MESH: A Flexible Distributed Hypergraph Processing System

Benjamin Heintz
University of Minnesota
Minneapolis, MN
heintz@umn.edu

Rankyung Hong
University of Minnesota
Minneapolis, MN
hongx293@umn.edu

Shivangi Singh
University of Minnesota
Minneapolis, MN
singh486@umn.edu

Gaurav Khandelwal
University of Minnesota
Minneapolis, MN
khand052@umn.edu

Corey Tesdahl
University of Minnesota
Minneapolis, MN
tesd0005@umn.edu

Abhishek Chandra
University of Minnesota
Minneapolis, MN
chandra@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 only 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 based on an underlying graph processing system, and enables different design choices for the key implementation issues of partitioning a hypergraph representation. We implement MESH on top of the popular GraphX graph processing framework in Apache Spark. Using a variety of real datasets and experiments conducted on a local 8-node cluster as well as a 65-node Amazon AWS testbed, we demonstrate that MESH provides flexibility based on data and application characteristics, as well as scalability with cluster size. We further show that it is competitive in performance to HyperX, another hypergraph processing system based on Spark, while providing a much simpler implementation (requiring about 5X fewer lines of code), thus showing that simplicity and flexibility need not come at the cost of performance.**

I. Introduction

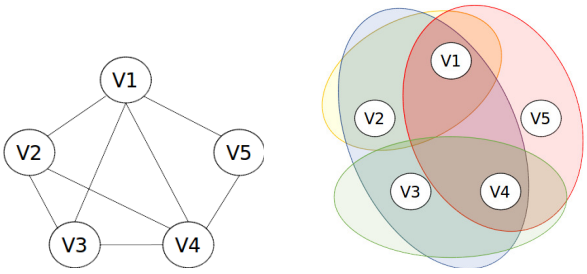
The advent of online social networks and communities such as Facebook and Twitter has led to unprecedented growth in user interactions (such as "likes", comments, photo sharing, and tweets), and collaborative activities (such as document editing and shared quests in multi-player games). This has resulted 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]–[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).

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] ranging from large-scale social graphs [6] to disease-gene networks [7]. 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.

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 [8] (as opposed to exactly two in a graph). Figure 1 illustrates the difference between a graph and a hypergraph. This figure shows a 5-vertex network, consisting of four groups ($\{v_1, v_2\}, \{v_1, v_2, v_3, v_4\}, \{v_1, v_4, v_5\}, \{v_3, v_4\}$). As can be seen from the figure, a graph can only capture binary relations (e.g., $\{v_1, v_2\}, \{v_3, v_4\}$, etc.), some of which may correspond to distinct overlapping groups (e.g., $\{v_3, v_4\}$ belongs to two distinct groups). On the other hand, a hypergraph can model all the groups unambiguously compared to a graph.

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



(a) Graph (b) Hypergraph

Fig. 1: A hypergraph can model groups unambiguously compared to a simple graph. Here, we have four groups $(\{v_1, v_2\}, \{v_1, v_2, v_3, v_4\}, \{v_1, v_4, v_5\}, \{v_3, v_4\})$.

From a system 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. One approach is to build a specialized system for hypergraph processing from scratch (e.g., HyperX [9]). While this 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. A different approach is to overlay a hypergraph processing system *on top of* an existing graph processing system. This approach 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. We take this approach, exploring the issues and tradeoffs involved therein to show its efficacy.

In this paper, we present MESH[2], a distributed hypergraph processing system based on a graph processing framework. We use our system to explore the key challenges on how to partition a hypergraph representation to allow efficient distributed computation in implementing a hypergraph processing system on top of a graph processing system. For our implementation, we choose the GraphX framework [1] in Apache Spark [10] due to its popularity and mature software eco-system, though 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, based on a graph processing framework.
- 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 (Section III).
- We explore the impact of the key design question in building a hypergraph processing system: how to partition hypergraph representations for distributed computation. We present multiple hypergraph partitioning algorithms and show how to map them to graph partitioning algorithms (Section IV).
- We implement a MESH prototype[3] on top of the GraphX graph processing system built on Apache Spark (Section V). Using this prototype and a number of real datasets and algorithms, we conduct experiments on a local 8-node cluster as well as a 65-node Amazon AWS testbed (Section V). We experimentally demonstrate that MESH provides the flexibility to make design choices based on data and application characteristics, and achieves scalability with cluster size. We further show that our MESH implementation is competitive in performance to HyperX [9] hypergraph processing system, while providing a much simpler implementation (requiring about 5X fewer lines of code), thus showing that simplicity and flexibility need not come at the cost of performance.

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.

[3]We have released the source code for our implementation, but do not reveal the repository for double-blind reasons.

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 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 [11], 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 [10]. As Figure 2 shows, MESH is positioned as a middleware layer between hypergraph applications and GraphX.

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. Application Programming Interface

In this section, we first discuss the features of hypergraph algorithms and then present the HPS API that can enable expressing such algorithms easily.

A. Hypergraph Algorithms

Many hypergraph algorithms can be viewed as generalizations of corresponding graph algorithms, but they can have

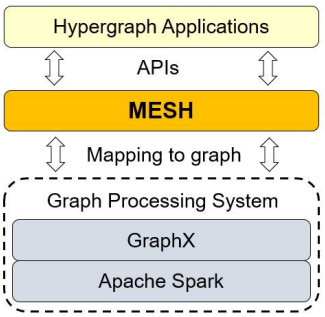


Fig. 2: MESH is implemented on top of a graph processing engine and provides an expressive and easy-to-use API to hypergraph applications.

richer attributes and computations, particularly those defined for hyperedges in addition to vertices. We examine some example hypergraph algorithms below to illustrate these aspects.

