HyperX: A Scalable Hypergraph Framework

Wenkai Jiang, Jianzhong Qi, Jeffrey Xu Yu, Jin Huang, and Rui Zhang†

Abstract—Hypergraphs are generalizations of graphs where the (hyper)edges can connect any number of vertices. They are powerful tools for representing complex and non-pairwise relationships. However, existing graph computation frameworks cannot accommodate hypergraphs without converting them into graphs, because they do not offer APIs that support (hyper)edges directly. This graph conversion may create excessive replicas and result in very large graphs, causing difficulties in workload balancing. A few tools have been developed for hypergraph partitioning, but they are not general-purpose frameworks for hypergraph processing. In this paper, we propose HyperX, a general-purpose distributed hypergraph processing framework built on top of Spark. HyperX is based on the computation paradigm "Pregel", which is user-friendly and has been widely adopted by popular graph computation frameworks. To help create balanced workloads for distributed hypergraph processing, we further investigate the hypergraph partitioning problem and propose a novel label propagation partitioning (LPP) algorithm. We conduct extensive experiments using both real and synthetic data. The result shows that HyperX achieves an order of magnitude improvement for running hypergraph learning algorithms compared with graph conversion based approaches in terms of running time, network communication costs, and memory consumption. For hypergraph partitioning, LPP outperforms the baseline algorithms significantly in these measures as well.

Index Terms—Hypergraph, HyperX, graph framework, graph partitioning, label propagation partitioning

1 Introduction

USUAL graphs allow each edge to connect two vertices representing a certain relationship between them. For example, in a social network graph, an edges connecting two vertices may represent a "friendship" between two users. In a wide range of applications, a relationship may be formed by more than two objects. For example, a research paper is likely to be coauthored by multiple researchers; a tweet may be reposted by many users. In such applications, modeling objects and their relationships with a usual graph may incur information loss [1]. A common approach to address this problem is representing the objects and their relationships by vertices and *hyperedges* in a *hypergraph*. A hypergraph is a generalized graph where an edge (now called a hyperedge) can connect more than two vertices. Hypergraph models have shown great effectiveness in capturing high-order relationships [2]–[8]. Table 1 summarizes some representative examples of hypergraph applications.

TABLE 1: Hypergraph Applications

Application	Algorithm	Vertex	Hyperedge
Recommendation	[2]	Songs and users	Listening histories
Text retrieval	[3]	Documents	Semantic similarities
Image retrieval	[4]	Images	Descriptor similarities
Multimedia	[5]	Videos	Hyperlinks
Bioinformatics	[6]	Proteins	Interactions
Social mining	[7]	Users	Communities
Machine Learning	[8]	Records	Labels

While applications of hypergraphs are emerging, there has been little work on developing a general-purpose processing framework for hypergraphs. Several tools have been designed for specific hypergraph operations. For example, Parkway [9] focuses on hypergraph partitioning, but it is difficult to be used, for example, to find the shortest path in a hypergraph or to run hypergraph learning algorithms over the partitions obtained. Popular general-purpose graph frameworks such as GraphX [10] and PowerGraph [11], on the other hand, support a large variety of traditional graph analytic tasks, but they do not support hypergraph representations. These graph frameworks cannot process hypergraphs without converting hypergraphs into graphs. Converting a hypergraph into a graph may inflate the size of the original hypergraph, because every hyperedge needs to be replaced by a clique which increases the number of edges and vertices. For example, a hypergraph studied previously [12] with 2 million vertices and 15 million hyperedges is converted to a bipartite with 17 million vertices and 1 billion edges. Such inflation causes huge difficulty in processing the hypergraph. With the rapid growth of hypergraph applications such as those mentioned before, there is an increasing need for a general-purpose hypergraph processing framework that can handle common operations such as traversal on hypergraph directly and efficiently.

In this paper, we address this need by proposing HyperX, a distributed hypergraph processing framework. HyperX is a thin layer built upon Apache Spark [13]. It provides flexible and expressive interfaces for the ease of implementation of hypergraph learning algorithms, operating directly on the hypergraph representation. To ease the use of the framework, HyperX provides a *hyperedge program* and a *vertex program* which are consistent with the *edge program* and *vertex program* used in popular graph frameworks such as GraphX. HyperX uses the Bulk Synchronous Parallel (BSP) message passing scheme, which is commonly used

- *W. Jiang, J. Qi and R. Zhang are with the School of Computing and Information Systems, The University of Melbourne, Australia. E-mail: {jiang.w, jianzhong.qi, rui.zhang}@unimelb.edu.au*
- *J. Yu is with the Department of Systems Engineering and Engineering Management, The Chinese University of Hong Kong, China. E-mail: yu@se.cuhk.edu.hk*
- *J. Huang was previously with the School of Computing and Information Systems, The University of Melbourne, Australia. E-mail: soone@iojin.com*

† *R. Zhang is the corresponding author of this paper.*

in synchronous graph processing frameworks. This allows the ideas of HyperX to be adapted to other distributed frameworks such as Flink [14] instead of being constrained on Apache Spark.

HyperX builds a foundation that supports processing hypergraphs at large scale. When hypergraphs are large, HyperX distributes the computation over across many workers. This calls for a hypergraph partition algorithm to create partitions that can be processed in a distributed manner with a balanced workload and low communication costs among the workers. The efficiency of a hypergraph processing algorithm running on HyperX may be significantly impacted by the hypergraph partitions.

Partitioning problem over both graphs and hypergraphs has been studied for many years. Partitioning a hypergraph is much more challenging than partioning a graph as the ratio of the cardinality of vertices over hyperedges can be vastly skewed, where the number of this ratio is basically constant for a graph. As a result, the computation load associated with *vertex program* and *hyperedge program* is hard to balance. This is also because the partitioning over vertices and hyperedges set up constrains on each other. Delicately design an algorithm to partition hyperedges without considering vertices allocation may lead to severe network communication cost. A good hypergraph partitioning algorithm should make a trade off between balancing the workload and minimizing the communication cost at the same time.

Hypergraph partitioning problem is acknowledged as a NP-hard problem [15], so exact partitioning algorithms are applicable only for small hypergraphs, therefore do not meet our need. All practical methods are heuristics. Their goal is intuitively to divide a hypergraph into a number of equal-sized components, while minimizing the number of components hyperedges span. In this paper, we investigate this problem and add a strong constraint to get better results. During the hypergraph partitioning, we only replicate vertices but not hyperedges. Replicating a hyperedge requires replicating all the vertices it involves. Therefore, doing so substantially reduces the number of replicas. Consequently, the excessive communication cost during the value synchronization among replicas is waived. We propose a novel label propagation partitioning (LPP) algorithm to fulfill this purpose. LPP monitors a load balance factor and network communication costs at the same time. In each iteration, it has two steps: it assigns hyperedges to partitions that can reduce the number of replicas in order to achieve least communication; and it relocates vertices to partitions less full to make partitions more balance. LPP is based on *label propagation*, but differs from the classic label propagation algorithm [16] in that it labels both vertices and hyperedges while updating the labels separately.

In summary, we make the following contributions:

- This work systematically investigate scalable hypergraph processing. Hypergraphs are extensively used as a popular model in optimizing distributed systems, but scaling computation over hypergraphs has not been thoroughly explored.
- We design a general-purpose framework, HyperX, to directly process large hypergraphs in a distributed manner. In HyperX, we provide two primary operations and tackle several implementation issues such as the representation of a distributed hypergraph.
- We implement HyperX and evaluate its performance with extensive experiments on three hypergraph learning algorithms. Compared with the graph conversion approaches implemented on GraphX [10], HyperX saves up to 77% memory usage and up to 98% communication costs, and runs up to 49 times faster.

This article is an extension of our earlier conference paper [17]. In the conference paper, we proposed the HyperX framework and described its implementation. In this extension, we focus on the hypergraph partitioning problem, which is essential to make the framework efficient. We make the following new contributions.

- We investigate factors that may affect the performance of hypergraph applications from a general perspective. We find that the performance can be largely impacted by the number of replicas and the balance level of the partitions (Section 5.1).
- We design a new optimization objective aimed to minimize the number of replicas and to achieve the balanced partitions. We formulate it as a constrained minimization problem (Section 5.2). The hardness of the above minimization problem on distributed environment is discussed (Section 5.3). We show its inapproximability under a strict case, and present a solution with a logarithmic approximation factor under a loose case.
- We propose a novel *label propagation partitioning* (LPP) algorithm (Section 5.4) to achieve the optimization goal in an effective way. LPP is an iterative algorithm that runs in a parallel behaviour. It has linear time complexity in each iteration. LPP relies on a vertex-centric computation paradigm, and it may scale to large hypergraphs with billions of potential relations.
- We explore a number of heuristics to further boost the efficiency of the LPP algorithm (Section 5.5). We conduct extensive experiments on LPP and compare it with the state-of-the-art hMetis, Parkway, and Zoltan over real data (Section 6.3). The results show that LPP generates partitions that speed up distributed hypergraph applications consistently, while it runs faster than the baseline algorithms.

The rest of the article is organized as follows: We summarize related work in Section 2. In Section 3, we show how to process hypergraphs distributively and present the HyperX framework. We then present its implementation in Sections 4. In Section 5, we study hypergraph partitioning. We report the experimental results in Section 6 and conclude the paper in Section 7.

2 Related Work

Distributed graph processing frameworks. Distributed graph processing has been extensively studied [10], [11], [18], [19]. A number of issues that HyperX addresses for hypergraphs have been studied for graphs: replicating data [11], aggregating messages [10], partitioning the data [11], [19], and providing common APIs for a wide range of applications [10], [18]. However, as discussed in Section 3.2, there are major efficiency issues in adopting these graph techniques to process hypergraphs.

Graph and hypergraph partitioning. Classic graph partitioning studies focus on the *minimum bisection paradigm*. It is the most commonly used approach for computing *k-way partitioning*. A more general approach is called the *(k,v)-balanced partitioning* [15] which partition a graph into k sets where each set has at most v times of the average number of vertices. An important variant of the *(k,v)-balanced partitioning* is the balanced edge partitioning where the goal is to partition the edges into two disjoint sets with an equal number of edges, while minimizing the number of vertices spanning over different sets. A recent study [20] shows that such a formulation suits distributed processing better. Works [21], [22] show that finding dominating sets and matching subgraph before partitioning are also useful. There are also greedy heuristics to minimize the marginal cost [11], [20]. However, all these techniques are based on graph models and therefore cannot be applied to hypergraphs without significant modifications. Furthermore, existing studies on graph partitioning usually balance either edges or vertices but not both, which does not satisfy our goal.

