Scalable Hypergraph Learning and Processing

Jin Huang †1, Rui Zhang †2, Jeffrey Xu Yu ‡3
† *Department of Computing and Information Systems, University of Melbourne, Australia*
{1 huang.j, 2 rui.zhang}@unimelb.edu.au
‡ *Department of Systems Engineering and Engineering Management, The Chinese University of Hong Kong, China*
3 yu.@se.cuhk.edu.hk

***Abstract*—A hypergraph allows a hyperedge to connect more than two vertices, using which to capture the high-order relationships, many hypergraph learning algorithms are shown highly effective in various applications. When learning large hypergraphs, converting them to graphs to employ the distributed graph frameworks is a common approach, yet it results in major efficiency drawbacks including an inflated problem size, the excessive replicas, and the unbalanced workloads. To avoid such drawbacks, we take a different approach and propose HyperX, which is a thin layer built upon Spark. To preserve the problem size, HyperX directly operates on a distributed hypergraph. To reduce the replicas, HyperX replicates the vertices but not the hyperedges. To balance the workloads, we investigate the hypergraph partitioning problem aiming at minimizing the space and the communication cost subject to two separate constraints on the hyperedge and the vertex workloads. With experiments on both real and synthetic datasets, we verify that HyperX significantly improves the efficiency of the learning algorithms when compared with the graph conversion approach.**

I. Introduction

Recently, a number of hypergraph learning algorithms have demonstrated high effectiveness in various applications (Table I). Such effectiveness is achieved because a hypergraph allows a hyperedge to connect multiple vertices, perfectly capturing the high-order relationships of interest, e.g., the gene expressions in protein-protein interactions, and the interest based communities among the users in a social network.

When implementing an algorithm as those in Table I, a common approach is to convert the hypergraph to a bipartite by treating the hyperedges as the vertices. For large hypergraphs, the well-studied techniques for processing large scale graphs[12], [4], [5], [14] can be applied to the converted bipartite. However, adopting the distributed graph frameworks, this conversion approach has major efficiency drawbacks.

An Inflated Problem Size. The bipartite has orders of magnitude more vertices and edges, e.g., a hypergraph [19] with 2 million vertices and 15 million hyperedges results in a bipartite with 17 million vertices and 1 billion edges.

An Enormous Replication Cost. Most distributed graph frameworks employ vertex replication to reduce the communication. When partitioned, the bipartite produces excessive replicas because 1) the inflated problem size; 2) in addition to the vertices, the hyperedges are replicated. This poses enormous space and communication costs.

A Great Difficulty in Balancing Workloads. As the partitioning algorithm may not distinguish the vertices and the hyperedges, the unbalanced workloads may drastically penalize the efficiency of the distributed computation.

TABLE I: Various hypergraph learning algorithms

Application	Algorithm	Vertex	Hyperedge
Recommendation	[16]	Songs and users	Listening histories
Text retrieval	[6]	Documents	Semantic similarities
Image retrieval	[11]	Images	Descriptor similarities
Multimedia	[15]	Videos	Hyperlinks
Bioinformatics	[7]	Proteins	Interactions
Social mining	[17]	Users	Communities
Machine learning	[18]	Records	Labels

To avoid these efficiency drawbacks, we take a different approach and propose HyperX, a thin layer built upon Spark, for easily implementing efficient and scalable hypergraph learning algorithms. We overcome the three drawbacks with the following three designs in HyperX .

First, to preserve the problem size, HyperX directly operates on a hypergraph rather than a converted graph. Specifically, it stores the hyperedges and the vertices using the *Resilient Distributed Dataset* (RDD) [20], a fault-tolerant data collection partitioned and persisted in the distributed memory. Because the hyperedges are not converted to vertices, for the ease of implementing hypergraph learning algorithms, HyperX provides the *hyperedge program* in addition to the widely adopted *vertex program* in the graph frameworks. While the vertex program computes vertex values based on the incident hyperedges, the hyperedge program computes hyperedge values based on the incident vertices. Both operations are executed independently on distributed machines. Together, they update the values in a hypergraph based on its structure.

Second, by replicating only vertices, HyperX avoids the prohibitive cost of replicating hyperedges that is attributed to the unrestrictive number of vertices in each hyperedge. Since the hyperedges and the vertices are both partitioned, such a replication suggests that while the hyperedge program accesses the vertex values via the local replicas, the vertex program accesses the hyperedge values via network communications.

