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, 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 \( \{v_1, v_2\}, \{v_1, v_3, v_4\}, \{v_1, v_4, v_5\}, \{v_3, v_4\} \). 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.
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, a distributed hypergraph processing system designed for scalable hypergraph processing, based on a graph processing framework. We implement a MESH prototype 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 VI). 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. A hypergraph processing system should therefore be expressive enough to allow hyperedges to have attributes and computational functions just as vertices do. In addition, a hypergraph system should also provide ease of use, enabling application developers to easily write a diverse variety of hypergraph applications such as Label Propagation and Page Rank.

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 and ease of Implementation: A hypergraph processing system must be flexible, allowing various design choices (such as the choice of partitioning algorithm) to be based on data and application characteristics. In addition, the system itself must be easy to implement, enabling faster development.

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 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 VI). 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.

2Minnesota Engine for Scalable Hypergraph analysis

3We have released the source code for our implementation, but do not reveal the repository for double-blind reasons.
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. We examine some example hypergraph algorithms below to illustrate these aspects.

trait HyperGraph[V, E] {  
def compute[HE, V](  
maxIters : Int = 5,  
initialMsg : V,  
vProgram : Program[VD, V, VE],  
eProgram : Program[HD, HE, V]) : HyperGraph[VD, HE]  
}

object HyperGraph {  
trait Program[A, InM, OutM] {  
def messageCombiner : MessageCombiner[OutM]  
}

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

type Procedure[A, InM, OutM] =  
(Step, Noded, Attr, InM, Context[OutM]) => Unit  

trait Context[A, OutM] {  
def become(attr : Attr) : Unit  
def send(msgF : Noded => OutM, to : Dst) : Unit  
def broadcast(msg : OutM) : Unit = send(msg, All)  
}

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

1) Label Propagation: Consider a Label Propagation algorithm [9], [12], 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 propagating information 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.

2) PageRank: Consider PageRank [13], 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. 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.

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

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. 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 [10], the wildly popular and rapidly growing general-purpose data-intensive computing platform. Spark achieves good performance—especially on the iterative workloads that graph and hypergraph computing require—through its novel Resilient Distributed Dataset abstraction, which enables fault-tolerant distributed in-memory computing.

III. APPLICATION PROGRAMMING INTERFACE

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

In particular, we choose GraphX [1] as our foundation. 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 [10], the wildly popular and rapidly growing general-purpose data-intensive computing platform. Spark achieves good performance—especially on the iterative workloads that graph and hypergraph computing require—through its novel Resilient Distributed Dataset abstraction, which enables fault-tolerant distributed in-memory computing.

Along these same lines, hypergraph extensions can be derived for many popular graph algorithms, such as connected components, shortest path, 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).

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) broadcasts 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. Listings 2 and 3 show how concisely we can implement the Label Propagation and PageRank algorithm respectively using our API. As seen from the pseudocode, it is fairly simple to implement the algorithms, requiring only a few lines of code.

```
// Vertex procedure
val vertex: Procedure =
(superstep, id, attr, msg, ctx) => {
  val (vertexData, _) = attr
  // Set its own vertex value
  ctx.broadcast((vertexData, newLabel))
  // Send data to neighbor vertices
  ctx.broadcast((newLabel))
}

// Hyperedge procedure
val hyperedge: Procedure =
(superstep, id, attr, msg, ctx) => {
  val (hyperedgeData, _) = attr
  // Set its own hyperedge value
  ctx.broadcast((hed, newLabel))
  // Send data to neighbor hyperedges
  ctx.broadcast((newLabel))
}
```

Listing 2: Label Propagation algorithm implementation.

```
// 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.broadcast((vertexData, newLabel))
  // Send data to neighbor vertices
  ctx.broadcast((newLabel))
}

// Hyperedge procedure
val hyperedge: Procedure =
(superstep, id, attr, msg, ctx) => {
  val ((cardinality, weight), _) = attr
  val newRank = msg * weight
  // Set its own hyperedge value
  ctx.broadcast((cardinality, weight, newRank))
  // Send data to neighbor hyperedges
  ctx.broadcast((newLabel))
}
```

Listing 3: PageRank algorithm implementation.

Note that for Label Propagation and PageRank, the MessageCombiner is derived automatically. Several other hypergraph algorithms can be implemented in a similar fashion. Examples include a richer version of PageRank algorithm which computes the entropy of each hyperedge (PageRank-Entropy) and the single source shortest path algorithm. You can find other algorithms such as a richer version of PageRank algorithm which computes the entropy of each hyperedge (PageRank-Entropy) and the single source shortest path al-

\(^4\)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
algorithm in the MESH github. 6.

