High-Level Programming Abstractions for Distributed Graph Processing

Vasiliki Kalavri, Vladimir Vlassov, and Seif Haridi

Abstract—Efficient processing of large-scale graphs in distributed environments has been an increasingly popular topic of research in recent years. Inter-connected data that can be modeled as graphs appear in application domains such as machine learning, recommendation, web search, and social network analysis. Writing distributed graph applications is inherently hard and requires programming models that can cover a diverse set of problems, including iterative refinement algorithms, graph transformations, graph aggregations, pattern matching, ego-network analysis, and graph traversals. Several high-level programming abstractions have been proposed and adopted by distributed graph processing systems and big data platforms. Even though significant work has been done to experimentally compare distributed graph processing frameworks, no qualitative study and comparison of graph programming abstractions has been conducted yet. In this survey, we review and analyze the most prevalent high-level programming models for distributed graph processing, in terms of their semantics and applicability. We review 34 distributed graph processing systems with respect to the graph processing models they implement and we survey applications that appear in recent distributed graph systems papers. Finally, we discuss trends and open research questions in the area of distributed graph processing.

Index Terms—Distributed graph processing, large-scale graph analysis, big data

1 Introduction

Graphs are immensely useful data representations, vital to diverse data mining applications. Graphs capture relationships between data items, like interactions or dependencies, and their analysis can reveal valuable insights for machine learning tasks, anomaly detection, clustering, recommendations, social influence analysis, bioinformatics, and other application domains.

The rapid growth of available data sets has established distributed shared-nothing cluster architectures as one of the most common solutions for data processing. Vast social networks with millions of users and billions of user-interactions, web access history, product ratings, and networks of online game activity are a few examples of graph data sets that might not fit in the memory of a single machine. Such massive graphs are usually partitioned over several machines and processed in a distributed fashion. However, coping with huge data sizes is not the sole motivation for distributed graph processing. Graphs rarely appear as *raw* data. Instead, they are most often derived by transforming other data sets into graphs. Data entities of interest are extracted and modeled as graph nodes and their relationships are modeled as edges. Thus, graph representations frequently appear in an intermediate step of some larger distributed data processing pipeline. Such intermediate graph-structured data are already partitioned upon creation and, thus, distributed algorithms are essential in order to efficiently analyze them and avoid expensive data transfers.

The increasing interest in distributed graph processing is largely visible in both academia and industry. Plenty of research and industrial papers on parallel and distributed graph processing have been published over the past few years [20], [23] and graph processing is a prominent topic in the proceedings of prime data management conferences [13], [14], [36], [37], [45], [46], [80], [81]. At the same time, open-source distributed graph processing systems are gaining popularity [1], [36], [53], while general-purpose distributed data processing systems invest in implementations of graph libraries and connectors to graph databases [14], [25], [50], [51].

Writing distributed graph analysis applications is inherently hard. Computation parallelization, data partitioning, and communication management are major challenges of developing efficient distributed graph algorithms. Furthermore, graph applications are highly diverse and expose a variety of data access and communication patterns [19], [54]. For example, iterative refinement algorithms, like PageRank, can be expressed as parallel computations over the local neighborhood of each vertex. Graph traversal algorithms produce unpredictable access patterns, while graph aggregations require grouping of similar vertices or edges together.

To address the challenges of distributed graph processing, several high-level programming abstractions and respective system implementations have been recently proposed [1], [2], [11], [24], [55]. Even though some have gained more popularity than others, each abstraction is optimized for certain classes of graph applications. For instance, the popular vertex-centric model [2] is well-suitable for iterative value propagation algorithms, while the neighborhood-centric model [24] is

- *V. Kalavri is with ETH Zurich, Zurich 8092, Switzerland. E-mail: kalavriv@inf.ethz.ch.*
- *V. Vlassov and S. Haridi are with KTH Royal Institute of Technology, Stockhol 114 28, Sweden. E-mail: {vladv, haridi}@kth.se.*