Third, to minimize the replicas and to balance the workloads, we investigate the problem of simultaneously partitioning the hyperedges and the vertices. Considering all the costs involved, we formuate the problem as a minimization problem with hard constraints on the vertex and hyperedge workloads. Unfortunately, as this optimization problem does not admit efficient (approximate) solution, we apply a propagation style heuristic partitioning algorithm[1].

Our contributions in this paper are as follows.

[1]Due to the limit of space, details of the theoretic analysis and the partitioning algorithm can be found in the technical report http://iojin.com/resources/hyperx-report.pdf

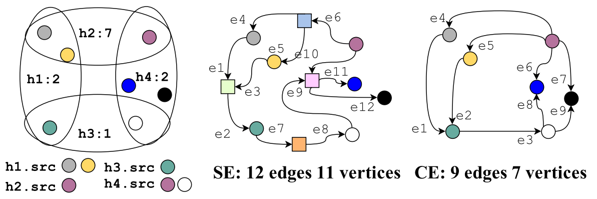


Fig. 1: Converting a hypergraph to a graph: SE and CE

1) This is the first study to systematically tackle scalable hypergraph processing. Interestingly, hypergraphs are extensively used as a popular model in optimizing distributed systems [13], yet scaling computation over data represented as hypergraphs has not been explored.
2) We design a new approach, HyperX, to directly process large hypergraphs in a distributed manner. We implement HyperX and evaluate its performance with extensive experiments on three hypergraph learning algorithms. The results are as follows.
   - Compared with the graph conversion approaches implemented on GraphX [5], for the learning algorithms, HyperX saves up to **77%** space, communicates up to **98%** fewer data, and runs up to **49** times faster.
   - HyperX gracefully scales out on a commodity cluster.

In Section II, we explain the drawbacks of the conversion approach in details. We then present an overview of HyperX in Section III, and discuss the implementation details in Section IV. The related work in discussed in Section V. The experiments are shown in Section VI and the paper concludes in Section VII.

II. Graph Conversion Drawbacks

Following the seminal study of Pregel [12], most distributed graph frameworks choose a vertex-centric approach and provides the *vertex program*, which updates the vertex value based on the values of the neighboring vertices. To avoid extensive communication over the network, vertices are replicated to the distributed partitions [5], [4], [1], [14]. When the vertex value changes, the new value is sent to its replicas; a local aggregation that combines values sent to the same destination is employed to enable the batch update. To adopt these frameworks to process large hypergraphs, we need to convert a hypergraph to a graph. Two common approaches can realize this conversion [21], i.e.,the *clique-expansion* (CE) which treats each hyperedge as a clique among the incident vertices, and the *star-expansion* (SE) which treats each hyperedge as a new vertex (therefore converting the hypergraph to a bipartite). Fig. 1 demonstrate an example on converting a hypergraph to a graph following these two approaches. Though intuitive, applying these approaches to adopt the distributed graph frameworks poses substantial drawbacks.

1) CE is not applicable to the algorithms that update $h.val$ as it no longer has records corresponding to the original hyperedges in the converted graph.
2) The graph grows orders of magnitude larger. Fig. 1 illustrates a substantial growth even in a tiny hypergraph. In this regard, we compare the conversion approaches with a direct hypergraph representation in Section IV-A, and a quick comparison is in Table II on page 5.

