HelP: High-level Primitives For Large-Scale Graph Processing

Semih Salihoglu
Stanford University
semih@cs.stanford.edu

Jennifer Widom
Stanford University
widom@cs.stanford.edu

ABSTRACT

Large-scale graph processing systems typically expose a small set of functions, such as the *compute()* function of Pregel, or the *gather()*, *apply()*, and *scatter()* functions of PowerGraph. For some computations, these APIs are too low-level, yielding long and complex programs, but with shared coding patterns. Similar issues with the MapReduce framework have led to widely-used languages such as *Pig Latin* and *Hive*, which introduce higher-level primitives. We take an analogous approach for graph processing: we propose *HelP*, a set of high-level primitives that capture commonly appearing operations in large-scale graph computations. Using our primitives we have implemented a large suite of algorithms, some of which we previously implemented with the APIs of existing systems. Our experience has been that implementing algorithms using our primitives is more intuitive and much faster than using the APIs of existing distributed systems. All of our primitives and algorithms are fully implemented as a library on top of the open-source GraphX system.

1. INTRODUCTION

Processing large-scale graph-structured data is an important task for many applications across different domains, such as the web [36], social networks [27], biology [39], and many others. As graphs grow to sizes that exceed the memory of a single machine, applications perform their computations on distributed and highly-parallel shared-nothing systems, such as *MapReduce* [8] and *Hadoop* [14], Hadoop's iterative extensions [9, 4], or more specialized graph systems such as *Pregel* [21] and *PowerGraph* [12]. At the core of the APIs of these systems is a small set of functions, such as the *map()* and *reduce()* functions of MapReduce, the *compute()* function of Pregel, or the *gather()*, *apply()*, and *scatter()* functions of PowerGraph.

The benefit of these frameworks is that programmers concentrate on implementing a few specific functions, and the framework automatically scales the computation by executing these functions in parallel across machines. However, sometimes the functions are too low-level or restricted for the algorithmic task. For example, as observed in references [15, 17, 31], additional code and workarounds that use global data structures can be required to control the flow of some algorithms, yielding complex and long programs. In addition, custom code can be required for some commonly appearing operations, such as initializing vertex values or checking for convergence of computations. For performing large-scale data analysis tasks on Hadoop [14], there has been an emergence of higher-level languages such as Pig Latin [24] and Hive [35], which have seen wide adoption in the industry and research communities [10, 35]. These languages express data analysis tasks with high-level constructs and primitives, such as filters and grouping, and compile to the lower-level *map()* and *reduce()* functions of Hadoop. We believe similar higher-level primitives for graph analysis tasks would be very useful for programming large-scale graph computations.

GRADES'14, June 22 - 27 2014, Snowbird, UT, USA

http://dx.doi.org/10.1145/2621934.2621938 .

In a previous paper [33], we implemented in detail several graph algorithms on an open-source Pregel clone. In the process, we observed certain patterns emerge from our implementations that we believed could be abstracted to a useful set of higher-level graph processing primitives. We implemented an additional suite of algorithms to verify our sense of the most generally useful primitives. This paper reports on the primitives we identified, called *HelP*, and the implementation of many graph algorithms using the primitives.[1] For reference, Tables 1 and 2 list our primitives and the algorithms we have implemented using our primitives, respectively.

This paper is a short version of a longer online technical report [32]. Readers and interested developers wishing for full coverage, including the implementation of our primitives on the GraphX system [37], are referred to that paper.

Our HelP primitives can be grouped broadly into three areas, according to the operations they perform:

1. **Vertex-centric Updates**: Update the values of some or all of the vertices in parallel. An update of a vertex value may use information from edges incident to the vertex (hereafter "local edges"), or from the values of other vertices. Updates can happen in iterations or in a single iteration.
2. **Topology Modifications**: Modify the topology of the graph by removing some vertices or edges based on filtering conditions, or merging multiple vertices together to form new vertices.
3. **Global Aggregations**: Perform a global aggregation operation over some or all of the graph (e.g., find the average degree, or find the vertex furthest from a given one).

All of the primitives in HelP abstract parallel operations that are suitable for scalable distributed implementations. We have implemented the HelP primitives fully in GraphX [37].

