Distributed Hypergraph Processing Using Intersection Graphs

Yu Gu, Kaiqiang Yu, Zhen Song, Jianzhong Qi, Zhigang Wang, Ge Yu, *Member, IEEE,* Rui Zhang

Abstract—The advent of online applications such as social networks has led to an unprecedented scale of data and complex relationships among data. Hypergraphs are introduced to represent complex relationships that may involve more than two entities. A hypergraph is a generalized form of a graph, where edges are generalized to hyperedges. Each hyperedge may consist of any number of vertices. The flexibility of hyperedges also brings challenges in distributed hypergraph processing. In particular, a hypergraph is more difficult to be partitioned and distributed among k workers with balanced partitions. In this paper, we propose to convert a hypergraph into an intersection graph before partitioning by leveraging the inherent shared relationships among hypergraphs. We explore the intersection graph construction method and the corresponding partition strategy which can achieve the goal of evenly distributing vertices and hyperedges across workers, while yielding a significant communication reduction. We also design a distributed processing framework named $Hyraph$ that can directly run hypergraph analysis algorithms on our intersection graphs. Experimental results on real datasets confirm the effectiveness of our techniques and the efficiency of the $Hyraph$ framework.

Index Terms—Hypergraphs, shared relationships, intersection graphs, distributed processing, graph processing

1 INTRODUCTION

WITH the rise of applications such as social networks and the rapid development of technologies such as cloud computing, data scale has been increasing rapidly, and the relationships among data are becoming more and more complex. Hypergraphs are powerful tools to model complex relationships. Each hyperedge in a hypergraph can contain any number of vertices and hence the hypergraph model can capture complex and high order relationships among data objects. Take the author-cooperation as an example. A hyperedge h_1 with vertices $\{v_1, v_2, v_3\}$ can naturally express the semantics that the paper h_1 is cooperatively written by authors v_1, v_2 and v_3. However, such information cannot be expressed easily in a regular graph with vertices as authors and edges as cooperation relationships. Further, if we run a pagerank algorithm on the regular graph to evaluate authors, we might get inaccuracy evaluation. This is because an author with high scores will have positive impact on his/her co-authors, even though some of their co-authored papers have less significant contribution. Because of the strong expression and rich semantics, numerous hypergraph analysis algorithms have been proposed in data mining and information retrieval tasks [1], [2], [3], [4], [5], many of which are naturally iterative.

Existing hypergraph iterative processing methods include SE (Star-Expansion), CE (Clique-Expansion) and HyperX [6], [7]. SE converts the hypergraph into a bipartite graph and then executes on the distributed graph processing framework such as Giraph [8]. However, the overall performance is still far from ideal since the characteristics of hypergraph have not been fully considered. CE converts the hypergraph into a homogeneous graph, which is inapplicable to hypergraph analysis tasks that need to update the values of hyperedges. HyperX updates the values of hyperedges and vertices in the same superstep sequentially but does not consider the structure of the original hypergraph, usually yielding suboptimal performance. Differently, we propose a hypergraph iterative processing method based on intersection graphs denoted by IG. IG considers the hypergraph structure under its construction and partition, which solves the limitations mentioned above.

In this paper, we observe the sharing and symmetry properties in hypergraphs and take advantages of them to improve the efficiency of hypergraph processing. In hypergraphs, each hyperedge consists of a non-empty subset of vertices. Different hyperedges may share common vertices, i.e., they *intersect*. Similarly, for a vertex, there is a set of hyperedges incident to it, and different vertices may share common incident hyperedges. In distributed hypergraph processing, hyperedges with shared common vertices and vertices with shared incident hyperedges should be assigned to the same partition as much as possible. For example, suppose the intersection of hyperedge h_i and hyperedge h_j is S. When h_i and h_j are assigned into the same partition, vertices in S need to send messages to only one of h_i and h_j, instead of both, since the other can fetch these shared messages directly. This design yields a significant communication reduction. The symmetry structure of hypergraphs makes it possible to process hyperedges and vertices in the same way.

Furthermore, we propose an intersection graph to represent the shared component relationships among the hyperedges and vertices. A hypergraph can be modeled as a

- *Y. Gu, K. Yu, Z. Song and G. Yu are with the Department of Computer Science, Northeastern University, Shenyang 110819, P. R. China.*
  E-mail: guyu, yuge@mail.neu.edu.cn.
  E-mail: yukaiqiang1994, songzhen_neu@163.com.
- *J. Qi and R. Zhang are with the School of Computing and Information Systems, The University of Melbourne, Australia.*
  E-mail: jianzhong.qi, rui.zhang@unimelb.edu.au
- *Z. Wang is with the College of Information Science and Engineering, Ocean University of China, Qingdao, Shandong, China.*
  E-mail: wangzhigang@ouc.edu.cn.

bipartite graph where the original vertices and hyperedges constitute the vertex set. The message communication will only occur along the edges in this bipartite graph when conducting iterative computation. Therefore, we aim to construct compressed bipartite graphs named intersection graphs based on the shared relationships, so that the smallest number of edges need to be reserved. When converting a bipartite graph into an intersection graph, our goal is to minimize the edges in the intersection graph while guaranteeing the correctness of computations.

Based on the intersection graph, we propose a distributed iterative hypergraph processing framework $Hyraph$ on top of Giraph [8]. It provides ease of use interfaces for users to implement various hypergraph analysis algorithms. For efficiently distributed computations, we propose a heuristic partitioning algorithm to divide the intersection graph among different workers. The basic idea is to assign vertices (hyperedges) that share the incident hyperedges (vertices) into the same partitions. In this way, the shared relationships among the same shared hyperedge (vertex) list can be preserved. In addition, vertices, hyperedges and their corresponding outgoing edges will be distributed in a balanced manner.

We now summarize our contributions below:

- We analyze the shared components among hyperedges and vertices and propose to use intersection graphs to represent hypergraphs.
- We propose algorithms to compute intersection graphs from hypergraphs and a partitioning algorithm to partition the intersection graphs for distributed hypergraph processing.
- We further propose a distributed hypergraph processing framework $Hyraph$ to facilitate computations based on our intersection graphs. Extensive experiments on real datasets show that the response time can be reduced significantly in comparison with the state-of-the-art hypergraph frameworks.

The rest of the paper is organized as follows. Sec.2 reviews related work and introduce necessary preliminary knowledge. Sec.3 presents an overview of $Hyraph$. Sec.4 shows how to convert a hypergraph to an intersection graph and partition the intersection graph for distributed execution. In Sec.5, we detail the implementation of $Hyraph$. We then report the evaluation results in Sec.6 and finally conclude this paper in Sec.7.

2 Related Work and Preliminaries

In this section, we review studies on the processing systems and partitioning algorithms of graphs and hypergraphs. Also, some key preliminaries related to hypergraph processing are introduced for better understanding our contributions.

2.1 Graph Processing

Iterative large-scale graph processing systems have been studied extensively in recent years. $Pregel$ developed by Google [9] as one of the early representative pioneers employs a vertex-centric message-passing design in distributed environments. That is also inherited by its open-source implementation $Giraph$ [8]. $Pregel$ has been driving much of the research on enhancing performance in perspectives of communication [10], [11], convergence [12], [13], and disk-based extension in centralized and distributed settings [14], [15], [16]. Another system $GraphX$ [17] provides APIs similar to $Pregel$ on top of a general-purpose $Spark$ [18] to utilize its ecosystem. Also, some related works like $GBASE$ [19] and $GraphMat$ [20] try to compute graphs with matrix-based methods.

More recently, $Gimini$ [21] extends the hybrid push-pull computation model from shared-memory to distributed scenarios. It adopts a sparse-dense dual engine design, in which computation and communication are handled differentially in the two modes. $KickStarter$ [22] attempts to accelerate computations over streaming graphs for a class of monotonic algorithms. When changes happen, $KickStarter$ incrementally maintains existing results by quickly detecting the affected range and then correcting invalid results. $GraphBolt$ [23] is a dependency-driven streaming graph processing system that aims to minimize redundant computations upon graph mutation. Compared with $KickStarter$, it removes the monotonic constraint.

2.2 Hypergraph Processing

Typically, hypergraphs are first converted into graphs and then processed using the graph frameworks mentioned above. Our study also follows this paradigm, but the novelty is that we convert hypergraphs into special types of graphs, i.e., intersection graphs, in order to optimize communication overheads. Below, we first review two existing classic approaches of converting hypergraphs: *star-expansion* (SE) and *clique-expansion* (CE), and then discuss hypergraph processing systems and well-known learning algorithms.

Star-expansion. In SE, each hyperedge h is replaced by a new vertex that connects to the vertices of h. The resultant graph has two types of vertices, one from the vertices of the original hypergraph and the other converted from hyperedges of the original hypergraph. These two types of vertices form a bipartite graph. Fig.1(a) shows a hypergraph, where the black dots represent the vertices and the ellipses represent the hyperedges. The resultant bipartite graphs computed by SE is shown in Fig.1(b), where the black dots are vertices from the hypergraph, and the red dots are new vertices converted from hyperedges of the hypergraph.

Once a hypergraph has been converted, an iterative hypergraph processing algorithm can be run as follows. Here, we call an iteration a superstep. During two consecutive supersteps, vertices and hyperedges are updated separately in each superstep. Specifically, we take the label propagation algorithm as an example. In superstep i, vertex v updates its value using the messages received from the incident hyperedges in superstep $i-1$, based on Eq.1. After v has been updated, new messages are generated based on Eq.2 and sent to its incident hyperedges so that the latter can update their values via Eq.3 in superstep $i+1$. Of course newly updated hyperedges will continuously generate messages via Eq.4 for incident vertices. This process continues until a predefined number of iterations has been reached or the algorithm converges, e.g., the vertex/hyperedge values do not change any more. However, SE doesn't consider the unique structure of hypergraphs.

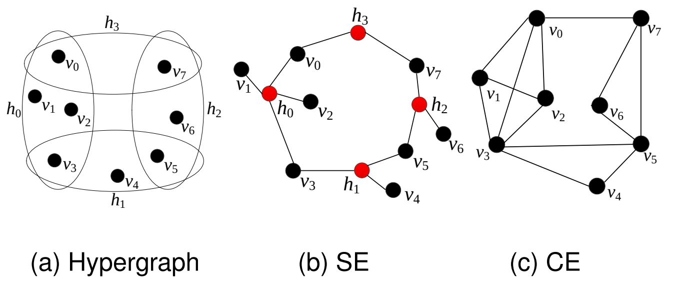


Fig. 1. Converting a hypergraph with SE and CE

$$v.val = \begin{cases} v.id & itr = 1 \\ max(h.msgs) & itr > 1 \end{cases} \quad (1)$$

$$v.msg = v.val \quad (2)$$

$$h.val = max(v.msgs) \quad (3)$$

$$h.msg = h.val \quad (4)$$

Clique-expansion. In CE, each hyperedge is expanded into a clique formed by all vertices in the hyperedge, i.e., an edge is added to connect every pair of vertices in a hyperedge. Fig.1(c) shows a graph converted from the hypergraph shown in Fig.1(a) via CE. Since no unique entity is used in the converted graph to represent the hyperedges, the CE method cannot be applied straightforwardly to hypergraph processing algorithms that need to update hyperedge values. For better understanding, we still use the author-cooperation hypergraph given in Introduction. Clearly, in the converted graph output by CE, it just maintains the relationships among authors, while the information between authors and papers has been discarded. Further, the CE method generates $O(\sum_{h \in H} h.deg^2)$ additional edges where H represents the set of hyperedges and $h.deg$ represents the degree of each edge. This may lead to substantial storage and communication overheads. For example, both $MESH$ [24] and $HyperX$ [7] have tested CE and SE, and reported that CE can be slower up to 10 times than SE.