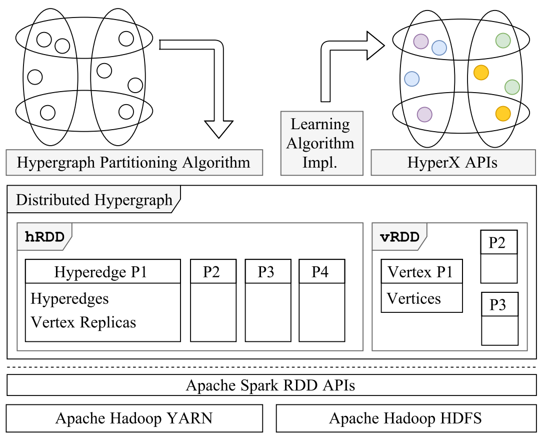


Fig. 2: An overview of HyperX

3) When partitioned, the converted graph produces excessive replicas that leads to high space and communication cost.
   - The graph partitioning problem deals with an inflated graph. Partitioning this graph naturally produces more replicas. This is especially fatal for CE as it increases the number of edges quadratically.
   - For SE, as the hyperedges are converted to vertices, not only the vertices but also the hyperedges are replicated. In real applications the hyperedge arity tends to be larger than the vertex degree (e.g., a community could easily have thousands of members yet the number of connections of each person could establish is usually at most hundreds). Replicating hyperedges could easily produce a replica spike, which further poses prohibitive space cost on storing the replicas and communication cost on updating the replicas.
4) For SE, there are two types of vertex program, i.e., one for the vertices and one for the hyperedges. As demonstrated in right-hand part of Fig. 3, when executing the vertex program iteratively, it takes two iterations to update $h.val$ and $v.val$. This is a problem due to the following.
   - Updating the vertex replicas for the changed vertex values is a constant overhead thanks to the local aggregation. However, two iterations results in doubling the cost of such an overhead.
   - When partitioning a graph following the classic techniques, either the vertices or the edges are balanced. On the converted graph, such a partitioning does not necessarily yield balanced distributions on *either* the real vertices *or* the vertices that represent hyperedges. As a result, in each iteration, there is no guarantee that the execution of either type of vertex program generates balanced workloads on the distributed machines.

III. HyperX Overview

HyperX has a similar architecture (demonstrated Fig. 2) to the existing graph frameworks: 1) it builds on top of Spark; 2) it runs on the Hadoop platform, i.e. YARN and HDFS; 3) it shares all the optimization techniques with GraphX [5]. HyperX *directly stores* a hypergraph as two RDDs, vRDD for the vertices and hRDD for the hyperedges. Moreover, HyperX differs from the graph frameworks in two major design choices.

- We provide two common operations for implementing the hypergraph learning algorithms. In addition to the widely

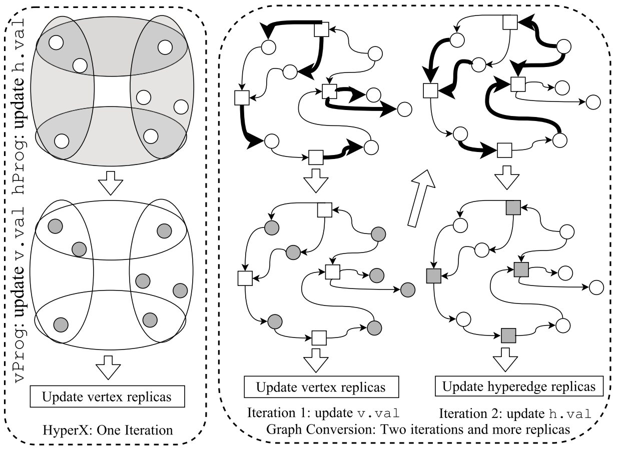

vProg: update v.val hProg: update h.val
Update vertex replicas
HyperX: One Iteration
Update vertex replicas
Update hyperedge replicas
Iteration 1: update v.val
Iteration 2: update h.val
Graph Conversion: Two iterations and more replicas

Fig. 3: Comparing HyperX with SE, the gray shapes and bold arrows indicate the running `vProg` (`hProg`) in each step

adopted vertex program (denoted as `vProg`) in the graph frameworks, the hyperedge program (denoted as `hProg`) is essential for HyperX.

- We *simultaneously* partition the hyperedges *and* the vertices. This extends the classic vertex partitioning and edge partitioning techniques. Only the vertices are replicated to avoid excessive replicas.

We elaborate these two choices in the remainder of this section.

A. Hyperedge Program and Vertex Program

Hypergraph learning algorithms usually involve accessing and updating $h.val$, e.g., the weight of relations between the visual descriptors are gradually learned during the computation [3]. Hence, in addition to `vProg` that computes $v.val$ based on $h.val$ on the incident hyperedges, we provide `hProg` that operates on every hyperedge to update $h.val$ according to $v.val$ on its incident vertices. Both `hProg` and `vProg` execute independently on each hyperedge and vertex partition, respectively. By executing `hProg` and `vProg` sequentially in one iteration, both $h.val$ and $v.val$ are updated. This sequential execution does not degrade the parallelism because both the hyperedges and the vertices are fully partitioned; all the computing resources are utilized in `hProg` and `vProg`.

As demonstrated in Fig. 3, to update $h.val$ and $v.val$, HyperX takes only one iteration, while SE takes two iterations. Meanwhile, this makes it much easier to balance the workloads in the two steps during each iteration: all the vertices participate in `vProg` and all the hyperedges participate in `hProg`.

B. Partitioning with Hybrid-Cut and Vertex Replication

HyperX brings the computation to the distributed data. To distribute the workloads is therefore to partition vertices and hyperedges and assign them to the distributed workers (the unit of resource in Spark). This requires a *hybrid-cut* that disjointly partitions the vertices *and* the hyperedges. This differs from the *vertex-cut* that cuts the vertices to disjointly partition the edges[1] and the *edge-cut* that cuts the edges to disjointly partition the vertices [9].