IV. HYPERGRAPH PARTITIONING

Next, we discuss one of the key issues for implementing a distributed hypergraph processing system: hypergraph partitioning. Before executing a hypergraph algorithm, the hypergraph data must be partitioned across the machines in the system. Since MESH is built on top of a graph processing engine, we first map the hypergraph to an underlying graph representation. We use a bipartite graph representation of the hypergraph, where one partition comprises exclusively vertices, and the other exclusively hyperedges, with low-level graph edges connecting hyperedges to their constituent vertices. We select this representation due to its general expressive power as well as compactness7. We next discuss the challenges in hypergraph partitioning and present some of the algorithms we have developed as part of MESH.

A. Partitioning 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/degree8, 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.

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.

B. Partitioning Algorithms

MESH utilizes the underlying graph partitioning framework to implement hypergraph partitioning algorithms. Many graph processing frameworks either partition vertices (cutting edges9) 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 its source and destination, then both vertices and hyperedges are effectively cut.

We explore a range of alternative partitioning algorithms that approach this goal from different angles. These algorithms fall into three classes: Random, Hybrid, and Greedy. We illustrate each of these algorithms by showing how they would partition our example hypergraph bipartite representation (Figure 3) 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 3(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 \( h_{e_1} \) and \( h_{e_2} \) (on machine 1), and \( h_{e_3} \) (on machine 2). The Random Hyperedge-cut algorithm, on the other hand, partitions hyperedges and cuts vertices. 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 \( \rho \)-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 3(b), the algorithm cuts vertices \( v_1 \)

---

6 https://github.com/mesh-umn/MESH
7 Other possible representations, such as a clique-expanded graph, which expands each hyperedge into a clique of its members, are highly unscalable (See section VI).
8 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.
9 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.
and $v_3$ while partitioning hyperedges and cuts hyperedge $he_2$
and 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.

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 3(c) illustrates the details of Greedy partitioning
strategies, showing how the strategies incrementally assign
vertices and hyperedges. In Figure 3(c), 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. $T = t_1$ in Figure 3(c) 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. $T = t_2$ 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 works similarly,
except it assigns vertices based on the overlap between their
incident hyperedges and the hyperedges already assigned to
each partition.

V. 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 Ver-
texRDD 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 as a bipartite
diagram by creating edges between vertices and their hyperedges.

MESH has parallel edge triplets joining a vertex, a hyper-
edge, and their edge, and allocates the parallel edge triplets to
multiple processes across cluster nodes through a partitioning
algorithm. Based on a bulk-synchronous parallel messaging
abstraction, MESH executes a series of super steps for the
parallel hypergraph processing. There are two sub-super steps
in a single super step. First sub-super step is that hyperedges
receive the sum of their inbound messages from vertices in
the previous super step. Second sub-super step is that vertices
receive the sum of their inbound messages from hyperedges in
the same super step. GraphX uses an edge-partitioning algo-

```python
def getPartition(
    src: VertexId,
    dst: VertexId,
    numPart: PartitionId): PartitionId
Listing 4: Original GraphX partitioning abstraction.
```

```python
def getAllPartitions[VD, ED](
    graph: Graph[VD, ED],
    numPartitions: PartitionId,
    degreeCutoff: Int)
: RDD[((VertexId, VertexId), PartitionId)]
Listing 5: Extended GraphX partitioning abstraction.
```

We use the built-in GraphX partitioning algorithms to
implement the baseline Random partitioning algorithms de-
scribed above. Our Greedy and Hybrid algorithms, however,
require a broader view of the graph (to compute “overlap”
and degree/cardinality, respectively). To satisfy this require-
ment, 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.

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 5. Partitioning algorithms in MESH use this extended partitioning interface.

Hybrid Vertex-cut PartitionStrategy uses different partitioning policy for low and high degree vertices. In this PartitionStrategy, 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 6.

```
// 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)
}
```


VI. EVALUATION

A. Experimental Setup

1) Deployment: We use two testbeds for our experiments: a local 8-node cluster and a 65-node Amazon AWS testbed. The local cluster consists 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 TB hard drive with at least 75% free space, and nodes are connected via gigabit ethernet. The Amazon AWS testbed consists of up to 65 Amazon AWS m4.2xlarge instances (8 cores and 32GB memory per instance). Each Spark worker uses 20GB memory, and the number of workers (cores) allocated for a single Spark application varies as per the experimental settings. We implement and run our MESH prototype on top of Apache Spark 1.6.0, and input data are stored in HDFS 2.7.2, which runs across all the nodes. For each cluster size, the input dataset was reloaded into HDFS restricted to the given set of cluster nodes. We show both partitioning time and execution time in our results. Throughout this paper, we run each experiment multiple times and plot the mean, with error bars denoting 95% confidence intervals.

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.

From Table I, we also observe that the bipartite representation is much more scalable compared to the clique-expanded graph representation, whose space overhead can be large or even prohibitive. For instance, the Friendster and Orkut hypergraphs 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.