The specific aspects of our work covered in this short paper are

[1] primitives and algorithms target synchronous graph processing engines, e.g., [11, 21, 31]. Identifying high-level primitives for asynchronous graph systems is an interesting future research direction [12, 20].

Primitive	Description
Filter	Removes some vertices or edges from the graph.
Aggregating Neighbor Values (ANV)	Some vertices aggregate some or all of their neighbor values to update their own values. Appears both as a one-step computation and as an iterative process. In the iterative version, some or all of the vertices start propagating a value to their local neighbors, which are aggregated and propagated further by receiving vertices in the next iteration. The propagations continue until vertex values converge.
Local Update of Vertices (LUV)	Updates vertex values, possibly using global information or local edges of each vertex. Used mainly for initializing vertex values.
Update Vertices Using One Other Vertex (UVUOV)	Updates vertex values by using a value from one other vertex (not necessarily a neighbor). Commonly used in matching-like algorithms.
Form Supervertices (FS)	Merges groups of vertices.
Aggregate Global Value (AGV)	Computes a single global value over the graph. A commonly used special case is picking a random vertex from the graph.

Table 1: Primitives.

Name	Filter	ANV	LUV	UVUOV	FS	AGV
PageRank [21]		x	x			
HITS [19]		x	x			x
Single Source Shortest Paths [21]		x	x			
Weakly Connected Components [18]		x	x			
Conductance [2]		x				x
Semi-clustering [21]		x	x			x
Random Bipartite Matching [21]	x		x	x		
Approx. Betweenness Centrality [1]		x	x			x
Diameter Estimation (Double Fringe) [28]		x	x			x
Strongly Connected Components [33]	x	x	x			x
Minimum Spanning Forest [33]	x		x	x	x	
Approx. Maximum Weight Matching [33]	x			x		
Graph Coloring [33]	x	x	x			
Maximal Independent Set [33]	x	x	x			
K-core [28]	x					x
Triangle Finding [28]						
Clustering Coefficient [28]	x					
K-truss [28]						
Multilevel clustering [23]	x			x	x	

Table 2: Algorithms.

as follows:

- In Section 2, we describe three primitives and their variations, all of which perform vertex-centric updates: (1) `updateVertices`; (2) `aggregateNeighborValues`; and (3) `updateVertexUsingOneOtherVertex`.
- In Section 3, we describe the `filter` and `formSupervertices` primitives and their variations, all of which modify the topology of the graph.
- In Section 4, we describe the `aggregateGlobalValue` primitive, which performs a global aggregation operation over the vertices of the graph.

Section 5 briefly covers related work; for more discussion see [32]. Section 6 concludes and proposes future work.

All of the HelP primitives and algorithms described in the paper are implemented on top of GraphX and publicly available [16]. Background on Spark [38] and GraphX [37], as well as the implementation of our primitives, is covered in [32].

Notably absent from this work is any experimental evaluation. Evaluating programmer productivity is difficult to do objectively and convincingly. Lines of code is one possible metric to evaluate productivity gains of new APIs and languages, however the shortcomings of LOC as a measure have long been observed [30]. For the algorithms in Table 2, we saw up to 2x code reduction using our primitives against coding in GraphX without them. We could also evaluate the performance of algorithms implemented using our primitives against using other libraries on top of GraphX, or programming in GraphX directly. However, all of these approaches, when programmed carefully, translate to similar GraphX and Spark calls at the lowest levels, and thus similar expected performance. Finally, evaluating HelP against other graph processing systems would yield the same comparison as those systems against GraphX, as covered in [37].

2. VERTEX-CENTRIC UPDATES

In this section and the following two we present our HelP primitives. For each primitive we first give a high-level description of the primitive, then we give one or more examples from our algorithms that use the primitive. For interested readers, our full online technical report [32] describes the implementations of our primitives in GraphX. We note that except for global aggregations (Section 4), all of our primitives return a new graph. In our code snippets, we omit the assignment of the returned graph to a new variable.