Hypergraph Processing Systems. $HyperX$ [7] is a system that processes hypergraphs without converting them into graphs. It draws upon optimization techniques of $GraphX$ [17] and is also built on top of $Spark$ [18]. $HyperX$ stores hypergraph data in *resilient distributed datasets* (RDD). There are two main differences between $HyperX$ and graph processing frameworks: (1) In addition to the vertex-centric programming model widely used in graph processing frameworks, $HyperX$ also provides a hyperedge-centric programming model; (2) $HyperX$ considers vertices and hyperedges at the same time when partitioning a hypergraph, which extends traditional vertex-cut and edge-cut techniques. Again, $HyperX$ doesn't take the structure of hypergraphs into account, yielding suboptimal performance. $MESH$ [24] is implemented on top of a graph processing system $GraphX$ [17], which mainly concentrates on three aspects, namely ease of use, scalability, and ease of implementation. While $MESH$ can provide good expressiveness and flexibility, it has no advantage in computation efficiency. Shun et al. [25] propose many parallel hypergraph algorithms in centralized environments, which are extended from classic graph algorithms. However, due to hardware limitations, the centralized system lacks scalability and may cause inefficiency for increasingly larger hypergraphs.

Hypergraph Learning. Considerable hypergraph learning algorithms have been studied in various scenarios. Berlt et al. [1] model the web with a hypergraph which is derived from the web graph by dividing the set of web pages into non-overlapping blocks. Random Walks [26] on the hypergraphs can identify the items that may be of interest to a user. Gao et al. [3] propose a hypergraph shortest path algorithm that allows dynamic changes of the values and topologies of a hypergraph. Somu et al. [4] propose to use hypergraph to represent the complex relationships among features to solve feature selection problems. These works focus on designing effective hypergraph algorithms, which is different from our efforts about optimizing hypergraph processing frameworks.

2.3 Graph and HyperGraph Partitioning

This subsection outlines key techniques of partitioning graphs and hypergraphs, followed by a brief discussion about dynamic partitioning.

Graph Partitioning. There exist two important research branches for graph partitioning: streaming and non-streaming. The latter scans the input graph multiple times, which is time-consuming but can gradually refine partitioning quality. The well-known representatives include METIS [27], PuLP [28] and their parallel/distributed variants [29], [30]. While, the former assumes graph data arrive streamingly and then computes the placement of newly arrived data based on the distribution of already placed data. This largely improves partitioning efficiency although it compromises the quality. Many widely-used techniques employ such a design, like centralized LDG [31], FENNEL [32], and distributed PowerGraph [11] and PowerLyra [33]. These techniques mentioned above work well in the traditional homogeneous scenario where graph computations transfer messages among only vertices, which is different from the heterogeneous hypergraph computations where messages are exchanged between vertices and hyperedges. The two scenarios naturally generate different quality metrics from perspectives of communication reduction and load balance, and then the heuristics employed are technically orthogonal.

Hypergraph Partitioning. A few studies propose hypergraph partitioning algorithms, mainly including *multilevel* hMetis [34], PaToH [35], Mondriaan [36], and the distributed variants Parkway [37] and Zoltan [38]. They gradually coarsen input hypergraph, divide the coarsest one, and then project the partition back towards the original hypergraph. Recently, some partitioning tools are also been released: rFM [39] supports replication and relocation of vertices; UMPa [40] allows optimizing multiple objective functions simultaneously. All these hypergraph partitioning solutions adopt the k-way partition strategy which aims to partition vertices into k parts such that more vertices belonging to the same hyperedges can be assigned into the same part. The goal of such partitioning is to solve the applications such as VLSI design, data storage of large databases on disks and transportation management, instead of facilitating

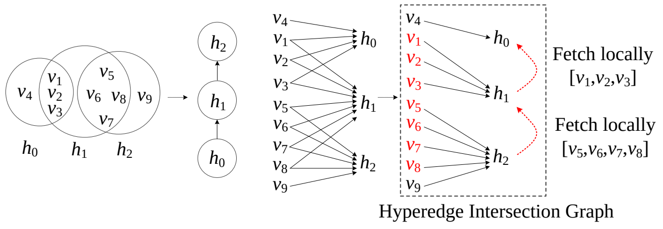


Fig. 2. The hyperedge intersection graph

consequent iterative computation based on SE. Besides, the balance of the hyperedges is not considered in these methods. We are also aware that Turk et al. [41] model social networks with temporal hypergraph to answer queries. They partition the hypergraph by predicting possible user actions, in order to reduce the average query span and balance the server load. However, the idea cannot be extended to general hypergraph computations.

Dynamic Partitioning. Wang et al. [42] discuss the definition, topological structure and systems for the time-dependent graphs. Given an original partition and updates to a graph, the partitioners like [43], [44], [45] incrementally compute changes to the old partition instead of re-partitioning, which avoids high partitioning costs.

3 Solution Overview

Fig.3 shows the four layers framework of $Hyraph$. The pre-processing layer converts original hypergraph into intersection graph by algorithms introduced in Sec.4. The storage layer divides an intersection graph into k partitions and then stores them onto HDFS. The execution layer implements hypergraph processing on top of $Giraph$ with ease of use APIs for programming various application algorithms. In the following, we first present the basic concepts and our key observation that enables hypergraph processing using intersection graphs. The detailed techniques and implementation will be given in Sec. 4 and Sec. 5, respectively.

3.1 Definitions

Let $G = (V, H)$ be a hypergraph where V is the vertex set and H is the hyperedge set. Each hyperedge $h \in H$ is a non-empty subset of V. For a hyperedge $h \in H$, its degree, denoted by d_h, is the number of vertices in h. For a vertex $v \in V$, its degree, denoted by d_v, is the number of hyperedges that are incident to v. We use $\Gamma(v)$ and $\Gamma(h)$ to denote the set of incident hyperedges of v and the set of incident vertices of h respectively.

Intersection Graphs. An intersection graph is modeled as $IG = (G_H, G_V)$, where G_H and G_V represent the hyperedge and vertex intersection graphs, respectively. The hyperedge intersection graph is denoted by $G_H = (V \bigcup H, E_H)$, and E_H indicates the set of edges. E_H contains two categories, i.e., E_{rmt} and E_{loc}, where E_{rmt} is the remote edge set, and E_{loc} is the local one. Similarly, the vertex intersection graph is denoted by $G_V = (V \bigcup H, E_V)$, where the edge set is denoted by $E_V = (E_{rmt}, E_{loc})$. The hyperedge intersection graph is shown in Fig.2

Shared hyperedge and shared vertex. We call two hyperedges h_i and h_j *shared hyperedges* if their sets of incident vertices overlap. Among all the shared hyperedges of h_i, the one that has the largest number of common incident vertices with h_i is called the *dominant shared hyperedge* of h_i, denoted as $\theta(h_i)$. Similarly, we call two vertices v_i and v_j *shared vertices* if their sets of incident hyperedges overlap. Among all the shared vertices of v_i, the one that has the largest number of common incident hyperedges with v_i is called the *dominant shared vertex* of v_i, denoted as $\theta(v_i)$.

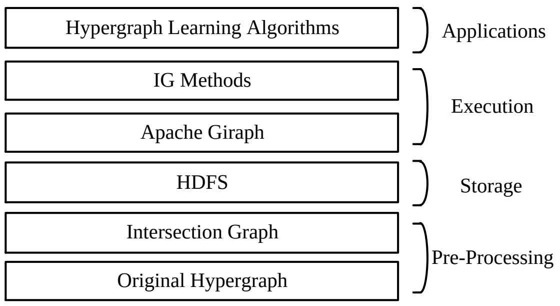


Fig. 3. The overall framework of Hyraph

Final dominant shared hyperedge and final dominant shared vertex. The *final dominant shared hyperedge* $\theta'(h_i)$ of hyperedge h_i is either the dominant shared hyperedge $\theta(h_i)$ of hyperedge h_i or empty, depending on whether a ring occurs when constructing a shared hyperedge list. Similarly, the *final dominant shared vertex* $\theta'(v_i)$ of vertex v_i is either the dominant shared vertex $\theta(v_i)$ of vertex v_i or empty, depending on whether a ring occurs when constructing a shared vertex list.

Shared hyperedge list and shared vertex list. Given a hyperedge h_i, we can compute a series of dominant shared hyperedges as $h_i, \theta(h_i), \theta(\theta(h_i)), \theta(\theta(\theta(h_i))), ..., \theta^n(h_i)$. Let the last hyperedge in this series be h_j. Then, we call this series the *shared hyperedge list* (SHL) between h_i and h_j. Similarly, we can compute a series of dominant shared vertices between v_i and v_j. We call such a series the *shared vertex list* (SVL) between v_i and v_j. The last hyperedge in an SHL is denoted as $hLast$ and the last vertex in an SVL is denoted as $vLast$.

Base hyperedge and base vertex. If $\theta'(hLast)$ does not exist, $hLast$ is the *base hyperedge* for all hyperedges in the current SHL; Otherwise, the base hyperedge of $\theta'(hLast)$ is the *base hyperedge* for all hyperedges in the current SHL. The *base hyperedge* of hyperedge h is denoted as $\beta(h)$. Similarly, the *base vertex* of vertex v is denoted as $\beta(v)$.

Symmetry structure of hypergraphs. We observe that hyperedges and vertices are structurally symmetrical. This is illustrated with Fig.4. On the left of Fig.4, the hypergraph is represented as a set of hyperedges, each of which is a set of vertices. The right of Fig.4 is a redraw of the same hypergraph, and we can also regard each vertex as a non-empty subset of hyperedges. The symmetry between hyperedges and vertices of hypergraph structure makes it possible to handle hyperedges and vertices in the same way using intersection graphs, which is detailed in the following sections. The commonly used symbols are listed in Tab.1.