1) PageRank: Consider PageRank [12], 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.

2) Label Propagation: Consider a Label Propagation algorithm [9], [13], 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.

3) Shortest Paths: Consider the Single Source Shortest Paths algorithm that computes the shortest paths from a source

vertex to all other vertices in the network. In the hypergraph context, a path would be defined in terms of the hyperedges that are traversed from the source to each destination, and the path length would depend on the number of hyperedges along the path as well as any weights assigned to them. As an example, this can allow us to compute the degree of separation between two users in terms of the group structure of a social network. Conversely, one can also compute shortest paths between hyperedges (e.g., to identify how far two groups are in terms of the connectivity of their users).

Along these same lines, hypergraph extensions can be derived for many popular graph algorithms, such as connected components, centrality estimation [14], and more. The key to this expressiveness is the elevation of hyperedges to first-class status.

B. Core API

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).

```
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[Attr, InMsg, OutMsg] {
    def messageCombiner: MessageCombiner[OutMsg]
    def procedure: Procedure[Attr, InMsg, OutMsg]
  }
  type MessageCombiner[Msg] = (Msg, Msg) => Msg
  type Procedure[Attr, InMsg, OutMsg] =
    (Step, NodeId, Attr, InMsg, Context[Attr, OutMsg]) =>
        Unit
  trait Context[Attr, OutMsg] {
    def become(attr: Attr): Unit
    def send(msgF: NodeId => OutMsg, to: Dst): Unit
    def broadcast(msg: OutMsg): Unit = send(msg, All)
  }
}
```

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

Listing 1 shows the core of the MESH API[4]. 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). When a vertex (resp., hyperedge) broadcases a message, the message is sent to all hyperedges (resp., vertices) to which the vertex (resp., hyperedge) is incident on.

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[5] 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

We next show how we can use the MESH API to implement some of the algorithms discussed in Section III-A. Listing 2 shows the implementation of a hypergraph variant of the PageRank algorithm which computes ranks for both hyperedges and vertices iteratively. As seen from the pseudocode, it is fairly simple to implement the algorithm, requiring only a few lines of code. As shown in Listing 3, a richer version of PageRank which also computes the entropy of each hyperedge (PageRank-Entropy, as described in Section III-A1) requires a simple three-line helper function to compute entropy and changes to only a few other lines (broadcast and rank computation).

Implementing a Label Propagation Algorithm is also simple. Listing 4 shows how concisely we can implement this algorithm using our API. Finally, Listing 5 shows an implementation of the Single Source Shortest Paths algorithm using our API. In this algorithm, the attribute values of both hyperedges and vertices are updated incrementally: if the path length increases, this update is broadcast to neighbors. The algorithm

[4]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.

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

terminates when the attribute values of every hyperedge and vertex are less than the updated values they received. The major difference between the Shortest Path algorithm and the other algorithms above is that only a subset of hyperedges and vertices are active during any iteration (ones which were updated with newValue in the previous iteration). In contrast, every hyperedge and vertex is active in every iteration for the other algorithms. Note that for Label Propagation, PageRank and Shortest Paths, the **MessageCombiner** is derived automatically.

```
// Vertex procedure
val vertex: Procedure =
(superstep, id, attr, msg, ctx) => {
  val (totalWeight, rank) = msg
  val (vertexData, _) = attr
  // alpha = 0.15 (input from user)
  val newRank = alpha + (1.0 - alpha) * rank
  // Set its own vertex value
  ctx.become((vertexData, newRank))
  // Send data to neighbor hyperedges
  ctx.broadcast(newRank / totalWeight)
}
// Hyperedge procedure
val hyperedge: Procedure =
(superstep, id, attr, msg, ctx) => {
  val ((cardinality, weight), _) = attr
  val newRank = msg * weight
  // Set its own hyperedge value
  ctx.become(((card, weight), newRank))
  // Send data to neighbor vertices
  ctx.broadcast((weight, newRank / cardinality))
}
```

Listing 2: PageRank algorithm implementation.

```
// Entropy function
def entropy(ranks: Seq[Double]): Double = {
  val totalRank = ranks.sum
  val normalizedRanks = ranks.map(_ / totalRank)
  normalizedRanks.map {
    p => p * math.log(1/p) }.sum / math.log(2)
  }
// Vertex procedure
val vertex: Procedure =
(superstep, id, attr, msg, ctx) => {
    ...
    ctx.broadcast(Seq(newRank -> totalWeight))
}
// Hyperedge procedure
val hyperedge: Procedure =
(superstep, id, attr, msg, ctx) => {
    ...
    val newRank = msg.map {
      case (rank, totalWeight) =>
      rank * weight / totalWeight
    }.sum
    val newEnt = entropy(msg.map(_._1))
    ...
}
```