Hypergraph partitioning, as a generalization of graph partitioning, is an even more complex problem. Hypergraph partitioning is previously explored in the context of integrated circuit design (VLSI) with a minimum cut-size objective on hyperedges in order to minimize bisections on a printed circuit board. Exact partitioning algorithms are expensive in both computation and storage space usage. A well known exact partitioning algorithm is the *spectral clustering* which is for partitioning bipartites (note that bipartites and hypergraphs are equivalent). It has been shown that a real-value relaxation under the cut criterion leads to the eigen-decomposition of a positive semidefinite matrix [23]. This means that cuts based on the second eigenvector always gives a guaranteed approximation to the optimal cut. A series of techniques based on spectural clustering have been proposed [24]–[26]. However, these techniques are inefficient as the size of a bipartite converted from a hypergraph can be very large. Two popular methods to compute eigen-decomposition are Lanczos [27] and SVD [24]. They both have the time complexity of $O(k(N_x + N_y)^{3/2})$, where N_x and N_y are the number of vertices in each group, respectively. For large hypergraphs where the numbers of vertices and hyperedges are up to 100 millions, spectral clustering may not have satisfactory efficiency.

Heuristic based partitioning algorithms such as hMeTis [28], PaToH [29], Parkway [9], and Zoltan [30] have been developed for a higher partitioning efficiency. The algorithms hMetis and PaToH are single-machine based algorithms, while the rest of the algorithms can run in a distributed manner. All these algorithms share the same multi-level coarsen-uncoarsen technique to partition a hypergraph. This technique coarsens the original hypergraph to a sequence of smaller ones. Then, heuristic partitioning algorithms are applied to the smallest hypergraph. Finally, the partitioned hypergraph is uncoarsened back to produce partitions of the original hypergraph. These algorithms require random accesses to the hypergraph located either in the memory or in other nodes. Thus, they do not scale well. Furthermore, these algorithms use MPI APIs and cannot be easily reimplemented on parallel frameworks such as Spark. Another technique called *hMulti-phase refinement* [31] considers hypergraph partitioning as a global optimization problem but it shares the same limitations. There are more recent tools for hypergraph partitioning. UMPa [32] is a serial partitioner that aims at minimizing several objective functions simultaneously; rFM [33] allows relocating vertices in partitioning; HyperSwap [34] partitions hyperedges rather than vertices. Other heuristic partitioning algorithms such as Random, Greedy, and Aweto [35] generate relatively balanced partitions with intuitive optimization strategies. These algorithms have low time complexities and scale well as they run iteratively in a distributed environment by message passing. However, these algorithms may produce more replicas which lead to higher space and communication costs. The difference among them is the optimization function used in the iterative refinement. They make different trade-offs between communication and computation costs.

In our experimental study, we compare our LPP partitioning algorithm with Random, Greedy, Aweto, hMetis, Parkway, and Zoltan because they are the most commonly used baseline algorithms in hypergraph partitioning [9], [32]–[35].

3 HYPERGRAPH PROCESSING

We first describe hypergraph notations. Then we discuss two traditional representations based on graph conversion and their limitations. We then present the HyperX framework.

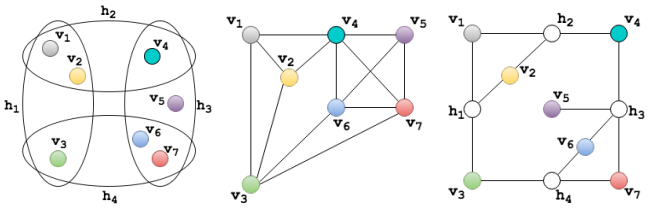
3.1 Hypergraph

We denote a hypergraph as $\mathcal{G} = \langle \mathcal{V}, \mathcal{H} \rangle$, where $\mathcal{V} = \{v_1, v_2, \ldots, v_m\}$ is a set of m vertices and $\mathcal{H} = \{h_1, h_2, \ldots, h_n\}$ is a set of n hyperedges. The *degree* of a vertex v, denoted by d_v, is the number of hyperedges that are incident to v. The *arity* of a hyperedge h, denoted by a_h, is the number of vertices in h, i.e., the number of vertices that are incident to h. Every vertex v and every hyperedge h is associated with some attributes of interest called a vertex value (e.g., a label), denoted by $v.val$ and a hyperedge value (e.g., a weight), denoted by $h.val$, respectively.

Both undirected and directed hyperedges are considered. An undirected hyperedge h is a nonempty subset of $\mathcal{V}$. For example, in Fig. 1a, there are four undirected edges, h_1, h_2, h_3 and h_4, represented by four ellipses. Each is a subset of $\mathcal{V} = \{v_1, v_2, \ldots, v_7\}$, e.g., $h_1 = \{v_1, v_2, v_3\}$. Since there are three vertices in h_1, the $arity$ of h_1 is 3, i.e., $a_{h_1} = 3$. Meanwhile, since v_1 is in both h_1 and h_2, its $degree$ is 2, i.e., $d_{v_1} = 2$. A directed hyperedge h is a mapping on two disjoint nonempty vertex sets of $\mathcal{V}$: a source set $\mathcal{S}$ and a destination set $\mathcal{D}$, i.e., $h : \mathcal{S} \rightarrow \mathcal{D}$. For example, in Fig. 1a, we can change hyperedge h_1 to a directed hyperedge by assigning $\{v_1, v_2\}$ as the source set and $\{v_3\}$ as the destination set, i.e., $h_1 : \{v_1, v_2\} \rightarrow \{v_3\}$.

3.2 Traditional Representations and limitations

Following the seminal study of Pregel [18], most distributed graph frameworks choose a vertex-centric approach and provide a *vertex program*, which updates a vertex value based on the values of neighboring vertices. To avoid extensive communication over the network, vertices are replicated to the distributed partitions [10], [11], [19], [20]. When a vertex value changes, the new value is sent to its



(a) Hypergraph (b) CE (c) SE

Fig. 1: Converting a hypergraph to a graph: CE and SE

replicas; a local aggregation that combines values sent to the same destination is employed to enable batch update. To adopt these distributed graph frameworks for hypergraph processing, we need to convert a hypergraph to a graph. Two traditional representations are used for this conversion [1]: 1) *clique-expansion* (CE), which replaces each hyperedge with multiple edges forming a clique among the incident vertices of the hyperedges, and 2) *star-expansion* (SE), which replaces each hyperedge with a new vertex connected to its incident vertices. Fig. 1 illustrates these two approaches where the hypergraph in Fig. 1a is converted to a graph shown in Fig. 1b by CE and a graph shown in Fig. 1c by SE, respectively. Although these approaches are simple to implement, they have substantial limitations.

1) CE is inapplicable to algorithms that update hyperedge values as it no longer has records corresponding to the original hyperedges in the converted graph.
2) The converted graph may have orders of magnitude more vertices and edges compared with the original hypergraph. Fig. 1 shows a substantial growth even in a tiny hypergraph. The hypergraph with 4 hyperedges and 7 vertices is converted by CE into a graph with 13 edges and 7 vertices and by SE into a graph with 13 edges and 11 vertices.
3) For SE, there are two types of vertices, those from the original hypergraph and those converted from the hyperedges of the original hypergraph. Two vertex programs are used for updating these two types of vertices. When executing these two vertex programs, it takes two iterations to update the vertex values and hyperedge values, which is a drawback, because the two iterations double the overhead of updating the vertex replicas.

3.3 HyperX Framework

HyperX has a similar architecture (cf. Fig. 2) to an existing graph framework, GraphX [10]: 1) it builds on top of Spark; 2) it runs on the Hadoop platform, i.e., YARN and HDFS; and 3) it shares all the optimization techniques with GraphX. HyperX *directly stores* a hypergraph as two RDDs [13], `vRDD` for the vertices and `hRDD` for the hyperedges. It differs from existing graph frameworks in two design choices.

– We provide two essential operations for implementing hypergraph learning algorithms, a *vertex program* denoted by `vProg` and a *hyperedge program* denoted by `hProg` (detailed in Section 3.3.1).
– We *simultaneously* distribute hyperedges *and* vertices, during which we only replicate vertices but not hyperedges to avoid excessive replicas (detailed in Section 3.3.2). This extends the classic storage strategy

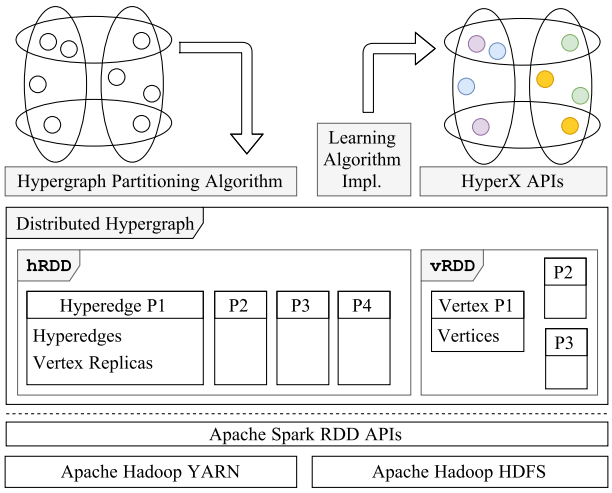


Fig. 2: An overview of HyperX

where both hyperedges and vertices are stored in multiple copies.

3.3.1 Computation Model

Algorithms running on hypergraphs usually involve accessing and updating not only $v.val$ but also $h.val$. For example, when mining relationships among social media networks [36], the weight of relations ($h.val$) between visual descriptors ($v.val$) needs to be gradually learned during the computation. Thus, we provide both a vertex program `vProg` and a hyperedge program `hProg`. The vertex program runs on each vertex, and computes $v.val$ of the vertex based on all the $h.val$s of the incident hyperedges which have that vertex inside. The hyperedge program runs on every hyperedge to update $h.val$ according to all the $v.val$s of its incident vertices.

As illustrated in Fig. 3, to update $h.val$ and $v.val$, HyperX takes only one iteration, while SE takes two iterations. Meanwhile, having `hProg` makes it much easier to balance the workloads in the two steps during each iteration because in the first step all the vertices participate in `vProg`, and in the second step, all the hyperedges participate in `hProg`. These two steps are fully decoupled. The hyperedge program provides another benefit on efficiency: it avoids extensive communication costs between vertices by adding a local aggregation step on the hyperedge partitions before sending messages to other partitions.

