of vertices and edges are both enabled, UPPS requires that their length be equal. This restriction avoids inter-layer communication for certain operations (see Section 4.3). It is already the case for graph problems formulated from matrix-based problems. 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 edges. But, for many bipartite graph based MLDM algorithms, only one direction is needed (see more details in Section 4.5).

4.2 Data Partitioning

UPPS allows users to divide each vertex/edge into several sub-vertices/edges so that each of them has a copy of $DShare$ (the indivisible part) and a **disjoint subset** of $DColle$ (the divisible property vector). Based on UPPS, a 3-D 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 2-D partitioning algorithm $\mathcal{P}$. The 3-D partitioner is denoted as $(\mathcal{P}, L)$.

To be specific, we should first guarantee that N is divisible by L. After that, 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 2-D partitioner $\mathcal{P}$; and *3)* randomly separate vertex set V into N/L sub-sets. $Node_{i,j}$ represents the j^{th} node of the i^{th} layer. E_j and V_j denote the j^{th} subset of E and V, respectively. So $Node_{i,j}$ 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 $LowerBound(i) \le k < LowerBound(i + 1)$;

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

In a 3-D 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. The communication cost is purely determined by L. 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 as the 2-D partitioning $\mathcal{P}$. 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. It is the responsibility of programmers to investigate the tradeoff and choose the best setting. To help users choose the appropriate L, we provide the equations to calculate communication costs for different UPPS operations that are used as building blocks for real applications (see Section 7.2). Within a layer, one can choose any 2-D partitioning $\mathcal{P}$ and it is orthogonal to L.

4.3 Computation

There are four types of operations in UPPS (**U**pdate, **P**ush, **P**ull, and **S**ink). The definition of these operations is given in Table 2. All possible variant forms of computations allowed in UPPS are also encoded in these APIs.

Update The *Update* operation takes **all** the information of each vertex/edge to calculate the new value. Roughly, it operates on all elements of an edge or vertex in the *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 All of these three operations handle updates in the *horizontal* direction: the updates follow the dependency relations determined by the 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 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). A similar restriction applies to *Sink*. In other words, $Node_{i,j}$ only communicates to $Node_{i,*}$ in *Push/Pull/Sink*.

4.4 Bipartite Graph

In many MLDM problems, the input graphs are modeled 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 $\mathbb{V}$, respectively. A recent study [13] demonstrates the unique properties of bipartite graphs and the special need for 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 $\mathbb{U}$ or $\mathbb{V}$. We use the bipartite-specialized 2-D partitioner *bi-cut* [13] as $\mathcal{P}$ for bipartite graphs.

4.5 Examples

To demonstrate the usages of UPPS, we implemented two different algorithms that both solve the Collaborative filtering (CF) problem. CF is a kind of problem that estimates the missing ratings based on a given incomplete set of (user, item) ratings. Let N denote the number of users and M denote the number of items, $R = \{R_{u,v}\}_{N\times M}$ is a sparse user-item matrix where each item $R_{u,v}$ represents the rating of item v given from user u. Let P and Q represent the user feature matrix and item feature matrix, respectively. P_u and Q_v are feature vectors with size D that represent the feature of user u and item v. $Err_{u,v}$ represents 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 $\sum_{(u,v)\in R} Err_{u,v}^2$.

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($F_3$, $F_4$, +);
```

GD Gradient Descent (GD) algorithm [14] is a classical solution to solve CF problem, which starts with randomly initializing feature vectors and improving them iteratively. The parameters of it 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, which is almost a straightforward translation of the above equations. Here, + is an abbreviation of the simple "sum" function. We do not show the regularization code for simplicity. In GD, S_C is set to D and the $DShare$ part of each vertex is not used. Each edge e contains the corresponding rating value ($e.DShare.Rate$), the current prediction error ($e.DShare.Err$) and a computation buffer whose length is D ($e.DColle$). Then, the algorithm is implemented by a *Sink* operation, an *UpdateEdge* operation and the last *Pull* and *Push* operation.

ALS Alternating Least Squares (ALS) [15] 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, calculates 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, calculates the best P in a similar way.

Step 4 Repeat Steps 2 and 3 until convergence.

Algorithm 2 Program for ALS.

```
Data:
    $S_C$ :— $D + D * D$
    $DShare_u$ :— NULL;    $DShare_{u \to v}$ :— {double Rate}
    $DColle_u$ :— vector<double>($S_C$);    $DColle_{u \to v}$ :— 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($F_3$);
```

As a typical bipartite algorithm, we implement ALS with the specialized APIs described in Section 4.4. Algorithm 2 presents our program, where the regularization code is also omitted. In ALS, the collection size S_C is set to "$D + D * D$" and contains two parts: *1)* a feature vector Vec with size D for user/item vertex and *2)* a buffer Mat with size $D \times D$ to keep the result of $Vec^T Vec$. Step 2 is implemented as an *UpdateVertexU* to calculate $Vec^T Vec$ and store it in Mat. Then, a *Push* is used to aggregate the corresponding $\sum_{(u,v) \in R} R_{u,v} P_u^T$ (stored in *DColle[0:D-1]*) and $\sum_{(u,v) \in R} P_u^T P_u$ (stored in *DColle[D:D+D*^{2}*-1]*) for each $v \in \mathbb{V}$. Finally, the optimal value of Q_v is calculated by solving a linear equation (calling the DSYSV function in LAPACK [16]). Step 3 is implemented similarly.

5 CUBE

To adopt the UPPS model in the distributed scenario, we build a new distributed graph computing engine CUBE, which is written in C++ and based on MPICH2. For optimizing performance, CUBE uses the matrix-based backend data structures because the matrix-based execution engines can be $2\times - 6\times$ faster than a naive vertex-centric programming model [17], [18], [19]. This strategy is the same with a single-machine system [18] while we use the data structures in a distributed environment. Next, we will describe the pre-processing procedure and the implementation of UPPS in CUBE.

5.1 Pre-processing

At initialization, each node loads a separate part of the graph and the data is re-dispatched by a global shuffling phase. The 3-D partitioning algorithm in CUBE consists of a 2-D partitioning algorithm $\mathcal{P}$ and a user-defined layer count L. Since *Hybrid-cut* [5] works well on real-world graphs, We deploy it as the default 2-D partitioner. And *Bi-cut* [13] is used for bipartite graphs. They are the best 2-D partitioning algorithms for the three representative datasets used in experiments (see more details in Section 7.1). After partition, 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 $LowerBound(i) \le k < LowerBound(i+1)$.

5.2 Implementation

Update In an *Update*, all the elements of $DColle$ properties are needed. 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 subsets of edges and vertices in j^{th} partition determined by a 2-D 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 the 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. The different operations could be specified by *MPI_OP* in the implementation. We leverage the existing *MPI_AllReduce* instead of gather and scatter to further reduce network traffic.

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 layer.