Listing 3: PageRank-Entropy algorithm implementation: Changes from PageRank implementation are shown here.

```
// Vertex procedure
val vertex: Procedure =
(superstep, id, attr, msg, ctx) => {
  val (vertexData, _) = attr
  val newLabel =
    if (superstep == 0) id
    else msg.max()
    // Set its own vertex value
    ctx.become((vertexData, newLabel))
    // Send data to neighbor hyperedges
    ctx.broadcast((newLabel))
}
// Hyperedge procedure
val hyperedge: Procedure =
(superstep, id, attr, msg, ctx) => {
  val (hyperedgeData, _) = attr
  val newLabel = msg.max()
  // Set its own hyperedge value
  ctx.become((hed, newLabel))
  // Send data to neighbor vertices
  ctx.broadcast((newLabel))
}
```

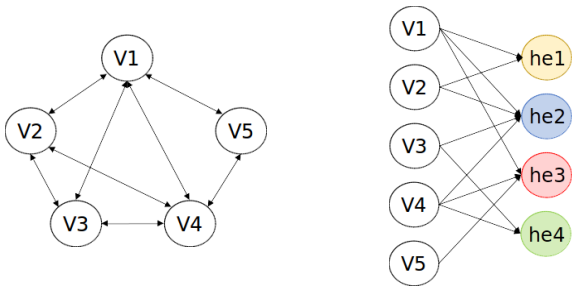
Listing 4: Label Propagation algorithm implementation.

```
// Vertex procedure
val vertex: Procedure =
(superstep, id, attr, msg, ctx) => {
  val (vertexData, currentHop) = attr
  val (_, newHop) = msg
  if (currentHop > newHop) {
    // Set its own vertex value
    ctx.become((vertexData, newHop))
    // Send data to neighbor hyperedges
    ctx.broadcast(newHop + 1.0)
  }
}
// Hyperedge procedure
val hyperedge: Procedure =
(superstep, id, attr, msg, ctx) => {
  val ((card, _), currentHop) = attr
  val newHop = msg
  if (currentHop > newHop) {
    // Set its own hyperedge value
    ctx.become((card, _), newHop)
    // Send data to neighbor vertices
    ctx.broadcast((_, newHop))
  }
}
```

Listing 5: Shortest Path algorithm implementation.

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 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



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

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

choices and the tradeoffs in making these decisions, as well as our implementation on top of GraphX.

A. Representation

The first question we must address is how to represent a hypergraph as an underlying graph that is understandable by a graph processing system, and we consider two alternatives here.

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. We refer to this representation as the *clique-expanded graph*. Figure 3(a) shows the clique-expanded representation for the example hypergraph shown in Figure 1(b). 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 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. Figure 3(b) shows the bipartite representation for the example hypergraph from Figure 1(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 a 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[6], and associated computation. Further, MESH computation runs on only one of these sets at a time. An effective partitioning algorithm must therefore *differentiate between hyperedges and vertices*.

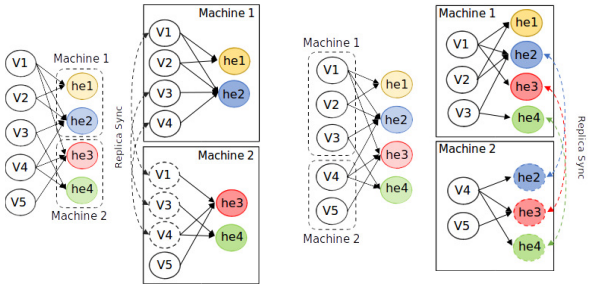
At the same time, hyperedge and vertex partitioning are fundamentally interrelated; an effective algorithm must *holistically partition hyperedges and vertices*. For example, an algorithm that partitions hyperedges without regard to vertex partitioning may achieve good computational load balance, but 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 (cutting edges[7]) or partition edges (thus cutting vertices) across machines. Many current systems [1], [11] 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, hypergraph vertices (resp., hyperedges) are each 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

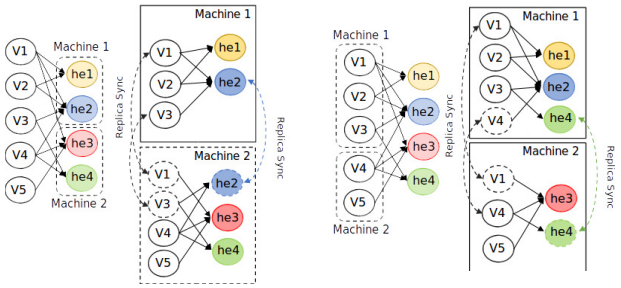
[6]The *degree* of a vertex denotes the number of hyperedges of which that vertex is a member. Similarly, the *cardinality* of a hyperedge denotes the number of vertices belonging to that hyperedge.

[7]Note that we use "edge" to refer to an edge in the *underlying* graph representation, and it is not to be confused with a hyperedge in the provided hypergraph.



(a) Random Vertex-cut (b) Random Hyperedge-cut

Fig. 4: Random partitioning strategies.



(a) Hybrid Vertex-cut (cutoff = 3) (b) Hybrid Hyperedge-cut (cutoff = 2)

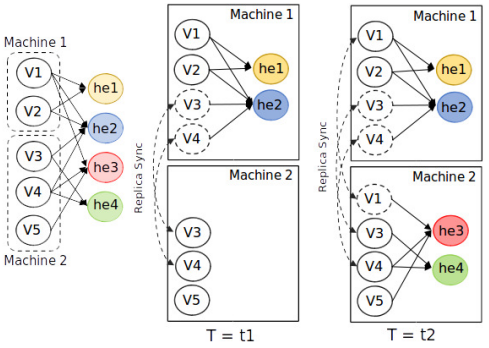
Fig. 5: Hybrid partitioning strategies.

its source and destination, then both vertices and hyperedges are effectively cut.