2.1 Local Update of Vertices

One of the most commonly appearing operations in distributed graph algorithms is to initialize the values of some or all of the vertices, either at the beginning of an algorithm, or at the start of each phase of an algorithm. Depending on whether the local edges of the vertices are used in the updating of vertices, we provide two primitives for this operation: `updateVertices` and `updateVerticesUsingLocalEdges`. `updateVertices` takes two inputs:

Input	Description
dir	The direction of the local neighbors whose values will be aggregated
nbrP	A predicate to select which neighbors to aggregate
vP	A predicate to select which vertices to update
aggregated-ValueF	A function that takes a neighbor vertex value and returns the relevant part to be aggregated, possibly the entire neighbor value itself
aggregateF	The aggregation function
updateF	A function that takes a vertex and an aggregated value of the neighbors and returns a new value for the vertex

(a) aggregateNeighborValues.

Input	Description
dir	The direction of the local neighbors to propagate values to
startVP	A predicate to select which vertices to start propagating from
propagated-ValueF	A function that takes a vertex value and returns the relevant part to be propagated
propagate-AlongEdgeF	A function that takes two inputs: (1) the propagated value of a vertex; and (2) an edge value, through which the vertex will propagate its value to a neighbor; and computes the final propagated value along the edge
aggregateF	A function to aggregate propagated values
updateF	A function that takes a vertex and an aggregated value of the propagated values from neighbors and returns a new value for the vertex

(b) propagateAndAggregate.

Figure 1: Inputs to aggregateNeighborValues and propagateAndAggregate.

- vP: A predicate to select which vertices to update.
- updateF: A function that takes a vertex and returns a new value for the vertex.

Not surprisingly, the behavior of updateVertices is to update the values of all vertices for which vP evaluates to true, with the updated vertex value returned by updateF. Other vertices remain unchanged. updateVerticesUsingLocalEdges takes an additional edge direction input dir which specifies whether the incoming, outgoing, or both types of incident edges are used in updateF.

2.1.1 Examples of Use

Consider the first step of PageRank [3], which initializes the value of each vertex to $\frac{1.0}{|V|}$. Using updateVertices, we can express this operations as follows:

```
g.updateVertices(v → true, v → { v.val.pageRank = 1.0/ g.numVertices; v;}
```

As an example use of updateVerticesUsingLocalEdges, consider the bipartite matching algorithm from [21]. The vertices of the input bipartite graph are divided into *L* and *R*, for left and right, respectively. In each iteration of the algorithm, every unmatched vertex v_l in *L* randomly picks one of its neighbors from *R*, say v_r, and stores the ID of v_r in the pickedNbr field of its value. We can express this operation using updateVerticesUsingLocalEdges as follows:

```
g.updateVerticesUsingLocalEdges(EdgeDirection.Out, v → v.isLeft,
  (v, edges) → {v.val.pickedNbr = edges.get(random(edges.size)).dstID})
```

2.2 Aggregating Neighbor Values

In another common form of vertex-centric update operation, vertices aggregate some or all of their neighbors' values to update their own values in parallel. PageRank, HITS, finding shortest paths from a single source, finding weakly-connected components, or computing the conductance of a graph, are some of the example algorithms that perform this operation. Aggregating neighbor values appears in algorithms as a one-step computation, or as an iterative process that continues until vertex values converge. We provide the aggregateNeighborValues primitive for the one-step version, and propagateAndAggregate for the iterative version.

aggregateNeighborValues: The inputs to aggregateNeighborValues are listed in Table 1a. vP is a predicate that selects which vertices to update. If v is a vertex whose value should be updated, the input dir and the nbrP predicate determine the neighbors of v whose values should be aggregated in updating the value of v. Function aggregatedValueF is applied to the values of these neighbors, the outputs of aggregatedValueF are aggregated using aggregateF, and finally updateF is applied on v and the output of aggregateF to compute the new value for v.

propagateAndAggregate: In some computations the aggregation of neighbor values continues in iterations until all vertex values converge. The common pattern of such computations is the following: In the first iteration, one or more vertices propagate a value to their neighbors. Vertices that receive propagated values aggregate them and update their own values. In the next iteration, all vertices whose values have changed propagate a new value to their neighbors. The propagation of values continues in iterations until all vertex values are stable.

The inputs of propagateAndAggregate are listed in Table 1b. Inputs dir, aggregateF, and updateF are the same as in aggregateNeighborValues. The startVP predicate selects which vertices propagate values in the first iteration. Similar to the aggregatedValueF input to aggregateNeighborValues, propagatedValueF extracts the relevant value to be propagated from a vertex. In addition, there is a propagateAlongEdgeF function that takes a propagated value val from a vertex, and an edge through which val will be propagated, and computes a possibly modified value to be propagated along the edge.