3.3.2 Storage Model

To distribute the workload of hypergraph processing is to separate and assign vertices and hyperedges of a hypergraph to the distributed workers (unit of resources in Spark). This requires a *hybrid-cut* that disjointedly separates the vertices *and* the hyperedges. This differs from either the *vertex-cut* that cuts the vertices to disjointedly separate the edges [20] or the *edge-cut* that cuts the edges to disjointedly separate the vertices [37].

Following the convention, in HyperX, the vertices whose incident hyperedges are assigned to different workers are replicated to those workers. Hyperedges are not replicated because replicating hyperedges is prohibitive as each hyperedge connects an unrestricted number of vertices, which

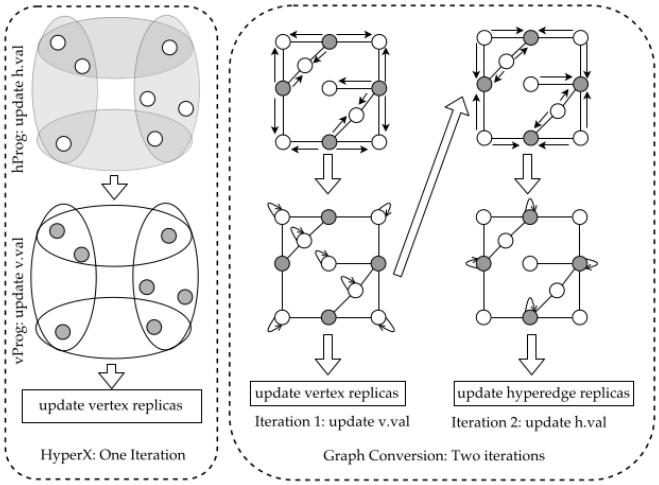


Fig. 3: Comparing HyperX with Graph Conversion, the gray shapes and bold arrows indicate the running of `vProg` and `hProg` in each step

need to be replicated with the hyperedge. This helps avoid the excessive replicas observed in CE and SE.

As a result, `vProg` does not operate locally. Instead, hyperedge values are sent to the vertex partitions over the network. The communication cost of `vProg` is thus attribute to the number of vertex replicas, as $h.val$ only needs to be sent to a partition where there are replicas of the vertices in the partition of h. Another network communication cost comes from updating $h.val$ according to the changed $v.val$, which is also attribute to the number of replicas. Both types of communications have been optimized by employing local aggregation, which means that we combine the values destined to the same partition together into one package before sending it out. Thus, HyperX avoids extensive communications between the partitions.

4 HyperX Implementation

We first describe the distributed hypergraph representation and the APIs supported in HyperX. Then we show the applicability of HyperX by discussing the implementation of three popular hypergraph algorithms on HyperX.

4.1 Distributed Hypergraph Representation

HyperX stores a hypergraph as one `vRDD` and one `hRDD`. Conceptually, each vertex and each hyperedge is stored as one row in its corresponding RDD. Let `vid` and `hid` denote the id of a vertex and a hyperedge, respectively. While `vRDD` simply stores $(\texttt{vid}, v.val)$ pairs, `hRDD` deals with an arbitrary number of vertices in each hyperedge. Directly storing a vertex set in one row introduces an overhead of the object headers and the linking pointers. Instead, we flatten each hyperedge by storing every vertex of it as a tuple of three values $\langle \texttt{vid}, \texttt{hid}, \texttt{isSrc} \rangle$, where `vid` is the id of the vertex, `hid` is the id of the hyperedge, and `isSrc` is a boolean denoting whether the vertex is a source vertex for directed hyperedge. For undirected hyperedge, `isSrc` is not used. This enables an efficient (columnar) array implementation. Now each hyperedge may span multiple consecutive rows in the `hRDD`. Given a `hid`, we cannot access the corresponding hyperedge directly. To resolve this, we create an additional map structure to associate a `hid` with the first row where the hyperedge is stored in the `hRDD`. Compared with the cost of directly storing hyperedges which is attribute to $O(\sum_{h\in\mathcal{H}} a_h)$, the cost of this additional structure is only attribute to $O(n)$, where n is the number of hyperedges. We conducted a set of experiments to evaluate the space cost of this design on various datasets. The results show that by flattening the hyperedges, we save up to 88% memory space for persisting the `hRDD` in the memory.

TABLE 2: Comparison of the number of replicas

Representation	Number of Replicas ($m = \lvert\mathcal{V}\rvert, n = \lvert\mathcal{H}\rvert$)
HyperX	$R(\mathbf{x}, \mathbf{y})$, $\mathbf{x} = m$, $\mathbf{y} = n$
GraphX-CE	$R(\mathbf{x}, \mathbf{y})$, $\mathbf{x} = m$, $\mathbf{y} \doteq \sum_{h\in\mathcal{H}} {a_h}^2 - a_h)/2$
GraphX-SE	$R(\mathbf{x}, \mathbf{y})$, $\mathbf{x} = m + n$, $\mathbf{y} = \sum_{h\in\mathcal{H}} a_h$

We show a comparative analysis on the space required for hypergraph representation by HyperX and the graph conversion based approaches in Table 2. Let $R(x, y)$ be a function that returns the number of replicas required by a distributed graph (hypergraph) representation. For simplicity, let us assume that $R(x, y)$ has the same form for HyperX and the graph representations based on CE and SE. This means that the function value depends only on the number of vertices x and the number of edges (hyperedges) y of a graph (hypergraph). For HyperX, x and y are simply the number of vertices $|\mathcal{V}|$ and the number of hyperedges $|\mathcal{H}|$ of a hypergraph. Graph representations based on CE and SE, on the other hand, inflates the graph size. For CE, y is approximated by $\sum_{h\in\mathcal{H}}({a_h}^2 - a_h)/2$ which is the sum of edges in each clique. It is an approximation because two cliques may share edges. For SE, x becomes $|\mathcal{V}| + |\mathcal{H}|$ as every hyperedge is converted to a new vertex, and y becomes $\sum_{h\in\mathcal{H}} a_h$ as every new vertex is connected with every original vertex in that hyperedge.

4.2 HyperX APIs

HyperX is built upon Apache Spark's RDD abstraction and its corresponding dataflow APIs. Here, an RDD can be seen as a horizontally distributed table. HyperX provides eight major APIs: `vertices`, `hyperedges`, `tuples`, `mrTuples`, `joinV`, `subV` `mapV`, and `mapH`, as listed in Table 3. The first three functions provide tabular views of a hypergraph, which are used to read data. The last two functions are setters for $v.val$ and $h.val$, which are used for hypergraph initialization. The middle three functions `mrTuples`, `joinV`, `subV` are essential for hypergraph processing. We detail them below.

Function `mrTuples` corresponds to `hProg` and includes the execution of three steps on a hyperedge: 1) aggregating the incident $v.val$ from the local replicas, 2) computing the new $h.val$, and 3) aggregating the $h.val$ destined to the same vertex partition.

Function `joinV` corresponds to `vProg` and includes the execution of two steps: 1) computing the new $v.val$ based on the $h.val$ received and 2) updating the replicas for each updated $v.val$.

Function `subH` restricts the computation on a sub-hypergraph. It is mainly for efficiency considerations.

These core functions enables the implementation of an iterative computation paradigm similar to Pregel [18].

TABLE 3: The APIs of `Hypergraph[V, H]` in HyperX

Functions	Return	Usage
vertices	`RDD[(Id, V)]`	View
hyperedges	`RDD[(Id, H)]`	View
tuples	`RDD[(Map[Id, srcV], Map[Id, dstV], H)]`	View
mrTuples	`RDD[(Id, M)]`	Update
joinV	`Hypergraph[V2, H]`	Update
subH	`Hypergraph[V, H]`	Update
mapV	`Hypergraph[V2, H]`	Set
mapH	`Hypergraph[V, H2]`	Set

Algorithm 1: HyperPregel

input : hypergraph $\mathcal{G}$= `Hypergraph[V,H]`, vertex program `vProg` to compute vertices values, hyperedge program `hProg` to update tuples, aggregation function `combine` defines rules of tuples aggregation, initial vertices values $initial$

output: A `RDD[(Id, V)]` in which all vertices values are updated

```
1 /*Generate a set of messages msg using initial*/
2 G ← G.mapV(vProg(initial))
3 msg ← G.mrTuples(hProg, combine)
4 /*When msg is not empty, it is broadcasted*/
5 while |msg| > 0 do
6 |   G ← G.joinV (vProg(msg))
7 |   msg ← G.mrTuples(hProg, combine)
8 /*No more msg means values of vertices and
    hyperedges are converged*/
9 return G.vertices
```

The paradigm is provided as an API in HyperX, named `HyperPregel`. Its pseudo-code is listed in Algorithm 1. The motivation for using the "pseudo-scala" code is to highlight that our algorithms are built on top of the HyperX framework, which offers scala-style APIs.

4.3 Algorithm Examples on Using HyperX APIs

To showcase the applicability of HyperX, we describe how three popular hypergraph algorithms are implemented using the APIs of HyperX.

4.3.1 Random Walks

On a hypergraph, *random walks* (RW) rank unlabeled data with regard to their high-order relationships with the labeled data. Random walks are carried out on a hypergraph by iteratively executing the following two steps independently: 1) compute the stationary probability on each vertex by aggregating the incident hyperedge values and 2) aggregate the probabilities from incident vertices to update the value on each hyperedge.

We show the implementation of directed random walks with restart on a hypergraph using HyperX APIs in Algorithm 2. Here, `joinV` is used to set up the value for a vertex to its corresponding out degree, which can be trivially obtained by $\mathcal{G}$.`mrTuples` with `map` generating $(u, 1)$ for every source vertex in the tuples, and `combine` summing the messages to the same vertex. Next, `mapV` is used to distinguish the vertices in the starting set (i.e., objects already labeled) from the other vertices. Then, `HyperPregel` is used to

Algorithm 2: Random Walks (RW) with Restart

input : hypergraph $\mathcal{G}$, restart probability rp, initial message $initial$ = 0

output: A `RDD[(Id, V)]` in which all vertices values are calculated

```
1 /*Define vProg, hProg, and combine for later use
    as parameters of HyperPregel*/
2 /*vProg updates vertex values through linearly
    combining msg and value*/
3 vProg(value, msg) = (1 − rp) × msg + rp × value
4 /*hProg updates tuple values according to the
    in-degree of hyperedges*/
5 hProg(Src, Dst, SrcDegree) =
    ∑_{i≤|Src|} Src_i / (SrcDegree_i × |Dst|)
6 /*The aggregation function is simply an addition of
    two numbers*/
7 combine(a,b) = a + b
8 /*Initialize hypergraph G to set the vertex values as
    the out-degree of vertices*/
9 G ← G.joinV (G.outDeg)
10 /*Call HyperPregel by passing above parameters to
    calculate vertex values*/
11 G.HyperPregel(G, vProg, hProg, combine,
    initial)
```

execute the random walk procedure iteratively with `vProg` to compute the new stationary probability and `hProg` to aggregate the probabilities from the incident vertices. As demonstrated in Algorithm 1, `vProg` and `hProg` will be executed in an interleaving manner in each iteration.