Manuscript received 8 July 2016; revised 13 July 2017; accepted 3 Oct. 2017. Date of publication 12 Oct. 2017; date of current version 9 Jan. 2018. (Corresponding author: Vasiliki Kalavri.) Recommended for acceptance by F. Li. For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below. Digital Object Identifier no. 10.1109/TKDE.2017.2762294

TABLE 1
Notation Used for the Execution Semantics Pseudo-Code and Interfaces

Notation	Meaning
$G = (V, E)$	Directed graph
V	Set of vertices
E	Set of edges
(u, v)	Edge from u to v
S_v	State (value) associated with vertex v
$S_{(u,v)}$	State (value) associated with edge (u, v)
N_v^{out}	Set of first-hop out-neighbors of vertex v
N_v^{in}	Set of first-hop in-neighbors of vertex v
I	Vertex identifier
VV	Type of the vertex state (value)
EV	Type of the edge state (value)
M	Message type
T	Data type

designed to efficiently support operations on custom subgraphs, like ego networks. Unfortunately, there is no single model yet that can efficiently and intuitively cover all classes of graph applications.

Although several studies have experimentally compared the available distributed graph frameworks [16], [17], [18], [22], [44], there exists no *qualitative* comparison of the graph programming abstractions they offer. In this survey, we review prevalent high-level abstractions for distributed graph processing, in terms of semantics, expressiveness, and applicability. We identify the classes of graph algorithms that can be naturally expressed by each abstraction and we give examples of applications for which a model may appear to be non-intuitive. We further analyze popular distributed graph processing systems, with regards to their implementations of programming models and semantic restrictions. Our analysis can help graph application developers choose the appropriate model and system for their use-cases, and provides distributed systems researchers with open questions and challenges in the graph processing area.

This survey makes the following contributions.

- We present an overview of high-level programming abstractions for distributed graph processing and analyze their execution semantics and user-facing interfaces. We further consider performance limitations of some graph programming models and we summarize proposed extensions and optimizations.
- For each model, we identify classes of graph algorithms that can be intuitively and easily expressed. Additionally, we look into examples of algorithms that are hard or problematic to express with some graph processing abstractions.
- We categorize 34 distributed graph processing systems, with reference to the graph programming models they expose. We discuss execution models and communication mechanisms used.

The rest of this survey is organized as follows. In Section 2, we describe several high-level programming abstractions for distributed graph processing. For each abstraction, we discuss semantics, applications, and existing performance optimizations. Section 3 provides an overview of a number of general-purpose distributed programming models that also can be and have been used for graph processing. In Section 4 we categorize distributed graph processing systems implementations, with regard to the programming model they expose. We discuss our findings, challenges and open questions in the area of distributed graph processing in Section 5. We review related work in Section 6 and conclude this survey in Section 7.

1.1 Notation

Table 1 summarises the notation used for the execution semantics pseudo-code and interfaces in the rest of this survey. For the user-facing APIs, we use Java-like notation.

2 Programming Abstractions for Distributed Graph Processing

In this section, we review high-level programming models for distributed graph processing. A distributed graph programming model is an abstraction of the underlying computing infrastructure that allows for the definition of graph data structures and the expression of graph algorithms. We consider a distributed programming model to be *high-level* if it hides data partitioning and communication mechanisms from the end user. Thus, programmers can concentrate on the logic of their algorithms and do not have to care about data representation, communication patterns, and underlying system architecture. High-level programming models are inevitably less flexible than low-level models, and limit the degree of customization they allow. On the other hand, they offer simplicity and facilitate the development of automatic optimization.

A programming abstraction for distributed graph processing provides *a partitioned view* of the input graph and *a set of methods* that operate on each partition and allow programmers to read and modify its data and communicate with other partitions. A programming abstraction is well-designed when it offers an easy and intuitive way to express a set of algorithms using the provided views and methods. If the expression of an algorithm with a certain programming model requires extending the provided view or using a method in an unintended way, we then consider this abstraction to be *unintuitive* for this algorithm. In the following, we consider an algorithm difficult to express with a given abstraction if it requires bypassing the functionality provided by the abstraction's views and methods.

