# Technical Report

Department of Computer Science
and Engineering
University of Minnesota
4-192 Keller Hall
200 Union Street SE
Minneapolis, MN 55455-0159 USA

TR 16-038

MESH: A Flexible Distributed Hypergraph Processing System

Benjamin Heintz, Shivangi Singh, Corey Tesdahl, Abhishek Chandra

December 5, 2016

# MESH: A Flexible Distributed Hypergraph Processing System

Benjamin Heintz, Shivangi Singh, Corey Tesdahl, and Abhishek Chandra
*Department of Computer Science & Engineering*
*University of Minnesota*
*Minneapolis, MN, USA*
heintz@cs.umn.edu, singh486@umn.edu, tesd0005@umn.edu, chandra@cs.umn.edu

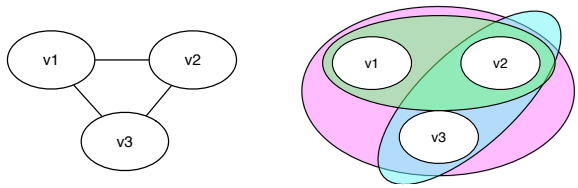
***Abstract*—With the rapid growth of large online social networks, the ability to analyze large-scale social structure and behavior has become critically important, and this has led to the development of several scalable graph processing systems. In reality, however, social interaction takes place not just between pairs of individuals as in the graph model, but rather in the context of multi-user groups. Research has shown that such group dynamics can be better modeled through a more general *hypergraph* model, resulting in the need to build scalable hypergraph processing systems. In this paper, we present MESH, a flexible distributed framework for scalable hypergraph processing. MESH provides an easy-to-use and expressive application programming interface that naturally extends the "think like a vertex" model common to many popular graph processing systems. Our framework provides a flexible implementation that enables different design choices for the key implementation issues of hypergraph representation and partitioning. We implement MESH on top of the popular GraphX graph processing framework in Apache Spark. Using a variety of real datasets, we experimentally demonstrate that MESH provides flexibility based on data and application characteristics, and is competitive in performance to HyperX, another hypergraph processing system based on Spark, showing that simplicity and flexibility need not come at the cost of performance.**

***Keywords*-Distributed computing, graph processing, data-intensive computing**

## I. Introduction

The advent of online social networks and communities such as Facebook and Twitter has led to unprecedented growth in user interactions (such as "likes", comments, photo sharing, and tweets), and collaborative activities (such as document editing and shared quests in multi-player games). This has resulted in massive amounts of rich data that can be analyzed to better understand user behavior, information flow, and social dynamics. The traditional way to study social networks is by modeling them as *graphs*, where each vertex represents an entity (e.g., a user) and each edge represents the relation or interaction between two entities (e.g., friendship). Myriad graph analytics frameworks [1], [2], [3] have been introduced to scale out the computation on massive graphs comprising millions or billions of vertices and edges.

While graph analytics has enabled a better understanding of social interactions between individuals, there is a growing interest [4] in studying *groups* of individuals as entities on their own. A group is an underlying basis for many social interactions and collaborations, such as users on Facebook commenting on an event of common interest, or a team of programmers collaborating on a software project. In these cases, individuals interact in the context of the overall group, and not simply in pairs. Further, the dynamics of many such systems may also be driven through group-level events, such as users joining or leaving groups, or finding others based on group characteristics (e.g., common interest).




(a) Graph (b) Hypergraph

Figure 1. A hypergraph can model groups unambiguously compared to a simple graph. Here, we have three groups: two consisting of pairwise interactions ($\{v_1, v_2\}, \{v_2, v_3\}$), and one including all three vertices. The graph model can only represent them as three pairwise interactions.

Since such group-based phenomena involve multi-user interactions, it has been shown that many natural phenomena can be better modeled using *hypergraphs* than by using graphs [5]. Formally, a *hypergraph* is a generalization of a graph[1], and is defined as a tuple $H = (V, E)$, where $V$ is the set of entities, called *vertices*, in the network, and $E$ is the set of subsets of $V$, called *hyperedges*, representing relations between one or more entities [6] (as opposed to exactly two in a graph). As illustrated in Figure 1, a hypergraph can model groups unambiguously compared to a graph.

Recent work [7] has shown that hypergraph models can achieve a significant improvement in modeling accuracy compared to graph-based models for social interactions. While hypergraph algorithms have received much less attention than graph algorithms, there has been work on developing hypergraph counterparts for problems such as centrality estimation [8], shortest path computation [9], and others. These algorithms will likely receive more attention

[1]In this paper, we use "graph" to refer to a traditional dyadic graph.

as the study of group dynamics matures. Hypergraphs have also been applied outside the social network analytics context, for example to analyze disease-gene networks [10], to optimize VLSI design [11], and to optimize on-disk storage of database files [12]. As a result, there is a growing need for scalable hypergraph processing systems that can enable easy implementation and efficient execution of such algorithms on real-world data.

From a systems standpoint, a hypergraph processing system must satisfy several design goals. First, for easy adoption by users, a hypergraph processing system must provide an interface that is **expressive and easy-to-use** by application programmers. Second is the ability to handle data at different scales, ranging from small hypergraphs to massive ones (with millions or billions of vertices and hyperedges). As a result, similar to a graph processing system, a hypergraph processing system must be **scalable**, both in terms of memory and storage utilization, as well as by enabling distributed computation across multiple CPUs and nodes for increased parallelism as needed. Third, it must be **flexible** in order to perform well in the face of diverse application and data characteristics. Finally, any novel design for a hypergraph processing system should strive for **ease of implementation**, as this allows faster development, enhancement, and maintenance.

From a high level, there are two main approaches to building a hypergraph processing system: build a standalone system (e.g., HyperX [13]), or overlay a hypergraph processing system on top of an existing graph processing system. Both approaches have pros and cons. While the first approach has the benefit of allowing hypergraph-specific optimizations at a lower level, it can be limited in terms of its flexibility and may require a sophisticated implementation effort. The second approach, on the other hand, can leverage many mechanisms and optimizations already available in existing mature graph processing systems, and hence, can be simpler to implement, and can provide flexibility in terms of design choices. As we will show, this flexibility can be achieved without performance cost.

In this paper, we take the latter approach, and present MESH[2], a distributed hypergraph processing system based on a graph processing framework. We use our system to explore two key challenges in implementing a hypergraph processing system on top of a graph processing system: how to represent the hypergraph and how to partition this representation to allow efficient distributed computation. For our implementation, we choose the GraphX framework [1] in Apache Spark [14]. While we choose GraphX due to the popularity and continued growth of the underlying Spark platform, we expect our ideas to be applicable or extensible to other graph processing frameworks as well.