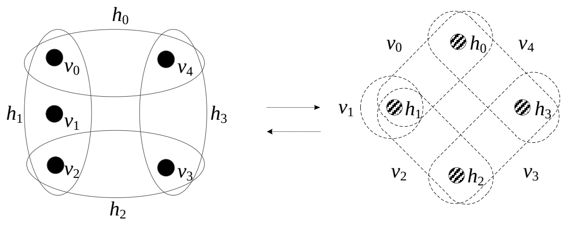


Fig. 4. Symmetry structure of a hypergraph

TABLE 1
Symbols and explanations

Symbols	Explanations
$\Gamma(h)$	The set of incident vertices of hyperedge h
$\Gamma(v)$	The set of incident hyperedges of vertex v
$\Delta(h_i, h_j)$	Intersection of hyperedge h_i and h_j
$\Delta(v_i, v_j)$	Intersection of vertex v_i and v_j
$\theta(h)$	The dominant shared hyperedge of hyperedge h
$\theta'(h)$	The final dominant shared hyperedge of hyperedge h
$\theta(v)$	The dominant shared vertex of vertex v
$\theta'(v)$	The final dominant shared vertex of vertex v
$\beta(h)$	The base hyperedge of hyperedge h
$\beta(v)$	The base vertex of vertex v

3.2 Key Observation

In hypergraphs, each hyperedge is a non-empty subset of vertices. Hyperedge h_i and hyperedge h_j can have common vertices. In this case, we say that h_i *intersects* h_j. Such intersections form the basis of constructing intersection graphs to represent hypergraphs. Take Fig.2 as an example. There are three hyperedges h_0, h_1, and h_2 with different sets of vertices. For h_0, the dominant shared hyperedge is h_1, that is, $\theta(h_0) = h_1$ and the intersection between h_0 and h_1 is $\{v_1, v_2, v_3\}$. For h_1, the size of the intersection between h_1 and h_2 is larger than that between h_1 and h_0, and hence the dominant shared hyperedge of h_1 is h_2. Recall that in a specific iteration, either every vertex sends messages to its incident hyperedges or every hyperedge sends messages to its incident vertices. In a bipartite graph converted from a hypergraph (via SE described in Section 2.2), we call the incident hyperedges of vertex v the **outgoing edges of** v and the incident vertices of hyperedge h as the **outgoing edges of** h. In Fig.2, vertices in $\{v_1, v_2, v_3, v_4\}$ need to send messages to h_0, vertices in $\{v_1, v_2, v_3, v_5, v_6, v_7, v_8\}$ need to send messages to h_1 and vertices in $\{v_5, v_6, v_7, v_8, v_9\}$ need to send messages to h_2. Since $\{v_1, v_2, v_3\}$ are shared by h_0 and h_1, and $\{v_5, v_6, v_7, v_8\}$ are shared by h_1 and h_2, if we can assign h_0, h_1 and h_2 into the same partition, the messages generated by $\{v_1, v_2, v_3\}$ only need to be sent to h_1 and the messages generated by $\{v_5, v_6, v_7, v_8\}$ only need to be sent to h_2, instead of both. Hyperedges h_0 and h_1 can fetch these shared messages within the partition directly. This avoids redundant message operations. Note that the analysis of the shared vertices is similar because of the symmetry of hypergraphs.

4 Intersection Graph Computation

In this section, we discuss the key steps of generating intersection graphs and the corresponding partition method.

4.1 Overview of Intersection Graph Converting

To benefit from the shared hyperedges and vertices in hypergraphs, our basic idea is as follows. Given a hypergraph with $|H|$ hyperedges and $|V|$ vertices, we number the hyperedges from 0 to $|H| - 1$ and the vertices from $|H|$ to $|H| + |V| - 1$ for simplicity. We use curH and curV to represent the hyperedge and the vertex that is currently being processed; curH starts from 0 while curV starts from $|H|$. Because of the symmetry of the hypergraph structure, we use the same approach to process hyperedges and vertices.

First, we need to acquire the dominant shared hyperedge/vertex for each hyperedge/vertex. The hyperedge which has the most common vertices with curH is the dominant shared hyperedge of curH. Two basic approaches are introduced for this stage which will be introduced later in Sec.4.2. Second, based on shared hyperedges/vertices, we estimate the number of outgoing edges for each hyperedge and vertex in the intersection graph, which is used to ensure the balance of the number of messages sent by every worker. Clearly, we need to control the total number of outgoing edges of hyperedges and vertices in each worker. We detail the approach of estimating the number of outgoing edges in Sec.4.3. Third, based on the shared hyperedges and the estimates of the outgoing edges of every hyperedges and vertices, we are able to generate shared hyperedge lists and shared vertex lists, both of which constitute an intersection graph. We detail the approach of generating shared hyperedge/vertex lists in Sec.4.4.

After accomplishing the constructing of shared hyperedge lists and shared vertex lists, we finally get the intersection graph converted from the original hypergraph. We propose a heuristic partitioning algorithm to partition the resulted intersection graph and a second-order intersection graph to further enhance the sharing effect in Sec.4.5.

We should stress that there exist two scenarios when using intersection graphs, i.e., reusable and non-reusable. We firstly briefly introduce several classic reusable scenarios. CGP (Concurrent Graph Processing) states a classic scenario where many different graph jobs are run over the same input graph, which clearly provides opportunities to share common operations and data (of course including our intersection graphs). Many efforts have been devoted into this issue [46], [47]. Another is two-phased partitioning [48] where the input graph is offline over-partitioned into fine-grained m parts and then can be re-organized and assigned onto k workers in a quickly online manner when a job arrives. Also, for evolving graphs, we can build the intersection graph and partition it for the initial snapshot. Then, we can incrementally maintain existing results for the subsequently arrived snapshots. Secondly, we optimize the construction process by a pruning technique so that its cost can be acceptable even in the non-reusable scenario. Alg.1 summarizes the construction procedures above.

4.2 Dominant Shared Hyperedge/Vertex Computation

Two approaches are proposed for this procedure, and the hyperedge case is mentioned as an example. The first basic approach is to find all appearing hyperedge pairs (h_i,h_j), and then select neighboring hyperedges with the maximal shared size as the dominant shared hyperedge. Assume

Algorithm 1: ConvertToIG

```
Input : A hypergraph $G$
Output: An intersection Graph $IG$
foreach hyperedge $h$ do
    //Get the dominant shared hyperedge for $h$
    $hSharedMap \leftarrow$ getDSH ($h$)
foreach hyperedge $h$ do
    //Get the dominant shared vertex for $v$
    $vSharedMap \leftarrow$ getDSV ($v$)
$hEMap \leftarrow$ getEstimatesForV ($hSharedMap$)
$vEMap \leftarrow$ getEstimatesForH ($vSharedMap$)
/*Generate shared hyperedge lists*/
foreach hyperedge $h$ do
    if !processed ($h$) then
        //Algorithm 2
        SHLs $\leftarrow$ buildSHLs ($h, hSharedMap, hEMap$)
/*Generate shared vertex lists*/
foreach vertex $v$ do
    if !processed($v$) then
        SVLs $\leftarrow$ buildSVLs ($v, vSharedMap, vEMap$)
partitionHyperedges(SHLs)//Algorithm 4
partitionVertices(SVLs)
```

the average degrees of vertices and hyperedges are $\overline{d}_H$ and $\overline{d}_V$ separately. Since the sets of vertices contained in hyperedges are stored by HashSet in initial hypergraphs, we can describe the time complexity as $O(\overline{d}_H \cdot \overline{d}_V \cdot |H| \cdot \overline{d}_H)$ for the hyperedge case, i.e., $O(\overline{d}_H^2 \cdot \overline{d}_V \cdot |H|)$. The total time complexity is $O(\overline{d}_H^2 \cdot \overline{d}_V \cdot |H|) + \overline{d}_V^2 \cdot \overline{d}_H \cdot |V|)$, which is clearly unacceptable for big graphs. To make it feasible, we present another improved approach. For the hyperedge pair (h_i,h_j), if $h_i \in \Gamma(v_k)$ and $h_j \in \Gamma(v_k)$, v_k is the shared part between h_i and h_j. Motivated by this, we traverse the vertex set to calculate the intersection size of hyperedge pairs. The time complexity is decreased to $O(\sum_v^{|V|} d_v^2)$. Furthermore, real graphs usually have power-law degree distribution. We find that several vertices with very large degrees can seriously affect the efficiency. Based on our test, the vertex with the degree of 40k takes nearly 0.8B calculations. For better performance, we propose to **prune such high-degree vertices when computing hyperedge shared pairs**. That is reasonable because pruned vertices tend to appear in quite a lot hyperedge pairs, which contribute less to finding the dominant shared hyperedge. Note that such vertices still participate in sharing in intersection graph. The pruning technique can decrease the entire time complexity to $O(\sum_v^{|V_p|} d_v^2 + \sum_h^{|H_p|} d_h^2)$, where $|V_p|$ and $|H_p|$ are the sizes of the vertex set and hyperedge set after pruning, respectively. Since the average and maximal degrees in V_p and H_p are extremely less than those in V and H, we achieve an excellent effect.

4.3 Obtaining the Estimations of Outgoing Edges

For the sake of balancing the workload in each worker, it is necessary to estimate the outgoing edges of each hyperedge and vertex. From Sec.3.2 we can see that when taking the shared hyperedges into account, the outgoing edges of the vertices are no longer the same as the original bipartite graph. Let's take vertex v_1 as an example. In the original hypergraph, the outgoing edges of v_1 are $\{h_0, h_1\}$, which are exactly the incident hyperedges of v_1. When considering the shared relationships, the outgoing edges of v_1 become $\{h_1\}$, which is a subset of $\Gamma(v_1)$. When dealing with the shared hyperedges, vertices need to send messages to corresponding hyperedges. Therefore, the outgoing edges of those vertices need to be determined. Similarly, the outgoing edges of hyperedges can be determined when processing vertices. Therefore, the total number of outgoing edges from all hyperedges can be determined only after all the vertices are processed. And only after all the hyperedges are processed can the total number of the outgoing edges of the vertices be determined. However, when generating shared hyperedge lists, we need to acquire the total amount of the outgoing edges in the current shared hyperedge lists to prevent the number from being too large.