Fig. 4 is a running example of the algorithm. Consider a hypergraph with four directed hyperedges as shown in Fig. 4. In Step 1, we assign each vertex with their out degree as the vertex value. Vertices v_5 and v_7 only appear in the destination set, hence their out degree is zero. We randomly choose the starting points, e.g., v_1 and v_6. Thus, in Step 2, the first values of v_1 and v_6 are both 1.0, while those of the other vertices are all 0. The value of each vertex is now represented by a tuple (v, d) where v denotes the vertex label and d denotes the vertex value, e.g., for v_1, $(v, d) = (1.0, 1)$. Step 3 executes line 1 of Algorithm 2 to initialize vertex values with initial $msg = 0$ and restart probability $rp = 0.7$. Then `HyperPregel` is run to update values of vertices and hyperedges. Steps 5 to 7 correspond to two iterations of `HyperPregel`. Note: 1) in Step 4, the numbers $1, 2, 3$, and 2 correspond to the size of destination vertex set $|D|$ of the hyperedges. This step updates hyperedge values following line 2 of Algorithm 2, and 2) in both Steps 4 and 6, msgs need to be aggregated before updating the value of v_7 because v_7 receives messages sent from two hyperedges. In this example, we see unlabeled vertices $\{v_2, v_3, v_4, v_5, v_7\}$ in Step 2 labeled after two iterations as shown in Step 7.

4.3.2 Label Propagation

Label propagation (LP) on a hypergraph finds communities among the vertices according to the high-order relationships among them. The procedure is straightforward: each vertex is assigned a label to start and then iteratively exchange its

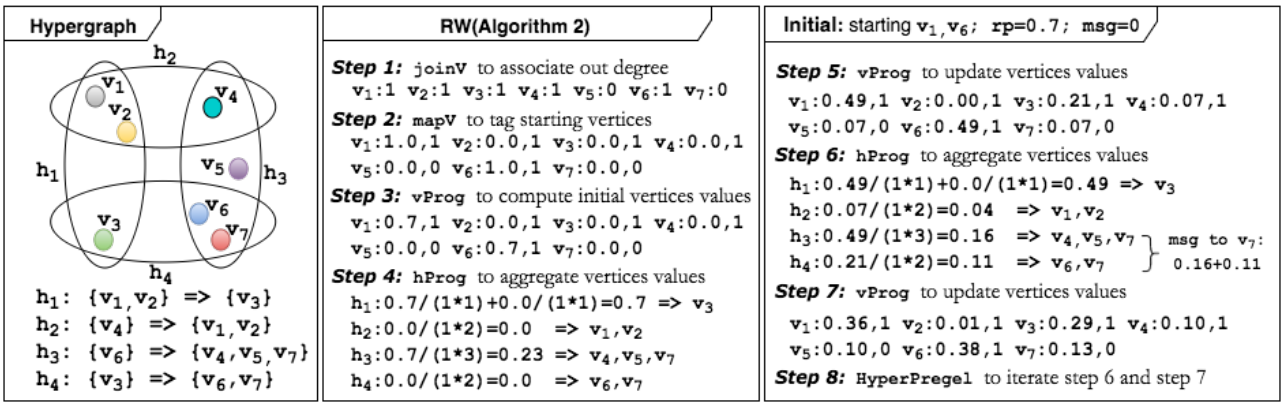


Fig. 4: Running examples of implementing Random Walks on HyperX

label with neighboring vertices through hyperedges. In each iteration, every vertex updates its label to a new one by a majority vote from its neighbors.The procedure is similar to RW, except that now $h.val$ and $v.val$ are the labels instead of the stationary probabilities, and there is no starting vertices. We omit the pseudo-code due to space limit.

4.3.3 Spectral Learning

Spectral learning (SP) covers a wide range of hypergraph Laplacian based clustering and semi-supervised learning techniques. Given a hypergraph $\mathcal{G}$, let D denote its vertex degree diagonal matrix, H denote its vertex-hyperedge incident matrix, A denote its hyperedge arity diagonal matrix, and W denote its hyperedge weight diagonal matrix. Then the normalized Laplacian is: $\mathcal{L}_\mathcal{G} = I - D^{-1/2}HWA^{-1}H^TD^{-1/2}$, where I is an $m \times m$ identity matrix. Straightforwardly, $\mathcal{L}_\mathcal{G}$ can be obtained by multiplying the matrices. In HyperX, $\mathcal{L}_\mathcal{G}$ can be computed via HyperX APIs with simpler computation, leveraging the fact that diagonal matrix multiplication is simply scaling the corresponding entries. This avoids the expensive matrix multiplication. The algorithm consists of two subtasks: 1) computing the Laplacian matrix, and 2) eigen-decomposing the matrix. We employ the Lanzcos method [27] On HyperX, Laplacian matrix can be implicitly computed during the matrix-vector multiplication phase. The idea is that a matrix-vector multiplication is basically a series of multiply-and-add operations, which can be decomposed and plugged into the Laplacian computation. The multiplication can be realized using `mrTuples` while the addition is simply `mapV`. We omit the details of lanzcosSRO since most steps are identical to that in [38] and are orthogonal to HyperX.

5 Hypergraph Partitioning

When processing a large hypergraph that is beyond the processing capability of a single worker, it is necessary to distribute the computation across multiple workers, which requires a hypergraph partitioner to divide a hypergraph into smaller partitions. This hypergraph partitioner, as an intermediate layer between the hypergraph storage and processing framework, has a significant impact on hypergraph processing algorithms in terms of workload balance and communication costs.

We investigate the hypergraph partitioning problem in this section, both in the context of HyperX and as a fundamental problem for hypergraph processing. The aim is to design a partitioning algorithm to minimize partitioning costs as well as processing costs on the partitions obtained.

We start with analyzing the limitations of existing hypergraph partitioning algorithms and propose a new optimization objective function. Based on this objective, we present our hypergraph partitioning algorithm – the *Label Propagation Partitioning* (LPP) algorithm.

5.1 Objective Analysis

Hypergraph processing costs consist of *computation costs* and *communication costs*. To minimize these costs, we find the minimization can be carried out solely on the number of replicas. This is because these costs come from synchronization among partitions that contain replicas of the same vertex (or hyperedges). Without replicas, processing a hypergraph in a distributed manner would have the same costs as processing the hypergraph on a single machine, and would not incur extra costs.

There are two types of replicas, i.e., hyperedge replicas and vertex replicas. We only use vertex replicas and aim to partition a hypergraph with a minimum number of vertex replicas to minimize the costs. The reasons for doing so are as follows. Existing partitioning solutions, either for graphs or for hypergraphs, do not distinguish the costs of these two types of replicas. This indiscrimination impacts the quality of the partitions substantially for hypergraph partitioning. In hypergraphs, a hyperedge may contain multiple vertices. Replicating a hyperedge would mean replicating more data (and hence higher costs) than replicating a vertex. In applications such as analyzing the co-worker relationships in professional networks (e.g., LinkedIn) or storage sharding where a hyperedge may relate to a large number of vertices, the difference in the costs of hyperedge replicas and vertex replicas are even more significant. To avoid the high overhead of hyperedge replicas, we only allow vertex replicas.

We propose an iterative partition update algorithm to compute the optimal partitions efficiently. Each iteration of our algorithm only takes a time linear to the number of vertices on one worker, which guarantees the high efficiency of our partitioning algorithm. This is different from traditional

k-way partitioning algorithms, such as Parkway [9] and Zoltan [30] which suffer from scalability issues. Parkway and Zoltan aim to divide the vertices of a hypergraph into a number of equal-sized components, during which the total number of subsets that the hyperedges in the hypergraph intersect is minimized. Their optimization goal is thus minimizing the *cut-size* of a hypergraph. Using a multi-level coarsen-uncoarsen technique, these algorithms partition the coarsest hypergraph on a single machine before it gets uncoarsened. This may create a single-machine bottleneck as the coarsest hypergraph may still be too large to be processed. The optimization goal of our algorithm and Parkway is different, thus the performance is not directly comparable. However, we both agree that jointly optimizing the partitions of hyperedges and vertices yields better performance than optimizing the partitions of either hyperedges or vertices.

Another consideration is to balance the workloads on the partitions. When running hypergraph applications distributively, there are two types of computations, i.e., vertex computation `vProg` and hyperedge computation `hProg`. These computations do not overlap during algorithm execution. As a result, we may balance these two types of workloads separately. These two types of workloads are not restricted to HyperX. They are common operations in modern distributed frameworks. Balancing them makes our partitioning technique extensible to these distributed frameworks as well.