2.2.1 Examples of Use

As an example use of aggregateNeighborValues, consider executing a fixed number of iterations, 10 say, of the HITS algorithm [19] for ranking web-pages. In HITS, each vertex has a hub and an authority value. In one iteration of the algorithm, vertices first compute the sum of all of their incoming neighbors' hub values to update their authority values. Then, vertices aggregate their outgoing neighbors' authority values to update their hub values. We can express this computation using aggregateNeighborValues as follows:

```
for (i = 0; i < 10; ++i) {
  // Aggregate in-neighbors' hub values to update authorities
  g.aggregateNeighborValues(
    EdgeDirection.In /* direction of neighbors */,
    nbr → true /* which neighbors to aggregate */,
    v → true /* which vertices to update */,
    nbrVal → nbrVal.hub /* relevant neighbor value to aggregate */,
```

```
  AggrFnc.SUM /* aggregate neighbors' hub values by summation */,
  (v, aggrHubVal) → { v.val.authority = aggrHubVal; v; })

// Aggregate out-neighbors' authorities to update hub values
g.aggregateNeighborValues(
  EdgeDirection.Out, nbr → true, v → true,
  nbrValue → nbrValue.authority, AggrFnc.SUM,
  (v, aggrAuthVal) → { v.val.hub = aggrAuthVal; v; })}
```

As an example use of `propagateAndAggregate`, consider the weakly-connected components algorithm from [18]. The input is an undirected graph, and vertices store a `wccID` value, initialized to their own IDs. Initially each vertex `v` propagates its `wccID` to all of its neighbors. In iterations, each vertex `v` updates its `wccID` to the maximum of its own `wccID` and the `wccID` values propagated by its neighbors, then propagates its `wccID` further if it has changed. At convergence, all vertices with the same `wccID` value belong to the same connected component. We can express this computation using `propagateAndAggregate` as follows:

```
g.updateVertices(v → true, v → { v.val.wccID = v.ID}
g.propagateAndAggregate(
  EdgeDirection.Either /* direction of neighbors */,
  v → true /* which vertices to start propagating from */,
  vVal → vVal.wccID /* propagated value */,
   /* do not change propagated value along each edge */
  (propagatedVal, edgeVal) → propagatedVal,
  AggrFnc.MAX /* aggregate wccIDs by taking their max */,
  (v, aggrWccIDVal) → {
   v.val.wccID = AggrFnc.MAX(v.val.wccID, aggrWccIDVal); v;})
```

2.3 Update Vertices Using One Other Vertex

In some algorithms, vertices store a pointer to (actually an ID of) one other vertex, not necessarily a neighbor, in a field of their vertex value, and either in parallel update their own value using the value of the vertex they point to, or vice-versa. This operation appears commonly in matching algorithms, but it also appears in the minimum spanning tree algorithm from [7, 33] or the clustering algorithm from [23]. If `v` stores the ID of `w` in its value, we refer to `v` as the *pointer* vertex and `w` as the *pointed* vertex. Depending on whether the pointer or the pointed vertex is updated, we provide two primitives for this operation: `updateSelfUsingOneOtherVertex` and `updateOneOtherVertexUsingSelf`. The inputs and behavior of the two primitives are very similar, so we specify them only for `updateSelfUsingOneOtherVertex`.

The inputs of `updateSelfUsingOneOtherVertex` are listed in Table 3. The behavior of `updateSelfUsingOneOtherVertex` is to update each vertex `v` for which `vP` evaluates to true in three steps: (1) compute the ID of the vertex `w` that `v` points to by applying `otherVertexIDF` on `v`; (2) apply `relevantPointedVertexValueF` to `w` to extract the relevant value of `w` that will be used in updating `v`; and (3) apply `updateF` on `v` and the output of `relevantPointedVertexValueF` from the second step to compute the new value for `v`. We note that in `updateOneOtherVertexUsingSelf`, multiple vertices might point to and update the same vertex. As a result `updateOneOtherVertexUsingSelf` takes an additional input function `aggregateF`, which aggregates the relevant pointer vertex values, before applying`updateF`.