[2]**M**innesota **E**ngine for **S**calable **H**ypergraph analysis

### A. Research Contributions

- We present MESH, a distributed hypergraph processing system designed for scalable hypergraph processing.
- We present an expressive API for hypergraph processing, which extends the popular "think like a vertex" programming model [2] by treating hyperedges as first-class computational objects with their own state and behavior.
- We explore the impact of two key design questions in building a hypergraph processing system: how to represent the hypergraph and how to partition this representation for distributed computation.
- We implement a MESH prototype on top of the GraphX graph processing system built on Apache Spark. Using this prototype and a number of real datasets, we experimentally demonstrate that there is no one-size-fits-all answer to the design questions, and that MESH provides the flexibility to make the best choice based on data and application characteristics.
- We experimentally compare our MESH implementation with the HyperX [13] hypergraph processing system, and demonstrate that MESH achieves comparable performance. This shows that a simple and flexible implementation need not come at the cost of performance.

Throughout this paper we explore two key research challenges: developing an expressive and easy-to-use API (Section III), and implementing this API on top of an existing graph processing system (Section IV).

## II. MESH Overview

### A. Design Goals

**Expressiveness & Ease of Use:** Hypergraph algorithms are fundamentally more general than graph algorithms. Many hypergraph algorithms treat hyperedges as first-class entities on par with vertices. A hypergraph processing system should therefore be *expressive* enough to allow hyperedges to have attributes and computational functions just as vertices do. It is critical that these attributes and functions be as general for hyperedges as they are for vertices. In addition to this expressiveness, a hypergraph system should also provide *ease of use*, enabling application developers to easily write a diverse variety of hypergraph applications.

**Scalability:** Many real-world datasets range in size from small to massive, comprising millions or billions of vertices and hyperedges. Similar to popular graph processing systems, hypergraph processing systems must be designed to scale to massive inputs, and they must allow distributed processing over multiple machines, while efficiently processing small datasets as well.

**Flexibility:** A hypergraph processing system must answer two key questions of how to represent hypergraphs, and how to partition this representation for distributed computation. As we show in Section IV, the right answer to

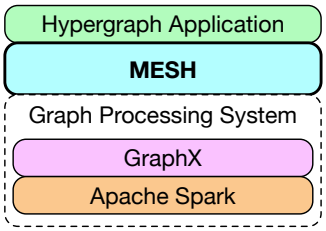



Figure 2. MESH is implemented on top of a graph processing engine and provides an expressive and easy-to-use API to hypergraph applications.

these questions depends on many factors related to the input dataset and algorithm characteristics. A hypergraph processing system must therefore be *flexible*, allowing the appropriate answers for these questions to be made at runtime based on data and application characteristics.

**Ease of Implementation:** A hypergraph processing system should be designed to simplify implementation as much as practical. This not only allows for faster development with fewer defects, but it allows the system to evolve more rapidly as it gains adoption. This is especially important as hypergraph processing is a novel area, where applications and systems will need to evolve rapidly in tandem.

Existing graph processing systems such as Pregel [2], PowerGraph [15], and GraphX [1] provide the foundation for scalability. They also provide a useful pattern we can follow to achieve programmability, namely the "think like a vertex" programming model, where graph processing applications are expressed in terms of vertex-level programs that iteratively receive messages from their neighbors, update their state, and send message to their neighbors. As we will show, however, these existing systems lack the flexibility required to handle diverse hypergraph applications and data.

### B. MESH Hypergraph Processing System

In order to meet the requirements of scalability and ease-of-implementation, we focus on implementing our hypergraph processing system, called MESH, on top of an existing graph processing system rather than from scratch. We assume that the underlying graph processing framework provides us with a graph representation consisting of vertices and edges, a graph partitioning framework to partition the input data across multiple machines, and a distributed execution framework that supports computation and communication across multiple machines, along with some fault tolerance mechanisms. For our implementation, we choose the GraphX framework [1] in Apache Spark [14]. As Figure 2 shows, MESH is positioned as a middleware layer between the hypergraph application and GraphX, while GraphX itself is implemented on top of Apache Spark [14], the wildly popular and rapidly growing general-purpose data-intensive computing platform.

Given such a system architecture, we explore two key research challenges throughout this paper: developing an expressive and easy-to-use API for enabling diverse hypergraph algorithms (Section III), and implementing this API on top of an existing graph processing system (Section IV).

## III. The MESH API

### A. Hypergraph Algorithms

Many hypergraph algorithms can be viewed as generalizations of corresponding graph algorithms, but they can have richer attributes and computations, particularly those defined for hyperedges in addition to vertices.

As an example, consider a Label Propagation algorithm [13], [16], which determines the community structure of a hypergraph. Here, in addition to identifying the community to which each vertex belongs, we may also assign to each *hyperedge* the community to which it belongs. Such an algorithm proceeds by iteratively passing messages from vertices to hyperedges and back. At each step along the way, the solution is refined as vertices and hyperedges update their attributes to record the community to which they belong.

As another example, consider PageRank [17], a widely used algorithm in graph analytics to determine the relative importance of different vertices in a graph. It is used in a variety of applications, such as search, link prediction, and recommendation systems.

We can extend PageRank to the hypergraph context in many ways. The most straightforward extension is to compute the PageRank for vertices based on their membership in different hyperedges. In a social context, this would correspond to determining the importance of a user based on her group memberships (e.g., a user might be considered more important if she is part of an exclusive club).

At the same time, it is possible to compute the PageRank for hyperedges based on the vertices they contain. This corresponds to estimating the importance of groups based on their members (e.g., a group with Fortune 500 CEOs is likely to be highly important). This extension also illustrates the fact that hyperedges can be considered first-class entities associated with similar state and computational functions as vertices in typical graph computation.

This elevation of hyperedges to first-class status enables further extensions to PageRank: we can compute additional attributes for hyperedges using arbitrary functions of their member vertices. For example, we can use an entropy function to determine the uniformity of each hyperedge; i.e., the extent to which its members contribute equally to its importance.

Along these same lines, hypergraph extensions can be derived for many popular graph algorithms, such as connected components, shortest paths [9], centrality estimation [8], [18], and more. The key to this expressiveness is the elevation of hyperedges to first-class status.

```
trait HyperGraph[VD, HED] {
  def compute[ToHE, ToV](
    maxIters: Int,
    initialMsg: ToV,
    vProgram: Program[VD, ToV, ToHE],
    heProgram: Program[HED, ToHE, ToV])
    : HyperGraph[VD, HED]
}

object HyperGraph {
  trait Program[A, InMsg, OutMsg] {
    def messageCombiner: MessageCombiner[OutMsg]
    def procedure: Procedure[A, InMsg, OutMsg]
  }

  type MessageCombiner[Msg] = (Msg, Msg) => Msg

  type Procedure[A, InMsg, OutMsg] =
    (Int, NodeId, A, InMsg, Context[A, OutMsg]) => Unit

  trait Context[A, OutMsg] {
    def become(attr: A): Unit
    def send(msgF: NodeId => OutMsg, to: Recipients): Unit
  }
}
```

Listing 1: Key abstractions from our hypergraph API (expressed in Scala).