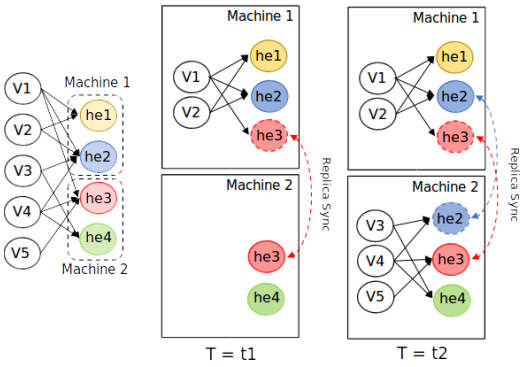
Any graph partitioning algorithm leads to a tradeoff between balancing computational load and minimizing network communication. While balancing the number of edges across machines could lead to good load balance, a high degree of replication of vertices can lead to increased network I/O and execution time due to increased syncing and state updates. 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*. We illustrate each of these algorithms by showing how they would partition our example hypergraph bipartite representation (Figure 3(b)) on two machines.

a) 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. For example, in Figure 4(a), the algorithm assigns each hyperedge to either machine1 or machine2 using a hash function. It then assigns a replica of each vertex to every machine which contains its incident hyperedges. E.g.: v_1 is assigned to machines 1 and 2, as it is incident on he_1 and he_2 (on machine 1), and he_3 (on machine 2). The *Random Hyperedge-cut* algorithm, on the other hand, partitions hyperedges and cuts vertices, as shown



(a) Greedy Vertex-cut



(b) Greedy Hyperedge-cut

Fig. 6: Greedy partitioning strategies.

in Figure 4(b).

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

b) Hybrid: The Hybrid algorithms we consider are based on the balanced ρ-way hybrid cut from PowerLyra [15]. These algorithms cut both vertices and hyperedges, but unlike Random Both-cut, they differentiate between vertices and hyperedges in doing so. The *Hybrid Vertex-cut* variant cuts vertices while partitioning hyperedges, except that it also cuts hyperedges with high cardinality (greater than 100 in our experiments). In Figure 5(a), the algorithm cuts vertices v_1 and v_3 while partitioning hyperedges and cuts hyperedge he_2 since it has cardinality greater than the cutoff value of 3 in the example. Similarly, the *Hybrid Hyperedge-cut* variant cuts hyperedges while also cutting high-degree vertices, as shown in Figure 5(b).

c) Greedy: Based on the Aweto [16] algorithm, the Greedy algorithms that we consider holistically partition hypergraphs with the goal of reducing replication and the resulting synchronization overhead. At 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 [16].)

Figure 6 illustrates the details of Greedy partitioning strate-

gies, showing how the strategies incrementally assign vertices and hyperedges. In Figure 6(a), the Greedy Vertex-cut algorithm first hash-partitions the bipartite graph edges based on their vertices and then assigns one hyperedge at a time to an appropriate machine. T = t1 in Figure 6(a) shows the intermediate state after hyperedges he_1 and he_2 have been assigned. Hyperedge he_1 is assigned to machine 1 because the machine contains maximum number of incident vertices at this time. Because load and overlap are even across machines at this time, hyperedge he_2 is randomly assigned to machine 1, and v_3 and v_4 are cut accordingly. T = t2 in the figure shows the final state of the partitioning. Hyperedges he_3 and he_4 are assigned to machine 2 because it contains maximum overlapping edges and the machine is also lightly loaded at this time. The *Greedy Hyperedge-cut* variant in Figure 6(b) works similarly, except it assigns vertices based on the overlap between their incident hyperedges and the hyperedges already assigned to each partition.

C. Implementation on 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 a VertexRDD and an EdgeRDD which internally extend Spark's Resilient Distributed Datasets (RDDs), an immutable and partitioned collection of elements [10]. VertexRDD contains information about vertex ids and vertex attributes. EdgeRDD contains information about edges (pairs of vertices) and edge attribute properties.

In our implementation, Hypergraph contains an additional HyperEdgeRDD which is similar to VertexRDD and contains information about hyperedge ids and hyperedge attributes. Moreover, EdgeRDD now contains information about (vertex, hyperedge) pairs and the attribute properties for the relation between them. We can represent a hypergraph using a clique 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 6.

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

Listing 6: Original GraphX partitioning abstraction.

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

Listing 7: Extended 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 7.

Unlike the getPartition method of GraphX, which receives source and destination vertices for an edge and returns partition number for that edge, the getAllPartitions method receives property graph corresponding to a pair of VertexRDD and EdgeRDD and returns an RDD which maps source and destination vertices with their associated partition number as shown in Listing 7. Next, we discuss how Hybrid and Greedy partitioning algorithms use this extended partitioning interface.

```
// mPrime: large prime number for better random assignment
val in_degrees = graph.edges.map(
   e => (e.dstId, (e.srcId, e.attr))).
   join(graph.inDegrees.map(e => (e._1, e._2))
)
in_degrees.map { e =>
   var part: PartitionID = 0
   if (Degree > degreeCutoff) {
      part = ((math.abs(srcId) * mPrime) % numParts).toInt
   } else {
      part = ((math.abs(dstId) * mPrime) % numParts).toInt}
   ((srcId, dstId), part))
}
```