Following the convention, the vertices whose incident hyperedges are assigned to different workers are replicated to those workers. The hyperedges are not replicated because replicating the hyperedges is prohibitive as each hyperedge connects an unrestrictive number of vertices, whose ids need to be replicated with the hyperedge value. As a result, `vProg` does not operate locally. Instead, the hyperedge values are sent to the vertex partitions over the network. The communication cost during `vProg` is thus attributed to the number of vertex replicas, as $h.val$ is sent to a vertex partition precisely because there are vertex replicas in that hyperedge partition. Another network communication of updating replicas according to the changed $v.val$, is also attributed to the number of replicas. Both communications employ the local aggregation to combine the values destined to the same partition.

IV. HyperX Implementation

We briefly discuss the distributed hypergraph representation and the APIs in HyperX, and then show how three widely used hypergraph learning algorithms can be easily implemented over HyperX.

A. Representing A Distributed Hypergraph

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 vertex and the hyperedge id, respectively. While the `vRDD` simply stores $(\texttt{vid}, v.val)$ pairs, the `hRDD` deals with the arbitrary number of source and destination vertices in each hyperedge. Directly storing the source and the destination vertex sets in one row introduces the overhead of the object headers and the linking pointers. Instead, we flatten each hyperedge to multiple `(vid, hid, isSrc)` tuples in `hRDD`. This enables an efficient (columnar) array implementation.

However, as now that each hyperedge spans multiple consecutive rows in the `hRDD`, given a `hid`, we cannot access the hyperedge directly. To resolve this, we create an additional map structure to associate `hid` with the first row the hyperedge is stored in the `hRDD`. Compared with the cost of directly storing hyperedges that is attributed to $O(\sum_{h\in\mathcal{H}} a_h)$, the cost of this additional structure is only attributed to $O(n)$. We conducted a set of experiments to evaluate the space efficiency of this design on various datasets. The results show that by flattening the hyperedges, we save **41%** to **88%** memory space for persisting the `hRDD` in the memory.

We compare the representation of HyperX with that of SE and CE implemented over GraphX in Table II. Intuitively, SE and CE increase the number of vertices and hyperedges, as shown by x' and y' in the first column. In real datasets such as Ork [19], this increase could be orders of magnitude, as demonstrated in the last two columns in the table. Moreover, the resultant number of replicas in these approaches could be drastically larger than that in HyperX as 1) they deal with a problem orders of magnitude larger; 2) a large number of hyperedges are subject to replication as well.

B. HyperX APIs

HyperX has five major APIs: `mrTuples`, `joinV`, `mapV`, `mapH`, and `subV`. The 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 $h.val$; 3) aggregating the $h.val$ destined

TABLE II: Comparison on the representations

Representation	# of Replicas)	m in Ork	n in Ork
HyperX	$R(\mathbf{x},\mathbf{y})$, $\lvert\mathbf{x}\rvert = km$, $\lvert\mathbf{y}\rvert = nk$	2,322,299	15,301,901
GraphX-SE	$R(\mathbf{x}',\mathbf{y}')$, $\lvert\mathbf{x}'\rvert = k(m+n)$, $\lvert\mathbf{y}'\rvert = k\sum_{h\in\mathcal{H}} a_h$	17,624,200	1,086,434,971
GraphX-CE	$R(\mathbf{x},\mathbf{y}')$, $\lvert\mathbf{x}\rvert = km$, $\lvert\mathbf{y}'\rvert = k\sum_{h\in\mathcal{H}} \frac{({a_h}^2-a_h)}{2}$	2,322,299	122,956,922,990

Algorithm 1: HyperPregel

```
input  : $\mathcal{G}$: Hypergraph[V,H], vProg: (Id,V) $\Rightarrow$ V,
         hProg: Tuple $\Rightarrow$ M, combine: (M,M) $\Rightarrow$ M,
         $initial$: M
output: RDD[(Id, V)]
1 $\mathcal{G} \leftarrow \mathcal{G}$.mapV(($id$, $v$) $\Rightarrow$ vProg($id$, $v$, $initial$))
2 $msg \leftarrow \mathcal{G}$.mrTuples(hProg, combine)
3 while $|msg| > 0$ do
4     $\mathcal{G} \leftarrow \mathcal{G}$.joinV ($msg$) (vProg).subH(v', t')
5     $msg \leftarrow \mathcal{G}$.mrTuples(hProg, combine)
6 return $\mathcal{G}$.vertices
```

to the same vertex partition. The function `joinV` corresponds to `vProg` and includes the execution of two steps: 1) computing the new $v.val$ based on the $h.val$ received; 2) updating the replicas for each updated $v.val$. The function `subH` restricts the computation on a sub-hypergraph, and it is mainly for efficiency considerations. The functions `mapV` and `mapH` are simply the setters for $v.val$ and $h.val$. With these APIs, we can easily implement an iterative computation paradigm similar to Pregel for hypergraphs, i.e., `HyperPregel` (Algorithm 1).