Then, 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*/*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 *Update*, the sum function $\oplus$ is used as a local combiner, thus the gather results are locally aggregated before sending. In the bipartite mode, only a subset of vertex data is synchronized in *Push* and *Pull* ($\mathbb{U}$ for *Push* and $\mathbb{V}$ for *Pull*).

6 SINGLECUBE

To use 3-D partitioning in the out-of-core scenario, we build SINGLECUBE, which is a new single-machine out-of-core graph computing system. In this section, we present its programming model and system implementation. We also use a real application ALS as the example to demonstrate the usages of SINGLECUBE.

TABLE 3: The programming model for SINGLECUBE.

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	$DColle_u[i]$	—	the i^{th} element of $DColle_u$
$D_u[i]$	—	abbreviation of $\{DShare_u, DColle_u[i]\}$	$D_{u\to v}[i]$	—	abbreviation of $\{DShare_{u\to v}\}$

Computation		
UpdateVertex($\mathcal{F}$)	—	**foreach** vertex $u \in V$ **do** $D_u^{new} := \mathcal{F}(D_u)$;
Push($\mathcal{U}$)	—	**foreach** vertex $v \in V$, index $i \in [0, S_C)$ **do** **foreach** $(u, v) \in E$ **do** $D_v[i] := \mathcal{U}(D_u[i],\ D_{u\to v}[i],\ D_v[i])$;
Pull($\mathcal{U}$)	—	**foreach** vertex $u \in V$, index $i \in [0, S_C)$ **do** **foreach** $(u, v) \in E$ **do** $D_u[i] := \mathcal{U}(D_v[i],\ D_{u\to v}[i],\ D_u[i])$;

6.1 Programming model

Existing programming models assume that all elements of the property vector for a specific vertex are indivisible. However, this assumption is not true for 3-D partitioning. As a result, we present the UPPS model that accommodates 3-D partitioning in Section 4. Similarly, in SINGLECUBE, we try to reduce total disk I/O amount by partitioning the vector of data elements associated to each vertex, thus GridGraph's programming model is insufficient for our system. Therefore, we propose a new model for SINGLECUBE.

As shown in Table 3, the programming model of SINGLECUBE is a simpler version of the UPPS model described in Section 4. As for data model, we still model the user defined data as a directed data graph G. The data contains two parts: an indivisible property $DShare$ and a divisible collection of property vector elements $DColle$. However, in SINGLECUBE, only data attached to vertices is partitioned into these two classes, while data attached to edges contains $DShare$ alone. This is because we follow GridGraph's streaming-apply model which stores values on vertices and only requires one (read-only) pass over the edges. Through read-only access to the edges, it can reduce the write amount compared with systems who write values on edges such as GraphChi. That is to say, SINGLECUBE sacrifices the ability to modify edge values for better performance. The same limitation exists in GridGraph. However, GridGraph has proved that the streaming-apply model is workable for most applications since they do not need to modify the edge values. Since modifying the edge data is not allowed, we eliminate the $UpdateEdge$ and $Sink$ functions in the programming model. In addition, since SINGLECUBE is executed in a single-machine environment where properties for both vertices of an edge (u, v) could be accessed immediately, it can operate corresponding vertices directly in $Pull/Push$, rather than using an $update$ for relaying.

6.2 Implementation

Since SINGLECUBE is a single-machine system, the implementation of our programming model is easier and more direct. Specifically, in SINGLECUBE, edges are stored in 2-D grids (edge data files), and sub-vertices of each layer are stored continuously (vertex data files) on disks. To implement $UpdateVertex$, the system only needs to go through all vertices and then write updates back. Therefore, the I/O amount will be $2 * S_V * |V|$. As for the $Push$ operation, the execution procedure is exactly identical as in GridGraph, except that all layers should be processed one by one. In each layer, SINGLECUBE accesses all edge grids in the column-oriented order (the same with GridGraph), and the Update function $\mathcal{U}$ will be executed on every edge. Similar to $Push$, the $Pull$ function also needs to access all grids for each layer. However, in order to ensure that updated vertices are written only once, $Pull$ accesses grids in a row-oriented order instead of the column-oriented order. These two accessing ways are demonstrated in Figure 2. Since $Push$ and $Pull$ are both element-wise operators, the I/O amount of one operation is definitive to be $L * S_E * |E| + (P + 2) * S_V * |V|$, as we have analyzed in Section 3.

Other execution implementations of SINGLECUBE are as same as GridGraph since our system is based on it. To calculate each layer, all edge blocks are streamed one by one. And before processing an edge block, the corresponding source vertex chunk is first loaded into memory. Then, a main thread will continuously push reading and processing tasks to the queue, while other worker threads fetch tasks from the queue, read data from specified location and process each edge. After all edge blocks for a specific destination vertex chunk are processed, updates to those vertices will be written back to the disk. By using this parallel pipeline way, the usage of disk bandwidth is increased.

6.3 Examples

To demonstrate the usages of SINGLECUBE, we use the ALS algorithm as an example. The general description of ALS has been provided in Section 4.5. In fact, the implementation of ALS in SINGLECUBE (shown by Algorithm 3) is similar to the implementation using UPPS in CUBE.

Algorithm 3 Program for ALS in SINGLECUBE.

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

TABLE 4: A collection of real-world graphs.

Dataset	$\lvert\mathbb{U}\rvert$	$\lvert\mathbb{V}\rvert$	$\lvert E\rvert$	Best 2-D Partitioner	Description
Libimseti	135,359	168,791	17,359,346	Hybrid-cut	Dating data from libimseti.cz. [20]
Last.fm	359,349	211,067	17,559,530	Bi-cut	Music data from Last.fm. [21]
Netflix	17,770	480,189	100,480,507	Bi-cut	Movie review data from Netflix. [15]

Besides, because of the simplicity of single-machine operations, the $DColle$ part of edge for saving intermediate results is not necessary. Instead, the results of vector Vec and $Vec^T Vec$ can be added directly on associated vertices to calculate $\sum_{(u,v)\in R} R_{u,v} P_u^T$ and $\sum_{(u,v)\in R} P_u^T P_u$. After that accumulation procedure, there is a simple $UpdateVertexV$ or $UpdateVertexU$ to complete the final update.

7 Evaluation

To study the effectiveness of our 3-D partitioning algorithm, we conduct experiments on both CUBE and SINGLECUBE, and systematically analyze the system performance. In Section 7.2, we analyze the basic operations and the layer count L in the novel UPPS model. In Section 7.3, we present evaluation results of CUBE and compare it with two existing frameworks, PowerGraph and PowerLyra. We compare our work with PowerGraph/PowerLyra because the partitioning algorithms in them produce significant fewer replicas than the others and hence PowerGraph/PowerLyra performs better than other distributed graph processing systems. We also present other aspects of CUBE such as memory consumption and scalability. In Section 7.4, we present the evaluation results of SINGLECUBE and compare it with the state-of-the-art out-of-core system GridGraph. We compare our work with GridGraph because it is reported to outperform other works, including GraphChi and X-Stream. Although Cagra [22] uses a novel technique to improve the cache performance and outperforms GridGraph, it is an in-memory system thus not able to process large-scale graphs. We provide the calculation of total disk I/O and the experimental performance, both of which show that our method is efficient. We also compare SINGLECUBE with CUBE, both similarities and distinctions of them are presented. Besides, to get a thorough understanding of our work, we discuss other aspects in Section 7.5.