Listing 8: Hybrid Vertex-cut PartitionStrategy.

```
// Count the number of edges corresponding to dstId
val groupedDst = graph.edges.map{
   e => (e.dstId, e.srcId)}.groupByKey
// Count the overlap for dstId with each partition .
val dstEdgeCount = groupedDst.map{ e =>
   var dstOverlap = new Array[Long](numParts)
   e._2.map{srcs => dstOverlap(
      (math.abs(srcs * mPrime) % numParts).toInt) += 1}
   (e._1, dstOverlap)
}
// Iterate to find most overlapping partitions
val FinalDstAssignment = dstEdgeCount.map { e =>
   var mostOverlap : Double =
      src.apply(part) - math.sqrt(1.0* current_load(part))
   for (cur <- 0 to numParts-1){
      val overlap : Double =
         src.apply(cur) - math.sqrt(1.0* current_load(cur))
      if (overlap > mostOverlap){
         part = cur
         mostOverlap = overlap
      }
   }
   (dst, part)
}
```

Listing 9: Greedy Vertex-cut PartitionStrategy.

Hybrid Vertex-cut PartitionStrategy uses different partitioning policy for low and high degree vertices. In this Partition-

Strategy, if the cardinality of a particular hyperedge exceeds the provided threshold (degreeCutoff), it cuts the hyperedge and partitions it based on the hashing of source vertex; otherwise, it partitions based on the hashing of destination vertex as shown in Listing 8. It is done to reduce communication overhead due to high degree hyperedges.

Listing 9 shows Greedy Vertex-cut PartitionStrategy which uses overlap and load for partitioning. This PartitionStrategy considers one hyperedge at a time and greedily chooses a partition that the hypervertices contained in this hyperedge most overlapped with. Additionally, if the load on the chosen partition is high, it picks the next most overlapped partition.

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. We show both partitioning time and execution time in our results.

2) Datasets: As inputs for our experiments, we use publicly available data [17] 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 [18] 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 the four applications described in Section III-C in our experiments: the Label Propagation algorithm (Listing 4), the two PageRank variants: PageRank (Listing 2) and PageRank-Entropy (Listing 3), and the Shortest Paths algorithm (Listing 5).

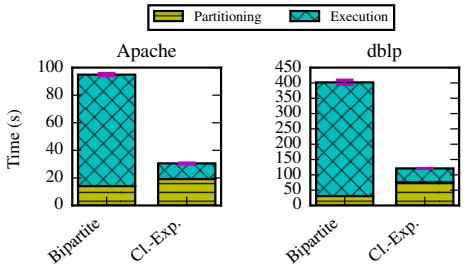


Fig. 7: Run time comparison for bipartite and clique-expanded representations for the PageRank algorithms on the Apache and dblp datasets. Friendster and Orkut results are not shown since their clique-expansions could not be materialized.

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 edges required for each of these representations, while Figure 7 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 multiple 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 in some cases, 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. As seen from Table I, these datasets would result in approx. 10 billion and 54 billion clique-expanded edges respectively, which is orders of magnitude higher than that for their bipartite graph representation.

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. Thus, while the clique-expansion representation might be beneficial for some use cases (specific algorithms and small datasets), it is neither expressive enough nor scalable in general. 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. Due to space constraints, we omit results for the Apache dataset here. Figure 8 shows both partitioning time and subsequent execution time for the Label Propagation algorithm for each of these policies for the dblp, Friendster, and Orkut datasets. Figures 9, 10, and 11 repeat these experiments for the PageRank, PageRank-Entropy, and Shortest Paths algorithms respectively.

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. First considering Figures 8 (Label Propagation) and 9 (PageRank), we see that the greedy hyperedge-cut algorithm is the best for the Friendster hypergraph (Figures 8(b) and 9(b)), 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 8(c) and 9(c)), 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 8(a) and 9(a)), 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.

Figures 10 and 11 shows the results for the PageRank-Entropy and Shortest Path algorithms, and their trends are very similar to those of the PageRank and Label Propagation algorithms, including the best partitioning strategy for each dataset. Note that the execution times of Shortest Path algorithm shown in Figure 11 are smaller than those of other algorithms because it terminates when messages are passed through the maximum distance between any two vertices, i.e., the diameter of the graph, whereas the other algorithms run more iterations until the values of vertices and hyperedges are converged or exceed the maximum number of iterations (30 for our experiments).

These results show that no one partitioning algorithm dominates all others in all cases. 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.

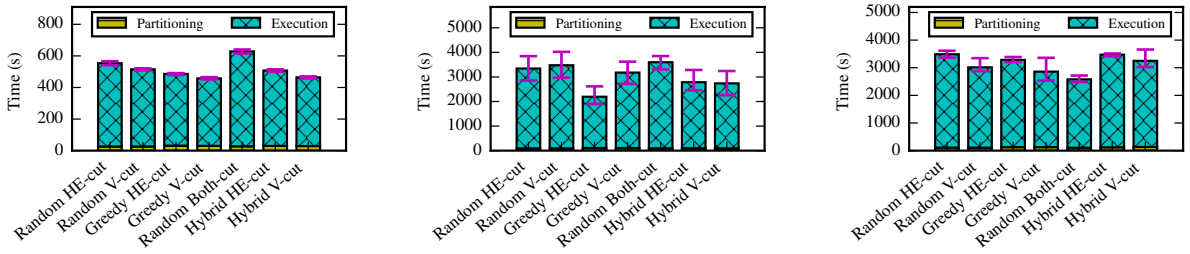
D. Scaling

Our next set of experiments examine the scaling of MESH based on available computing resources (size of the cluster and size of the Spark workers). These experiments are carried out on the 65-node Amazon AWS testbed. We show results for *strong scaling*, i.e., keeping the total dataset size the same, while changing the number of nodes and the number of Spark workers in the cluster. We execute Label Propagation algorithm for hybrid vertex-cut partitioning strategies and for the largest dataset, Orkut, with up to 65 amazon AWS m4.2xlarge instances, and measured the partitioning and execution times. We omit results for other smaller datasets due to space constraints.