C. Implementing Learning Algorithms

In this section, we briefly describe how three common hypergraph learning algorithm can be implemented easily using HyperX. Fig. 4 gives the running examples of the random walks and the spectral learning.

1) Random Walks: We show the implementation of directed random walks with restart on a hypergraph [16] using the APIs in Algorithm 2. Here, `joinV` is used to set up the attribute 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 (e.g., the songs already labeled) from the other vertices. Then, `HyperPregel` is used to execute the random walk procedure iteratively with the `vProg` to compute the new stationary probability and the `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.

2) Label Propagation: Algorithm 3 demonstrates the the implementation of a label propagation on HyperX. 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 vertex set.

3) Spectral Learning: We demonstrate how the hypergraph spectral learning (e.g., clustering and embedding) [21], can be implemented using HyperX APIs in Algorithm 4. The technique consists of two subtasks, 1) computing the Laplacian matrix and 2) eigen-decomposing the matrix. We employ

Algorithm 2: Random Walks (RW) with restart

```
input  : $\mathcal{G}$, label vertex set $L$, restart probability $rp$
output: RDD[(Id, Double)]
1 vProg(id,(v,d),msg)= $((1-rp)\times msg + rp \times v,\ d)$
2 hProg($\mathcal{S}$,$\mathcal{D}$,$Sd$,$Dd$,$h$)= $\sum_{i\le|\mathcal{S}|}\frac{\mathcal{S}_i}{Sd_i\times|\mathcal{D}|}$
3 combine(a,b)= $a+b$
4 $\mathcal{G} \leftarrow \mathcal{G}$.joinV ($\mathcal{G}$.outDeg, ($id$, $v$, $d$) $\Rightarrow$ $d$)
5 $\mathcal{G} \leftarrow \mathcal{G}$.mapV(($id$, $v$) $\Rightarrow$ if $id \in L$ (1.0, $v$) else (0.0, $v$))
6 $\mathcal{G}$.HyperPregel($\mathcal{G}$, vProg, hProg, combine,0)
```

Algorithm 3: Label Propagation (LP)

```
input  : $\mathcal{G}$
output: RDD[(Id, Id)]
1 vProg(id,(v,d),[(l,c)]) = $\arg\max_{l\in[(l,c)]} c$
2 hProg($\mathcal{S}$,$\mathcal{D}$,$Sl$,$Dl$,$h$)= $(Sl+Dl)$.map($l \Rightarrow (l,1)$)
3 agg(a,b)= $(a+b)$.reduceByKey($(c,c') \Rightarrow c+c'$)
4 $\mathcal{G} \leftarrow \mathcal{G}$.mapV(($id$, $v$) $\Rightarrow$ $id$)
5 $\mathcal{G}$.HyperPregel ($\mathcal{G}$,vProg,hProg, agg, null)
```

the Lanzcos method with selective reorthogonalization (lanzcosSRO) as the numeric method for the eigen-decomposition. The lanzcosSRO method is an iterative procedure that involves matrix-vector multiplication. On HyperX, a straightforward approach would first explicitly compute Laplacian and then perform lanzcosSRO. However, as demonstrated in Algorithm 4, the Laplacian matrix can be implicitly computed during the matrix-vector multiplication phase in the second step. 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. Specifically, the multiplication could be realized using `MRTuples` while the addition is simply `mapV`. We omit the details of lanzcosSRO in Algorithm 4 since most steps are identical to that in [8] and are orthogonal to HyperX.