7.1 Setup

We conduct the experiments of CUBE on an 8-node Intel® Xeon® CPU E5-2640 based system, while we use a single node for testing SINGLECUBE. 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 [23]. Table 4 shows the basic characteristics of each dataset.

Since in CUBE, our 3-D partitioning algorithm relies on a 2-D partitioner within each layer. We first select the *best* 2-D partitioner for each dataset. To do so, we evaluated **all** existing 2-D partitioning algorithms in PowerGraph and PowerLyra, including the heuristic-based Hybrid-cut [5], the bipartite-graph-oriented algorithm Bi-cut [13] 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 4 also shows the best 2-D partitioner for each data set: Hybrid-cut is the best Libimseti, while Bi-cut is the best for LastFM and Netflix. For LastFM, the source set should be used as the favorite subset, while for Netflix, the target set should be used as the favorite subset in Bi-cut.

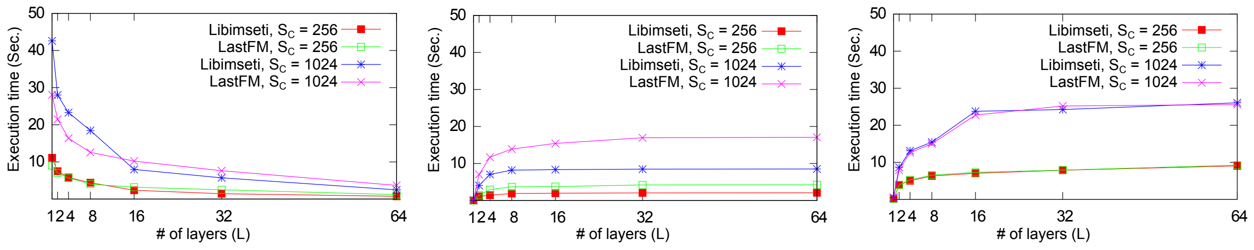
7.2 Basic operations

We use several micro benchmarks to analyze the characteristics of the basic operations of the UPPS model. We also give a guideline to decide the parameter L. As analysis for disk I/O amount of the basic operations in SINGLECUBE has been provided in Section 6.2, we conduct the experiments on CUBE and analyze the network traffic. We use micro-benchmarks first instead of full applications is two-fold: 1) each benchmark only requires a *single* operation in UPPS so that we can isolate it from other impacts; 2) the equations obtained for each case can be used as building blocks to construct communication traffic equations for real applications.

7.2.1 Push/Pull

We use the Sparse Matrix to Matrix Multiplication (SpMM) application to discuss the *Push/Pull* operation since it can be implemented by a single *Push* (or *Pull*) operation. Specifically, the 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, such as in training phase of a Deep Learning algorithm [24]. 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 $DShare = NULL$. Then, the computation of SpMM is implemented by a single *Push* (or *Pull*) operation.

Figure 5 (a) shows the execution time of SpMM on 64 workers with L from 1 to 64. Since different L is based on the same 2-D partitioning $\mathcal{P}$, reduction on execution time is mainly due to the reduction on network traffic. With a 3-D partitioner $(\mathcal{P}, L)$, a total of $\lambda_{N/L} * |V|$ exist in all nodes in a layer. *Push or Pull* only involve intra-layer communication and only *DColle* elements of vertices need to be synchronized. For the general graph, the total network traffic can be calculated by summing the number of *DColle* elements sent in each layer, which is $(S_C/L)*(\lambda_{N/L}-1)*|V|$. The amount of network traffic is the same for *Push* and *Pull*. 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 $(S_C/L)*(\lambda^{\mathbb{U}}_{N/L}-1)*|\mathbb{U}|$. $\lambda^{\mathbb{U}}_{N/L}$ and $\lambda^{\mathbb{V}}_{N/L}$ are replication factor for $\mathbb{U}$ and $\mathbb{V}$, respectively.



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

Fig. 5: The impact of layer count on average execution time for running the micro benchmarks with 64 workers.

Then, we can calculate the amount of network traffic in a SpMM operation by the following equations. 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.

$$\text{Traffic}(\text{SpMM}_{\text{Push}}) = 2 * \text{S} * S_C * (\lambda^{\mathbb{V}}_{N/L} - 1) * |\mathbb{V}| \quad (3)$$

$$\text{Traffic}(\text{SpMM}_{\text{Pull}}) = 2 * \text{S} * S_C * (\lambda^{\mathbb{U}}_{N/L} - 1) * |\mathbb{U}| \quad (4)$$

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

$$\text{Traffic}(\text{Push/Pull}) = 2 * \text{S} * S_C * (\lambda_{N/L} - 1) * |V| \quad (5)$$

For the Libimseti dataset, about 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. And Figure 5 (a) shows that the reduction on network traffic incurs a $7.78\times$ and $7.45\times$ speedup on average execution time when S_C is set to 256 and 1024, respectively.

7.2.2 *UpdateVertex*

For *Push/Pull*, the best performance is always achieved by having as many layers as possible (i.e., L is the number of workers) because it does not incur any inter-layer communication. However, for operations that need all elements in nodes from different layers, 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 of each vertex and stores the result in *DShare* of each vertex (i.e., $DShare_u := sum(DColle_u)$). It can be implemented by a single *UpdateVertex*. since we intend to measure the overhead of general cases.

Figure 5 (b) 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 execution time increase is decreased 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

$$\begin{aligned}\text{Traffic}(\text{SumV}) &= \text{Traffic}(\text{UpdateVertex}) \\ &= 2 * \text{S} * \frac{L-1}{L} * S_C * |V| + \text{S} * (L-1) * |V|\end{aligned} \quad (6)$$

Since the collection size S_C is usually large, 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 a very similar trend in Figure 5 (b).

7.2.3 *UpdateEdge*

To discuss *UpdateEdge*, we implement SumE, which is a micro benchmark similar to SumV. It does the same operations but for all edges. Figure 5 (c) 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. The communication cost is:

$$\begin{aligned}\text{Traffic}(\text{SumE}) &= \text{Traffic}(\text{UpdateEdge}) \\ &= 2 * \text{S} * \frac{L-1}{L} * S_C * |E| + \text{S} * (L-1) * |E|\end{aligned} \quad (7)$$

As a result, data lines in Figure 5 (c) share the same tendency of the lines in Figure 5 (b).

7.2.4 *The Layer Count*

Given a real-world algorithm which uses the basic operations in UPPS as building blocks, programmers could obtain the equations of communication cost and estimate a good layer count L that achieves low cost.