Figure 12(a) shows the scaling results for the Orkut dataset by increasing the number of Spark workers, while keeping the number of nodes fixed. MESH is evaluated in a cluster with 1 master and 16 slaves (8 cores per node) and number of workers from ranging from 4 to 128. As the number of workers increases, both partitioning and execution time decrease, and we get diminishing returns in performance improvement as we saturate the physical cores of the cluster nodes.

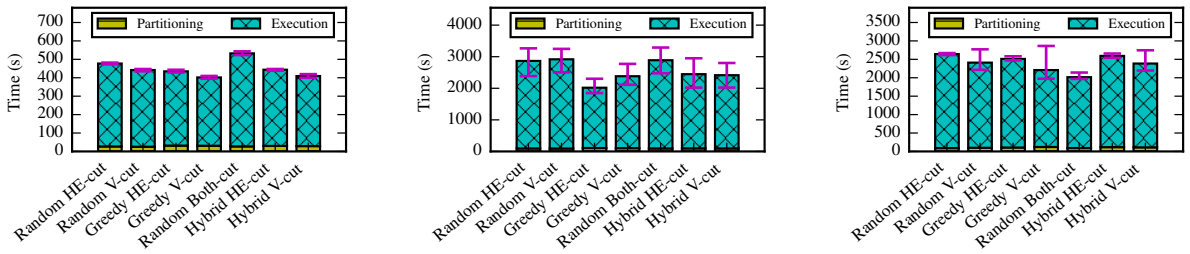
Figure 12(b) shows the scaling results for the Orkut dataset by increasing the number of nodes (number of slaves of a cluster) from 8 to 64 nodes. For example, 64-node cluster indicates 64 m4.2xlarge instances used for slaves and 1 m4.2xlarge for the master of the cluster. The partitioning and execution times for each cluster size are based on the best Spark cores setting for that cluster size. The figure shows the execution time decreases as the computing resources increase from 8 to 64 nodes. However, the partitioning time remains relatively constant even with larger set of computing resources, because a minimum amount of time is necessary for the completion of a partitioning task regardless of the cluster size.

Figure 14 shows the scaling results for the four datasets using the Random Both-cut partitioning strategy and Figure 13 shows detailed results for all partitioning strategies (Apache is omitted due to space constraints). We make the following



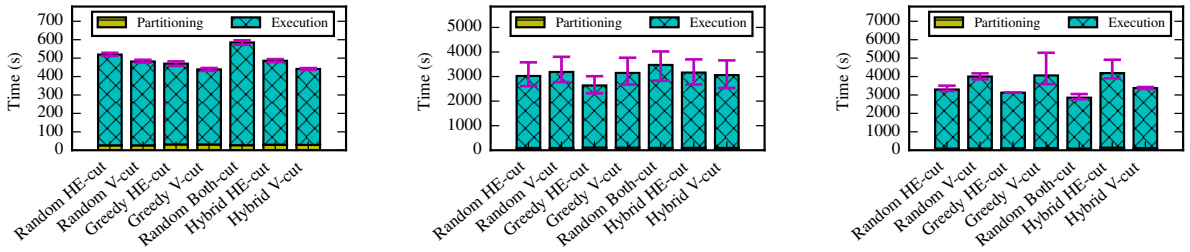
(a) dblp (b) Friendster (c) Orkut

Fig. 8: Label Propagation: Partitioning and execution time using several partitioning algorithms in MESH.



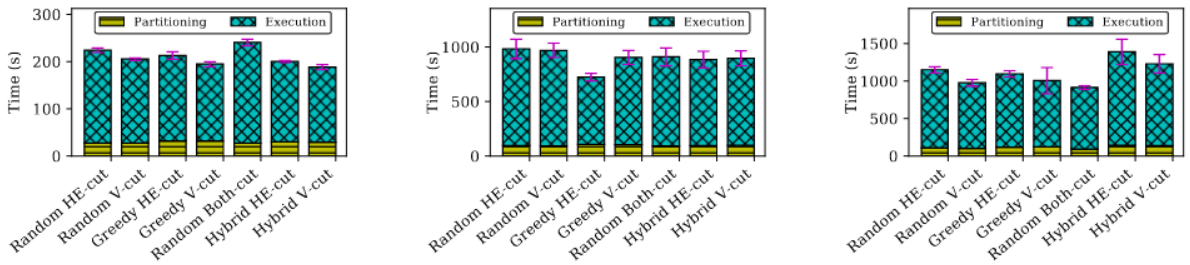
(a) dblp (b) Friendster (c) Orkut

Fig. 9: PageRank: Partitioning and execution time using several partitioning algorithms in MESH.



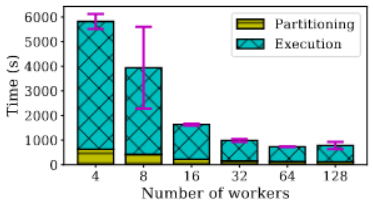
(a) dblp (b) Friendster (c) Orkut

Fig. 10: PageRank-Entropy: Partitioning and execution time using several partitioning algorithms in MESH.