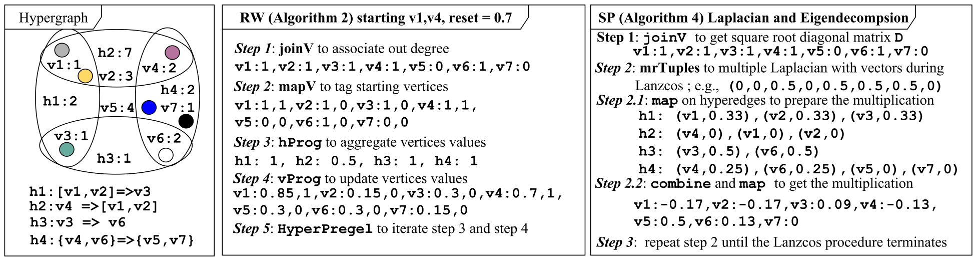
V. Related Work

A. Distributed Graph Processing Frameworks

Distributed graph processing has been intensively investigated in recent years [14], [12], [4], [5]. A number of issues that HyperX addresses for hypergraphs have been studied for graphs: replicating data [4], aggregating messages [5], partitioning the data [14], [4], and providing common APIs for a wide range of applications [12], [5]. However, as described in Section II, there are major efficiency issues in adopting these graph techniques when processing a hypergraph.

B. Hypergraph Learning

The applications of hypergraphs include music and news recommendations [16], [10], multimedia retrieval [2], [15], bioinformatics [7], social mining [17], and information retrieval [3]. All these studies have demonstrated that the hypergraph model is a highly effective tool for harvesting the rich relationships not captured by a graph model. However, these studies evaluate their techniques on datasets containing a few hundred thousand records. To the best of our knowledge, there is no study specifically addressing the scalability issues in hypergraph learning with a cluster of commodity machines.



RW (Algorithm 2) starting v1,v4, reset = 0.7

Step 1: **joinV** to associate out degree
`v1:1,v2:1,v3:1,v4:1,v5:0,v6:1,v7:0`
Step 2: **mapV** to tag starting vertices
`v1:1,1,v2:1,0,v3:1,0,v4:1,1,`
`v5:0,0,v6:1,0,v7:0,0`
Step 3: `hProg` to aggregate vertices values
`h1: 1, h2: 0.5, h3: 1, h4: 1`
Step 4: `vProg` to update vertices values
`v1:0.85,1,v2:0.15,0,v3:0.3,0,v4:0.7,1,`
`v5:0.3,0,v6:0.3,0,v7:0.15,0`
Step 5: `HyperPregel` to iterate step 3 and step 4

SP (Algorithm 4) Laplacian and Eigendecompsion

Step 1: `joinV` to get square root diagonal matrix `D`
`v1:1,v2:1,v3:1,v4:1,v5:0,v6:1,v7:0`
Step 2: **mrTuples** to multiple Laplacian with vectors during Lanzcos ; e.g., `(0,0,0.5,0,0.5,0.5,0.5,0)`
Step 2.1: `map` on hyperedges to prepare the multiplication
`h1: (v1,0.33),(v2,0.33),(v3,0.33)`
`h2: (v4,0),(v1,0),(v2,0)`
`h3: (v3,0.5),(v6,0.5)`
`h4: (v4,0.25),(v6,0.25),(v5,0),(v7,0)`
Step 2.2: `combine` and `map` to get the multiplication
`v1:-0.17,v2:-0.17,v3:0.09,v4:-0.13,`
`v5:0.5,v6:0.13,v7:0`
Step 3: repeat step 2 until the Lanzcos procedure terminates

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

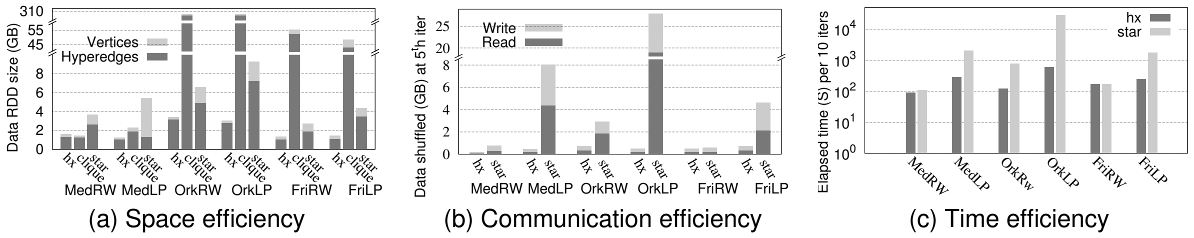


Fig. 5: Comparing HyperX with graph conversion approaches

Algorithm 4: Laplacian Spectral Learning (SP)

```
input : $\mathcal{G}$, $eigK$
output: Eigenvectors $[eigC]$, eigenvalue $[eigV]$
1 map(mapS,mapD,w)=$(mapS+mapD)$.map($(id, val) \Rightarrow (id, val \times ( (\frac{w}{|mapS|+|mapD|} \sum (\mathcal{S}+\mathcal{D})$.map$(u \Rightarrow u.val \times vec(u.id)))$
2 combine(a,b)=$a+b$
3 multiple($\mathcal{G}$,$vec$)=$\mathcal{G}$.mrTuples(map,combine).map($(id, val) \Rightarrow vec(id) - val$)
4 $\mathcal{G} \leftarrow \mathcal{G}$.joinV ($\mathcal{G}$.deg,($id$,$v$,$d$) $\Rightarrow d^{-\frac{1}{2}}$)
5 lanzcosSRO($\mathcal{G}$, multiple, $eigK$)
```