In CUBE, *Update* becomes slower as L increases while *Push/Pull/Sink* becomes faster. Since most applications use two kinds of operations at the same time (such as GD and ALS), L is a key factor determining the tradeoff between the intra-layer and inter-layer communication amount. Two extreme values for L are: 1, where the inter-layer communication is zero and 3-D partitioning degenerates to 2-D partitioning; and N (the number of workers), where the intra-layer communication is zero. Still, it is difficult to get the best L directly because the communication cost of *Push/Pull/Sink* depends on the replica factor λ, which is influenced by the 2-D partitioner. Fortunately, some 2-D partitioning algorithms (e.g., Hybrid-cut [5]) perform a theoretical analysis of the expected λ , which is a function of the number of sub-graphs (i.e., N/L in CUBE) for a given input graph. By taking λ into our communication cost equations, L becomes the single variable, hence it is possible to estimate a good L.

As for SINGLECUBE, the I/O amount of an *UpdateVertex* operation is fixed to be $2 * S_V * |V|$. And the reduction on I/O amount comes from the *Push/Pull* operation, in which the I/O amount is $L * S_E * |E| + (P+2) * S_V * |V|$. Since the memory needed is $M = 2 * \lceil S_V/L \rceil * |V|/P$, we can define $K = P * L$, which is obtained according to the memory

TABLE 5: Results on execution time (in Second). Each of the cell gives data in the format of "PowerGraph / PowerLyra / CUBE". 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)

capacity M. Therefore, the I/O amount of *Push/Pull* is $L * S_E * |E| + (K/L+2) * S_V * |V|$, which is a hyperbolic function of L. Since L is the only variable, it is easy to estimate a best available L after bringing other values into the equation. We further explain this method to decide L through a real application in Section 7.4.2.

7.3 CUBE

To illustrate the efficiency and generality of CUBE, we implemented the GD and ALS algorithm that we explained in Section 4.5. 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. ALS explores the specialized APIs for bipartite graphs while GD uses the normal ones. The implementation of the two algorithms covers all common patterns of CUBE. Also, many other algorithms can be constructed by 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).

Both PowerGraph and PowerLyra have provided their implementation of GD and ALS, we use *oblivious* [4] for PowerGraph. And for PowerLyra, the corresponding best 2-D partitioners (as listed in Table 4) are used, which are the same as in CUBE. For CUBE, the implementation of GD and ALS are given in Section 4.5. The 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. Next, we first demonstrate the performance of CUBE and then present the network traffic calculations.

7.3.1 Overall Performance

Table 5 shows the execution time results. D is the size of the latent dimension that was not exploited in previous systems. We report the execution time of GD and ALS on three datasets (Libimseti, LastFM and Netflix) with three different numbers of workers (8, 16 and 64). For each case, we conduct the execution on three systems: PowerGraph [4], PowerLyra [5] 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 the best performance. "Failed" means that the execution in this case failed due to exhausted memory.

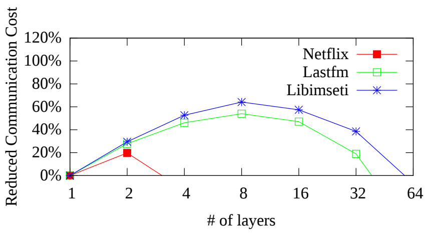


Fig. 6: Reduction on GD (64 workers, $D = 128$).

The results show that CUBE outperforms PowerLyra by up to $4.7\times$ and $3.1\times$ on GD and ALS respectively. The speedup to 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 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, GD is mainly memory bounded and hence is sensitive to memory locality.

7.3.2 GD

The network traffic of GD can be calculated with the equations given in Section 7.2. Since a local combiner is used for *UpdateEdge*, its communication cost is only $2 * 8\text{byte} * (L-1) * |E|$ (in Equation 7, $S = 8$ and divide the first term by D/L, the second term is zero because $DShare$ is $NULL$). The network traffic for a *Sink* is half of *Push/Pull*, so the communication cost of each GD iteration is:

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

The reduced network traffic is plotted in Figure 6. We see that the network traffic reduction 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 will be grouping all the 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. Even for Libimesti of which the density is 57, our 3-D algorithm can reduce about 64% network traffic. However, if D is set to 2048, for Netflix, the best L becomes 8 with 64 workers, achieving 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 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 GraphMat [18], which also maps a vertex program to matrix backend.

7.3.3 ALS

As we have discussed, we merged the successive *UpdateVertexU* and *UpdateVertexV* in ALS for reducing the needed

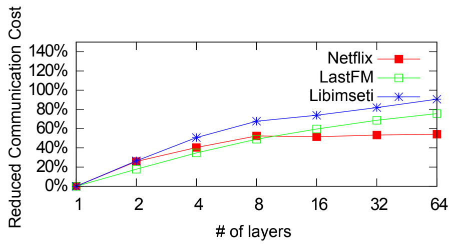


Fig. 7: Reduction on ALS (64 workers, $D = 128$).

synchronizations. After this 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 7.2 (i.e, Equation 3, Equation 4 and Equation 6), the network traffic needed in each iteration is:

$$\begin{aligned}\text{Traffic(ALS)} =& 2 * 8\text{byte} * (\lambda_{N/L} - 1 + \frac{L-1}{L}) * S_C * (|\mathbb{U}| + |\mathbb{V}|) \\ &+ 8\text{byte} * (L-1) * (|\mathbb{U}| + |\mathbb{V}|)\end{aligned} \tag{9}$$

According to Equation 9, our 3-D partitioner can achieve more significant network traffic reduction on a graph if it is hard to reduce replicas (i.e. λ_N is large). For example, Figure 7 shows the relationship between layer count L and the proportion of reduced network traffics when executing ALS with 64 workers and $D = 128$. For Libimseti, $\lambda_{64} = 11.52$, by partitioning the graph into 64 layers, network traffic is drastically reduced by 90.6%. Table 5 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 on Libimseti.

7.3.4 Memory Consumption

L affects 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. In ALS, since each edge has *DShare* data with type *double*, the total memory needed is $(\lambda_{N/L} * S_C * |V| + L * |E|) * 8$ bytes, where $S_C = D^2 + D$. Figure 8 shows the total memory consumption (sum of the memory needed on all nodes) with different L when running ALS on Libmesti with 64 workers.

Figure 8 shows 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. Nevertheless, we see that the total memory consumption at $L = 1$ is *much* larger than cases when $L > 1$. Therefore,

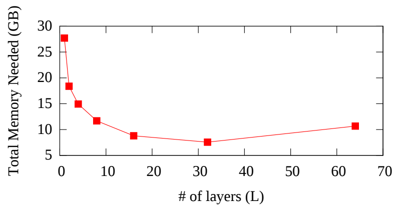


Fig. 8: Total memory needed for running ALS with 64 workers and $D = 32$, $S_C = 1056$.