### B. Core Interface

To make MESH easy to use, its API builds upon programmers' existing familiarity with the "think like a vertex" model [2], by providing a "think like a vertex *or hyperedge*" model. MESH provides an iterative computational model similar to Pregel, but with the introduction of hyperedges as first-class entities with their own computational behavior and state. In this model, computation proceeds iteratively in a series of alternating "supersteps" (alternating between vertex and hyperedge computation). Within a superstep, vertices (resp., hyperedges) update their state and compute new messages, which are delivered to their incident hyperedges (resp., vertices).

Listing 1 shows the core of the MESH API[3]. The key abstraction is the **`HyperGraph`**, which is parameterized on the vertex and hyperedge attribute data types. Similar to the GraphX **`Graph`** interface, the **`HyperGraph`** provides methods (not shown) such as `vertices` and `hyperEdges` for accessing vertex and hyperedge attributes, `mapVertices` and `mapHyperEdges` for transforming the hypergraph, `subHyperGraph` for computing a subhypergraph based on user-defined predicate functions, and so on.

The iterative computation model described above is implemented via the core computational method, `compute`. To use the `compute` method to orchestrate their iterative computation, users encode their vertex (resp., hyperedge) behavior in the form of a **`Program`** comprising a **`Procedure`** for consuming incoming messages, updating state, and producing outgoing messages, as well as a **`MessageCombiner`** for aggregating messages destined to a common hyperedge (resp., vertex). The **`Context`** provides methods that enable the **`Procedure`** to update vertex (resp., hyperedge) state, and to send messages to neighboring hyperedges (resp., vertices).

[3]We show Scala code for our API/algorithms. Scala `traits` are analogous to Java `interfaces`, and the `object` keyword here is used to define a module namespace.

```
// Attributes have been 'augmented' to
// carry algorithm state.
type VAttr = (VD, Weight, Rank)
type HEAttr = (HED, Weight, Cardinality, Rank)

// Vertex procedure: update vertex rank, and
// broadcast to incident hyperedges.
val prVProc: Procedure[VAttr, Rank, Rank] =
  (ss, id, attr, msg, ctx) => {
    val (vd, totalHeWeight, _) = attr
    val newRank = alpha + (1.0 - alpha) * msg
    ctx.become((vd, totalHeWeight, newRank))
    ctx.broadcast(newRank / totalHeWeight)
  }

// Hyperedge procedure: update hyperedge rank,
// and broadcast to member vertices.
val prHeProc: Procedure[HEAttr, Rank, Rank] =
  (ss, id, attr, msg, ctx) => {
    val (hed, ourWeight, card, _) = attr
    val newRank = msg * ourWeight
    ctx.become((hed, ourWeight, card, newRank))
    ctx.broadcast(newRank / card)
  }
```

Listing 2: PageRank algorithm implementation.

In this model, hyperedges are elevated to first-class status; they can maintain their state, carry out computation, and send messages just as vertices do. The MESH API therefore meets our expressiveness requirements. The generality and conciseness of the API aid in making the API easy to use.

To further improve ease of use, we observe that, in many cases, it is possible to determine the **`MessageCombiner`** automatically based on the message types. We implement this convenient feature using Twitter's Algebird[4] library, and allow programmers to enable it with a single **`import`** directive. With this feature enabled, users need only specify a **`Procedure`**.

### C. Example MESH Applications

Using the MESH API, it is simple to implement a hypergraph variant of the PageRank algorithm which computes ranks for both hyperedges and vertices, as shown in Listing 2. A richer version which also computes the entropy of each hyperedge (not shown) requires a simple three-line helper function and changes to only three lines.

Implementing a Label Propagation Algorithm [13], [16] is even simpler. Listing 3 shows how concisely we can implement this algorithm using our API. Note that for both Label Propagation and PageRank, the **`MessageCombiner`** is derived automatically.

## IV. Implementation

In order to implement a scalable hypergraph processing system, we must address two key challenges: how to represent the hypergraph, and how to partition this representation

[4]`https://github.com/twitter/algebird`

```
// Attributes have been 'augmented' to
// carry algorithm state.
type VAttr = (VD, Community)
type HEAttr = (HED, Community)

// Use the same message type in both directions.
// The automatically derived MessageCombiner
// performs key-wise addition.
type Msg = Map[Community, Int]

// Compute the most frequent community
def mostFrequent(msg: Msg): Community =
  msg.maxBy(_.swap)._1

// Vertex procedure: join the most frequent neighboring
// community and broadcast this decision.
val lpVProc: Procedure[VAttr, Msg, Msg] =
  (ss, id, attr, msg, ctx) => {
    val (vd, _) = attr
    // To begin, join our own
    // singleton community
    val newCommunity =
      if (ss == 0) id
      else mostFrequent(msg)
    ctx.become((vd, newCommunity))
    ctx.broadcast(Map(newCommunity -> 1))
  }

// Hyperedge procedure: join the most frequent
// neighboring community and broadcast this decision.
val lpHeProc: Procedure[HEAttr, Msg, Msg] =
  (ss, id, attr, msg, ctx) => {
    val (hed, _) = attr
    val newCommunity = mostFrequent(msg)
    ctx.become((hed, newCommunity))
    ctx.broadcast(Map(newCommunity -> 1))
  }
```

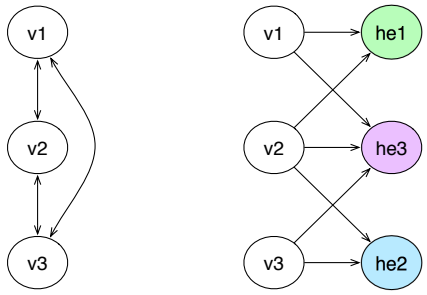
Listing 3: Label Propagation algorithm implementation.

for distributed computation. As discussed in Section II, MESH leverages the capabilities of an underlying graph processing system to address these challenges. Thus, it converts a hypergraph into an underlying graph representation, and utilizes a graph partitioning framework to implement a variety of hypergraph partitioning algorithms. We next discuss the design choices and the tradeoffs in making these decisions, as well as our implementation on top of GraphX.

### A. Representation

*1) Clique-Expanded Graph:* One possibility is to represent a hypergraph as a simple graph by expanding each hyperedge into a clique of its members, as shown in Figure 3(a). We refer to this representation as the *clique-expanded graph*. In order to enable this representation, our **`HyperGraph`** interface provides a `toGraph` transformation method, which logically replaces the connectivity structure of the hypergraph with edges rather than hyperedges. The attributes of an edge from $v_1$ to $v_2$ are determined by user-defined functions applied to the set of all hyperedges common to $v_1$ and $v_2$. Applying this transformation to produce a clique-expanded graph may be costly—even prohibitively so—in terms of both space and time.

Another major disadvantage of the clique-expanded graph is its limited applicability. Because hyperedges do not appear in this representation, it is only appropriate for algorithms




(a) Clique-expanded. (b) Bipartite.