TABLE III: Datasets presented in the empirical study

Dataset	n	m
Medline Coauthor (Med) [2]	3,228,002	8,007,214
Orkut Communities (Ork)[19]	2,322,299	15,301,901
Friendster Communities (Fri)[19]	7,944,949	1,620,991
Synthetic (Zipfian $s = 2$)	2M - 10M	8M - 24M

VI. Empirical Study

We evaluate HyperX (shortened to `hx` in the figures) against the alternative techniques via extensive experiments. We measure the memory space consumption of data RDDs, i.e., `vRDD` and `hRDD` (`eRDD` with no edge values for GraphX), the network communication, and the elapsed time of the three learning algorithms. Due to the limit of space, we only show the elasped time when varying the dataset cardinality and the number of workers[3].

A. Datasets and Experimental Settings

The datasets used are listed in Table III. The synthetic datasets are generated using Zipfian distribution with exponent s set to 2.

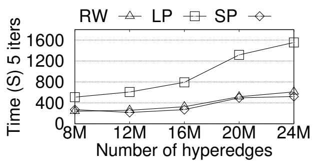
Discussion on the size of the datasets. These datasets may seem small yet they are adequate for the purpose of evaluation for three reasons. First, the interconnection between vertices in a hypergraph is rather intense. For example, the clique-expansion graph in the Fri and the Ork datasets respectively contain 1,834,786,185 and 122,956,922,990 edges, which is of similar magnitude to the size of datasets used in recent publications such as [5]. Second, the data RDDs may appear small in the figures, yet because of significant intermediate RDDs, these datasets are the largest we can evaluate in our settings. Third, the hypergraph learning algorithms can be of high complexity, where the previous studies only tackle problems with hundreds of thousands of hyperedges [10], [17].

The experiments are carried out on an 8 virtual-node cluster created from an academic computing cloud running on OpenStack. Each virtual-node has 4 cores running at 2.6GHz and 16GB memory. Note that each worker corresponds to one core, *4 workers effectively simulate a single node running with 4 processes.* 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 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.

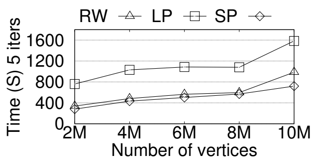
B. Comparing with the Graph Conversion Approaches

We implement the graph conversion approaches on GraphX and compare them with HyperX. GraphX is a great competitor for evaluating HyperX because 1) it also executes on Spark, Hadoop, and JVM; 2) it is also implemented in Scala; 3) it shares all the optimization techniques with HyperX such as filtered index scanning and automatic join elimination. For GraphX, we use `Edge2DPartition` as recommended [5], while for HyperX, we use LPP. The results are illustrated in Fig. 5. `CE` is not a practical choice in real applications due to its $O(\sum_{h \in \mathcal{H}} a_h^2)$ extra edges, it consumes up to two orders of magnitude more memory than both SE and HyperX. We therefore omit it thereafter. When compared with SE, HyperX is clearly the better choice in all criteria: data RDDs consume 48% to **77%** less memory; communication transfers 19% to **98%** less data while exchanging 27% to **93%**

[3]the results on the memory space consumption and the network communication can be found in the report http://iojin.com/resources/hyperx-report.pdf

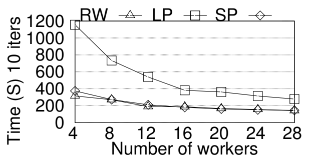


(a) Time Efficiency, Hyperedges

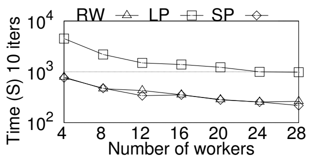


(b) Time Efficiency, Vertices

Fig. 6: Varying dataset cardinality



(a) Time Efficiency, Medline



(b) Time Efficiency, Orkut

Fig. 7: Varying number of workers

fewer messages; the elapsed time is shortened by up to **49.1** times. This verifies the drawbacks we identified in Section II: graph conversion enlarges the problem size, produces excessive replicas, and makes it difficult to balance workloads for the hypergraph learning algorithms.