CUBE using a 3-D partitioning algorithm always consumes less memory than PowerGraph and PowerLyra, who do not support 3-D partitioning.

7.3.5 Scalability

The communication cost of graph algorithms usually grows with the number of nodes used. Because the network time may soon dominate the whole execution time and the reducing of computation time could not offset the increase of network time, the scalability could be limited for those algorithms on small graphs. Although the potential scalability limitations exist, since CUBE reduces the network traffic, it scales better than PowerGraph and PowerLyra.

For example, 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 [25], 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 with around one billion edges. The results show that CUBE can scale to 128 workers easily. Moreover, existing techniques [26], [27] that could improve Pregel/PowerGraph's scalability can also be used to improve our system.

7.4 SINGLECUBE

To evaluate SINGLECUBE, we implement two basic applications SpMM and ALS in SINGLECUBE and evaluate the performance under setting the different number of layers. All of our experiments are conducted on a single node of the cluster introduced in Section 7.1. As we have mentioned, the computation of SpMM could be implemented by a single Push/Pull operation. And the implementation of the ALS algorithm in SINGLECUBE has been discussed in Section 6. We don't provide an implementation of the GD algorithm in SINGLECUBE because our system is built based on GridGraph, which does not support the modification of edges.

In the following sections, we first present the overall performance of SINGLECUBE. After that, for each application, we present the total I/O amount needed and analyze the speedup under the different settings of L. At last, we make a comparison between SINGLECUBE and CUBE.

7.4.1 Overall Performance

Table 6 shows the overall execution time results. We report the execution time of SpMM and ALS on three datasets (Libimseti, LastFM and Netflix) with two different sizes of the latent dimension ($D = 256, 1024$ for SpMM, and $D = 16, 64$ for ALS). We set $K = L * P = 32$ in our experiments, i.e., the initial 2-D partitioning (when $L = 1$) contains 32×32 grids. For each case, we conduct the execution with L varying from 1 to 32 and report the execution time. In fact, since our system is implemented based on GridGraph, when $L = 1$, the result represents the performance of GridGraph. As a summary of the results, SINGLECUBE can outperform GridGraph by up to $4.5\times$ and $3.0\times$ on the SpMM and ALS algorithm, respectively.

TABLE 6: Execution time (in Second) for SINGLECUBE.

SpMM		D = 256			D = 1024		
L	P	Libimseti	LastFM	Netflix	Libimseti	LastFM	Netflix
1	32	3.34	4.27	10.0	9.48	19.7	38.9
2	16	1.73	2.73	7.49	6.46	10.8	27.9
4	8	1.61	2.07	6.40	4.82	7.99	23.1
8	4	1.77	1.75	6.78	3.91	5.22	20.7
16	2	2.14	1.27	7.66	3.25	4.40	19.3
32	1	2.47	2.50	10.6	3.31	5.04	19.9
ALS		D = 16			D = 64		
L	P	Libimseti	LastFM	Netflix	Libimseti	LastFM	Netflix
1	32	4.32	7.32	8.38	84.4	107	136
2	16	3.25	4.96	8.19	56.9	80.6	122
4	8	2.47	3.73	8.06	44.9	60.6	122
8	4	2.06	3.52	9.31	43.8	50.1	124
16	2	2.67	4.38	17.4	37.7	43.0	130
32	1	4.44	6.91	24.3	28.0	37.0	138

7.4.2 SpMM

Since the SpMM application could be implemented by a single Push/Pull operation, we can calculate the amount of total disk I/O using the method presented in Section 6.2. Given P and L, the I/O amount can be calculated by the following formula, where $|V|$ is the total number of synchronized vertices for a general graph:

$$\text{Traffic(P, L)} = L * S_E * |E| + (P+2) * S_V * |V| \qquad (10)$$

The first part of the equation represents that SINGLECUBE reads the edge data ($S_E * |E|$) by L times, while the second part of the equation represents that it repeatedly reads the vertex data ($S_V * |V|$) by $(P+1)$ times and write once.

We present the I/O amount with varying L under different fixed K ($P*L$) in Figure 9. To get close to real results, we set $|V|$ and $|E|$ as the actual size of the Last.fm dataset, thus $|V| = 570416$ and $|E| = 17359346$. At the same time, we also set S_V and S_E as a common configuration when running SpMM, that is $S_E = 8byte$, $S_V = 256 * 4 = 1024byte$. That is to say, the smaller dimension of the smaller matrix in SpMM is 256. From the results we can see that, the total amount of disk I/O for our system will increase with K increases (i.e., the memory needed decreases). In the meantime, by increasing L with fixed K (i.e., fixed memory size M), the amount of disk I/O will first decrease and then increase. Therefore, we can significantly reduce the I/O amount (about 71.5% to 86.5%) by carefully choosing the number of layers. Table 6 shows that this reduction on I/O amount incurs a significant speedup (up to 3.4× and 4.5×) on average execution time when D is set to 256 and 1024, respectively. Typically, the larger D is (means that the larger S_V is), SINGLECUBE can achieve a larger speedup. That is because when D is very large, the second part accounts for the most of Equation 10, thus we can reduce I/O amount by increasing L and decreasing P.

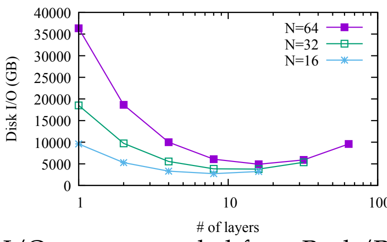


Fig. 9: Total I/O amount needed for a Push/Pull operation for calculating SpMM on Last.fm with $S_C = 256$.

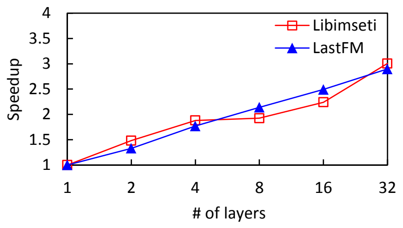


Fig. 10: Speedup on ALS ($D = 64$).

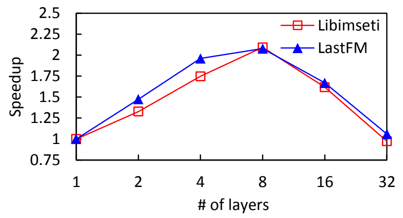


Fig. 11: Speeup on ALS ($D = 16$).