Figure 3. Underlying graph representations of the hypergraph in Figure 1(b).

that do not modify hyperedge state, and thus, for instance cannot be used for our Label Propagation algorithm. Further, the hyperedge and vertex programs must meet additional requirements, such as sending the same message type in both directions. Overall, therefore, this representation is best viewed as a potential optimization for a small set of use cases rather than a general approach.

*2) Bipartite Graph:* An alternative approach is to represent the hypergraph internally as a bipartite graph, where one partition comprises exclusively vertices, and the other exclusively hyperedges, with low-level graph edges connecting hyperedges to their constituent vertices, as shown in Figure 3(b). This representation can concisely encode any hypergraph, and it allows us to run programs that treat both vertices and hyperedges as first-class computational entities. By using directed edges (in our implementation exclusively from vertices to hyperedges) we provide a means to differentiate between vertices and hyperedges. Due to the general expressive power of this representation, *we focus our attention throughout this paper on its efficient implementation in a graph processing system.*

### B. Partitioning

*1) Challenges:* To scale to large hypergraphs, it is essential to distribute computation across multiple nodes. The decision of how to partition the underlying representation can significantly affect performance, in terms of both computational load and network I/O. An effective partitioning algorithm—whether for a graph or hypergraph—must simultaneously balance computational load and minimize communication. Hypergraph partitioning, however, presents several challenges beyond those for partitioning graphs.

For one, hypergraphs contain two distinct sets of entities: vertices and hyperedges. In general, these two sets can differ significantly in terms of their size, skew in cardinality/degree[5], and associated computation. Further, MESH

[5]The *degree* of a vertex denotes the number of hyperedges of which that vertex is a member. Similarly, the *cardinality* of a hyperedge denotes the number of vertices belonging to that hyperedge.

computation runs on only one of these sets at a time. An effective partitioning algorithm must therefore *differentiate between hyperedges and vertices*.

At the same time, hyperedge and vertex partitioning are fundamentally interrelated; an effective algorithm must *holistically partition hyperedges and vertices*. For example, an algorithm that partitions hyperedges without regard to vertex partitioning may achieve good computational load balance, but will suffer from excessive network I/O.

*2) Algorithms:* MESH utilizes the underlying graph partitioning framework to implement hypergraph partitioning algorithms. Many graph processing frameworks either partition vertices (thus cutting edges[6]) or partition edges (cutting vertices) across machines. Many current systems [1], [15] use edge partitioning since it has been shown to be more efficient for many real-world graphs. In what follows, we describe mapping hypergraph partitioning algorithms to such edge partitioning graph algorithms. We expect that mapping to vertex partitioning algorithms could be done in a similar fashion, and we leave such mapping as future work.

Concretely, we assume the underlying graph partitioning framework partitions the set of edges, while replicating each vertex to every partition that contains edges incident on that vertex. In our bipartite graph representation, edges are directed exclusively from (hypergraph) vertices to hyperedges. As a result, if we partition based only on the source (resp., destination) of an edge, then hypergraph vertices (resp., hyperedges) will each be assigned to a unique partition, while hyperedges (resp., vertices) will be replicated—i.e., "cut"—across several partitions. If we choose the partition for an edge based on both its source and destination, then both vertices and hyperedges are effectively cut.

Any graph partitioning algorithm leads to a tradeoff between balancing computational load and minimizing network communication. While balancing the number of edges across machines could lead to good load balance, a high degree of replication of vertices can lead to increased network I/O and execution time. In order to distribute a hypergraph, however, replication is unavoidable. The goal is therefore to choose which set(s) (vertices or hyperedges) to cut, and how to partition the other set so as to balance computational load while minimizing replication.

We explore a range of alternative partitioning algorithms that approach this goal from different angles. These algorithms fall into three classes: *Random*, *Greedy*, and *Hybrid*.

*Random:* We explore three Random partitioning algorithms. The Random Vertex-cut algorithm hash-partitions bipartite graph edges based on their destination (i.e., by hyperedge), effectively cutting hypergraph vertices. The Random Hyperedge-cut algorithm, on the other hand, partitions vertices and cuts hyperedges.

[6]Note that we use "edge" to refer to an edge in the *underlying* graph representation. This should not be confused with a hyperedge in the application-level hypergraph.

The Random Both-cut algorithm hash-partitions bipartite graph edges by both their source and destination, effectively cutting both vertices and hyperedges.

*Hybrid:* The Hybrid algorithms we consider are based on the balanced $\rho$-way hybrid cut from PowerLyra [19]. These algorithms cut both vertices and hyperedges, but unlike Random Both-cut, they differentiate between vertices and hyperedges in doing so. In particular, the Vertex-cut variant cuts vertices while partitioning hyperedges, except that it also cuts hyperedges with high cardinality (greater than 100 in our experiments). Similarly, the Hyperedge-cut variant cuts hyperedges while also cutting high-degree vertices.

*Greedy:* Based on the Aweto [20] algorithm, the Greedy algorithms that we consider holistically partition hypergraphs with the goal of reducing replication and the resulting synchronization overhead. From a high level, the Greedy Vertex-cut variant aims to assign each hyperedge to a lightly-loaded partition with a large "overlap" between the vertices in that hyperedge and the vertices with replicas already on that partition based on (a heuristic estimate of) the assignments already made. (For a more rigorous definition, see [20].) The Greedy Hyperedge-cut variant, on the other hand, assigns vertices based on the overlap between their incident hyperedges and the hyperedges already assigned to each partition. Note that, unlike the Hybrid algorithms, the Greedy algorithms cut either vertices or hyperedges, but not both.

### C. Implementation in GraphX

As mentioned in Section II, we have implemented a MESH prototype on top of the GraphX [1] graph processing system. GraphX provides a graph representation consisting of vertices and edges represented as Resilient Distributed Datasets (RDDs). We can represent a hypergraph using the clique-expanded representation by mapping each hyperedge to a clique of its incident vertices. Similarly, we can represent a hypergraph as a bipartite graph by creating edges between vertices and their hyperedges.

GraphX uses an edge-partitioning algorithm for partitioning the graph across machines. In GraphX, the **`PartitionStrategy`** considers each edge in isolation, as in Listing 4.

```
def getPartition(
  src: VertexId,
  dst: VertexId,
  numPart: PartitionId): PartitionId
```

Listing 4: Original GraphX partitioning abstraction.

We use the built-in GraphX partitioning algorithms to implement the baseline Random partitioning algorithms described above. Our Greedy and Hybrid algorithms, however, require a broader view of the graph (to compute "overlap" and degree/cardinality, respectively). To satisfy this requirement, we extend the **`PartitionStrategy`** by adding a

new `getAllPartitions` method that allows partitioning decisions to be made with awareness of the full graph, as shown in Listing 5.

```
def getAllPartitions[VD, ED](
  graph: Graph[VD, ED],
  numPartitions: PartitionId,
  degreeCutoff: Int)
  : RDD[((VertexId, VertexId), PartitionId)]
```

Listing 5: Additional GraphX partitioning abstraction.