2.3.1 Examples of Use

Consider the minimum spanning tree algorithm from [22, 33]. In each iteration of this algorithm, each vertex `v` points to its minimum-weight neighbor, which is stored in a `pointedV` field. As shown in [7], the vertices and their picked neighbors form disjoint subgraphs called *conjoined-trees*: two trees joined by a cycle. Figure 2 shows an example conjoined tree. We refer to the vertex with the smaller ID in the cycle of a conjoined tree `T` as the *root* of `T`, for example vertex 5 in Figure 2. After picking their neighbors, vertices find the root of the conjoined-tree they are part of iteratively as follows. In the first iteration, each vertex `v` discovers whether it is the root by checking whether `v`'s `pointedV u` points back to `v`, and whether `v`'s ID is smaller than `u`. If both conditions hold, `v` is the root and sets its `pointsAtRoot` field to true. In the later iterations, each vertex `v` copies its `pointedV u`'s `pointedV` and `pointsAtRoot` values to itself until every vertex points to the root. We can express this computation using `updateSelfUsingOneOtherVertex` as follows (we will describe the `aggregateGlobalValue` primitive in Section 3):

Input	Description
`vP`	A predicate to select which vertices will update themselves
`otherVertexIDF`	A function that takes the vertex value, say of `v`, and returns the ID of the other vertex that `v` points to
`relevantPointedVertexValueF`	A function that takes the pointed vertex's value and returns the relevant part of it that will be used in updating the pointer vertex
`updateF`	Takes the vertex `v` and the relevant value of the vertex that `v` points to and returns a new value for `v`

Table 3: Inputs to `updateSelfUsingOneOtherVertex`.

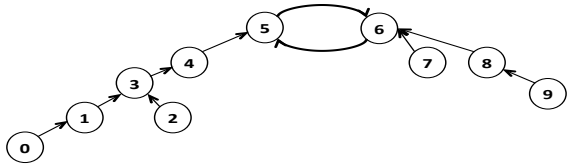

Figure 2: Example of a conjoined-tree.

```
// discover the root of each conjoined-tree
g.updateSelfUsingOneOtherVertex(
  v → true, /* which vertices to update */
  v → v.val.pointedV /* ID of the other vertex */,
  otherV → (otherV.val.pointedV, otherV.ID), // relevant pointedV value
  // updateF: set pointsAtRoot to true if v is the root
  (v, (otherVsPointedV, otherVID) → {
    if (v.ID == otherVsPointedV  v.ID < otherVID) {
      v.val.pointsAtRoot = true}}; v;))

var numNotPointing = g.aggregateGlobalValue(
  v  → { (v.val.pointsAtRoot) ? 0 : 1), AggrFnc.SUM)
while (numNotPointing > 0) {
  g.updateSelfUsingOneOtherVertex(
    v → !v.val.pointsAtRoot,
    v → v.val.pointedV,
    otherV → (otherv.val.pointedV, otherV.pointsAtRoot),
    (v, (otherVsPointedV, otherVPointsAtRoot) → {
      v.val.pointedV = otherVsPointedV;
      v.val.pointsAtRoot = otherVPointsAtRoot}}))
  numNotPointing = g.aggregateGlobalValue(
    v  → { (v.val.pointsAtRoot) ? 0 : 1), AggrFnc.SUM)}
```

3. TOPOLOGY MODIFICATIONS

We propose two primitives that change the topology of the graph: (1) **Filter**: removes certain vertices and edges from the graph; and (2) **Forming Supervertices**: merges multiple vertices together to form *supervertices*. Due to space constraints and the fact that filtering incurs no unusual challenges, we omit describing our three filter primitives: `filterVertices`, `filterVerticesUsingLo-`

Input	Description
superver-texIDF	A function that takes the vertex value, say of v, and returns the ID of v's supervertex
mergeVer-texValuesF	A function that takes a set of vertex values and returns a single merged value for the supervertex
mergeEdge-ValuesF	A function that takes a set of edge values for edges between the same pair of supervertices and returns a single merged value for the edge

Table 4: Inputs to formSupervertices.

calEdges, and filterEdges. The interested reader is referred to the full online technical report [32].