3) Applications: We use the two applications described in Section III-C in our experiments: the Label Propagation (Listing 2) and PageRank (Listing 3) algorithms. Results for experiments conducted with other applications, PageRank-Entropy and Shortest Paths algorithms, are omitted due to space constraints. However, those experimental results and details can be found in our technical report10.

B. Partitioning

We evaluate the partitioning policies described in Section IV. Due to space constraints, we omit results for the Apache dataset here. Figure 4 (Label Propagation) and Figures 5 (RageRank) show both partitioning time and subsequent execution time on the local cluster for the Label Propagation and PageRank algorithms for each of these policies for the dblp, Friendster, and Orkut datasets.

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. We see that the greedy hyperedge-cut algorithm is the best for the Friendster hypergraph (Figures 4(b) and 5(b)), where vertices outnumber hyperedges. Here, cutting

TABLE I: Datasets used in our experiments.

<table>
<thead>
<tr>
<th>Dataset</th>
<th># Vertices</th>
<th># Hyperedges</th>
<th>Max. Degree</th>
<th>Max. Cardinality</th>
<th># Bipartite Edges</th>
<th># Clique-Expanded Edges</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apache</td>
<td>3316</td>
<td>78,080</td>
<td>4,507</td>
<td>179</td>
<td>408,231</td>
<td>196,342</td>
</tr>
<tr>
<td>dblp</td>
<td>899,393</td>
<td>782,659</td>
<td>368</td>
<td>2,803</td>
<td>2,624,912</td>
<td>21,707,061</td>
</tr>
<tr>
<td>Friendster</td>
<td>7,944,949</td>
<td>1,620,991</td>
<td>1,700</td>
<td>9,299</td>
<td>23,479,217</td>
<td>10.3 billion (approximate)</td>
</tr>
<tr>
<td>Orkut</td>
<td>2,322,299</td>
<td>15,301,901</td>
<td>2,958</td>
<td>9,120</td>
<td>107,080,530</td>
<td>54.5 billion (approximate)</td>
</tr>
</tbody>
</table>

Fig. 4: Label Propagation: Partitioning and execution time using several partitioning algorithms in MESH.

Fig. 5: PageRank: Partitioning and execution time using several partitioning algorithms in MESH.

hyperedges while partitioning the larger set of vertices might lead to better computational load balancing. On the other hand, for the Orkut hypergraph (Figures 4(c) and 5(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 4(a) and 5(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.

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.

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.

C. 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 6(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 6(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.

D. 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)

<table>
<thead>
<tr>
<th>LOC</th>
<th>MESH</th>
<th>HyperX</th>
</tr>
</thead>
<tbody>
<tr>
<td>System core</td>
<td>630</td>
<td>2,620</td>
</tr>
<tr>
<td>Partition core</td>
<td>30</td>
<td>1,295</td>
</tr>
<tr>
<td>Partition algorithm</td>
<td>5 - 40</td>
<td>10 - 60</td>
</tr>
<tr>
<td>Total system</td>
<td>795</td>
<td>4,050</td>
</tr>
<tr>
<td>Applications</td>
<td>LP: 35, RW: 40</td>
<td>LP: 50, RW: 75</td>
</tr>
</tbody>
</table>

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 2 for MESH, and the provided example implementation for HyperX.\footnote{We modify the implementation in HyperX to compute over undirected hypergraphs.}

Figure 7 shows the partitioning and Label Propagation execution times for MESH (using the best partitioning algorithm) and HyperX.

Figure 7 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.

VII. RELATED WORK

a) Graph Processing Systems: There has been a flurry of research on graph computing systems in recent years [2], [11], [19]–[21], and along with it, a great deal of work on performance evaluation and optimization [22]–[24].

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 [25] provided a more fine-grained interface along with support for asynchronous computation. ForeGraph [20] and Lux [21] proposed a scalable graph processing on multi-FPGA and multi-GPU systems, respectively.

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) [26]. GraphX, however, is particularly compelling due to the popularity and growth of Spark. Further, by facilitating diverse views of the same underlying data, 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, Spinner [27] partitions graphs with trillions of edges by leveraging vertex-centric Pregel abstraction. Metis [28] provides very effective graph partitioning, and has open-source implementations for both single-node and distributed deployment. Its hMetis [28] cousin partitions hypergraphs, but no distributed implementation yet exists. The Zoltan toolkit from Sandia National Laboratories [29] includes a parallel hypergraph partitioner [30] that cuts both vertices and hyperedges. Facebook introduced a distributed vertex-centric algorithm [31] to partition large-scale hypergraph.

In the distributed system context, PowerGraph [11] 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 [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], [6], [8], [32] and have been applied in many settings, ranging from bioinformatics [7] to VLSI design [33] to database optimization [34]. Social networks have generally been modeled using simple graphs, but hypergraph variants of popular graph algorithms (e.g., centrality estimation [14], [35], shortest path [36]) 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.

VIII. 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, CNS-1619254 and CNS-1717834.

REFERENCES