## V. Evaluation

### A. Experimental Setup

*1) Deployment:* We implement and run our MESH prototype on top of Apache Spark 1.6.0. Experiments are conducted on a cluster of eight nodes, each with two Intel Xeon E5-2620 v3 processors with 6 physical cores and hyperthreading enabled. Each node has 64 GB physical RAM, and a 1 TB hard drive with at least 75% free space, and nodes are connected via gigabit ethernet. Input data are stored in HDFS 2.7.2, which runs across these same eight nodes.

*2) Datasets:* As inputs for our experiments, we use publicly available data to build the hypergraphs described in Table I. These datasets differ in their characteristics, such as size, relative number of vertices and hyperedges, vertex degree/hyperedge cardinality distribution, etc.

The Apache hypergraph, derived from the Apache Software Foundation subversion logs, models collaboration on open-source software projects. Each vertex represents a committer, and each hyperedge represents a set of committers that have collaborated on one or more files.

The dblp dataset describes more than one million publications, from which we use authorship information to build a hypergraph model where vertices represent authors and hyperedges represent collaborations between authors.

In the Friendster and Orkut hypergraphs, vertices represent individual users, and hyperedges represent user-defined communities in the Friendster and Orkut social networking sites, respectively. Because membership in these communities does not require the same commitment as collaborating on software or academic research, these hypergraphs have very different characteristics from dblp and Apache, in particular in terms of the overall size of the data, and vertex degree and hyperedge cardinality. One difference between the two is that Friendster has many more vertices than hyperedges, whereas the opposite is true for Orkut.

*3) Applications:* We use three applications in our experiments, including the Label Propagation algorithm from Listing 3 as well as two PageRank variants. The first variant, which we refer to simply as PageRank (Listing 2), computes ranks for both vertices and hyperedges. The second variant, PageRank-Entropy, additionally computes the entropy of each hyperedge. The key difference between the two variants is that, in the PageRank-Entropy version, in order to compute entropy of each hyperedge, messages from vertices to hyperedges must be concatenated rather than summed, leading to a higher volume of data movement.

We run each algorithm for 30 iterations.

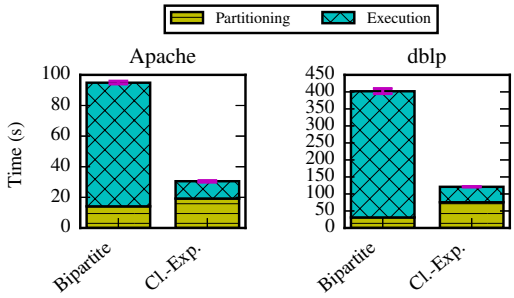



Figure 4. Execution time for bipartite and clique-expanded representations for the PageRank algorithms on the dblp dataset.

### B. Representation

We first briefly explore the relative strengths and weaknesses of two main representation alternatives: the clique-expanded graph and the bipartite graph. Table I shows the number of underlying graph edges required for each of these representations, while Figure 4 shows both partitioning time and subsequent execution time for the PageRank algorithm for the Apache and dblp hypergraphs. Throughout this paper, we run each experiment nine times and plot the mean, with error bars denoting 95% confidence intervals.

For the Apache hypergraph, the clique-expanded graph shows some promise in terms of both space and time. The clique-expanded representation uses only about 48% as many edges as the bipartite alternative, and although the initial partitioning phase (which includes running the `toGraph` transformation) is more time-consuming for this representation, the execution is significantly faster. For the dblp hypergraph, the clique-expanded representation again shows an execution time advantage, but this comes at the cost of space overhead, as this representation requires roughly 8x as many edges as the bipartite alternative.

The clique-expansion can be thought of as a constant-folding optimization applied at the time of constructing the representation. Although this can be helpful in terms of execution time, its space overhead can be large or even prohibitive. In fact, for the Friendster and Orkut hypergraphs, we are unable to even materialize the clique-expanded graphs on our cluster due to space limitations.

In addition to these scalability concerns, it is important to keep in mind that the clique-expanded representation does not apply for all algorithms, as discussed in Section IV. For example, we cannot use this representation for our Label Propagation or PageRank-Entropy algorithms, as these algorithms need explicit access to hyperedge attributes. Given

Table I
DATASETS USED IN OUR EXPERIMENTS.

| Dataset | # Vertices | # Hyperedges | Max. Degree | Max. Cardinality | # Bipartite Edges | # Clique-Expanded Edges |
|---|---|---|---|---|---|---|
| Apache | 3316 | 78,080 | 4,507 | 179 | 408,231 | 196,452 |
| dblp | 899,393 | 782,659 | 368 | 2,803 | 2,624,912 | 21,707,067 |
| Friendster | 7,944,949 | 1,620,991 | 1,700 | 9,299 | 23,479,217 | *10.3 billion (approximate)* |
| Orkut | 2,322,299 | 15,301,901 | 2,958 | 9,120 | 107,080,530 | *54.5 billion (approximate)* |

these limitations, we focus on the more general alternative of the bipartite representation throughout the remainder of our experiments.

### C. Partitioning

Next, we evaluate the partitioning policies described in Section IV. Figure 5 shows both partitioning time and subsequent execution time for the Label Propagation algorithm for each of these policies for the Apache, dblp, Friendster, and Orkut datasets. Figure 6 repeats these experiments for the PageRank algorithm.

Trends are similar between Label Propagation and PageRank. We see that the choice of the best partitioning algorithm depends on the data. One possible data characteristic having an impact could be the relative number of vertices and hyperedges in the hypergraph. For Apache, for example, MESH yields better performance by partitioning hyperedges while cutting vertices. The reason is that hyperedges vastly outnumber vertices, so partitioning hyperedges to balance computational load is well worth the relatively minor cost of replicating vertices. We see that the greedy hyperedge-cut algorithm is the best for the Friendster hypergraph (Figures 5(c) and 6(c)), where vertices outnumber hyperedges. Here, cutting hyperedges while partitioning the larger set of vertices might lead to better computational load balancing. On the other hand, for the Orkut hypergraph (Figures 5(d) and 6(d)), where hyperedges outnumber vertices, we see that while the vertex-cut algorithms seem to perform better than the corresponding hyperedge-cut variants, a Random Both-cut algorithm is the best. This suggests that cutting vertices is better than cutting hyperedges, but that cutting both sets may lead to even better load balancing. For dblp (Figures 5(b) and 6(b)), we see a much less pronounced difference between vertex-cut and hyperedge-cut algorithms, as the number of hyperedges and vertices in this dataset are roughly equal.

These experiments also show that the Greedy vertex-cut and hyperedge-cut algorithms outperform their Random counterparts. The reason is that they *holistically* partition the hypergraph; when partitioning the vertex (resp., hyperedge) set, they do so with an awareness of the interaction with hyperedge (resp., vertex) partitioning. This leads to reduced replication and in turn lower synchronization overhead.