3.1 Forming Supervertices

One operation that modifies the topology of the graph is to merge groups of vertices into *supervertices*. This operation appears in Boruvka's minimum spanning tree algorithm [22] and in some partitioning and clustering algorithms, such as the clustering algorithm from [23]. In these algorithms, after some computation every vertex identifies a supervertex (possibly itself) that it will merge into. Then:

- All vertices and their values that belong to the same supervertex are merged into a single vertex. How the vertex values are merged is specified by an input function.
- Consider an edge (u,v) and assume that vertices u and v are merged into supervertices s1 and s2, respectively. If s1 = s2, then (u,v) is removed from the graph. Otherwise, (u,v) becomes an edge between s1 and s2. If there are multiple edges between s1 and s2, then edges are merged. How the edge values are merged is specified by an input function.

The inputs of formSupervertices are listed in Table 4. As with all HelP primitives, formSupervertices is also suitable for parallel implementations. The details of its implementation on GraphX can be found in [32].

3.1.1 Examples of Use

As an example, consider Boruvka's minimum spanning tree algorithm, which we discussed in Section 2.3.1. In each iteration, vertices discover the root of the conjoined-tree they are part of and store its ID in a pointedV value. The algorithm uses these values to merge all vertices in each conjoined-tree into a single supervertex. For this algorithm, function mergeVertexValuesF simply returns a new empty vertex value because the algorithm does not need to merge the values of vertices that form the supervertex. However, edges are weighted; function mergeEdgeValuesF takes minimum of the edge values. We can express this computation using formSupervertices as:

```
g.formSupervertices(
  v → v.val.pointedV /* ID of the supervertex */
  vVals → new EmptyMSTVertexValue(), /* mergeVertexF */
  eVals → AggrFnc.MIN(eVals) /* mergeEdgeF */)
```

4. GLOBAL AGGREGATIONS

Many graph algorithms need to compute a global value over the vertices of the graph, such as counting the number of vertices with a particular value, or finding the maximum vertex value. Specifically, each vertex emits a value, and the values are aggregated in some fashion to produce a single value. These computations can thus be seen as special MapReduce computations with a single reducer.

We abstract global aggregation in our aggregateGlobalValue and aggregateGlobalValueUsingLocalEdges primitives. aggregateGlobalValue takes two inputs:

- mapF: A function that takes a vertex and produces a value.
- reduceF: A function that takes a pair of values and aggregates them into a single value.

Notice that function reduceF combines a pair of values, rather than the set of all emitted values. Function reduceF is first applied to a pair of mapped values to produce a single value, the function is then applied to the result value with another mapped value, and so on until the entire set of mapped values has been processed to produce the final aggregated value. The order of application is unpredictable, so for correctness of the aggregation operation, reduceF should be commutative and associative. Our requirement of a pairwise reduceF function is primarily for efficiency (see [32] for our implementation).

Function aggregateGlobalValueUsingLocalEdges takes an additional dir input, and in this case mapF additionally takes the set of edges incident to the vertex in the given direction.

We note that a frequently used special case of a global aggregation operation is to pick a random vertex from the graph, which we expose as a separate pickRandomVertex primitive in GraphX.

4.1 Examples of Use

One example use of aggregateGlobalValue is to detect termination of the root finding phase of the minimum spanning tree algorithm. Recall from Section 2.3.1 that the algorithm iterates in this phase until all vertices find their roots. To compute whether all vertices have found their roots, the algorithm counts the number of vertices whose pointsAtRoot value is false, using primitive aggregateGlobalValue as follows:

```
numNotPointing = g.aggregateGlobalValue(
  v → { (!v.val.pointsAtRoot) ? 1 : 0}, AggrFnc.SUM)
```

Another example is the approximate betweenness-centrality algorithm from [1], which performs a breadth-first search (BFS) from a source vertex and labels each vertex with its level in the BFS tree. Then, the algorithm computes the maximum depth of the tree using primitive aggregateGlobalValue as follows:

```
maxDepth = g.aggregateGlobalValue(v → v.val.level, AggrFnc.MAX)
```

5. RELATED WORK

No other work we know of proposes and implements high-level primitives for distributed graph computations. We provide a very brief discussion of related work in this short paper; more extensive discussion and comparison against our approach is provided in the full online technical report [32]. In brief, past work divides into five categories, none of which corresponds directly to our approach based on a set of high-level graph-specific primitives:

- **Vertex-centric APIs [12, 21]**: Programmers implement a small set of functions, such as the compute() function of Pregel, to specify local messaging and value updating performed by each vertex. A distributed framework invokes the functions iteratively.
- **MapReduce-based APIs [4, 8]**: Programmers implement a series of map() and reduce() functions for each graph operation, usually performed on relational tables that store the graph.
- **Higher-level Data Analysis Languages [24, 35]**: Instead of map() and reduce() functions, programmers use higher-level data primitives, such as relational joins and grouping, to express their graph operations.
- **Green-Marl [17]**: A domain-specific language allows programmers to write their algorithms in an imperative fashion as if the the graph is stored in a single machine, then compiles to different parallel and distributed backends.
- **MPI-based Graph Libraries [5, 13]**: Programmers use graph libraries based on the *Message Passing Interface* (MPI), a standard interface for building parallel and distributed message-passing

programs. These libraries can execute on distributed architectures that have an MPI implementation, such as Open MPI [25].

6. CONCLUSIONS AND FUTURE WORK

We presented HelP, a set of high-level graph processing primitives, which we believe abstract the most commonly appearing operations in distributed graph computations. We described the implementations of many graph algorithms using our primitives. We have implemented all of our primitives on the GraphX system (see [32]). Our experience has been that implementing algorithms using the HelP primitives is more intuitive and much faster than using the APIs of existing distributed graph systems. For a discussion on the limitations of our primitives, interested readers are referred to our full online technical report [32], where we give examples of graph algorithms that benefit from using additional "data primitives" such as joins and grouping, either instead of or in addition to our HelP primitives. These examples are naturally expressed in an "edge-centric" fashion (rather than "vertex-centric"), such as triangle-finding, clustering coefficient, or finding k-trusses.

We outline several broad directions for future work.

- **Extending Existing High-level Languages:** Some of the existing high-level data analysis languages, such as Pig Latin [24] and Scalding [34], could be extended first with graph-specific data structures, and then with our HelP primitives. For example, in hypothetically extended Pig Latin, an algorithm that first loads a web graph from raw files and then finds its weakly-connected components could be expressed by a program like:

```
vid_links = LOAD '/raw_links.txt' AS (fromId, toId)
g = LOAD GRAPH vid_links WITH VERTEX VALUE wccId
g = UPDATE VERTICES wccId = ID
g = PROPAGATE AND AGGREGATE wccId FROM ALL
    USING MAX IN g
STORE graph INTO 'graph_wcc_output.txt'.
```

  Primitives that already exist in Pig Latin are in normal font and additional HelP primitives in bold.

- **Primitives for Asynchronous Graph Computations:** The HelP primitives target algorithms suitable for synchronous distributed engines, such as Pregel [21], Giraph [11], or MapReduce [8]. It may be possible to similarly identify and implement commonly-appearing operations as high-level primitives in asynchronous graph computations, such as Gibbs sampling [6], loopy belief propagation [26], or collaborative filtering [29], which are suitable for executing on asynchronous engines [12, 20].

7. REFERENCES

[1] D. A. Bader, S. Kintali, K. Madduri, and M. Mihail. Approximating Betweenness Centrality. In *WAW*, 2007.

[2] B. Bollobas. *Modern Graph Theory*. Springer, 1998.

[3] S. Brin and L. Page. The Anatomy of Large-Scale Hypertextual Web Search Engine. In *WWW*, 1998.

[4] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst. HaLoop: Efficient iterative data processing on large clusters. In *VLDB*, 2010.

[5] A. Buluç and J. R. Gilbert. The Combinatorial BLAS: Design, Implementation, and Applications. *International Journal of High Performance Computing Applications*, 25(4), 2011.

[6] G. Casella and E. I. George. Explaining the Gibbs Sampler. *The American Statistician*, 46(3), 1992.

[7] S. Chung and A. Condon. Parallel Implementation of Boruvka's Minimum Spanning Tree Algorithm. In *IPPS*, 1996.

[8] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI*, 2004.

[9] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S. Bae, J. Qiu, and G. Fox. Twister: A Runtime for Iterative MapReduce. In *HPDC*, 2010.