5.2 Optimization Objective Formulation

Suppose we have k workers. The hypergraph partitioning problem is to allocate m vertices and n hyperedges to the k workers. Let binary variables $x_{h,i}$ and $y_{v,i}$ denote whether a hyperedge h and a vertex v are assigned to the i^{th} worker, where $i \in \{1, 2, 3, ..., k\}$. Then we can get Equations (1). A partition result is denoted as $\{X, Y\}$. It is a particular set of values for all the variables $x_{h,i} \in X = \{0, 1\}^{n \times k}$ and $y_{v,i} \in Y = \{0, 1\}^{m \times k}$.

$$\sum_{i=1}^{k} x_{h,i} = 1, \sum_{i=1}^{k} y_{v,i} = 1, x_{h,i} = 0, 1; y_{v,i} = 0, 1 \quad (1)$$

Given a vertex v, let $N(v)$ denote the set of its incident hyperedges and $R(X, Y_v)$ denote the number of replicas of vertex v given a partition result $\{X, Y\}$. Then

$$R(X, Y_v) = \sum_{i=1}^{k} max((1 - y_{v,i} - \prod_{h \in N(v)} (1 - x_{h,i})), 0) \quad (2)$$

This formulation of $R(X, Y_v)$ considers the local aggregation mechanism implemented in popular distributed framework, i.e., on each partition only one vertex replica is necessary no matter how many hyperedges in that partition are incident to the vertex .A vertex will only receive messages from a partition where it has a replica as the replica indicates the presence of incident hyperedges. Meanwhile, when vertex values change, the vertex replicas need to be updated, the communication cost of which is again attribute to the number of replicas. Thus, each replica incurs two units of communication cost. Let $C(X, Y)$ denote the overall communication costs:

$$C(X, Y) = 2 \times \sum_{v \in \mathcal{V}} R(X, Y_v) \quad (3)$$

As the space cost is proportional to $R(X, Y)$, i.e., the number of replicas given a partition result $\{X, Y\}$, the minimization of the communication costs minimizes the space cost as well. Thus, the optimization problem is as follows.

$$\begin{aligned} \textbf{minimize} \quad & C(X, Y) \\ \textbf{subject to} \quad & \sum_{h \in \mathcal{H}} x_{h,i} a_h \leq (1 + \alpha) \frac{\sum_{h \in \mathcal{H}} a_h}{k} \\ & \sum_{v \in \mathcal{V}} y_{v,i} R(X, Y_v) \leq (1 + \beta) \frac{\sum_{v \in \mathcal{V}} R(X, Y_v)}{k} \end{aligned} \quad (4)$$

Here, α and β are nonnegative relaxation factors. A value 0 for these factors suggests that every worker should have exactly the same workload. The inequalities are the load balancing constraints over hyperedges (the input of `hProg` is determined by a_h) and vertices (the input of `vProg` is determined by $R(X, Y_v)$), respectively.

5.3 Hardness and Theoretic Analysis

The constrained optimization problem described above involves a trade-off between the communication and space costs and the potential overheads for hypergraph learning algorithms to process the partitions. Setting proper values for α and β is vital for the optimization. However, the values of α and β may not be determined easily as the trade-off involves multiple factors, e.g., the data distribution, the computation complexity of `hProg` and `vProg`, the network bandwidth, etc. To overcome this limitation, we investigate a soft-constrained variation of the above problem.

5.3.1 The Strict Case

Consider a case where $\beta = +\infty$ and the variables in Y are configured such that every vertex is assigned to a worker that has its incident hyperedges. We optimize a strict instance in this case where $\alpha = 0$, i.e., each worker has exactly the same hyperedge workload. Then the optimization problem becomes:

$$\begin{aligned} \textbf{minimize} & \sum_{v \in \mathcal{V}} \sum_{i=1}^{k} (1 - \prod_{h \in N(v)} (1 - x_{h,i})) \\ \textbf{subject to} & \sum_{h \in \mathcal{H}} x_{h,i} a_h \leq \frac{\sum_{h \in \mathcal{H}} a_h}{k}, i \in \{1, 2, ..., k\}. \end{aligned} \quad (5)$$

We have the following proposition.

Proposition 1. *The above minimization problem has no polynomial time solution that can achieve a finite approximation factor unless P=NP.*

The proof follows a reduction from the strongly NP-Complete `3-Partition` problem where the goal is to partition the hyperedges set $\mathcal{H}$ into k workers with equal workload. This suggests that in general the strict case is

inapproximable (e.g., no polynomial-time approximation scheme (PTAS), no constant approximation factor, or even $\Omega(2^n)$ approximation factor).

5.3.2 A Variant with Soft Constraints

If we are able to quantitatively compare the cost of communication and computations over hyperedges and vertices, we can convert the hard constraints (Inequalities (4)) to soft constraints and integrate them into the optimization objective. This gives an alternative minimization objective $C'(X, Y)$.

$$\begin{aligned} C'(X,Y) = 2 \times & \sum_{v \in \mathcal{V}} R(X, Y_v) \\ & + w_h (\sum_{i=1}^{k} |\frac{1}{k} \sum_{h \in \mathcal{H}} |h| - \sum_{h \in \mathcal{H}} x_{h,i}|h||^p)^{\frac{1}{p}} \\ & + w_v (\sum_{i=1}^{k} |\frac{1}{k} \sum_{v \in \mathcal{V}} R(X, Y_v) - \sum_{v \in \mathcal{V}} y_{v,i} R(X, Y_v)|^p)^{\frac{1}{p}} \end{aligned} \tag{6}$$

Here, w_h and w_v denote the relative cost of a unit of computation cost on hyperedges and vertices compared with a unit of communication cost, respectively; p is the norm to aggregate the workload difference on the partitions.

To minimize this objective, we can reformulate it as a *generalized constraint satisfaction problem* (GSCP) [39] with three payoff functions C_0, C_1, and C_2 as follows.

$$\begin{aligned} C_0(X, Y_v) &= \prod_{h \in N(v)} (1 - x_{v,i}) + y_{v,i} - 1, \forall v \in \mathcal{V} \\ C_1(X) &= -w_h \frac{|\frac{1}{k} \sum_{h \in \mathcal{H}} a_h - \sum_{h \in \mathcal{H}} x_{h,i} a_h|}{\sum_{h \in \mathcal{H}} a_h}, \forall i \in [1, \ldots, k] \\ C_2(X, Y) &= -w_v \frac{|\frac{1}{k} \sum_{v \in \mathcal{V}} R(X, Y_v) - \sum_{v \in \mathcal{V}} y_{v,i} R(X, Y_v)|}{\sum_{v \in \mathcal{V}} R(X, Y_v)} \end{aligned} \tag{7}$$

The minimization of the objective $C'(X, Y)$ is therefore equivalent to maximize the total sum of all the payoff functions C_0, C_1, and C_2. This GCSP can be approached by a general semi-definite programming relaxation [39], which has been proven to deliver the best approximation for the GCSP under the unique game conjecture. According to Krauthgamer et al. [37], this relaxed semi-definite programming problem is computable in a polynomial (supercubic) time. However, it may not be sufficiently efficient for very large hypergraphs. We therefore design more efficient heuristics based algorithms.

5.4 Label Propagation Partitioning

We propose a new hypergraph partitioning algorithm to achieve the soft optimization goal with a high efficiency. This algorithm follows a label propagation procedure that labels the hyperedges and the vertices with the workers they are assigned to. A hyperedge (vertex) iteratively runs two steps: 1) propagating its label to its incident vertices (hyperedges) and 2) updating its labels based on the labels propagated to it. This algorithm differs from the classic graph label propagation algorithms in that it labels hyperedges in addition to vertices. This is essential because 1) both the hyperedges and the vertices need to be partitioned and 2) both `vProg` and `hProg` need to be balanced.

Existing techniques that label either (hyper)edges or vertices are insufficient because labeling only one type of data may obtain workload balance on that type of data but suffer from skewed workloads on the other type of data. By labeling both, we guarantee that both vertex and hyperedge workloads are balanced.

We name the proposed algorithm *Label Propagation Partitioning* (LPP). In each iteration of the LPP algorithm, when updating the label for a hyperedge (vertex), possible labels are the partitions that its incident vertices (hyperedges) have been assigned to. To find the optimal one, we first sort all candidates according to a score computed based on our objective function, i.e., balancing the workloads and reducing the replicas, and then choose the one with the highest score. The optimization problem is then reduced to designing two scoring functions $S(h, i)$ and $S(v, i)$ for a hyperedge h and a vertex v, respectively. Specifically, when updating hyperedge label, we focus on minimizing the number of replicas, since it is impossible to compute the arity distribution before all hyperedges are assigned. To reduce the number of replicas, the scoring function $S(h, i)$ for hyperedge h and worker number i is defined as the number of incident vertices of h that choose the worker. Let $N(h)$ denote the incident vertices of h, $L(h)$ and $L(v)$ denote the labels of h and v, respectively. The update rule for a hyperedge is:

$$\begin{aligned} L(h) &= \arg\max_{i \in [1,\ldots,k]} S(v, i) \\ &= \arg\max_{i \in [1,\ldots,k]} |\{v | v \in N(h) \wedge L(v) = i\}| \end{aligned} \tag{8}$$

Similarly, when updating the vertex labels, we focus on balancing the hyperedge arity and minimizing the replicas, since it is impossible to compute the replica distribution before all vertices are assigned. To balance the arity, we assign vertices to workers with smaller sum of arity. Formally, we compute the sum of hyperedge arity for each worker, denoted as $A_i = \sum_{L(h)=i} a_h$, and update vertex v as:

$$\begin{aligned} L(v) &= \arg\max_{i \in [1,\ldots,k]} S(h, i) \\ &= \arg\max_{i \in [1,\ldots,k]} (|\{h | h \in N(v) \wedge L(h) = i\}| \times e^{\frac{\bar{A}^2 - A_i^2}{\bar{A}^2}}) \end{aligned} \tag{9}$$

Here, $\bar{A} = \frac{\sum_{i \in K} A_i}{k}$, the cardinality $|\cdot|$ accounts for reducing the number of replicas; and the exponent accounts for weighting workers inversely to their sum of arity. The pseudo-code of LPP is listed in Algorithm 3.

Note that, while label propagation algorithms have good practical performance, they do not have a guaranteed approximation factor. They may not converge on general (hyper)graphs [40]. Acknowledging this, we restrict the scope to the empirical evaluation of LPP. Theoretic analysis on LPP is reserved for future study.

Discussion. LPP outperforms classic hypergraph partitioning algorithms in the following aspects.

1) LPP is scalable as it overcomes the limitation of single machine bottleneck. Unlike coarse-uncoarse techniques,

Algorithm 3: Label Propagation Partitioning

```
foreach v ∈ V do
    L(v) ← Random(k)
foreach j ← 1 to Iter do
    foreach h ∈ H do
        L(h) ← arg max_i S(h, i)
        A_{L(h)} ← A_{L(h)} + a_h
    foreach v ∈ V do
        L(v) ← arg max_{i∈K} S(v, i)
    foreach i ← i to k do
        A_i ← 0
```

workload on single machine is rather small and balanced in LPP. The updates of labels rely on message passing.

2) LPP is effective because it considers general constraints on both hyperedges and vertices. It does not replicate hyperedges which avoids high computation and communication costs for processing excessive replicas in hypergraph processing.
3) LPP is efficient because its time complexity of each iteration is linear. This outperforms the state-of-the-art technique [31], in which each subproblem has a super-cubic time complexity and is impractical to large hypergraphs.