We further observe that cutting both hyperedges and vertices can be effective. though the best specific partitioning algorithm depends on the dataset. For Apache, dblp, and Friendster, the Hybrid vertex-cut and hyperedge-cut algorithms outperform the Random both-cut baseline, and perform comparably to the Greedy algorithms, though with less sensitivity to the choice of which set (vertices or hyperedges) to cut. These hybrid algorithms *differentiate* between hyperedges and vertices, and attempt to cut only one of these sets. Only when it is warranted by high degree or high cardinality do these algorithms cut entities from the other set. This principled approach results in lower replication and in turn better performance for these datasets.

For Orkut, however, the Random Both-cut algorithm outperforms not just the Hybrid algorithms, but all other algorithms as well. This further highlights that the choice of partitioning algorithm depends on the dataset.

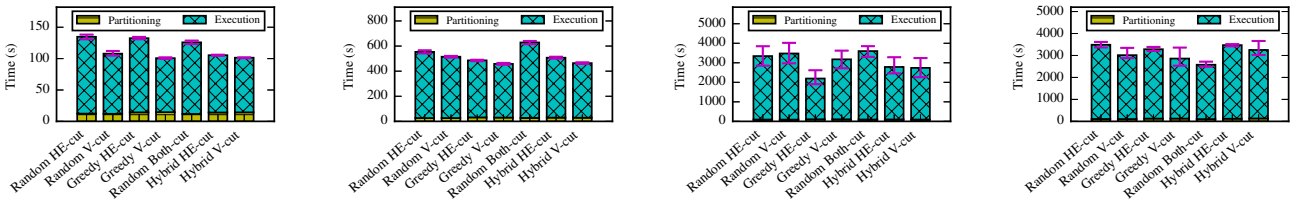
We repeat these experiments for the PageRank-Entropy algorithm and show the results in Figure 7. Results are similar to the Label Propagation and PageRank results for all but the Orkut dataset, where we see that, for Random and Greedy algorithms, hyperedge-cut partitioning outperforms vertex-cut. This apparent inconsistency may arise due to the larger message sizes in the PageRank-Entropy algorithm, though further investigation is required.

These results show that no one algorithm dominates all others. The best choice depends on the characteristics of the hypergraph. For instance, holistically partitioning the hypergraph, as done by the Greedy vertex-cut and hyperedge-cut algorithms, can be beneficial in some cases, while cutting both hyperedges and vertices can be effective in others. A promising next step is to develop a combined algorithm that partitions holistically as the Greedy algorithms do, while differentiating between hyperedges and vertices as the Random Both-cut and Hybrid algorithms do.

These results also show the value of the flexibility provided by MESH, where the choice of an appropriate partitioning algorithm can be based on data and application characteristics. Note that the vertex-to-hyperedge ratio is only one data characteristic that may be impacting the performance. Identifying all the relevant characteristics and their impact, and automatically making the design choices is an area of future work.

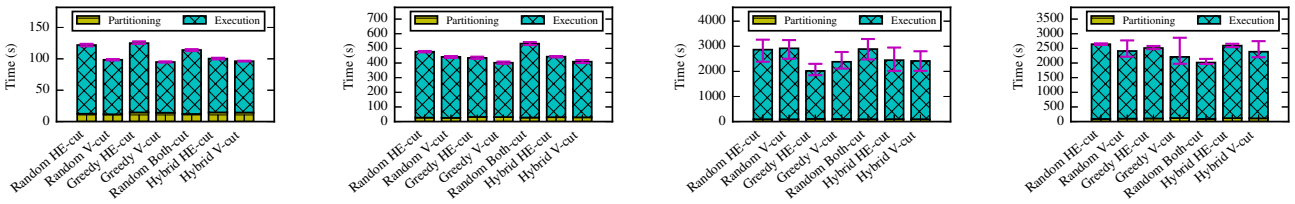
### D. Comparison with HyperX

To evaluate the overall performance of MESH, we compare against HyperX [13], a hypergraph processing system that is also built on top of Apache Spark. Unlike MESH, which builds on top of GraphX, HyperX implements a hypergraph layer—heavily inspired by GraphX—directly on



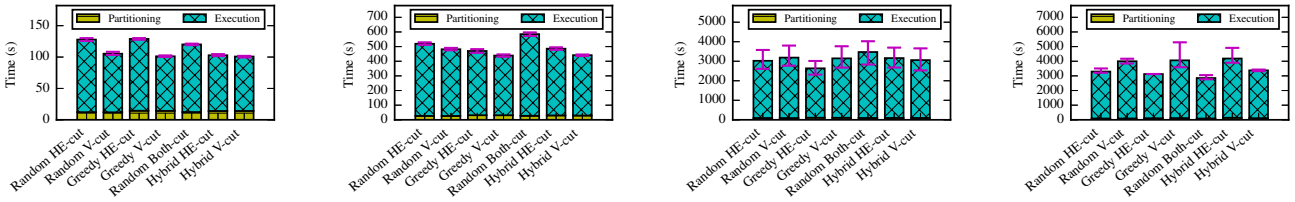

(a) Apache (b) dblp (c) Friendster (d) Orkut

Figure 5. Partitioning and Label Propagation execution time using several partitioning algorithms in MESH.




(a) Apache (b) dblp (c) Friendster (d) Orkut

Figure 6. Partitioning and PageRank execution time using several partitioning algorithms in MESH.




(a) Apache (b) dblp (c) Friendster (d) Orkut

Figure 7. Partitioning and PageRank-Entropy execution time using several partitioning algorithms in MESH.

top of Spark. This hypergraph layer is designed around certain assumptions, for example that hypergraphs tend to have more vertices than hyperedges.

We compare these two systems using a Label Propagation algorithm, specifically Listing 3 for MESH, and the provided example implementation for HyperX.[7]

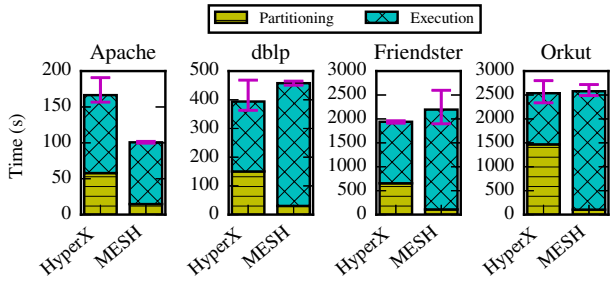



Figure 8. Partitioning and Label Propagation execution time for MESH (using the best partitioning algorithm) and HyperX.