C. Varying Dataset Cardinality and the Number of Workers

The results of varying the dataset cardinality are illustrated in Fig. 6. As depicted in the figures, both RW and SP are relatively insensitive to the dataset cardinality. 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.

The results of varying the number of available workers are shown in Fig. 7. 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 character is as follows. 1) The communication and the space cost 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. 3) The less significant speedup of RW and SP is because both of them only compute on a sub-hypergraph, where the CPU power for the distributed `hProg` and `vProg` may not be the bottleneck.

VII. Conclusions

We studied large scale hypergraph processing in a distributed environment. Our solution, HyperX, systematically overcomes the drawbacks in the graph conversion approach by preserving the problem size, minimizing replicas, and balancing the workload. We conducted an empirical study on both real and synthetic datasets to evaluate the performance of HyperX. The results confirmed that for the hypergraph learning tasks, HyperX is much more efficient than adopting the graph frameworks via a graph conversion. HyperX offers great scalability and the ease of implementation to the ever growing family of hypergraph learning algorithms. We intend to open source HyperX to the community shortly.

Acknowledgment

This work is supported in part by the Australian Research Council (ARC) Discovery Project DP130104587 and the Research Grants Council of the Hong Kong SAR, China, No. 14209314 and 418512. Dr. Zhang is supported by the ARC Future Fellowships Projects FT120100832.

References

[1] F. Bourse, M. Lelarge, and M. Vojnovic. Balanced graph edge partition. In *KDD*, 2014.

[2] A. Docournau and A. Bretto. Random walks in directed hypergraphs and applications to semi-supervised image segmentation. *CVIU*, 2014.

[3] Q. Fang, J. Sang, C. Xu, and Y. Rui. Topic-sensitive influencer mining in interest-based social media networks via hypergraph learning. *IEEE TM*, 2014.

[4] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *OSDI*, 2012.

[5] 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*, 2014.

[6] T. Hu, H. Xiong, W. Zhou, S. Y. Sung, and H. Luo. Hypergraph partitioning for document clustering: a unified clique perspective. In *SIGIR*, 2008.

[7] T. Hwang, Z. Tian, R. Kuang, and J.-P. Kocher. Learning on weighted hypergraphs to integerate protein interactions and gene expressions for cancer outcome prediction. In *ICDM*, 2008.

[8] U. Kang, B. Meeder, and C. Faloutsos. Spectral analysis for billion-scale graphs: Discoveries and implementation. In *PAKDD*, 2011.

[9] R. Krauthgamer, J. S. Naro, and R. Schwartz. Partitioning graphs into balanced components. In *SODA*, 2009.

[10] L. li and T. Li. News recommendation via hypergraph learning: Encapsulation of user behavior and news content. In *WSDM*, 2013.

[11] Q. Liu, Y. Huang, and D. N. Metaxas. Hypergraph with sampling for image retrieval. *Pattern Recognition*, 2011.

[12] G. Malewicz, M. H. Austern, A. J. C. Matthew, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: A system for large-scale graph processing. In *SIGMOD*, 2010.

[13] N. Selvakkumaran and G. Karypis. Multi-objective hypergraph partitioning algorithms for cut and maximum subdomain degree minimization. *IEEE TCAD*, 2006.

[14] B. Shao, H. Wang, and Y. Li. Trinity: a distributed graph engine on a memory cloud. In *SIGMOD*, 2013.

[15] H.-K. Tan, C.-W. Ngo, and X. Wu. Modeling video hyperlinks with hypergraph for web video ranking. In *MM*, 2008.

[16] S. Tan, J. Bu, C. Chen, B. Xu, C. Wang, and X. He. Using rich social media information for music recommendation via hypergraph model. *ACM TMCCA*, 2013.

[17] S. Tan, Z. Guan, D. Cai, X. Qin, J. Bu, and C. Chen. Mapping users across networks by manifold alignment on hypergraph. In *AAAI*, 2014.

[18] Y. Wang, P. Li, and C. Yao. Hypergraph canonical correlation analysis for multi-label classification. *Signal Processing*, 2014.

[19] J. Yang and J. Leskovec. Defining and evaluating network communities based on ground-truth. In *ICDM*, 2012.

[20] 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 clutser computing. In *NSDI*, 2012.

[21] D. Zhou, J. Huang, and B. Scholkopf. Learning with hypergraphs: Clustering, classification, and embedding. In *NIPS*, 2006.