Fortunately, we can get the estimated values of the amount of outgoing edges of hyperedges and vertices. When estimating the amount of outgoing edges of hyperedges, the information of dominant shared vertices will be used and vice versa. The estimating procedure of hyperedges and vertices is the same. Here we take the hyperedges' estimating as an example. We start with the hyperedge with id 0 to sequentially process every hyperedge according to their ids, and denote the hyperedge that is being processed as curH. If curH does not have the dominant shared hyperedge, all vertices in Γ(curH) need to add outgoing edges to curH, that is, the number of outgoing edges of each vertex in Γ(curH) needs to be increased by 1; otherwise, there exists the dominant shared hyperedge for curH. There are two cases : (1) the id of θ(curH) is larger than the id of curH, which means θ(curH) has not been processed yet. In this case, only the number of outgoing edges of each vertex in Γ(curH) $- \Delta$(curH, θ(curH)) needs to be increased by 1; (2) the id of θ(curH) is less than that of curH, which means θ(curH) has been processed before. In this case, we should add outgoing edges for all vertices in Γ(curH), instead of only for Γ(curH) $- \Delta$(curH, θ(curH)). Otherwise, vertices in Δ(curH, θ(curH)) will lose outgoing edges to curH. The time complexity of this stage is $O(|V| \cdot \overline{d}_V + |H| \cdot \overline{d}_H)$, i.e., $O(|E|)$, where $|E|$ indicates total edge number in the bipartite graph. After all hyperedges have been processed, we can get the sum of all the estimating outgoing edges of vertices, denoted as $VSumEdges$. Similarly, we denote the sum of all the estimated outgoing edges of hyperedges as $HSumEdges$. We set the hyperedges' outgoing edges threshold(HET) as $HSumEdges$ / k and the vertices' outgoing edges threshold(HVT) as $VSumEdges$ / k where k is the number of partitions. When generating the shared hyperedge lists and shared vertex lists, the sum of all the hyperedges' outgoing edges in a certain shared hyperedge list should not exceed HET and VET for the sum of all the vertices' outgoing edges in a certain shared vertex list.

For example, if curH is h_0 and its dominant shared hyperedge is h_1, then $\Delta(h_0, h_1) = \{v_1, v_2, v_3\}$. In this case, since the id of h_1 is larger than the id of h_0, we do not need to add outgoing edges for vertices in $\Delta(h_0, h_1)$. When h_0 has been processed, curH is h_1. If the dominant shared hyperedge for h_1 happens to be h_0, we still have $\Delta(h_1, h_0) = \{v_1, v_2, v_3\}$. In this case, the id of θ(curH) is less than that of curH. If we do not add outgoing edges for the vertices in $\Delta(h_1, h_0)$, neither h_1 nor h_0 will receive messages from vertices in $\{v_1, v_2, v_3\}$, that is, the messages from $\{v_1, v_2, v_3\}$ are lost for h_1 and h_0. Thus, we need to add outgoing edges for $\{v_1, v_2, v_3\}$ to hyperedge h_1.

4.4 Computing Shared Hyperedge/Vertex Lists

For simplicity, we write $\theta(\theta(h_i))$ as $\theta^2(h_i)$. From h_i, we start to search for the dominant shared hyperedge, and the dominant shared hyperedge of the k-th hyperedge h_{i+k} is denoted by $\theta^k(h_i)$. In the procedure of constantly looking for $\theta^k(h_i)$ from the initial hyperedge h_i, we will finally get a series of hyperedges that have shared relationships, all of which constitute a shared hyperedge list (SHL). Note that when constructing a SHL, we need to use the final dominant shared hyperedge $\theta'(h_i)$ for current hyperedge h_i, which is computed based on $\theta(h_i)$, instead of using $\theta(h_i)$ directly. The final dominant shared hyperedge $\theta'(h_i)$ of hyperedge h_i is either the dominant shared hyperedge $\theta(h_i)$ or empty, depending on whether a ring occurs, which is detailed in Sec.4.5. Similarly, the final dominant shared vertex $\theta'(v_i)$ of vertex v_i needs to be computed when generating shared vertex lists. After that, we can get the common vertex set of h_i and $\theta'(h_i)$.

During the procedure of constructing the shared hyperedge lists, the initial hyperedge is denoted as $start$, the shared hyperedge list obtained from $start$ hyperedge is denoted as $list$ and the sum of the outgoing edges of all hyperedges in $list$ is denoted as $outEdgeNum$. All hyperedges are encoded from 0 to $|H|$ and $start$ begins from the first hyperedge to constantly construct the shared hyperedge lists for obtaining the final dominant shared hyperedge and base hyperedge for every hyperedge. During the procedure, we need to guarantee that the sum of outgoing edges and the number of all hyperedges in each shared hyperedge list won't exceed the hyperedge outgoing edges threshold HET and the hyperedge number threshold HT respectively. HT equals to $|H|\ /\ k$ where k is the number of partitions. The procedure of computing the shared hyperedges lists are summarized in Alg.2. Assuming that the hyperedge currently being processed is curH, there are three cases:

Case 1: curH has not been processed, $outEdgeNum$ is less than HET and the number of hyperedges in $list$ is less than HT. We obtain the final dominant shared hyperedge θ'(curH) for curH. If θ'(curH) is not empty, the mapping relationship of curH and θ'(curH) need to be stored and we should add outgoing edges for vertices in Γ(curH) $-$ Δ(curH, θ'(curH)). What's more, curH needs to be added to $list$ and $outEdgeNum$ needs to increase by the estimate of the outgoing edges of curH. Finally, θ'(curH) becomes θ(curH) and the procedure is repeated. Otherwise, the final dominant shared hyperedge θ'(curH) for curH does not exist and we cannot continue to construct the current shared hyperedge list. At this time, curH becomes the base hyperedge of all the other hyperedges in $list$. Suppose hyperedge B is the base hyperedge and all vertices incident to B need to add edges to it. In addition, B might be other hyperedges' base hyperedge, so for those hyperedges whose base hyperedge is B, we need to store the number of hyperedges and the number of outgoing edges for B.

Algorithm 2: BuildSHLs

```
Input  : hyperedge curH, $hSharedMap$, $hEMap$
Output: A shared hyperedge list from hyperedge curH
1  $list \leftarrow \emptyset, outEdgeNum \leftarrow 0, nodeNum \leftarrow 0$
2  while curH $\neq \emptyset$ and !processd(curH) and
   $outEdgeNum < HET$ and $nodeNum$++$< HT$ do
3      $list$.add (curH)
4      $outEdgeNum$ += getEstimateOutEdgeNum (curH)
5      $\theta'$(curH) $\leftarrow$ getFDSH (curH) //Algorithm 3
6      computeIntersection (curH, $\theta'$(curH))
7      if $\theta'$(curH) $= \emptyset$ then
8          curH is the base hyperedge
9          $baseNum$[curH] += $list.size$
10         $edgeNum$[curH] += $outEdgeNum$
11     addOutgoingEdges ($\Gamma$(curH) $-$ $\Delta$(curH, $\theta'$(curH)))
12     curH $\leftarrow$ $\theta'$(curH)
13 if $outEdgeNum > HET$ then
14     if !processed (curH) then
15         curH is the base hyperedge
16         $list$.add (curH)
17         $curHOE$ = getEstimateOutEdgeNum (curH)
18         $baseNum$[curH] += $list.size$
19         $edgeNum$[curH] += $outEdgeNum$ + $curHOE$
20         addOutgoingEdges ($curHOE$)
21     else
22         $hLast$ is the base hyperedge
23         $baseNum[hLast]$ += $list.size$
24         $edgeNum[hLast]$ += $outEdgeNum$
25         addOutgoingEdges ($\Delta$(hLast, $\theta'$(curH)))
26 if $list.size > HT$ then
27     $base \leftarrow$ obtainBase (curH)
28     if $edgeNum[base] + outEdgeNum < HET$ then
29         $base$ is the base hyperedge
30         $baseNum[base]$ += $list.size$
31         $edgeNum[base]$ += $outEdgeNum$
32     else
33         $hLast$ is the base hyperedge
34         $baseNum[hLast]$ += $list.size$
35         $edgeNum[hLast]$ += $outEdgeNum$
36         addOutgoingEdges ($\Delta$($hLast$, $\theta'$(curH)))
```

Case 2: curH has already been processed before. We obtain the base hyperedge β(curH) for curH. If the sum of outgoing edges doesn't exceed HET and the number of hyperedges corresponding to β(curH) doesn't exceed HT, β(curH) is the base hyperedge for all hyperedges in $list$. Otherwise, the last hyperedge $hLast$ in $list$ becomes the base hyperedge for all the other hyperedges in $list$. Since $hLast$ no longer shares vertices with curH, vertices in $\Delta(hLast$, curH) need to add edges to $hLast$.

Case 3: $outEdgeNum$ exceeds the hyperedge outgoing edge threshold HET. The last hyperedge $hLast$ in the current shared hyperedge list becomes the base hyperedge for all other hyperedges in $list$. What's more, vertices in $\Delta(hLast$, curH) need to add edges to $hLast$.

Alg.2 summarizes the procedure of constructing shared hyperedge lists, and we call it the ***Building Shared Hyperedge Lists*** (buildSHLs) algorithm. Let's take the processing procedure of computing shared hyperedge lists as an example to make it more intuitive. In Fig.5, from the initial hyperedge S_1, we constantly look for the final dominant shared hyperedge. When hyperedge E_1 is reached, we have $\theta'(E_1) = \emptyset$, which means E_1 is the base hyperedge for all the hyperedges in SHL_1. The above procedure is repeated for S_2. When E_2 is reached, we have $\theta'(E_2) = \theta'(E_1)$ and E_1 has already been processed, so we can get $\beta(E_1) = E_1$. E_1 is the base hyperedge for all hyperedges in SHL_2. The processes of S_3 and S_4 are similar with S_2. Next, starting from hyperedge S_5, when E_5 is reached, we have $\theta'(E_5) = \theta'(E_3)$. E_3 has been processed and $\theta'(E_3) = \theta'(E_1)$.

If all hyperedges in SHL_5 take E_1 as their base hyperedge, the outgoing edge number will exceed the threshold HET, so E_5 is the new base hyperedge for all other hyperedges in SHL_5. The overall time complexity of Alg.2 is $O(|H| \cdot \overline{d}_H + |V| \cdot \overline{d}_V)$. Since the process of $computeIntersection$ (Line 7 in Alg.2) can complete in the stage of Sec.4.3, the time complexity becomes $O(|H| + |V|)$.

4.5 Final Dominant Shared Hyperedges/Vertices

For the original hypergraph, we can obtain the dominant shared hyperedge for each hyperedge and the dominant shared vertex for each vertex easily. However, the dominant shared hyperedges/vertices cannot be used directly when computing the shared hyperedge/vertex lists. Suppose the dominant shared hyperedge of hyperedge h_i is h_j, that is, $\theta(h_i)$=h_j. Only the vertices in $\Gamma(h_i)$ - $\Delta(h_i, h_j)$ need to add outgoing edges to h_i. When hyperedge h_j is being processed, suppose its dominant shared hyperedge happens to be h_i. The vertices in $\Gamma(h_j)$ - $\Delta(h_j, h_i)$ need to add outgoing edges to h_j. Since $\Delta(h_i, h_j)$ equals to $\Delta(h_j, h_i)$, the vertices in $\Delta(h_i, h_j)$ are lost outgoing edges related to h_i and h_j, which breaks the incident relationships of hyperedges and vertices in the original hypergraph. To avoid this, we need to compute the final dominant shared hyperedge $\theta'(h_i)$ for each hyperedge h_i and the final dominant shared vertex for each vertex v_i. Note that $\theta'(h_i)$ and $\theta'(v_i)$ either equals to $\theta(h_i)$ and $\theta(v_i)$ or empty sets.