Figure 8 shows the partitioning and Label Propagation execution times (for 30 iterations) for HyperX and MESH (using the best partitioning policies for the given dataset). Unlike MESH, HyperX uses an iterative partitioning algorithm (10 iterations in our experiments, based on the HyperX experiments [13]), leading to much higher partitioning times. This costly partitioning may be warranted for the Apache, dblp, Friendster and Orkut hypergraphs, as HyperX achieves lower execution time than MESH for those hypergraphs. For Apache, however, MESH achieves lower application runtime in spite of its less sophisticated partitioning algorithm.

Additionally, we observe that even the simple and unoptimized MESH prototype achieves comparable performance to HyperX, which features several low-level optimizations. In fact, for the Apache dataset, MESH outperforms HyperX. This is an interesting result, as the Apache hypergraph satisfies HyperX's design assumption that vertices outnumber hyperedges.

Further, HyperX assumes hypergraphs have certain properties (e.g., they contain many more vertices than hyperedges), In reality, however, hypergraphs are diverse, and as we have shown above, MESH provides the flexibility to handle this diversity.

[7]We modify the implementation in HyperX to compute over undirected hypergraphs.

From a higher level, our results suggest that high performance need not be at odds with a simple and flexible implementation. In fact, by layering on top of GraphX and leveraging its maturity and ongoing development, we can expect to reap the benefits of ongoing optimization. Backporting future optimizations to HyperX, on the other hand, would require significant engineering effort.

## VI. Discussion & Future Work

We plan to explore a number of enhancements to MESH, in terms of both its API and its implementation.

*Automatic Partitioning and Representation Selection:* We have demonstrated that there is no one-size-fits-all answer to the key questions of how to represent a hypergraph, and how to partition that representation for distributed processing. Instead, the best choice depends upon the particular hypergraph.

In some cases, analysts may focus on a single hypergraph or slowly evolving versions of a single hypergraph (e.g., Facebook group structure, Twitter followers, etc.). In these cases, it is practical to invest some up-front effort in determining the best representation and partitioning techniques.

In other cases, however, it may be necessary to process a varying set of hypergraphs and applications on a single deployment. In these cases, determining which representation and partitioning to use on a case-by-case basis could quickly become tedious. We are exploring approaches for automating this selection based on application and data characteristics. For example, our experiments in Section V showed that the relative number of hyperedges and vertices in a hypergraph can influence which of these sets should be partitioned and which should be cut. It may be possible to train a model to predict an appropriate representation and partitioning to (nearly) eliminate the need for manual selection of these parameters.

*Directed Hyperedges:* We currently model only undirected hyperedges; i.e., each hyperedge is simply a set of vertices. It may be useful to support directed hyperedges, where each hyperedge contains one or more source vertices and one or more destination vertices. The MESH API would need to be extended to allow applications to differentiate between sources and destinations, and the implementation would also need to be extended to model this distinction. Allowing this additional modeling flexibility would only increase the diversity of data and applications that MESH would be required to handle, and this may further motivate future work on partitioning and representation techniques.

Note that HyperX exclusively models directed hypergraphs, with the restrictions that 1) each hyperedge must contain at least one source and at least one destination, and 2) the source and destination sets must be disjoint. Undirected hyperedges can be emulated by arbitrarily partitioning each hyperedge into source and destination subsets and ignoring the distinction between source and vertex in the application code. Unfortunately, these restrictions effectively prohibit hyperedges with cardinality 1, which for example is required to model single-author papers in our dblp dataset. A more general model would allow the source and destination sets to overlap.

*Asynchronous Computation:* In its current form, the MESH API is purely synchronous; all vertex computation finishes before the next round of hyperedge computation begins, and so on. This is similar to the Pregel and GraphX models, and facilitates simpler programs and system implementations, though possibly at the cost of slower convergence [21]. Supporting asynchronous computation would impact both the API and implementation, and may restrict the choice of implementation platform.

## VII. Related Work

*Graph Processing Systems:* There has been a flurry of research on graph computing systems in recent years [2], [15], [22], [23], [24], [25], [21], [26], [27], and along with it, a great deal of work on performance evaluation and optimization [28], [29], [30].

Key among these systems, Pregel [2] introduced the "think like a vertex" model. GraphX [1], built upon Apache Spark [14], adopted a similar model while inheriting the scalability and fault tolerance of Spark's Resilient Distributed Datasets (RDD). GraphLab [15], [25], [21] provided a more fine-grained interface along with support for asynchronous computation.

These systems provide scalability, and their interfaces are easy to use in the graph computing context. Our MESH API can be viewed as an extension of the "think like a vertex" model. Although we have discussed challenges in implementing MESH on top of a graph processing system in general, and GraphX in particular, there is no fundamental requirement that MESH run on top of a specialized platform. For example, MESH could be implemented on top of a general-purpose relational database management system (RDBMS) [31]. GraphX, however, is particularly compelling due to the popularity and growth of Spark. Further, by facilitating diverse views of the same underlying data—e.g., collection-oriented, graph-oriented, tabular [32]—building on top of Spark allows easier integration in broader data processing pipelines.

*Graph and Hypergraph Partitioning:* Graph Partitioning is a significant research topic in its own right. In the high-performance computing context, metis [33] provides very effective graph partitioning, and has open-source implementations for both single-node and distributed deployment. Its hMetis [34] cousin partitions hypergraphs, but no distributed implementation yet exists. The Zoltan toolkit from Sandia National Laboratories [35] includes a parallel hypergraph partitioner [36] that cuts both vertices and hyperedges.

In the distributed systems context, PowerGraph [15] targets natural (e.g., social) graphs by cutting vertices rather

than edges. While this is effective for natural graphs, hypergraphs require different approaches. Chen et al. have proposed novel algorithms for bipartite graphs [1] and skewed graphs [19], which we have used as the basis for our Greedy and Hybrid algorithms, respectively. While these are already effective algorithms, there remains opportunity to combine holistic and differentiated approaches to improve hypergraph partitioning.

*Hypergraph Processing:* Hypergraphs have been studied for decades [5], [6], [37] and have been applied in many settings, ranging from bioinformatics [10] to VLSI design [11], [38] to database optimization [12]. Social networks have generally been modeled using simple graphs, but hypergraph variants of popular graph algorithms (e.g., centrality estimation [8], [18], shortest paths [9]) have been developed in recent years. HyperX [13] builds a hypergraph processing system on top of Spark, but does so by modifying GraphX rather than building on top of GraphX. Unlike HyperX, MESH does not make any static assumptions about the data characteristics, and instead provides the flexibility necessary to choose an appropriate representation and partitioning algorithm at runtime based on data and application characteristics.

## VIII. Conclusion

We presented MESH, a flexible distributed framework for scalable hypergraph processing. MESH provides an easy-to-use and expressive API that naturally extends the "think like a vertex" model common to many popular graph processing systems. We used our system to explore two key challenges in implementing a hypergraph processing system on top of a graph processing system: how to represent the hypergraph and how to partition this representation to allow distributed computation. MESH provides flexibility to implement different design choices, and by implementing MESH on top of the popular GraphX framework, we have leveraged the maturity and ongoing development of the Spark ecosystem and kept our implementation simple. Our experiments with multiple real-world datasets demonstrated that this flexibility does not come at the expense of performance, as even our unoptimized prototype performs comparably to HyperX.