LPP is designed for a general-purpose hypergraph processing framework. Its computational paradigm that iteratively updates labels of vertices and hyperedges that using the *vertex program* and the *hyperedge program* is commonly supported by graph frameworks such as GraphX [10] and PowerGraph [11]. This enables LPP to be implemented over any platform as long as appropriate data structures can be used to support the hypergraph representation. This requires an *object* that holds the attribute of a hyperedge along with the attributes of vertices involved in this hyperedge so that the label of a hyperedge can be updated according to the labels of its incident vertices. Particularly, HyperX provides a **`tuples`** object to view hyperedge and its vertices as a whole. In this sense, LPP fits HyperX very well. We therefore implement LPP over HyperX to evaluate its performance in the experimental study.

5.5 Other Partitioning Approaches

We discuss a few other heuristic hypergraph partitioning algorithms. They are competitors of LPP as they also run iteratively with a vertex-centric programming model, however, they have different optimization goals and label updating strategies which lead to different partition results.

Random partitioning. Random partitioning is a competent contender for partitioning hypergraphs that randomly assigns vertices and hyperedges to the partitions. If the number of partitions is small comparing with the data records to be partitioned, a round-robin style random partition may produce almost perfectly balanced partitions [41]. However, it may suffer from arbitrarily high network communication and replication costs, since no hypergraph topology is retained during the partitioning.

Greedy partitioning. Greedy partitioning is a straightforward approach to improve random partitioning. It migrates hyperedges and vertices between the partitions created by random partitioning so that the marginal cost of the objective function is minimized. This greedy approach has a linear time complexity. It will serve as a baseline in the experimental study.

Scalable bipartite partitioning. Recently, there is a heuristic bipartite partitioning algorithm proposed, named Aweto [35]. It distinguishes two different types of vertices and randomly partitions the one type that may incur cost when the computational workloads are unbalanced. Next, it partitions the other type of vertices by minimizing the replication cost. After the initial partitioning procedure, it again adopts greedy partitioning to refine the partitioning results.

6 Experiments

We evaluate HyperX (denoted by `hx` in the figures) against the alternative techniques via extensive experiments. We measure the memory space consumption of the data RDDs, i.e., `vRDD` and `hRDD` (`eRDD` with no edge values for GraphX), the network communication, and the elapsed time of running the three learning algorithms described in Section 4.3. The size of the data RDDs is a good indicator of space cost because although intermediate RDDs may persist in the memory and together they may be larger than the data RDDs, their sizes are roughly proportional to the size of the corresponding data RDDs that they are computed from.

6.1 Experimental Settings

The datasets used are listed in Table 4, where c_{vd} and c_{va} are the coefficient of variance of the vertex degree and the hyperedge arity, respectively. Three real datasets are used, Medline Coauthor (Med)[1], Orkut Communities (Ork) and Friendster Communities (Fri) [12]. Interconnections between vertices are rather intense in hypergraphs represented by these datasets, e.g., the CE graph of Ork and Fri datasets contain 122 billion and 1.8 billion edges, respectively, as shown in Table 5. This is of similar magnitude to the size of datasets used in recent studies such as [10]. The synthetic datasets are generated using Zipfian distribution with exponent $s = 2$.

The experiments are carried out on an 8 virtual-node cluster created from an academic computing cloud[2] running on OpenStack. Each virtual-node has 4 cores running at 2.6GHz and with 16GB memory. Note that each worker corresponds to one core. *A single node running with 4 processes effectively simulates 4 workers.* The network bandwidth is up to 600Mbps. One node acts as the master and the other 7 nodes act as slaves (i.e., up to $w = 28$ workers) using Apache Hadoop 2.4.0 with Yarn as the resource manager. The execution engine is Apache Spark 1.1.0-SNAPSHOT. HyperX is implemented in Scala.

6.2 Evaluation of HyperX

We implement the graph conversion approaches CE and SE described in Section 3.2 on GraphX and compare them

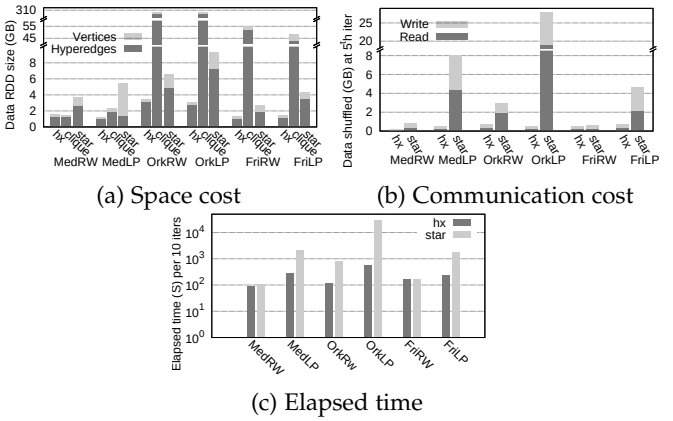
1. SBNS datasets: http://www.autonlab.org/autonweb/17433.html
2. Nectar: https://nectar.org.au/

TABLE 4: Datasets presented in the empirical study

Dataset	$\|\mathcal{H}\|$	$\|\mathcal{V}\|$	d_{min}	d_{max}	$\bar{d}$	σ_d	c_{vd}	a_{min}	a_{max}	$\bar{a}$	σ_a	c_{va}
Medline Coauthor (Med)	3,228,002	8,007,214	1	5913	10	36.91	3.69	2	744	4	2.15	0.54
Orkut Communities (Ork) [12]	2,322,299	15,301,901	1	2958	46	80.23	1.74	2	9,120	71	70.81	1.00
Friendster Communities (Fri) [12]	7,944,949	1,620,991	1	1700	5	5.14	1.03	2	9,299	81	81.39	1.00
Synthetic (Zipfian $s = 2$)	2,000,000	8,000,000	2	803	32	33.7	1.05	2	48,744	8	178.59	22.32
		12,000,000	5	1,173	48	50.27	1.05	2	49,526	8	174.07	21.76
		16,000,000	10	1,527	63	66.56	1.06	2	49,006	8	171.36	21.42
		20,000,000	15	1,893	79	83.40	1.06	2	49,963	8	175.52	21.94
		24,000,000	21	2,305	95	100.00	1.05	2	49,326	8	173.12	21.64
	4,000,000	16,000,000	1	1,102	32	36.04	1.13	2	49,843	8	173.12	21.64
	5,999,984		1	940	21	25.04	1.19	2	49,728	8	179.55	22.44
	7,999,535		1	799	16	19.42	1.21	2	49,526	8	173.84	21.73
	9,996,355		1	716	13	15.79	1.21	2	49,932	8	173.84	21.73

TABLE 5: Comparison on the size of datasets

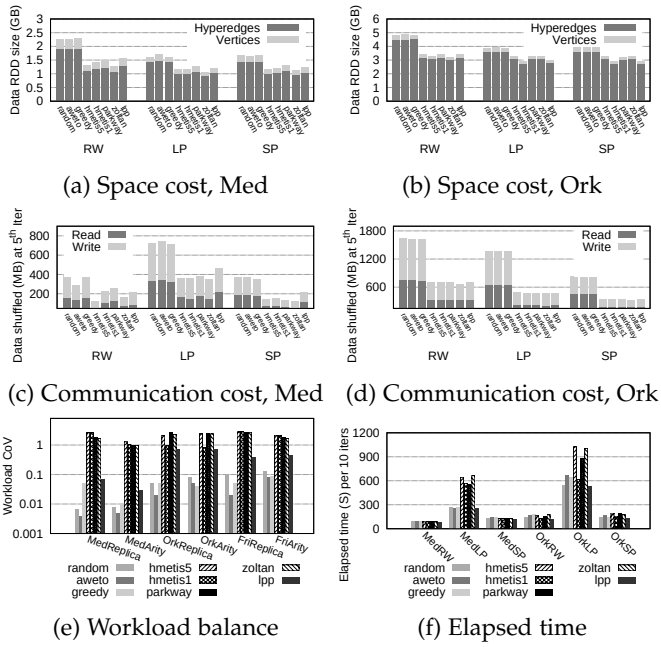
Representation	Ork		Fri	
	$\|H\|$	$\|V\|$	$\|H\|$	$\|V\|$
HyperX	2,322,299	15,301,901	7,944,949	1,620,991
GraphX-CE	2,322,299	122,956,922,990	7,944,949	1,806,067,135
GraphX-SE	17,624,200	1,086,434,971	9,565,940	643,540,869



(a) Space cost (b) Communication cost

(c) Elapsed time

Fig. 5: HyperX vs. graph conversion based approaches



(a) Space cost, Med (b) Space cost, Ork

(c) Communication cost, Med (d) Communication cost, Ork

(e) Workload balance (f) Elapsed time

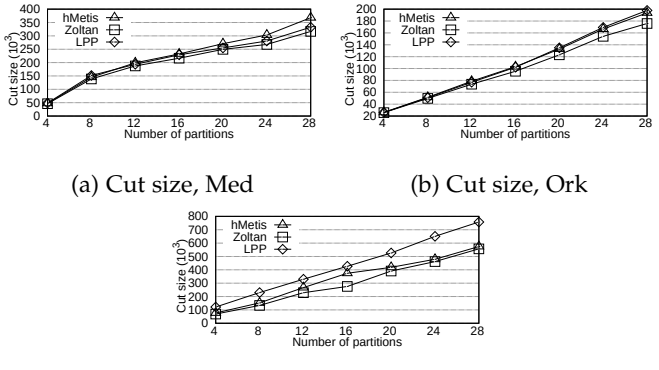
Fig. 6: Comparing partitioning algorithms

with HyperX. GraphX is a good competitor for evaluating HyperX because 1) it also runs on Spark, Hadoop, and JVM, 2) it is also implemented in Scala, and 3) it shares all the optimization techniques with HyperX such as filtered index scanning and automatic join elimination. We use `Edge2DPartition` for graph partition on GraphX as recommended [10], while for HyperX, we use LPP. For comparison, we run two hypergraph learning algorithms random walks (described in Section 4.3.1, denoted by RW) and label propagation (decribed in Section 4.3.2, denoted by LP) on HyperX and on GraphX. Figure 5 shows the result on the three real datasets, where a dataset name (e.g., Med) and a learning algorithm name (eg., RW) are combined (e.g., MedRM) to indicate the result of running the algorithm on the dataset. As Fig. 5a shows, running the learning algorithms on GraphX with CE graph conversion (denoted by `clique`) is impractical. It creates significantly more edges (in the scale of $O(\sum_{h\in\mathcal{H}} a_h^2)$), and consumes up to two orders of magnitude more memory space than those of both HyperX and SE (denoted by `star`). We therefore omit it in the rest of the figures. When compared with SE, HyperX (`hx`) performs better in all measures: its data RDDs consume 48% to 77% less memory; its communication transfers 19% to 98% less data and exchanges 27% to 93% fewer messages; and its elapsed time is up to 49.1 times shorter.