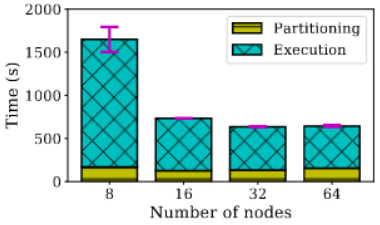


(a) dblp (b) Friendster (c) Orkut

Fig. 11: Shortest Paths: Partitioning and execution time using several partitioning algorithms in MESH.

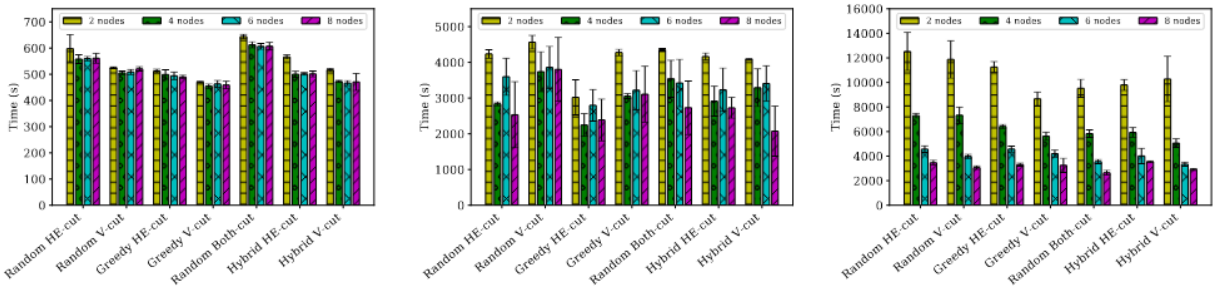


(a) Varying number of workers (Spark cores) on 16-node cluster, partitioning and execution time (10 iterations, Orkut)



(b) Varying the size of cluster, partitioning and execution time (10 iterations, Orkut).

Fig. 12: MESH system scalability.



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

Fig. 13: Partitioning and Label Propagation execution time for all partitioning strategies in various size of MESH clusters.

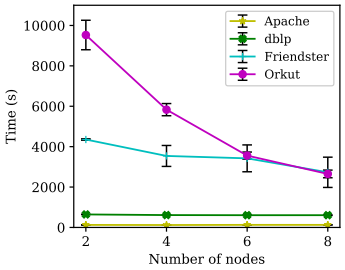


Fig. 14: Execution time for the Label Propagation algorithm on various size datasets with Random Both-cut partitioning strategy.

observations from these results. First, as seen from the figures, the execution times for all datasets either decreases or flattens out as the size of the cluster increases. The bigger the size of the dataset is, the greater the performance benefit. For instance, in Figure 13(c), for Orkut, the execution time keeps decreasing as the computing resources increase from 2 to 8 nodes, indicating a computational resource bottleneck at smaller cluster sizes. However, if the computing resources are sufficient to deal with a given dataset (e.g., Apache and dblp), the performance improvement obtained by running on a larger cluster becomes insignificant. For example, Figure 13(a) shows that for dblp, there is only a slight performance improvement when the size of the cluster increases from two to four nodes for all partition strategies. This means that a 2-node cluster is sufficient for the dblp dataset.

Figure 14 also shows that, although the Friendster and Orkut datasets have different input data sizes, their execution times are saturated around 3000 seconds with 8 nodes. Orkut dataset contains two times more the total numbers of vertices and hyperedges and 4.5 times more bipartite edges than Friendster dataset, but the maximum cardinality of Orkut is similar to that of Friendster (about 9,000 from Table I). Once the cluster is large enough that computational resources are sufficient, the network I/O becomes a bottleneck. Since the messages sent from one host to another are merged before they are sent out, the maximum cardinality influences the message sizes and hence, the total network traffic, leading to a similar performance for both Orkut and Friendster at larger cluster sizes.

E. Comparison with HyperX

To evaluate the overall performance, simplicity and flexibility of MESH, we compare it against HyperX [9], 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 top of Spark. While the HyperX implementation is optimized for hypergraph execution, our implementation relies on the GraphX optimizations designed for graph execution. Here, we evaluate the performance tradeoff given the simplicity and flexibility of our API and implementation.

TABLE II: MESH vs. HyperX (Scala Lines of Code)

LOC	MESH	HyperX
System core	630	2,620
Partition core	30	1,295
Partition algorithm	5 - 40	10 - 60
Total system	795	4,050
Applications	LP: 35, RW: 40	LP: 50, RW: 75

Simplicity and flexibility comparison. Table II shows the quantitative difference in the implementations of MESH and HyperX in terms of the lines of code (LOC) required for each system. System core code corresponds to the core system functionality such as handling hypergraph construction, processing, etc. Partition core code is specifically related to basic partitioning features. Partition algorithm indicates the range of lines of codes needed to implement a particular partitioning algorithm (e.g., Hybrid Vertex-cut) based on the partition and system cores. Total system is the sum of LOCs for core and partition algorithms.