[10] A. F. Gates, O. Natkovich, S. Chopra, P. Kamath, S. M. Narayanamurthy, C. Olston, B. Reed, S. Srinivasan, and U. Srivastava. Building a high-level dataflow system on top of map-reduce: the pig experience. *VLDB*, 2(2), 2009.

[11] Apache Incubator Giraph. http://incubator.apache.org/giraph/.

[12] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs. In *OSDI*, 2012.

[13] D. Gregor and A. Lumsdaine. The Parallel BGL: A Generic Library for Distributed Graph Computations. In *POOSC*, 2005.

[14] Apache Hadoop. http://hadoop.apache.org/.

[15] P. Haller and H. Miller. Parallelizing Machine Learning-Functionally: A Framework and Abstractions for Parallel Graph Processing. In *Scala*, 2011.

[16] Help Primitives. https://github.com/semihsalihoglu/incubator-spark/tree/Help_Primitives_1.

[17] S. Hong, H. Chafi, E. Sedlar, and K. Olukotun. Green-marl: a dsl for easy and efficient graph analysis. In *ASPLOS*, 2012.

[18] U. Kang, C. E. Tsourakakis, and C. Faloutsos. PEGASUS: A Peta-Scale Graph Mining System – Implementation and Observations. In *ICDM*, 2009.

[19] J. M. Kleinberg. Authoritative Sources in a Hyperlinked Environment. *Journal of the ACM*, 46, 1999.

[20] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. GraphLab: A New Parallel Framework for Machine Learning. In *UAI*, 2010.

[21] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: A System for Large-Scale Graph Processing. In *SIGMOD*, 2011.

[22] J. Nešetřil, E. Milková, and H. Nešetřilová. "Otakar Borůvka on Minimum Spanning Tree Problem Translation of Both the 1926 Papers, Comments, History". *"Discrete Mathematics "*, 233(1–3), 2001.

[23] S. Oliveira and S. C. Seok. Multilevel Approaches for Large-scale Proteomic Networks. *International Journal of Computer Mathematics*, 84(5), 2007.

[24] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig Latin: A Not-So-Foreign Language for Data Processing. In *SIGMOD*, 2008.

[25] Open MPI. http://www.open-mpi.org/.

[26] J. Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann Publishers Inc., 1988.

[27] O. Phelan, K. McCarthy, and B. Smyth. Using Twitter to Recommend Real-time Topical News. In *RecSys*, 2009.

[28] L. Quick, P. Wilkinson, and D. Hardcastle. Using Pregel-like Large Scale Graph Processing Frameworks for Social Network Analysis. In *ASONAM*, 2012.

[29] P. Resnick and H. R. Varian. Recommender Systems. *Communications of the ACM*, 40, Mar. 1997.

[30] J. Rosenberg. Some Misconceptions About Lines of Code. In *METRICS*, 1997.

[31] S. Salihoglu and J. Widom. GPS: A Graph Processing System. In *SSDBM*, 2013.

[32] S. Salihoglu and J. Widom. HelP: High-level Primitives For Large-Scale Graph Processing. Technical report, Stanford University, March 2014. http://ilpubs.stanford.edu:8090/1085/.

[33] S. Salihoglu and J. Widom. Optimizing Graph Algorithms on Pregel-like Systems. In *VLDB*, 2014.

[34] Scalding Github Repository. https://github.com/twitter/scalding.

[35] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive: A Warehousing Solution Over a Map-Reduce Framework. *VLDB*, 2(2), 2009.

[36] S. Vadapalli, S. R. Valluri, and K. Karlapalem. A Simple Yet Effective Data Clustering Algorithm. In *ICDM*, 2006.

[37] R. S. Xin, J. E. Gonzalez, M. J. Franklin, and I. Stoica. GraphX: A Resilient Distributed Graph System on Spark. In *GRADES*, 2013.

[38] Zaharia, M. and Chowdhury, M. and Franklin, M. J. and Shenker, S. and Stoica, I. Spark: Cluster Computing with Working Sets. In *HotCloud*, 2010.

[39] C. Zhong, D. Miao, and R. Wang. A Graph-theoretical Clustering Method Based on Two Rounds of Minimum Spanning Trees. *Pattern Recognition*, 43(3), 2010.