6.3 Evaluation of Hypergraph Partitioning

To evaluate the performance of the proposed LPP algorithm, we compare it with the following hypergraph partitioning algorithms: Random, Greedy, Aweto [35], hMetis [28], Parkway [9], and Zoltan [30].

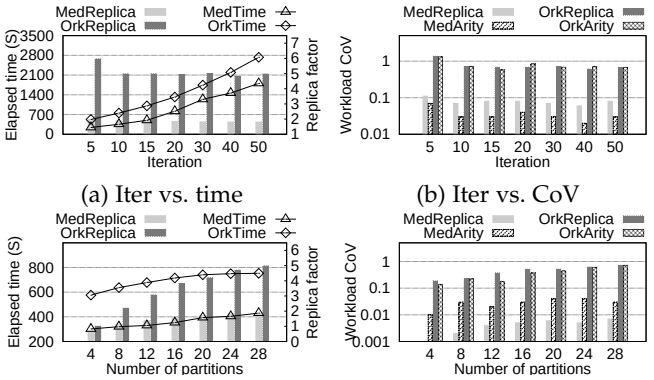
Quality of partitioning. We first compare the partitioning quality indirectly by evaluating the performance of hypergraph learning algorithms running on the partitions obtained. For hMetis, we set the workload balance factor to 5 and 1. For Zoltan and Parkway, we set the imbalance factor $\epsilon = 0.05$. For LPP, we execute `HyperPregel` for 10 iterations because we find 10 iterations are enough to produce high quality partitions and require a low running time. For all the algorithms, we set the number of partitions k to 28, which is determined by worker numbers. We then run the three hypergraph learning algorithms RW, LP and SP as described in Section 4.3 on the partitions created by these algorithms. The results are presented in Fig. 6.



(a) Cut size, Med (b) Cut size, Ork

(c) Cut size, Fri

Fig. 7: Comparing cut size of partitioning algorithms



(a) Iter vs. time (b) Iter vs. CoV

(c) # partitions vs. time (d) # partitions vs. CoV

Fig. 8: Effect of LPP parameters

TABLE 6: Partitioning time of different algorithms

Dataset	Algorithm	Time t (s)	k	w.r.t. LPP
Med	LPP	356	28	1.0
	hMetis5	14,796	1	1.5
	Zoltan	4,764	28	13.4
Ork	LPP	753	28	1.0
	hMetis5	88,936	1	4.2
	Zoltan	9,180	28	12.2
Fri	LPP	248	28	1.0
	hMetis5	6,766	1	1.0
	Zoltan	2,875	28	11.6

Among all the algorithms, hMetis, Parkway, Zoltan and LPP outperform the others in all the measures. But LPP is even better in achieving balanced workload. The workload on each partition is defined as the sum of the number of hyperedges and the number of vertices (including replicas) on that partition. In terms of workload balance, which is measured by the *Coefficient of variation* (CoV) of the workloads among different partitions. Random, Aweto, and Greedy all perform well in this metric as shown in Fig. 6e. However, their excessive numbers of replicas offset the advantages and result in high costs in the space and communication metrics. When comparing hMetis, Parkway, and Zoltan with LPP, we observe that they have different preferences on the trade-off between the workload balance and the number of replicas: LPP achieves more balanced workloads than the best of hMetis (Fig. 6e) even though it produces slightly more replicas (Figs. 6a,6b). As shown in Figs. 6a, 6b, 6c and 6d, the extra replicas in LPP do *not* result in significant space or communication overhead. According to Fig. 6f, LPP always outperforms hMetis, Parkway, and Zoltan and delivers up to 2.6 times speed-up for the learning algorithms. Another drawback of them is that they perform particularly poor in LP (even much worse than Random). This is because when all the vertices and hyperedges are active in all iterations, the unbalanced workloads outweigh the benefit gained from a slightly smaller number of replicas.

To further quantify the partitioning quality of different algorithms, we measure the *cut-size* of the partitions proposed by the algorithms. Note that while hMetis, Parkway, and Zoltan algorithms minimize the cut-size as their optimization goal, our LPP algorithm does not optimize the cut-size explicitly. However, as shown in Fig. 7, the cut size of LPP is not much worse than those of hMetis, Parkway, and Zoltan, while LPP shows a much better performance for the other quality measures as discussed above. We also find that Random, Greedy, and Aweto generally produce higher cut-size than those of LPP. Parkway and Zoltan produce very similar results. For simplicity, we only discuss Zoltan as well as hMetis and the proposed algorithm LPP in Fig. 7. Zoltan has the minimum cut-size over the three datasets. LPP has around 10-20% higher cut-size compared with those of Zoltan as the number of partitions grows larger, while the cut-size of hMetis lies in between those of LPP and Zoltan.

Partitioning time. The partitioning time of hMetis, Zoltan, and LPP on all the real datasets are listed in Table 6 (hMetis5 and hMetis1 are similar and hence only hMetis5 is shown), where k denotes the number of workers in use. Note that the hMetis and Zoltan implementations are serial, written in C, and highly optimized, while LPP is written in Scala and runs on multiple layers: HyperX, Spark, Hadoop, and JVM. Even assuming that a distributed hMetis implementation can speed up in a (unlikely) linear manner, i.e., dividing the time by $k = 28$, LPP is still faster than both hMetis and Zoltan as shown in the last column of Table 6.

Effect of LPP parameters. LPP involves two parameters: the number of iterations and the number of partitions. We report the effect of these parameters in Fig. 8. The elapsed time of LPP increases linearly as the number of iterations grows from 5 to 50. The number of replicas decreases significantly from iteration 5 to iteration 10 and from iteration 10, the decreases is less significant on the Med dataset. The number stays steady on the Ork dataset. This suggests that LPP only needs 10 iterations to achieve good minimization on the number of replicas. The workload balance demonstrates a similar trend, leaving iteration 10 a good termination point. When the number of partitions increases from 4 to 28, the elapsed time grows slowly, depicting insensitivity to the number of partitions. The number of replicas grows sub-linearly, which is reasonable considering that more vertices are separated when there are more partitions. The workload balance also degrades slowly, which verifies that the partitioning effectiveness of LPP is robust to the growth of the number of partitions.

Effect of number of workers. The results of varying the number of available workers are shown in Fig. 9. In terms of the size of data RDDs: in the Med dataset the additional workers do not significantly increase the cost; while in the Ork dataset the cost grows moderately. This is because the

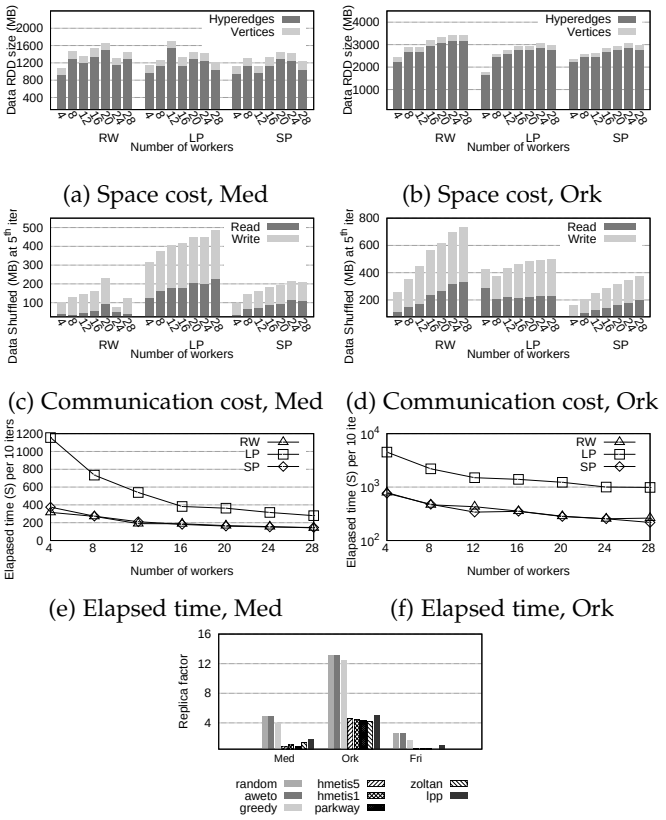


Fig. 9: Effect of number of workers

hyperedge arity in the Med dataset is rather small (average of 4 and the coefficient of variance is 0.54). The replica factor, which is defined as $\frac{|\mathcal{V}|+|R(X,Y)|}{|\mathcal{V}|}$, is only around 1.8 for LPP even when there are 28 workers as shown in Fig. 9g. This suggests that the effect of particular placement of hyperedges may overshadow the increasing replica factor, resulting in a slightly varying space consumption in different workers. In the Ork dataset, the replica factor is 4.8 when there are 28 workers. More workers lead to a larger replica factor, which dominates the space consumption and therefore depicts a small growth. In terms of the communication efficiency, more workers intuitively result in higher messages exchange cost. In terms of the time efficiency, all the algorithms on all the datasets speed-up at a sub-linear rate to the increasing number of workers. The reason for this sub-linear speed up is as follows: 1) The communication and the space costs intuitively increase due to more replicas when there are more partitions, 2) The overhead caused by YARN scheduler in each iteration is rather steady and will not diminish when there are more workers, and 3) The less significant speed up of RW and SP is because they only compute on a sub-hypergraph, where the CPU power for the distributed `hProg` and `vProg` may not be the bottleneck.

Effect of dataset cardinality. The results of varying the dataset cardinality are illustrated in Fig. 10. We use synthetic datasets described in Table 4. When the number of hyperedges grows, the size of the data RDDs grows linearly. On the other hand, when the number of vertices grows, the size grows marginally. This is because the hyperedges are the dominant factor in RDDs memory consumption.