7.4.3 ALS

The implementation of ALS has been presented in Section 6. Using the same method as SpMM, we can estimate the amount of total disk I/O by the following formula:

$$\begin{aligned}\text{Traffic(P, L)} = 2 * L * S_E * |E| + (P+2) * S_V * (|\mathbb{U}| + |\mathbb{V}|) \\ +2 * S_V * (|\mathbb{U}| + |\mathbb{V}|)\end{aligned} \qquad (11)$$

The first two parts of the equation are caused by the *Pull* and *Push* operation, which are the same as Equation 10 and have been discussed. And the third part of the equation represents the *UpdateVertexU*/*UpdateVertexV* operation, which needs to read all operated vertices and then write them back. According to Equation 11, the I/O amount of an *UpdateVertexU*/*UpdateVertexV* operation is fixed, no matter which L we set. As a result, the reduction on I/O amount for ALS also comes from the *Pull*/*Push* operation, which is as same as SpMM. Table 6 shows that such reduction leads to about 2.0× ∼ 3.0× speedup on the average execution time for calculating ALS on Libimseti and LastFM. However, our 3-D partitioning method brings little optimization on Netflix dataset because the density of Netflix (i.e., |E|/|V|) is more than 200, which is much larger than other datasets, hence the reduction in reading vertex data could not cover the overhead of repeatedly reading edge data. In this situation, we can set L as a small number (even as one).

Figure 10 shows the impact of layer count on speedup for running ALS on Libimseti and LastFM with $D = 64$ (i.e., $S_C = 4160$). The initial 2-D partitioning contains 32 × 32 grids and $K = P * L = 32$, thus L varies from 1 to 32. From this figure, we can see that the speedup increases with L increases, up to about 3.0× on both datasets, which shows that our method is very efficient. However, a larger number of layers does not lead to a larger speedup for every case. For example, Figure 11 shows another situation that $D = 16$ (i.e., $S_C = 272$). The performance for both datasets is best when $L = 8$, about 2.0× faster than the baseline. This is because that, the frequency of reading edge data increases with L increases, while the frequency of reading vertex data decreases, leading to a trade-off. Compared with Figure 10, Figure 11 adopts a smaller D, thus the I/O amount for accessing vertices accounts for a smaller part of total disk I/O amount. In a word, users of SINGLECUBE need to carefully set L to get the best performance.

TABLE 7: Comparisons between CUBE and SINGLECUBE.

	CUBE	SINGLECUBE
Supporting 3-D partitioning	✓	✓
Supporting vertex-centric programming	✓	✓
Supporting distributed environment	✓	×
Supporting using disks	×	✓
Supporting modifying edge values	✓	×

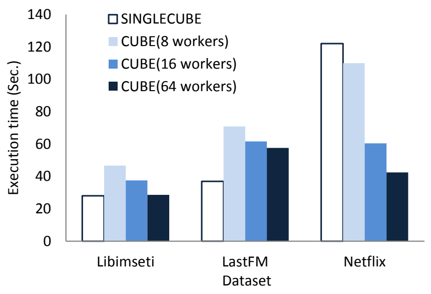


Fig. 12: Execution time for running ALS (D = 64).

7.4.4 *Compare with* CUBE

CUBE and SINGLECUBE are both based on our 3-D partitioning algorithm and the novel programming model UPPS. Still, these two systems have some differences. Similarities and distinctions between them are listed in Table 7.

CUBE is a distributed in-memory graph processing system. Since it holds all the graph data in memory, it needs the computing cluster to have enough nodes to process a large-scale graph. On the contrary, SINGLECUBE is a single-machine out-of-core system, which can largely eliminate the challenges of using a distributed framework and is much easier to use. The developer could decide how much data is loaded into memory every time according to the memory capacity by setting the parameters L and P. Because the disk capacity is usually far larger than that of memory, SINGLECUBE makes practical large-scale graph processing available to anyone with a single PC. To be specific, the total memory size required by CUBE can be calculated using the method that we have presented in Section 7.3.4. And SINGLECUBE requires the disk capacity of the machine to be larger than the graph data size.

However, SINGLECUBE has a restriction compared to CUBE. Since it is implemented based on the system GridGraph, which does not support the modification of edges, it also does not allow users to modify the edge data. As a result, the $UpdateEdge$ operation and the $Sink$ operation are eliminated in SINGLECUBE. Applications that require modification for edge values are not able to be implemented in SINGLECUBE, such as the GD algorithm.

As for system performance, we compare the execution time for running ALS ($D = 64$) of CUBE and SINGLECUBE. The evaluation results are illustrated in Figure 12, and every value in the figure represents the best system performance under setting different values for L. In general, the performance of SINGLECUBE is competitive and even better than CUBE when the cluster size of CUBE is small (e.g., 8 workers). Besides, because of the good scalability of CUBE, the execution time of it may decrease with more nodes contained in the cluster. In fact, according to our test, for those very large graphs used in industry, CUBE will be able to scale to hundreds of nodes. As a result, it usually performs better than SINGLECUBE for large graphs, if there are enough computing resources.

7.5 Discussion

Partitioning Cost Some works (e.g., [28]) indicated that intelligent graph partitioning algorithms might have a dominating time and hence actually increase the total execution time. However, according to [5], this is only partially true for simple heuristic-based partitioning algorithms. The partitioning complexity of a 3D partitioner is almost the same as the 2D partitioning algorithm it used. 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 our work focuses on, the ingress time typically only counts for a small partition of the overall computation time. So we believe that the partitioning time of CUBE/SINGLECUBE is negligible.

Applicability In general, our method is applicable to algorithms that: 1) the property vectors are divisible; and 2) the operators are element-wise (thus the inter-layer communication overhead will not offset its benefits). The algorithms presented in this paper are only examples but not all we can support. As an illustration, the SpMM and matrix factorization examples presented above are building blocks of many other MLDM algorithms. Thus, these problems (e.g., mini-batched SGD) can also benefit from our method. Moreover, some algorithms whose basic version has only indivisible properties (thus do not meet the first requirement), have advanced versions that involve divisible properties (e.g., Topic-sensitive PageRank, Multi-source BFS, etc.),which obviously can take advantage of a 3-D partitioner. The graph neural network (GNN) is also a kind of applications with divisible vertex/edge properties, and is gaining significant increasing popularity in the MLDM community recently. Some works [29] proved that the frequently-used skip-gram model can achieve better performance by partitioning the property vector. As a result, GNN applications that contain this model (including many graph embedding algorithms such as deepwalk and node2vec) can also make use of 3-D partitioning. And for some of the other GNN applications, since the computation is not always element-wise, they can not benefit from the 3-D partitioner.

We follow the popular “think like a vertex” philosophy, thus it is easy to rewrite an algorithm in our model. Besides, when 3-D partitioning is not applicable, our work can be used as a traditional vertex-centric graph system by grouping all data into one layer. Still, users of CUBE can take advantage of our efficient matrix backend. In conclusion, our work provides an alternative to partition a new dimension (which is common in MLDM problems), as well as keeps the ability to implement other algorithms efficiently.

8 OTHER RELATED WORK

There are many distributed ([3], [4], [5], [10], [11], [12], [30], [31], [32], [33], [34], [35]) and single-machine out-of-core ([6], [7], [8], [9], [36], [37], [38], [39]) systems 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 work, is fundamentally different from all of them with a novel 3-D partitioning strategy.