Algorithm 3: GetFDSH

```
Input  : hyperedge curH, $preHShareMap$, $hShareMap$
Output: $\theta'$(curH)
1  if !$preHSharedMap$.containsKey(curH ) then
2      $\theta'$(curH) $\leftarrow \emptyset$
3  else
4      $\theta$(curH) $\leftarrow$ $preHSharedMap$.get (curH)
5      if $hShareMap$.containsKey($\theta$(curH )) then
6          if from $\theta$(curH) can reach curH then
7              $\theta'$(curH) $\leftarrow \emptyset$
8          else
9              $\theta'$(curH) $\leftarrow$ $\theta$(curH)
10     else
11         $\theta'$(curH) $\leftarrow$ $\theta$(curH)
12 $hShareMap$.put (curH, $\theta'$(curH))
```

The method of obtaining the final dominant shared hyperedge for each hyperedge is the same as that of the vertex. Here the processing procedure for hyperedges is introduced as an example. For the current hyperedge curH, there are two cases that will make θ(curH) equal to $\emptyset$: (1) curH is the initial hyperedge and curH is an isolated hyperedge,

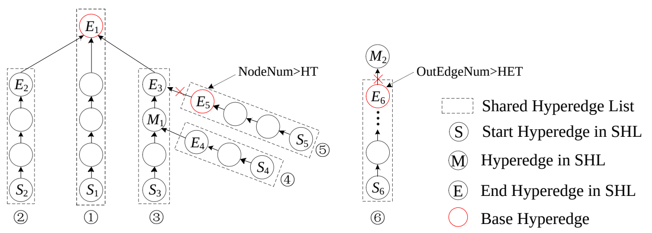


Fig. 5. Computing shared hyperedge lists

as shown in Fig.6(a); (2) curH has the dominant shared hyperedge θ(curH) and θ(curH) has been processed and starting from θ(curH) can eventually reach curH again, that is, a ring is formed. We need to set θ'(curH) to be empty, as shown in Fig.6(b).

For the current hyperedge curH, θ(curH) not equal to $\emptyset$ is also divided into two cases: (1) If the dominant shared hyperedge θ(curH) is not processed, then θ'(curH) is θ(curH), as shown in Fig.6(c); (2) Suppose the dominant shared hyperedge θ'(curH) has been processed, but it can not reach curH from curH, that is, no ring can be formed, as shown in Fig.6(d). Alg.3 summarizes the procedure of obtaining the final dominant shared hyperedges, and we call it the ***Getting Final DSH*** (getFDSH) algorithm. $hShareMap$ in the input of $getfinalDSH$ algorithm is used to store the mapping relationship of hyperedge h_i and its final dominant shared hyperedge $\theta'(h_i)$.

4.6 Partitioning Intersection Graphs

4.6.1 *Partition and Second-order Intersection Graph*

When partitioning an intersection graph, our goal is to balance the hyperedges, vertices, outgoing edges from hyperedges and vertices across different workers. According to the symmetry of hypergraph, only the hyperedge case is considered here. Assuming we partition the hyperedge set H into k disjoint subsets $H_1, H_2, \ldots, H_k$, where k represents worker number and H_i is a subset of hyperedges assigned to the i-th worker. To balance the workload, the number of hyperedge and outgoing edge of each worker needs to satisfy $|H_i| \le \frac{|H|}{k}, 1 \le i \le k$ and $HEdge_i \le \frac{HSumEdge}{k}, 1 \le i \le k$, respectively, where $HSumEdge$ indicates the sum of the hyperedge's outgoing edges. From Sec.4.4 we can realize that both the hyperedge curH and its final dominant shared hyperedge θ'(curH) correspond to the same base hyperedge, while each base hyperedge corresponds to a number of non-base hyperedges. We call each base hyperedge and all the corresponded hyperedges as a hyperedge allocation unit, denoted as HAU. We need to divide all the hyperedges in each allocation unit into the same partition to make full use of the shared relationships among the hyperedges.

As shown in Alg.4, for each base hyperedge B_i, all the corresponded hyperedges are collected to form the current allocation unit HAU. After all the hyperedge allocation units are obtained, the allocation units are sorted in descending order according to the sum of the amount of the outgoing edges to pre-divide the hyperedge allocation units with

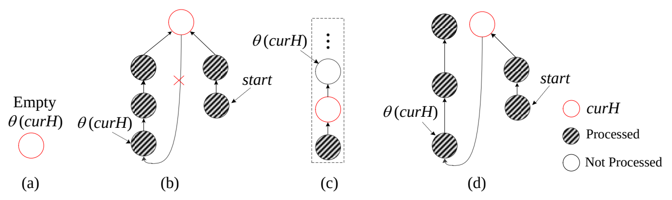


Fig. 6. Computing final dominant shared hyperedges

more outgoing edges. Given k workers, every worker holds 0 outgoing edges at the beginning. We partition the HAU with the most outgoing edges to the worker with the least loaded outgoing edges.

Algorithm 4: PartitionHyperedges

```
Input  : All shared hyperedges lists SHLs
Output: Each hyperedge $h$ and its corresponding
         assigned worker id
1  HAUList ← $\emptyset$
2  foreach $baseH$ in $hBaseMap$ do
3      HAU$_{\text{BaseH}}$.add ($hBaseMap$.get ($baseH$))
4      HAUList.add (HAU$_{\text{BaseH}}$)
5  sort HAUList in descending order according to the
   number of outgoing edges
6  foreach HAU in HAUList do
7      $workerId$ ← getMinOutgoingEdgeLoad ()
8      foreach hyperedge $h$ in HAU do
9          $h$.workerId ← $workerId$
10         foreach $v$ in $\Gamma(h)$ do
11             if vSetOfWorker[workerId].contains($v$) then
12                 $\Gamma(h)$.remove($v$);
13             else
14                 vSetOfWorker[WorkerId].add($v$)
```

Along the logical partitioning process, we make further efforts to improve the performance of intersection graph. We construct the second-order intersection for the various hyperedge (vertex) lists of the same worker (lines 11-14 in Alg.4). Then, we transform second-order sharing into hyperedge (vertex) structure to keep consistency with intersection graph.

The time complexity analysis of this stage is summarized as follows. The first part (Lines 2-4 in Alg.4) divides all SHLs according to baseH, which occupies $O(N_{SHL})$ time complexity, where N_{SHL} represents the number of SHLs. The time complexity of the second part (Lines 5 in Alg.4) is $O(N_{HAU} \cdot log(N_{HAU}))$, where N_{HAU} is the size of HAUList. For the last part (Lines 6-14 in Alg.4), the time complexity is $O(|H| \cdot \overline{d}_H)$. In general, the overall time complexity of the partitioning process is $O(N_{SHL} + N_{SVL} + N_{HAU} \cdot log(N_{HAU}) + N_{VAU} \cdot log(N_{VAU}) + (|H| \cdot \overline{d}_H) + (|V| \cdot \overline{d}_V))$. Since $N_{SHL} < |H|$ and $N_{SVL} < |V|$, we can reduce the time complexity to $O(N_{HAU} \cdot log(N_{HAU}) + N_{VAU} \cdot log(N_{VAU}) + (|H| \cdot \overline{d}_H) + (|V| \cdot \overline{d}_V))$.

4.6.2 *Analysis of Intersection Graphs*

Time Complexity. The time complexity of the process of intersection graph construction and partitioning has been analyzed in related sections. We merge them to generate a total time complexity of $O(\sum_v^{|V_p|} d_v^2 + \sum_h^{|H_p|} d_h^2 + N_{HAU} \cdot log(N_{HAU}) + N_{VAU} \cdot log(N_{VAU}) + |E|)$.

Intersection Graph Benefits. Assume that the final dominant shared hyperedge of h_i is h_j. Without considering the shared relationships, the total number of messages sent to h_i and h_j is $\Gamma(h_i) + \Gamma(h_j)$. When taking the shared relationships into account, the number is $\Gamma(h_i) + \Gamma(h_j) - |\Delta(h_i, h_j)|$. For h_i and h_j, the reduced number of messages is $|\Delta(h_i, h_j)|$. The total number of messages sent to hyperedges in an intersection graph is $\sum_{i=1}^{|H|}(\Gamma(h_i) - |\Delta(h_i, \theta'(h_i))|)$, and the same analysis holds true for the vertex case. As a result, the total number of reduced messages in an intersection graph is $\sum_{i=1}^{|H|}(|\Delta(h_i, \theta'(h_i))|) + \sum_{j=1}^{|V|}(|\Delta(v_j, \theta'(v_j))|)$. After constructing the second-order intersection graph, we partition hyperedges and vertices into two categories respectively, i.e., with/without the final dominant shared hyperedge (vertex). The former can benefit from other shared hyperedge/vertex lists in the same worker, while the latter cannot. Assume that W is the set of workers, θ'_H and θ'_V represent the set of hyperedges and vertices with the final dominant shared ones, respectively. Thus, the total message number of intersection graph is decreased to:

$$\sum_{k \in W}(\sum_{h \notin \theta'_H} |\Gamma(h)| + |\bigcup_{h \in \theta'_H} \Gamma(h)| + \sum_{v \notin \theta'_V} |\Gamma(v)| + |\bigcup_{v \in \theta'_V} \Gamma(v)|).$$

Limitations of Hyraph. Our *Hyraph* mainly tackles the communication-intensive scenario, which is generally the bottleneck of the distributed systems. Although the number of remote messages has been decreased, the total aggregation number of messages keeps the same with bipartite graph. Also, there remains a promotion space for dynamic hypergraphs, i.e., how to share new vertices or hyperedges through intersection graphs and how to keep the structure of intersection graphs when vertices or hyperedges are deleted.

5 Implementation

In this section, we discuss how the intersection graphs are stored and how iterative hypergraph processing algorithms are run based on the intersection graphs.

5.1 Storage of Intersection Graphs

We use adjacency lists to represent intersection graphs. Every hyperedge is stored as a 7-tuple: $\langle h_i, \Delta(h_i, \theta'(h_i)), hType, workerId, deg, E(h_i), B(E(h_i))\rangle$ Here, h_i is the ID of the hyperedge; $\Delta(h_i, \theta'(h_i))$ is the common vertices shared by h_i and $\theta(h_i)$; $hType$ indicates that this entity is a hyperedge; $workerId$ is the ID of the worker that h_i is to be assigned to; deg is the degree of h_i; $E(h_i)$ is the set of outgoing edges ids; $B(E(h_i))$ is the ID set of workers where outgoing edges are assigned.

Every vertex is stored as a 7-tuple: $\langle v_i, \Delta(v_i, \theta'(v_i)), vType, workerId, deg, E(v_i), B(E(v_i))\rangle$ Here, v_i is the ID of the vertex; $\Delta(v_i, \theta'(v_i))$ is the common hyperedges shared by v_i and $\theta(v_i)$; $vType$ indicates that this entity is a vertex; $workerId$ is the ID of the worker that v_i is to be assigned to; deg is the degree of v_i; $E(v_i)$ is the set of outgoing edges ids; $B(E(v_i))$ is the set of the IDs of the workers that the outgoing edges are to be assigned to.