## Acknowledgments

The authors would like to acknowledge the support of NSF Grants CNS-1413998 and CRI-1305237, as well as a University of Minnesota Doctoral Dissertation Fellowship.

## References

[1] J. E. Gonzalez *et al.*, "GraphX: Graph processing in a distributed dataflow framework," in *Proc. of OSDI*, 2014, pp. 599–613.

[2] G. Malewicz *et al.*, "Pregel: A system for large-scale graph processing," in *Proc. of SIGMOD*, 2010, pp. 135–146.

[3] D. Nguyen, A. Lenharth, and K. Pingali, "A lightweight infrastructure for graph analytics," in *Proc. of SOSP*, 2013, pp. 456–471.

[4] D. Lazer *et al.*, "Computational social science," *Science*, vol. 323, no. 5915, pp. 721–723, 2009.

[5] E. Estrada and J. Rodriguez-Velazquez, "Complex networks as hypergraphs," *Arxiv preprint physics/0505137*, 2005.

[6] C. Berge, *Graphs and hypergraphs*. Elsevier, 1976, vol. 6.

[7] A. Sharma, A. Chandra, and J. Srivastava, "Predicting multi-actor collaborations using hypergraphs," in *arXiv*, 2014.

[8] P. Bonacich, A. C. Holdren, and M. Johnston, "Hyper-edges and multidimensional centrality," *Social Networks*, vol. 26, no. 3, pp. 189 – 203, 2004.

[9] J. Gao, Q. Zhao, W. Ren, A. Swami, R. Ramanathan, and A. Bar-Noy, "Dynamic shortest path algorithms for hypergraphs," *Arxiv preprint arXiv:1202.0082*, 2012.

[10] S. R. Gallagher, M. Dombrower, and D. S. Goldberg, "Using 2-node hypergraph clustering coefficients to analyze disease-gene networks," in *Proc. of BCB*, 2014, pp. 647–648.

[11] C. J. Alpert and A. B. Kahng, "Recent directions in netlist partitioning: A survey," *Integr. VLSI J.*, vol. 19, no. 1-2, pp. 1–81, Aug. 1995.

[12] D.-R. Liu and S. Shekhar, "Partitioning similarity graphs: A framework for declustering problems," *Information Systems*, vol. 21, no. 6, pp. 475–496, 1996.

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

[14] M. Zaharia *et al.*, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proc. of NSDI*, 2012.

[15] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "PowerGraph: Distributed graph-parallel computation on natural graphs," in *Proc. of OSDI*, 2012, pp. 17–30.

[16] U. N. Raghavan, R. Albert, and S. Kumara, "Near linear time algorithm to detect community structures in large-scale networks," *Phys. Rev. E*, vol. 76, p. 036106, Sep 2007.

[17] L. Page, S. Brin, R. Motwani, and T. Winograd, "The PageRank citation ranking: Bringing order to the web," Stanford InfoLab, Tech. Rep., 1999.

[18] S. Roy and B. Ravindran, "Measuring network centrality using hypergraphs," in *Proc. of CoDS*, 2015, pp. 59–68.

[19] R. Chen, J. Shi, Y. Chen, and H. Chen, "PowerLyra: Differentiated graph computation and partitioning on skewed graphs," in *Proc. of EuroSys*, 2015, pp. 1:1–1:15.

[20] R. Chen, J. Shi, B. Zang, and H. Guan, "Bipartite-oriented distributed graph partitioning for big learning," in *Proc. of APSys*, 2014, pp. 14:1–14:7.

[21] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein, "GraphLab: A new parallel framework for machine learning," in *Proc. of UAI*, July 2010.

[22] Y. Bu, V. Borkar, J. Jia, M. J. Carey, and T. Condie, "Pregelix: Big(ger) graph analytics on a dataflow engine," *Proc. VLDB Endow.*, vol. 8, no. 2, pp. 161–172, Oct. 2014.

[23] R. Cheng *et al.*, "Kineograph: Taking the pulse of a fast-changing and connected world," in *Proc. of EuroSys*, 2012, pp. 85–98.

[24] A. Kyrola, G. Blelloch, and C. Guestrin, "GraphChi: Large-scale graph computation on just a PC," in *Proc. of OSDI*, 2012.

[25] Y. Low *et al.*, "Distributed GraphLab: a framework for machine learning and data mining in the cloud," *Proc. VLDB Endow.*, vol. 5, no. 8, pp. 716–727, Apr. 2012.

[26] N. Satish *et al.*, "Navigating the maze of graph analytics frameworks using massive graph datasets," in *Proc. of SIGMOD*, 2014, pp. 979–990.

[27] S. Salihoglu and J. Widom, "GPS: A graph processing system," in *Proc. of SSDBM*, 2013, pp. 22:1–22:12.

[28] M. Han *et al.*, "An experimental comparison of pregel-like graph processing systems," *Proc. VLDB Endow.*, vol. 7, no. 12, pp. 1047–1058, Aug. 2014.

[29] S. Salihoglu and J. Widom, "Optimizing graph algorithms on pregel-like systems," *Proc. VLDB Endow.*, vol. 7, no. 7, pp. 577–588, Mar. 2014.

[30] Y. Zhang, Q. Gao, L. Gao, and C. Wang, "PrIter: A distributed framework for prioritized iterative computations," in *Proc. of SOCC*, 2011, pp. 13:1–13:14.

[31] J. Fan, A. G. S. Raj, and J. M. Patel, "The case against specialized graph analytics engines," in *Proc. of CIDR*, 2015.

[32] M. Armbrust *et al.*, "Spark SQL: Relational data processing in spark," in *Proc. of SIGMOD*, 2015, pp. 1383–1394.

[33] G. Karypis and V. Kumar, "Parallel multilevel k-way partitioning scheme for irregular graphs," in *Proc. of SC*, 1996.

[34] G. Karypis, R. Aggarwal, V. Kumar, and S. Shekhar, "Multilevel hypergraph partitioning: applications in vlsi domain," *IEEE Transactions on VLSI Systems*, vol. 7, no. 1, 1999.

[35] http://www.cs.sandia.gov/Zoltan/.

[36] K. D. Devine, E. G. Boman, R. T. Heaphy, R. H. Bisseling, and U. V. Catalyurek, "Parallel hypergraph partitioning for scientific computing," in *Proc. of IPDPS*, April 2006.

[37] C. Berge, *Hypergraphs: combinatorics of finite sets*. Elsevier Science Publishers B. V., 1989.

[38] G. Karypis, R. Aggarwal, V. Kumar, and S. Shekhar, "Multilevel hypergraph partitioning: applications in VLSI domain," *IEEE Transactions on VLSI Systems*, vol. 7, no. 1, pp. 69–79, March 1999.