We see that HyperX requires 5 times more LOCs, compared to MESH since it directly builds on top of Spark. In terms of partition modules, HyperX needs 43 times more LOCs to partition a hypergraph in their system than MESH, as MESH is able to take advantage of the basic partitioning functionality provided by GraphX. Besides, HyperX requires slightly more LOCs to implement a particular partitioning algorithm compared to MESH. This shows that MESH is much simpler and more flexible compared to HyperX, allowing various system features and partitioning algorithms for distributed hypergraph computation. In addition, in terms of ease of use, MESH applications can be implemented with fewer LOCs as compared to HyperX. For example, as shown in Table II, Label Propagation (LP) and Random Walk (RW) require 35 and 40 LOCs on MESH, compared to 50 and 75 for HyperX.

Performance comparison. We compare the performance of these two systems using a Label Propagation algorithm, specifically Listing 4 for MESH, and the provided example implementation for HyperX.[8]

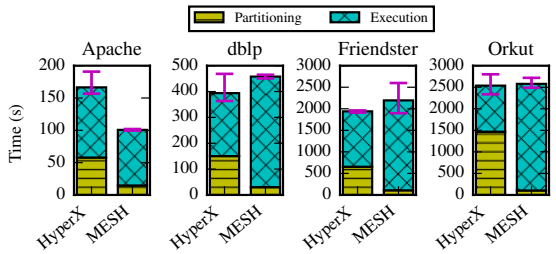


Fig. 15: Partitioning and Label Propagation execution time for MESH (using the best partitioning algorithm) and HyperX.

Figure 15 shows the partitioning and Label Propagation execution times (for 30 iterations) on the local 8-node cluster 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 [9]), leading to much higher partitioning times and comparable total running times.

In terms of performance, these results show the efficacy of MESH, which achieves comparable performance to HyperX, despite lacking several low-level optimizations. An additional qualitative benefit of MESH is its flexibility: hypergraphs are diverse, and MESH provides a simple interface that allows implementing different partitioning policies easily, as shown above.

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. Related Work

a) Graph Processing Systems: There has been a flurry of research on graph computing systems in recent years [2], [11], [19], and along with it, a great deal of work on performance evaluation and optimization [20]–[22]. Key among these systems, Pregel [2] introduced the "think like a vertex" model. GraphX [1], built upon Apache Spark [10], adopted a similar model while inheriting the scalability and fault tolerance of Spark's Resilient Distributed Datasets (RDD). GraphLab [23] 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) [24]. 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 [25]—building on top of Spark allows easier integration in broader data processing pipelines.

b) Graph and Hypergraph Partitioning: Graph Partitioning is a significant research topic in its own right. In the high-performance computing context, metis [26] provides very effective graph partitioning, and has open-source implementations for both single-node and distributed deployment. Its hMetis [26] cousin partitions hypergraphs, but no distributed implementation yet exists. The Zoltan toolkit from Sandia National Laboratories [27] includes a parallel hypergraph partitioner [28] that cuts both vertices and hyperedges.

In the distributed systems context, PowerGraph [11] targets natural (e.g., social) graphs by cutting vertices rather than edges. While this is effective for natural graphs, hypergraphs

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

require different approaches. Chen et al. have proposed novel algorithms for bipartite graphs [1] and skewed graphs [15], 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.

c) Hypergraph Processing: Hypergraphs have been studied for decades [5], [8] and have been applied in many settings, ranging from bioinformatics [7] to VLSI design [29] to database optimization [30]. Social networks have generally been modeled using simple graphs, but hypergraph variants of popular graph algorithms (e.g., centrality estimation [14], [31], shortest paths [32]) have been developed in recent years. HyperX [9] 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.

VII. Conclusion

We presented MESH, a flexible distributed framework for scalable hypergraph processing based on a graph processing system. 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 the key challenges in implementing a hypergraph processing system on top of a graph processing system: how to partition the hypergraph 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 and algorithms on a local cluster as well as an AWS testbed demonstrated that this flexibility does not come at the expense of performance, as even our unoptimized prototype performs comparably to HyperX.

Acknowledgments

This work is supported in part by NSF grant III-1422802.

References

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

[2] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: A system for large-scale graph processing," in *Proc. of SIGMOD ICMD*, 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, A. Pentland, L. Adamic, S. Aral, A.-L. Barabsi, D. Brewer, N. Christakis, N. Contractor, J. Fowler, M. Gutmann, T. Jebara, G. King, M. Macy, D. Roy, and M. Van Alstyne, "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] A. Sharma, T. J. Moore, A. Swami, and J. Srivastava, "Weighted simplicial complex: A novel approach for predicting small group evolution," in *Pacific-Asia Conference on Knowledge Discovery and Data Mining*. Springer, 2017, pp. 511–523.

[7] 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.

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

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

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

[11] 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.

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

[13] 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.

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

[15] 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.

[16] 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.

[17] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection," http://snap.stanford.edu/data, Jun. 2014.

[18] "The apache software foundation," http://www.apache.org/, 2017.

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

[20] M. Han, K. Daudjee, K. Ammar, M. T. Özsu, X. Wang, and T. Jin, "An experimental comparison of pregel-like graph processing systems," *Proc. VLDB Endow.*, vol. 7, no. 12, pp. 1047–1058, Aug. 2014.

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

[22] 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.

[23] 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.

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

[25] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia, "Spark SQL: Relational data processing in spark," in *Proc. of SIGMOD*, 2015, pp. 1383–1394.

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

[27] "Zoltan: Data management services for parallel applications," http://www.cs.sandia.gov/Zoltan/, 2016.

[28] 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.

[29] 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.

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

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

[32] 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.