5.2 Iterative Processing Based on Intersection Graphs

Running application algorithms with intersection graphs (IG) is very similar to that with bipartite graphs via SE as described in Sec.2.2. In two consecutive iterations, the vertices and hyperedges update their values based on their neighbouring vertices and hyperedges respectively, until a termination condition is reached, e.g., when there are no messages sent or a predefined number of iterations has been reached. The update procedure is shown in Alg.5.

Algorithm 5: VertexHyperedgeUpdate

```
Input  : Vertex $v$, Messages $msgs$
Output: New vertex value and messages
1  // Vertex Update
2  if turn==0 then
3      if $itr == 0$ then
4          initialize $v.val$
5      else
6          $v.val \leftarrow v$.update $(v.val, msgs)$
7      $outMsg \leftarrow$ generateMessages $(v.val)$
8      foreach incident hyperedge $h$ of $v$ do
9          sendMessage $(outMsg, h, \beta(h))$
10 // Hyperedge Update
11 else
12     $h.val \leftarrow h$.update $(h.val, msgs)$
13     $outMsg \leftarrow$generateMessages $(h.val)$
14     foreach incident vertex $v$ of $h$ do
15         sendMessage $(outMsg, v, \beta(v))$
```

The main difference between IG and the iterative updating procedures of SE lies in how the updated messages (vertex or hyperedge values) are obtained for the vertices and hyperedges. *Hyraph* adopts the push-based message acquisition mechanism, that is, messages generated in superstep i are sent along outgoing edges and these messages will be received and used for updates in superstep $i+1$. For SE, in superstep i, hyperedge h can receive all the messages from its incident vertices that generate and send the messages in superstep $i-1$. In this way, every hyperedge can receive all messages directly. For IG, since we know that the final dominant shared hyperedge $\theta'(h_i)$ (or vertex $\theta'(v_i)$) of a hyperedge h_i (or vertex (v_i)) is assigned to the same partition as h_i (v_i), we can directly fetch the update messages of the vertices shared with $\theta'(h_i)$ since $\theta'(h_i)$ can receive the shared messages between h_i and $\theta'(h_i)$, rather than requesting the update messages from other partitions where those vertices lie. Therefore, the update messages of the shared vertices only need to be fetched once from other partitions, which reduces the communication costs.

We denote the messages that hyperedge h_i receives as $ownMsgs$ and the messages shared with $\theta'(h_i)$ as $sharedMsgs$. These two types of messages constitute the messages used by h_i for updating its value. To enable the shared update procedure above, we create an array $msgArray$ (of size $|H|+|V|$) where each element stores the update message from a vertex or hyperedge. Before starting a superstep, $msgArray$ needs to be filled to support the processing of IG. If hyperedge h_i has shared hyperedge $\theta'(h_i)$, h_i needs to fetch the update messages of $\Delta(h_i, \theta'(h_i))$ from $msgArray$.

Example Hyraph Applications. Since hypergraph-related studies attracted a lot of attentions, many mining algorithms have already been proposed in existing works [7], [24], [25], like connected components, pagerank, label propagation, random walk and shortest path computation. These algorithms are initially designed for regular graph processing but now can be run over hypergraphs with little modification to discover more knowledge or improve the recommendation accuracy. Here, limited by the manuscript length, we give the pseudocodes of connected components as a classic representative (see Alg.6) to show existing algorithms can be easily implemented using our techniques. For more implementation details, please refer to related References [7], [24], [25].

Algorithm 6: ConnectedComponents

```
Input  : Vertex $v$, Hyperedge h, Messages $AllMsgs$
Output: New hyperedge/vertex value and messages
1  // Vertex Program
2  if turn==0 then
3      if itr==0 then
4          $v.val \leftarrow v.id$
5          $outMsg \leftarrow v.val$
6          foreach incident hyperedge $h$ of $v$ do
7              sendMessage $(outMsg, h, \beta(h))$
8      else if $AllMsgs \neq \phi$ & $max(AllMsgs) > v.val$ then
9          $v.val \leftarrow max(AllMsgs)$
10         $outMsg \leftarrow v.val$
11         foreach incident hyperedge $h$ of $v$ do
12             sendMessage $(outMsg, h, \beta(h))$
13     v.voteToHalt()
14 // Hyperedge Program
15 else
16     if $AllMsgs \neq \phi$ & $max(AllMsgs) > h.val$ then
17         $h.val \leftarrow max(AllMsgs)$
18         $outMsg \leftarrow h.val$
19         foreach incident vertex $v$ of $h$ do
20             sendMessage $(outMsg, v, \beta(v))$
21     h.voteToHalt()
```

6 Experiments

Now we run hypergraph analysis jobs on our proposed *Hyraph* framework to evaluate its performance.

6.1 Experimental Settings

The default experiments are run on a cluster of 9 physical machines (8 as workers and 1 as master) connected with Gigabit Ethernet, each of which has 4 Intel(R) Xeon(R) CPUs running at 3.30GHz and is equipped with 16GB memory. *Hyraph* is implemented on top of Apache Giraph 1.2.0.

Datasets and Algorithms As summarized in Tab.2, we use the following four datasets. **Reuters (RE)** dataset contains a set of story-word inclusion relationships extracted from Reuters news stories in the Reuters Corpus, Volume

TABLE 2
Dataset Statistics

Data	$\lvert H\rvert$	$\lvert V\rvert$	H_Avg	H_Max	V_Avg	V_Max	$Edge$
RE[1]	283,911	781,265	213.34	345,056	77.53	1585	60,569,726
FR[2]	1,620,991	7,944,949	14.48	9,299	2.96	1,700	23,479,217
DB[3]	5,699,408	1,973,769	2.44	315	7.04	42,385	13,901,295
OG[4]	8,730,857	2,783,196	37.46	318,240	117.50	40,425	327,037,487

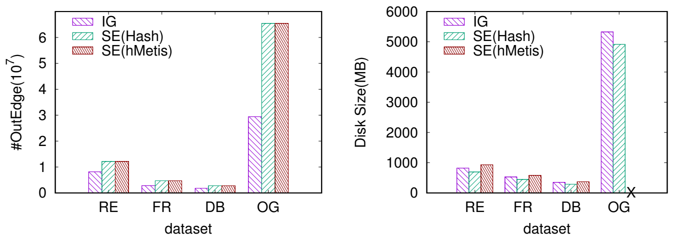
TABLE 3
Intersection Graph Size

Data	$\lvert H\rvert$	$\lvert V\rvert$	H_Avg	H_Max	V_Avg	V_Max	$HOutEdge$	$VOutEdge$
RE	283,911	781,265	118.06	192,444	61.34	857	33,517,552	47,921,289
FR	1,620,991	7,944,949	6.09	7,172	2.14	869	12,441,377	15,838,430
DB	5,699,408	1,973,769	1.48	70	2.38	11,664	9,613,126	8,612,777
OG	8,730,857	2,783,196	12.09	70,282	67.82	22,578	105,556,061	188,756,356

1 (RCV1). Each word can be seen as a vertex while a news story can be seen as a hyperedge. **Friendster (FR)** dataset contains an on-line gaming network. Each game player can be seen as a vertex while a game player group forms a hyperedge. **Dblp (DB)** dataset contains the authorship network of the DBLP computer science bibliography repository. Each author can be seen as a vertex while a publication can be seen as a hyperedge. **Orkut-group (OG)** dataset belongs to affiliation networks, which contains the membership of actors in groups. Each actor can be regarded as a vertex while the group refers to a hyperedge. We implement the following four hypergraph analysis algorithms on $Hyraph$. **PageRank** can compute the page rank for vertices (hyperedges) based on their memberships in different hyperedges (vertices). **Random Walks** can rank vertices and hyperedges,which updates the value of each vertex (hyperedge) based on its incident hyperedges (vertices). **Label Propagation** on a hypergraph can be used to find communities. Each vertex (hyperedge) updates its label to the label that the largest number of its incident hyperedges (vertices) have. **Connected Components** is to find the connected components where each vertex and hyperedge updates its id to the largest id that it receives. Note that the dynamic activation function is always enabled to prevent unchanged vertices/hyperedges from sending redundant messages. In addition, we run 30 iterations for all algorithms and then analyze the runtime performance. Based on our tests, connected components can quickly converge in advance. We then only count the runtime of the valid iterations. For other algorithms which converge slowly, a majority of computations can be completed (i.e., most vertices and hyperedges have been involved in computations) in 30 iterations, which can also validate the effectiveness of our proposals. Some existing works like $MESH$ also employ the similar test policy. **Baseline solutions.** We run the above hypergraph analysis algorithms on $Hyraph$ (denoted by **IG**) and compare them with counterparts running on HyperX [7] (denoted by **HX**). We also compare $Hyraph$ with the SE method (detailed in Section 2.2) which is also implemented on top of Giraph. We denote the SE method with hash partitioning [8] and hMetis [34] by **SE-Hash** and **SE-hMetis** respectively. For HX, we use an iterative partitioning algorithm LPP (10 by the setting of the HyperX paper [7]).

6.2 Features of Intersection Graphs

Tab.3 shows the size of the intersection graphs. Compared with the information of bipartite graphs in Tab.2, we can see that the average degree (H_Avg and V_Avg) and maximal degree (H_Max and V_Max) of hyperedges and vertices in the intersection graphs are consistently smaller than those in the original hypergraphs.



(a) Number of outgoing edges (b) Disk Size
Fig. 7. Comparation of disk size and number of outgoing edges

Further, Fig.7 shows the amount of disk space and the number of outgoing edges in both the bipartite graphs and the intersection graphs for the different datasets where SE (Hash) and SE (hMetis) denote the bipartite graphs partitioned by Hash [8] and hMetis [34] respectively and IG denotes the intersection graph. As Fig.7(a) shows, the number of outgoing edges of the bipartite graphs (Hash) and bipartite graph (hMetis) are the same. In comparison, the numbers of outgoing edges in the intersection graphs are 67%, 60%, 66%, and 45% smaller on the Reuters, Friendster, DBLP and Orkut-group datasets, respectively. Although the number of outgoing edges in the intersection graphs are smaller, we also need to store the partition information for each hyperedge and vertex in the intersection graphs, which causes the disks occupied by the intersection graphs to be slightly larger than the bipartite graphs (Hash), as shown in Fig.7(b). For the RE, FR, DB and OG datasets, the occupied disk space of intersection graphs is 18%, 19%, 21% and 8% larger. Bipartite graphs (hMetis) also need to store the partition information, and hence require more disk spaces than bipartite graphs (Hash).