Our 3-D partitioning algorithm is inspired by the 2.5D matrix multiplication algorithm [40], which is designed for multiplying two dense matrices. There are also many other algorithms proposed for partitioning large matrices ([29], [41], [42], [43], [44], [45]), and some of them discuss 3-D parallel algorithms. However, they are all designed for a specific problem or the standard matrix multiplication, and hence cannot be used in other graph applications. Instead, both CUBE and SINGLECUBE are general graph processing systems that provide a vertex-centric programming model. Furthermore, the system SINGLECUBE uses 3-D partitioning in the out-of-core environment to reduce disk I/O. This is a new scenario that is not considered by previous related works because they all focus on distributed computing.

There are also some works provide a simpler alternative to out-of-core graph systems, which are based on data caching mechanisms. For example, by leveraging the well-known memory mapping capability alone, MMap [37] outperforms some graph systems. Importantly, we do not intend to replace existing approaches with 3-D partitioning. Rather, our work can benefit these works, besides general graph processing systems that we focus on in this paper. Specifically, once the property vector is divided into different layers, more vertices/edges can be cached in memory, thus reduce the number of page replacement. In other words, 3-D partitioning still leads to a I/O reduction.

9 CONCLUSION

Disk I/O is the major performance bottleneck of existing out-of-core graph processing systems. And the total I/O amount is largely determined by the partitioning strategy. We found that the popular "task partitioning == graph partitioning" assumption is untrue for many MLDM algorithms and may result in suboptimal performance. We explore this feature and propose a category of *3-D partitioning* algorithm that considers the hidden dimension to partition the property vector. By 3-D partitioning, the I/O amount of the out-of-core system can be largely reduced. In fact, this 3-D partitioning algorithm is adaptive to both distributed and out-of-core scenarios. Based on it, we built a new distributed graph computation engine CUBE and an out-of-core graph processing system SINGLECUBE, both of which can perform significantly better than the state-of-the-art systems.

ACKNOWLEDGEMENT

This work is supported by National Key Research & Development Program of China (2018YFB1003505), Natural Science Foundation of China (61877035, 61433008, 61373145, 61572280), China Postdoctoral Science Foundation (2018M630162). Corresponding Authors: Kang Chen (chenkang@tsinghua.edu.cn) and Yongwei Wu (wuyw@tsinghua.edu.cn).

REFERENCES

[1] M. Zhang, Y. Wu, K. Chen, X. Qian, X. Li, and W. Zheng, "Exploring the hidden dimension in graph processing," in *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, vol. 16, 2016, pp. 285–300.

[2] 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*, 2010, pp. 135–146.

[3] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein, "Distributed graphlab: A framework for machine learning and data mining in the cloud," *Proc. VLDB Endow.*, vol. 5, pp. 716–727, 2012.

[4] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "Powergraph: Distributed graph-parallel computation on natural graphs," in *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, 2012, pp. 17–30.

[5] R. Chen, J. Shi, Y. Chen, B. Zang, H. Guan, and H. Chen, "Powerlyra: Differentiated graph computation and partitioning on skewed graphs," *ACM Transactions on Parallel Computing (TOPC)*, vol. 5, no. 3, pp. 1–39, 2019.

[6] A. Kyrola, G. E. Blelloch, and C. Guestrin, "Graphchi: Large-scale graph computation on just a pc," in *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, 2012, pp. 31–46.

[7] X. Zhu, W. Han, and W. Chen, "Gridgraph: Large-scale graph processing on a single machine using 2-level hierarchical partitioning," in *2015 USENIX Annual Technical Conference*, 2015, pp. 375–386.

[8] A. Roy, I. Mihailovic, and W. Zwaenepoel, "X-stream: Edge-centric graph processing using streaming partitions," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, 2013, pp. 472–488.

[9] Z. Ai, M. Zhang, Y. Wu, X. Qian, K. Chen, and W. Zheng, "Squeezing out all the value of loaded data: An out-of-core graph processing system with reduced disk i/o," in *2017 USENIX Annual Technical Conference*, 2017, pp. 125–137.

[10] A. Buluç and J. R. Gilbert, "The combinatorial BLAS: design, implementation, and applications," *IJHPCA*, vol. 25, pp. 496–509, 2011.

[11] 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 *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, 2014, pp. 599–613.

[12] R. Dathathri, G. Gill, L. Hoang, H.-V. Dang, A. Brooks, N. Dryden, M. Snir, and K. Pingali, "Gluon: A communication-optimizing substrate for distributed heterogeneous graph analytics," in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2018, pp. 752–768.

[13] 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*, 2014.

[14] D. Jannach, M. Zanker, A. Felfernig, and G. Friedrich, *Recommender systems: an introduction*. Cambridge University Press, 2010.

[15] Y. Zhou, D. Wilkinson, R. Schreiber, and R. Pan, "Large-scale parallel collaborative filtering for the netflix prize," in *International Conference on Algorithmic Applications in Management*, 2008, pp. 337–348.

[16] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, *LAPACK Users' Guide*, 3rd ed., 1999.

[17] C.-C. Huang, Q. Chen, Z. Wang, R. Power, J. Ortiz, J. Li, and Z. Xiao, "Spartan: A distributed array framework with smart tiling," in *USENIX Annual Technical Conference*, 2015, pp. 1–15.

[18] N. Sundaram, N. Satish, M. M. A. Patwary, S. R. Dulloor, M. J. Anderson, S. G. Vadlamudi, D. Das, and P. Dubey, "GraphMat: High performance graph analytics made productive," *Proc. VLDB Endow.*, vol. 8, no. 11, pp. 1214–1225, 2015.

[19] M. Zhang, Y. Wu, K. Chen, T. Ma, and W. Zheng, "Measuring and optimizing distributed array programs," *Proceedings of the VLDB Endowment*, vol. 9, no. 12, pp. 912–923, 2016.

[20] L. Brozovsky and V. Petricek, "Recommender system for online dating service," in *Znalosti*, 2007.

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

[22] Y. Zhang, V. Kiriansky, C. Mendis, S. Amarasinghe, and M. Zaharia, "Making caches work for graph analytics," in *2017 IEEE International Conference on Big Data (Big Data)*. IEEE, 2017, pp. 293–302.

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

[24] A. Coates, B. Huval, T. Wang, D. J. Wu, B. C. Catanzaro, and A. Y. Ng, "Deep learning with COTS HPC systems," in *International Conference on Machine Learning*, 2013, pp. 1337–1345.

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

[26] K. Awara, H. Jamjoom, and P. Kanlis, "To 4,000 compute nodes and beyond: Network-aware vertex placement in large-scale graph processing systems," in *ACM SIGCOMM Computer Communication Review*, 2013, pp. 501–502.

[27] Z. Khayyat, K. Awara, A. Alonazi, H. Jamjoom, D. Williams, and P. Kalnis, "Mizan: A system for dynamic load balancing in large-scale graph processing," in *Proceedings of the 8th ACM European Conference on Computer Systems*, 2013, pp. 169–182.