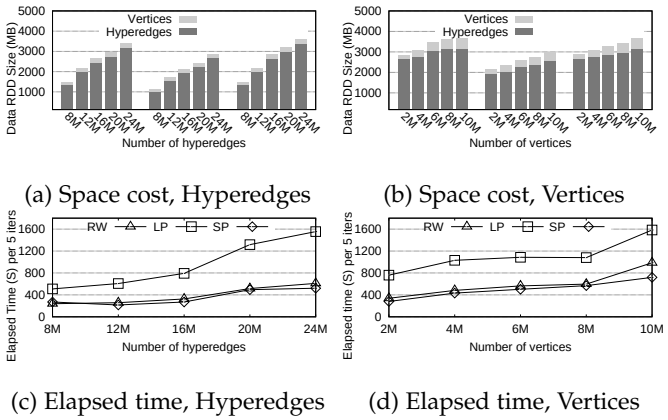


Fig. 10: Effect of datatset cardinality

The communication overhead shows a similar trend. In terms of the elapsed time, both RW and SP are relatively insensitive to the dataset cardinality as they are almost overlapped in Figs. 10c, 10d. This is because while RW starts with a particular number of vertices irrespective of the size of the datasets, SP leverages the sparseness in the matrix multiplication to avoid unnecessary computation. LP grows approximately linearly to the growth of the number of hyperedges and sub-linearly to the growth of the number of vertices.

7 Conclusions

We studied large scale hypergraph processing in a distributed environment. Our solution, HyperX, overcomes the drawbacks of traditional graph conversion based approaches by preserving the graph size, minimizing the number of replicas, and balancing the workload. We further investigated the hypergraph partitioning problem and proposed a novel label propagation partitioning (LPP) algorithm to achieve balanced and efficient partitioning. The results of an extensive empirical study on HyperX show that LPP not only generates more balanced hypergraph partitions, but also runs faster than the state-of-the-art algorithm.

HyperX offers great scalability and ease of implementation to the ever growing family of hypergraph learning algorithms. For future work, we plan to open source HyperX and use it for other hypergraph processing tasks such as building similarity based regularization classifier for recommender systems and studying biochemical interactions.

8 Acknowledgement

This work is supported by the Australian Research Council (ARC) Discovery Project DP180102050, ARC Future Fellowships Projects FT120100832, and the Research Grants Council of the Hong Kong SAR, China, No. 14221716.

References

[1] D. Zhou, J. Huang, and B. Schölkopf, "Learning with hypergraphs: Clustering, classification, and embedding," in *NIPS*, 2006, pp. 1601–1608.

[2] S. Tan, J. Bu, C. Chen, and X. He, "Using rich social media information for music recommendation via hypergraph model," in *Social media modeling and computing*, 2011, pp. 213–237.

[3] T. Hu, H. Xiong, W. Zhou, S. Y. Sung, and H. Luo, "Hypergraph partitioning for document clustering: A unified clique perspective," in *SIGIR*, 2008, pp. 871–872.

[4] Q. Liu, Y. Huang, and D. N. Metaxas, "Hypergraph with sampling for image retrieval," *Pattern Recognition*, vol. 44, no. 10-11, pp. 2255–2262, 2011.

[5] H.-K. Tan, C.-W. Ngo, and X. Wu, "Modeling video hyperlinks with hypergraph for web video reranking," in *ACM Multimedia*, 2008, pp. 659–662.

[6] T. Hwang, Z. Tian, R. Kuangy, and J.-P. Kocher, "Learning on weighted hypergraphs to integrate protein interactions and gene expressions for cancer outcome prediction," in *ICDM*, 2008, pp. 293–302.

[7] S. Tan, Z. Guan, D. Cai, X. Qin, J. Bu, and C. Chen, "Mapping users across networks by manifold alignment on hypergraph." in *AAAI*, vol. 14, 2014, pp. 159–165.

[8] Y. Wang, P. Li, and C. Yao, "Hypergraph canonical correlation analysis for multi-label classification," *Signal Processing*, vol. 105, pp. 258–267, 2014.

[9] A. Trifunovic and W. J. Knottenbelt, "Parkway 2.0: A parallel multilevel hypergraph partitioning tool," *Lecture notes in computer science*, pp. 789–800, 2004.

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

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

[12] J. Yang and J. Leskovec, "Defining and evaluating network communities based on ground-truth," *Knowledge and Information Systems*, vol. 42, no. 1, pp. 181–213, 2015.

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

[14] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache flink: Stream and batch processing in a single engine," *Bulletin of the IEEE CSTCDE*, vol. 36, no. 4, 2015.

[15] K. Andreev and H. Racke, "Balanced graph partitioning," *Theory of Computing Systems*, vol. 39, no. 6, pp. 929–939, 2006.

[16] J. Ugander and L. Backstrom, "Balanced label propagation for partitioning massive graphs," in *ICWSDM*, 2013, pp. 507–516.

[17] J. Huang, R. Zhang, and J. X. Yu, "Scalable hypergraph learning and processing," in *ICDM*, 2015, pp. 775–780.

[18] 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 *SIGMOD*, 2010, pp. 135–146.

[19] B. Shao, H. Wang, and Y. Li, "Trinity: A distributed graph engine on a memory cloud," in *SIGMOD*, 2013, pp. 505–516.

[20] F. Bourse, M. Lelarge, and M. Vojnovic, "Balanced graph edge partition," in *SIGKDD*, 2014, pp. 1456–1465.

[21] C. Pang, R. Zhang, Q. Zhang, and J. Wang, "Dominating sets in directed graphs," *Information Sciences*, vol. 180, no. 19, pp. 3647–3652, 2010.

[22] X. Lin, R. Zhang, Z. Wen, H. Wang, and J. Qi, "Efficient subgraph matching using gpus," in *Australasian Database Conference*, 2014, pp. 74–85.

[23] F. R. Chung, *Spectral graph theory*, 1997, no. 92.

[24] H. Zha, X. He, C. Ding, H. Simon, and M. Gu, "Bipartite graph partitioning and data clustering," in *CIKM*, 2001, pp. 25–32.

[25] I. S. Dhillon, "Co-clustering documents and words using bipartite spectral graph partitioning," in *KDD*, 2001, pp. 269–274.

[26] A. Trifunović and W. J. Knottenbelt, "Parallel multilevel algorithms for hypergraph partitioning," *JPDC*, vol. 68, no. 5, pp. 563–581, 2008.

[27] J. Shi and J. Malik, "Normalized cuts and image segmentation," *TPAMI*, vol. 22, no. 8, pp. 888–905, 2000.

[28] G. Karypis and V. Kumar, "Multilevel k-way hypergraph partitioning," *VLSI design*, vol. 11, no. 3, pp. 285–300, 2000.

[29] U. V. Catalyurek and C. Aykanat, "Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication," *IEEE TPDS*, vol. 10, no. 7, pp. 673–693, 1999.

[30] K. D. Devine, E. G. Boman, R. T. Heaphy, R. H. Bisseling, and U. V. Catalyurek, "Parallel hypergraph partitioning for scientific computing," in *IPDPS*, 2006, pp. 10–pp.

[31] N. Selvakkumaran and G. Karypis, "Multiobjective hypergraph-partitioning algorithms for cut and maximum subdomain-degree minimization," *TCAD*, vol. 25, no. 3, pp. 504–517, 2006.

[32] M. Deveci, K. Kaya, B. Uçar, and Ü. V. Çatalyürek, "Hypergraph partitioning for multiple communication cost metrics: Model and methods," *Journal of Parallel and Distributed Computing*, vol. 77, pp. 69–83, 2015.

[33] R. O. Selvitopi, A. Turk, and C. Aykanat, "Replicated partitioning for undirected hypergraphs," *Journal of Parallel and Distributed Computing*, vol. 72, no. 4, pp. 547–563, 2012.

[34] W. Yang, G. Wang, L. Ma, and S. Wu, "A distributed algorithm for balanced hypergraph partitioning," in *APSCC*, 2016, pp. 477–490.

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

[36] Q. Fang, J. Sang, C. Xu, and Y. Rui, "Topic-sensitive influencer mining in interest-based social media networks via hypergraph learning," *IEEE Transactions on Multimedia*, vol. 16, no. 3, pp. 796–812, 2014.

[37] R. Krauthgamer, J. S. Naor, and R. Schwartz, "Partitioning graphs into balanced components," in *SIAM*, 2009, pp. 942–949.

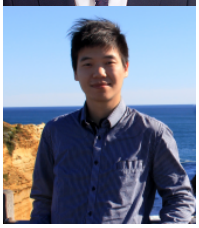
[38] U. Kang, B. Meeder, and C. Faloutsos, "Spectral analysis for billion-scale graphs: Discoveries and implementation," in *PAKDD*, 2011, pp. 13–25.

[39] P. Raghavendra, "Optimal algorithms and inapproximability results for every csp?" in *STOC*, 2008, pp. 245–254.

[40] T. Hazan and A. Shashua, "Norm-product belief propagation: Primal-dual message-passing for approximate inference," *IEEE Transactions on Information Theory*, vol. 56, no. 12, pp. 6294–6316, 2010.

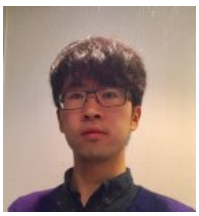
[41] P. Berenbrink, K. Khodamoradi, T. Sauerwald, and A. Stauffer, "Balls-into-bins with nearly optimal load distribution," in *SPAA*, 2013, pp. 326–335.

Wenkai Jiang received his B.E. degree in automation from Tsinghua University, Beijing, China in 2015. He is currently a Ph.D. candidate in the School of Computing and Information Systems at the University of Melbourne. His research interests include large scale data analytics and database modeling on public cloud.

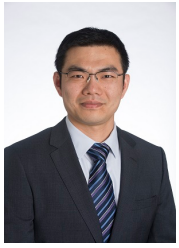


Jianzhong Qi is a lecturer in the School of Computing and Information Systems at the University of Melbourne. He received his Ph.D degree from the University of Melbourne in 2014. He has been an intern at Toshiba China R&D Center and Microsoft Redmond in 2009 and 2013, respectively. His research interests include spatio-temporal databases, locationbased social networks, information extraction, and text mining.

Jeffrey Xu Yu is currently a Professor in the Department of Systems Engineering and Engineering Management, The Chinese University of Hong Kong. He received his Ph.D. degree from University of Tsukuba. His current main research interests include keywords search in relational databases, graph mining, graph query processing, and graph pattern matching.



Jin Huang is a software engineer at Google NYC. He received his Ph.D degree from the University of Melbourne in 2016. His research interests include large scale data analytics, spatial temporal data mining, and data structures and indexes.



Rui Zhang is a Professor in the School of Computing and Information Systems at The University of Melbourne, Australia. He has won the Future Fellowship awarded by Australian Research Council in 2012, Chris Wallace Award in 2015 and Google Faculty Research Award in 2017. His research interests are data mining and databases, particularly in areas of spatial and temporal data analytics, recommender systems, moving object management and data streams.