The distribution of the outgoing edges on the 8 workers is shown in Fig.8(a) where H1 to H8 represent the outgoing edges from hyperedges to vertices on worker 1 to worker 8 and V1 to V8 represent the outgoing edges from vertices to hyperedges on worker 1 to worker 8. We see that we achieve a balanced distribution of outgoing edges. The distribution of the number of hyperedges and vertices on the 8 workers is shown in Fig.8(b). These figures reveal that the intersection graphs converted from different datasets have a balanced distribution of hyperedges and vertices.

6.3 Runtime Evaluation

We compare IG with other approaches mainly over the experimental indicators of partitioning time and executing time. In the last part, we measure the scalability of IG. Note that the partitioning time of IG contains the total time of the intersection graphs construction in Sec.4. We set the thresholds of hyperedges, vertices and their outgoing edges

1.http://konect.uni-koblenz.de/networks/reuters
2.https://snap.stanford.edu/data/com-Friendster.html
3.http://dblp.uni-trier.de/xml
4.http://socialnetworks.mpi-sws.org/data-imc2007.html

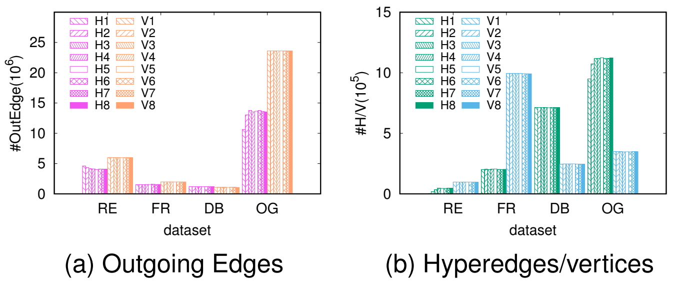


Fig. 8. Distribution of outgoing edges and hyperedges/vertices

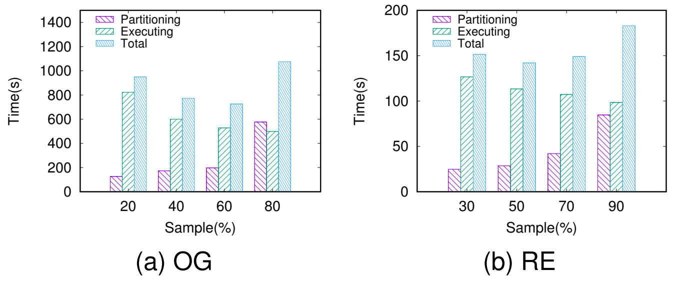


Fig. 9. Time under different sample ratios

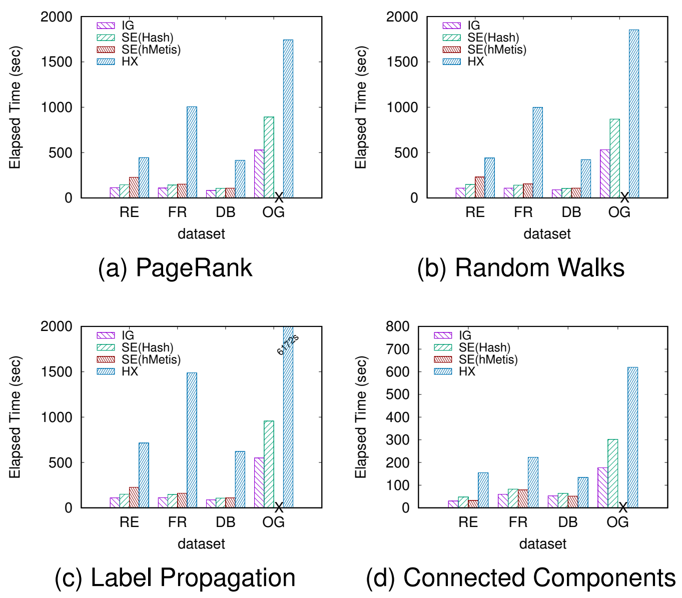


Fig. 10. Executing time

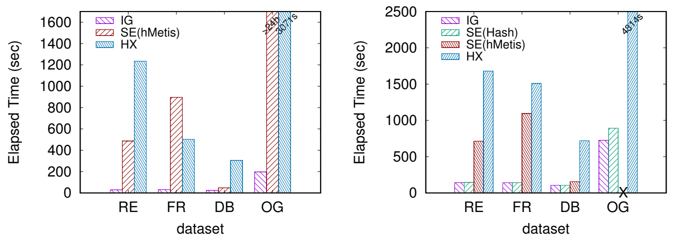
as the total number of them divided by the worker number, which can achieve excellent communication-reduction and acceptable load balance.

We run Pagerank on Orkut-group and Reuters to estimate how the pruning rate affects the effect of the intersection graphs. We select the pruning rate by analyzing the number of the remaining samples. We select the corresponding pruning rates nearby the remaining samples of 25%, 45%, 65%, 85%. As shown in Fig.9, with the sampling rate increasing, the executing time goes down and the partitioning time rises gradually. Experimental results indicate that the executing time is almost immutable when the sample rate s>50%. As a result, s=50% often shows the optimal total time, which corresponds to $0.2\times$ average degree for the two datasets. For the other two datasets, since the average degree is small, the optimal results appear in the average degree. For simplicity, we denote average degree as $\overline{d}$. We give an empirical criterion that the pruning rates of $0.2\times$ and $1\times$ $\overline{d}$ are set for $\overline{d} >= 30$ and $\overline{d} < 30$ respectively, and our experimental settings also follow this rule.

As can be seen from Fig.10, the executing time of IG is consistently superior to that of SE (Hash), SE (hMetis) and HyperX on all combinations of algorithms and datasets. In the connected components algorithm, we achieve similar performance with SE (hMetis), but the connected component consumes much more partition time. Note that we label the executing time of hMetis on OG with "x" because we fail to get results within 24 hours. Specifically, SE works better when using hash partitioning compared with hMetis when running most algorithms on these datasets. This is because hMetis is initially proposed for some classic problems like VLSI design and sharding storage of distributed databases where the complex relationship between the two kinds of entities (vertices and hyperedges) are not considered, which largely limits the communication gain for iterative computations. More importantly, hMetis achieves poor load balance for hypergraphs due to converting as many as possible remote messages to the local ones, which generates waiting costs and then offsets the communication gain. On RE, considering executing time, IG is 21.5%∼37.0% faster than that of SE (Hash), 6.6%∼50.2% faster than that of SE (hMetis) and 73.0%∼84.6% faster than that of HX. On FR, the improvements are 19.0%∼28.0%, 24.3%∼30.4% and 82.4%∼92.5%, respectively. On DB, we still achieve performance gain by 14.4%∼18.6%, 18.7%∼20.9% and 55.6%∼86.0% respectively. The exception is that for the connected components algorithm, SE (hMetis) achieves a comparable result with IG. Finally, on OG, the executing time of IG is 38.9%∼42.5% less than that of SE (Hash), and 69.7%∼91.1% less than that of HX.

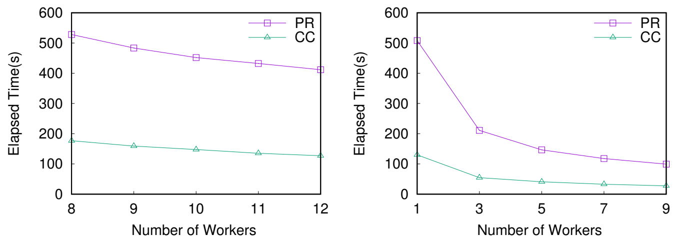
Fig.11(a) describes the partitioning time of IG, SE (hMetis) and HX, while we omit that of SE (Hash) since it does not require additional partition. Fig.11(b) reveals the result of total time on all of the four datasets with Pagerank for the three existing approaches and our IG. With the limitation of space, we just show the results of PageRank, and others have the similar results. The experiments demonstrate that our method shows the best performance on all datasets. Even for the end-to-end performance with partitioning and executing times, we can still produce the comparable effect with the SE-Hash method. More importantly, our IG tends to be more effective over larger datasets with fewer workers. This is principally because in this scenario more intensive communication is incurred, while partitioning time does not increase heavily with the pruning scheme.

Since the effectiveness of IG might be affected when varying the number of workers, it is essential to run the scalability test to explore the runtime change pattern. We run PageRank and Connected Components on datasets OG and RE starting with 8 workers and 1 worker respectively. The worker number settings are the least requirements by the corresponding datasets, so as to normally perform iterative computations [49]. Fig.12 shows experiments on the scalability of IG. We can see that with the increase of the number of workers, the efficiency of IG could degrade,



(a) Partitioning Time (b) Total Time

Fig. 11. Partitioning time and total time



(a) Scablability on OG (b) Scablability on RE

Fig. 12. Scalability

since it reduces the chance of intersection. Consequently, under the condition that the number of workers can meet the execution, fewer workers make the intersection graph gain more performance.

7 Conclusion

In this paper, we observe the symmetry of the hypergraph structure and propose to compute intersection graphs based on hyperedges and vertices, which enables hypergraph being partitioned with fewer outgoing edges across partitions. Accordingly, we design a hypergraph partitioning algorithm and an iterative distributed hypergraph processing framework named $Hyraph$. Experiments on real datasets show that $Hyraph$ outperforms state-of-the-art frameworks.

8 Acknoledgement

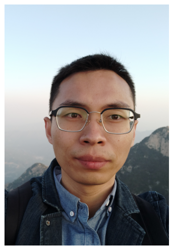
This work is supported by the National Key R&D Program of China (2018YFB1003404) and the National Natural Science Foundation of China (61872070 and 61902366).