[28] I. Hoque and I. Gupta, "Lfgraph: Simple and fast distributed graph analytics," in *Proceedings of the First ACM SIGOPS Conference on Timely Results in Operating Systems*, ser. TRIOS '13. New York, NY, USA: ACM, 2013, pp. 9:1–9:17. [Online]. Available: http://doi.acm.org/10.1145/2524211.2524218

[29] E. Ordentlich, L. Yang, A. Feng, P. Cnudde, M. Grbovic, N. Djuric, V. Radosavljevic, and G. Owens, "Network-efficient distributed word2vec training system for large vocabularies," in *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management*, 2016, pp. 1139–1148.

[30] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi, "Naiad: A timely dataflow system," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, 2013, pp. 439–455.

[31] M. Wu, F. Yang, J. Xue, W. Xiao, Y. Miao, L. Wei, H. Lin, Y. Dai, and L. Zhou, "Gram: Scaling graph computation to the trillions," in *The annual ACM Symposium on Cloud Computing*, 2015, pp. 408–421.

[32] X. Zhu, W. Chen, W. Zheng, and X. Ma, "Gemini: A computation-centric distributed graph processing system." in *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, 2016, pp. 301–316.

[33] A. Roy, L. Bindschaedler, J. Malicevic, and W. Zwaenepoel, "Chaos: Scale-out graph processing from secondary storage," in *Proceedings of the 25th Symposium on Operating Systems Principles*, 2015, pp. 410–424.

[34] J. Seo, J. Park, J. Shin, and M. S. Lam, "Distributed socialite: A datalog-based language for large-scale graph analysis," *Proc. VLDB Endow.*, vol. 6, no. 14, pp. 1906–1917, 2013.

[35] A. Corbellini, D. Godoy, C. Mateos, S. Schiaffino, and A. Zunino, "Dpm: A novel distributed large-scale social graph processing framework for link prediction algorithms," *Future Generation Computer Systems*, vol. 78, pp. 474–480, 2018.

[36] V. Prabhakaran, M. Wu, X. Weng, F. McSherry, L. Zhou, and M. Haridasan, "Managing large graphs on multi-cores with graph awareness," in *USENIX Annual Technical Conference*, 2012, pp. 4–4.

[37] Z. Lin, M. Kahng, K. M. Sabrin, D. H. P. Chau, H. Lee, and U. Kang, "Mmap: Fast billion-scale graph computation on a pc via memory mapping," in *2014 IEEE International Conference on Big Data*, 2014, pp. 159–164.

[38] D. M. Da Zheng, R. Burns, J. Vogelstein, C. E. Priebe, and A. S. Szalay, "Flashgraph: Processing billion-node graphs on an array of commodity ssds," in *Proceedings of the 13th USENIX Conference on File and Storage Technologies*, 2015, pp. 45–58.

[39] W.-S. Han, S. Lee, K. Park, J.-H. Lee, M.-S. Kim, J. Kim, and H. Yu, "Turbograph: a fast parallel graph engine handling billion-scale graphs in a single pc," in *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*, 2013, pp. 77–85.

[40] E. Solomonik and J. Demmel, "Communication-optimal Parallel 2.5D Matrix Multiplication and LU Factorization Algorithms," in *European Conference on Parallel Processing*, 2011, pp. 90–109.

[41] G. Ballard, A. Buluc, J. Demmel, L. Grigori, B. Lipshitz, O. Schwartz, and S. Toledo, "Communication optimal parallel multiplication of sparse random matrices," in *Proceedings of the twenty-fifth annual ACM symposium on Parallelism in algorithms and architectures*, 2013, pp. 222–231.

[42] G. Ballard, J. Demmel, O. Holtz, and O. Schwartz, "Graph expansion and communication costs of fast matrix multiplication: Regular submission," in *Proceedings of the twenty-fifth annual ACM symposium on Parallelism in algorithms and architectures*, 2011, pp. 1–12.

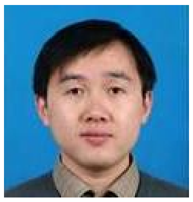
[43] P. Koanantakool, A. Azad, A. Buluç, D. Morozov, S.-Y. Oh, L. Oliker, and K. Yelick, "Communication-avoiding parallel sparse-dense matrix-matrix multiplication," in *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2016, pp. 842–853.

[44] E. Solomonik, M. Besta, F. Vella, and T. Hoefler, "Scaling betweenness centrality using communication-efficient sparse matrix multiplication," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2017, pp. 47:1–47:14.

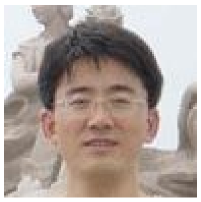
[45] A. Azad, G. Ballard, A. Buluc, J. Demmel, L. Grigori, O. Schwartz, S. Toledo, and S. Williams, "Exploiting multiple levels of parallelism in sparse matrix-matrix multiplication," *SIAM Journal on Scientific Computing*, vol. 38, no. 6, pp. C624–C651, 2016.

Xue Li is a PhD student in Department of Computer Science and Technology, Tsinghua University, China. Her research interests include graph processing and distributed systems. She received her B.E. degree from Beijing University of Posts and Telecommunications, China, in 2014. She can be reached at: lixue14@mails.tsinghua.edu.cn.

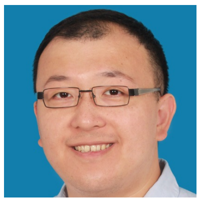
Mingxing Zhang received the PhD degree in computer science and technology from Tsinghua University, Beijing, China in 2017. His research interests include parallel and distributed systems. He received his B.E. degree from Beijing University of Posts and Telecommunications, China, in 2012. He is now with Graduate School at Shenzhen, Tsinghua University, and Sangfor Inc.



Kang Chen received the PhD degree in computer science and technology from Tsinghua University, Beijing, China in 2004. Currently, he is an Associate Professor of computer science and technology at Tsinghua University. His research interests include parallel computing, distributed processing, and cloud computing.



Yongwei Wu received the PhD degree in applied mathematics from the Chinese Academy of Sciences in 2002. He is currently a professor in computer science and technology at Tsinghua University of China. His research interests include parallel and distributed processing, and cloud storage. Dr. Wu has published over 100 research publications and has received four Best Paper Awards. He is an IEEE senior member.



Xuehai Qian received the PhD degree in Computer Science Department from University of Illinois at Urbana-Champaig, USA. Currently, he is an assistant professor at the Ming Hsieh Department of Electrical Engineering and the Department of Computer Science at the University of Southern California. His interests lie in the fields of computer architecture, architectural support for programming productivity and correctness of parallel programs.

Weimin Zheng received the BS and MS degrees, respectively, in 1970 and 1982 from Tsinghua University, China, where he is currently a professor of Computer Science and Technology. He is the research director of the Institute of High Performance Computing at Tsinghua University, and the managing director of the Chinese Computer Society. His research interests include computer architecture, operating system, storage networks, and distributed computing. He is a member of the IEEE.