References

[1] Klessius Berlt, Edleno Silva de Moura, André Luiz da Costa Carvalho, Marco Cristo, Nivio Ziviani, and Thierson Couto. A hypergraph model for computing page reputation on web collections. *XXII Simpsio Brasileiro De Banco De Dados*, pages 35–49, 2007.
[2] Jiajun Bu, Shulong Tan, Chun Chen, Can Wang, Hao Wu, Lijun Zhang, and Xiaofei He. Music recommendation by unified hypergraph:combining social media information and music content. In *ACM MM*, pages 391–400, 2010.
[3] Jianhang Gao, Qing Zhao, Wei Ren, Ananthram Swami, Ram Ramanathan, and Amotz Bar-Noy. Dynamic shortest path algorithms for hypergraphs. *IEEE/ACM Transactions on Networking*, 23(6):1805–1817, 2012.
[4] Nivethitha Somu, M. R. Gauthama Raman, Kirthivasan Kannan, and V. S. Shankar Sriram. Hypergraph based feature selection technique for medical diagnosis. *Journal of Medical Systems*, 40(11):239, 2016.
[5] Shulong Tan, Ziyu Guan, Deng Cai, Xuzhen Qin, Jiajun Bu, and Chun Chen. Mapping users across networks by manifold alignment on hypergraph. In *AAAI*, pages 159–165, 2014.
[6] Jin Huang, Rui Zhang, and J. X. Yu. Scalable hypergraph learning and processing. In *ICDM*, pages 775–780, 2015.
[7] Wenkai Jiang, Jianzhong Qi, Jeffrey Xu Yu, Jin Huang, and Rui Zhang. Hyperx: A scalable hypergraph framework. *IEEE Transactions on Knowledge and Data Engineering*, 31(5):909–922, 2019.
[8] Avery Ching, Sergey Edunov, Maja Kabiljo, Dionysios Logothetis, and Sambavi Muthukrishnan. One trillion edges: graph processing at facebook-scale. *PVLDB*, 8(12):1804–1815, 2015.
[9] Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel:a system for large-scale graph processing. In *SIGMOD*, pages 135–146, 2010.
[10] Yuanyuan Tian, Andrey Balmin, Severin Andreas Corsten, Shirish Tatikonda, and John McPherson. From "think like a vertex" to "think like a graph". *PVLDB*, 7(3):193–204, 2013.
[11] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: distributed graph-parallel computation on natural graphs. In *OSDI*, pages 17–30, 2012.
[12] Yanfeng Zhang, Qixin Gao, Lixin Gao, and Cuirong Wang. Maiter: An asynchronous graph processing framework for delta-based accumulative iterative computation. *IEEE Transactions on Parallel and Distributed Systems*, 25(8):2091–2100, 2014.
[13] Minyang Han and Khuzaima Daudjee. Giraph unchained: Barrierless asynchronous parallel execution in pregel-like graph processing systems. *PVLDB*, 8(9):950–961, 2015.
[14] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. X-stream: edge-centric graph processing using streaming partitions. In *SOSP*, pages 472–488, 2013.
[15] Zhigang Wang, Yu Gu, Yubin Bao, Ge Yu, and Jeffrey Xu Yu. Hybrid pulling/pushing for i/o-efficient distributed and iterative graph computing. In *SIGMOD*, pages 479–494, 2016.
[16] Peng Sun, Yonggang Wen, Ta Nguyen Binh Duong, and Xiaokui Xiao. Graphh: High performance big graph analytics in small clusters. In *IEEE International Conference on Cluster Computing*, pages 256–266, 2017.
[17] Joseph E. Gonzalez, Reynold S. Xin, Ankur Dave, Daniel Crankshaw, Michael J. Franklin, and Ion Stoica. Graphx: Graph processing in a distributed dataflow framework. In *OSDI*, pages 599–613, 2014.
[18] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, pages 2–2, 2012.
[19] U. Kang, Hanghang Tong, Jimeng Sun, Ching-Yung Lin, and Christos Faloutsos. GBASE: a scalable and general graph management system. In *KDD*, pages 1091–1099, 2011.
[20] Narayanan Sundaram, Nadathur Satish, Md. Mostofa Ali Patwary, Subramanya Dulloor, Michael J. Anderson, Satya Gautam Vadlamudi, Dipankar Das, and Pradeep Dubey. Graphmat: high performance graph analytics made productive. *PVLDB*, 8(11):1214–1225, 2015.
[21] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. Gemini: A computation-centric distributed graph processing system. In *OSDI*, pages 301–316, 2016.
[22] Keval Vora, Rajiv Gupta, and Guoqing (Harry) Xu. Kickstarter: Fast and accurate computations on streaming graphs via trimmed approximations. In *ASPLOS*, pages 237–251, 2017.
[23] Mugilan Mariappan and Keval Vora. Graphbolt: Dependency-driven synchronous processing of streaming graphs. In *EuroSys*, pages 25:1–25:16, 2019.
[24] Benjamin Heintz, Rankyung Hong, Shivangi Singh, Gaurav Khandelwal, Corey Tesdahl, and Abhishek Chandra. MESH: A flexible distributed hypergraph processing system. In *IC2E*, pages 12–22, 2019.
[25] Julian Shun. Practical parallel hypergraph algorithms. In *PPoPP*, pages 232–249, 2020.
[26] Dengyong Zhou, Jiayuan Huang, and Bernhard Schölkopf. Learning with hypergraphs: clustering, classification, and embedding. In *NIPS*, pages 1601–1608, 2006.
[27] George Karypis and Vipin Kumar. Multilevel k-way partitioning scheme for irregular graphs. *J. Parallel Distributed Comput.*, 48(1):96–129, 1998.

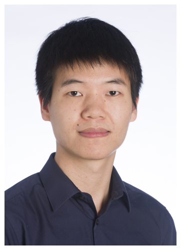
[28] George M. Slota, Kamesh Madduri, and Sivasankaran Rajamanickam. Complex network partitioning using label propagation. *SIAM J. Scientific Computing*, 38(5), 2016.
[29] George Karypis and Vipin Kumar. A parallel algorithm for multilevel graph partitioning and sparse matrix ordering. *J. Parallel Distributed Comput.*, 48(1):71–95, 1998.
[30] George M. Slota, Sivasankaran Rajamanickam, Karen D. Devine, and Kamesh Madduri. Partitioning trillion-edge graphs in minutes. In *IPDPS*, pages 646–655, 2017.
[31] Isabelle Stanton and Gabriel Kliot. Streaming graph partitioning for large distributed graphs. In Qiang Yang, Deepak Agarwal, and Jian Pei, editors, *SIGKDD*, pages 1222–1230. ACM, 2012.
[32] Charalampos E. Tsourakakis, Christos Gkantsidis, Bozidar Radunovic, and Milan Vojnovic. FENNEL: streaming graph partitioning for massive scale graphs. In *WSDM*, pages 333–342, 2014.
[33] Rong Chen, Jiaxin Shi, Yanzhe Chen, and Haibo Chen. Powerlyra: differentiated graph computation and partitioning on skewed graphs. In *EuroSys*, pages 1:1–1:15, 2015.
[34] George Karypis, Rajat Aggarwal, Vipin Kumar, and Shashi Shekhar. Multilevel hypergraph partitioning: applications in vlsi domain. *IEEE Transactions on Very Large Scale Integration Systems*, 7(1):69–79, 1999.
[35] Hypergraph-Partitioning-Based Decomposition for Parallel Sparse-Matrix Vector Multiplication. Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication. *IEEE Transactions on Parallel & Distributed Systems*, 10(7):673–693, 1999.
[36] Brendan Vastenhouw and Rob H. Bisseling. R.h.: A two-dimensional data distribution method for parallel sparse matrix-vector multiplication. *Siam Review*, 47(1):67–95, 2005.
[37] Aleksandar Trifunovic and William J. Knottenbelt. Par kway 2.0: A parallel multilevel hypergraph partitioning tool. In *Lecture notes in computer science*, pages 789–800, 2004.
[38] Karen D. Devine, Erik G. Boman, Robert T. Heaphy, Rob H. Bisseling, and Ümit V. Çatalyürek. Parallel hypergraph partitioning for scientific computing. In *International Conference on Parallel and Distributed Processing*, page 10, 2006.
[39] R. Oguz Selvitopi, Ata Turk, and Cevdet Aykanat. Replicated partitioning for undirected hypergraphs. *Journal of Parallel & Distributed Computing*, 72(4):547–563, 2012.
[40] Mehmet Deveci, Kamer Kaya, Bora Uçar, and Ümit V. Çatalyürek. Hypergraph partitioning for multiple communication cost metrics: Model and methods. *Journal of Parallel & Distributed Computing*, 77:69–83, 2015.
[41] Ata Turk, R. Oguz Selvitopi, Hakan Ferhatosmanoglu, and Cevdet Aykanat. Temporal workload-aware replicated partitioning for social networks. *TKDE*, 26(11):2832–2845, 2014.
[42] Yishu Wang, Ye Yuan, Yuliang Ma, and Guoren Wang. Time-dependent graphs: Definitions, applications, and algorithms. *Data Science and Engineering*, 4(4):352–366, 2019.
[43] Kirk Schloegel, George Karypis, and Vipin Kumar. Multilevel diffusion schemes for repartitioning of adaptive meshes. *J. Parallel Distributed Comput.*, 47(2):109–124, 1997.
[44] Daniel Nicoara, Shahin Kamali, Khuzaima Daudjee, and Lei Chen. Hermes: Dynamic partitioning for distributed social network graph databases. In *EDBT*, pages 25–36, 2015.
[45] Wenfei Fan, Muyang Liu, Chao Tian, Ruiqi Xu, and Jingren Zhou. Incrementalization of graph partitioning algorithms. *VLDB*, 13(8):1261–1274, 2020.
[46] Yu Zhang, Xiaofei Liao, Hai Jin, Lin Gu, Ligang He, Bingsheng He, and Haikun Liu. Cgraph: A correlations-aware approach for efficient concurrent iterative graph processing. In Haryadi S. Gunawi and Benjamin Reed, editors, *ATC*, pages 441–452.
[47] Jilong Xue, Zhi Yang, Shian Hou, and Yafei Dai. Processing concurrent graph analytics with decoupled computation model. *IEEE Trans. Computers*, 66(5):876–890, 2017.
[48] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M. Hellerstein. Distributed graphlab: A framework for machine learning in the cloud. *Proc. VLDB Endow.*, 5(8):716–727, 2012.
[49] Chang Zhou, Jun Gao, Binbin Sun, and Jeffrey Xu Yu. Mocgraph: scalable distributed graph processing using message online computing. *PVLDB*, 8(4):377–388, 2014.

Yu Gu received the PhD degree in computer software and theory from Northeastern University, China, in 2010. Currently, he is a professor and the PhD supervisor at Northeastern University, China. His current research interests include big data analysis, graph data management and spatial data management. He is a senior member of the China Computer Federation (CCF).

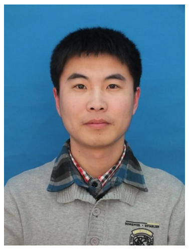


Kaiqiang Yu received the master degree in computer software and theory from Northeastern University, China, in 2019. His current research interests include big data analysis and graph data management.

Zhen Song received the master degree in computer software and theory from Northeastern University, China, in 2019. He is a Ph.D candidate in Northeastern University, China. His current research interests include distributed graph computation and distributed machine learning.



Jianzhong Qi received his Ph.D degree from The University of Melbourne in 2014. He is a lecturer in the School of Computing and Information Systems at the University of Melbourne. His research interests include spatio-temporal databases and natural language processing.



Zhigang Wang received the PhD degree in computer software and theory from Northeastern University, China, in 2018. He is currently a lecturer in the College of Information Science and Engineering, Ocean University of China. He has been a visiting PhD student in University of Massachusetts Amherst during December 2014 to December 2016. His research interests include cloud computing, distributed graph processing and machine learning.

Ge Yu received the PhD degree in computer science from Kyushu University of Japan, in 1996. He is currently a professor and the PhD supervisor at Northeastern University of China. His research interests include distributed and parallel database, OLAP and data warehousing, data integration, graph data management, etc. He is a member of the IEEE Computer Society, IEEE, ACM, and a Fellow of the China Computer Federation (CCF).

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 and data streams.