We use the PageRank algorithm as the running example throughout this paper to provide a common ground for comparison and demonstrate how different models can express a fairly simple but very common graph computation. While it is true that most of the surveyed abstractions can easily express this computation, using this example also reveals some subtle but important differences among the models. We highlight these differences in the respective subsections. We give examples of representative applications and examples of algorithms that are difficult to express with certain abstractions. For the most popular models, we also review implementation variants, known performance limitations, and proposed optimizations. Here, we describe six models that were developed *specifically* for distributed graph processing, namely vertex-centric, scatter-gather, gather-sum-apply-scatter, subgraph-centric, filter-process, and graph traversals.

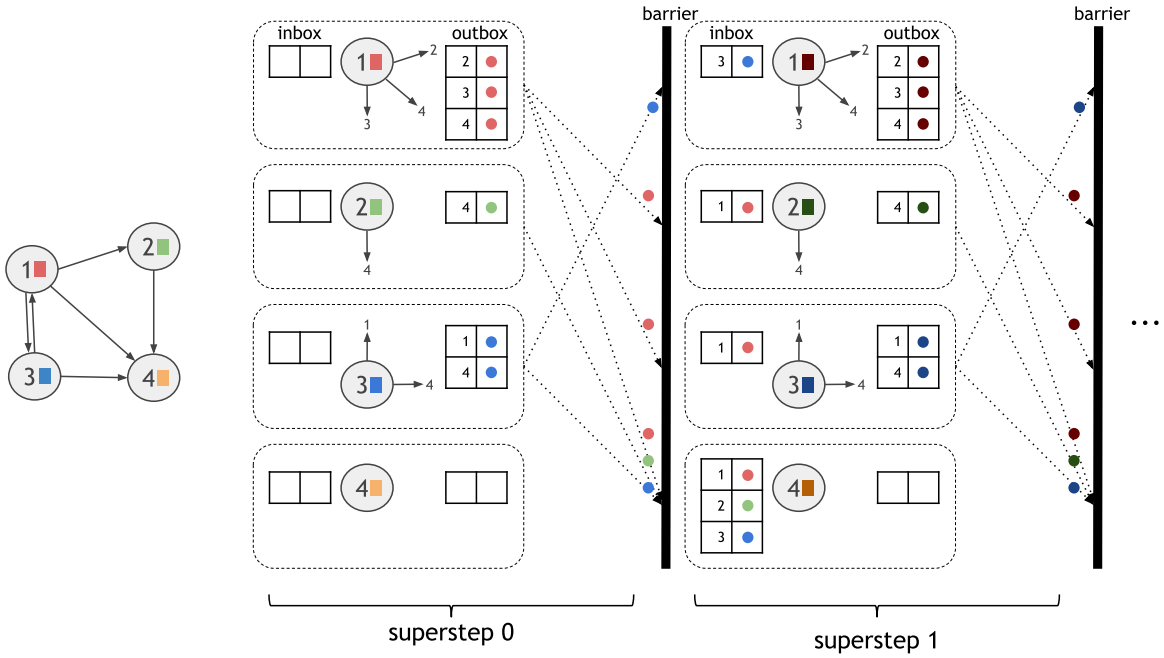

Fig. 1. Superstep execution and message-passing in the vertex-centric model. Graph edges can also have associated values, but we omit them here for simplicity. Each dotted box represents the computation scope of a vertex. Vertices in different scopes might reside on the same or different physical partitions. Arrows denote communication actions.

2.1 Vertex-Centric

The vertex-centric model is one of the most popular abstractions for large-scale distributed graph processing. Also known as the *think like a vertex* model, it forces the user to express the computation from the point of view of a single vertex. The input of a vertex-centric program is a directed graph and a vertex function. Each vertex has local state that consists of a unique ID, an optional vertex value, and its out-going edges, with optional edge values. Vertices communicate with other vertices through messages. A vertex can send a message to any other vertex in the graph, provided that it knows the destination's unique ID.

The execution workflow and computation parallelization of the vertex-centric model is shown in Fig. 1. The dotted boxes correspond to parallelization units. Vertices can be in two states: *active* or *inactive*. Initially, all vertices are active. The computation proceeds in synchronized iterations, called *supersteps*. In each superstep, all active vertices execute the same *vertex-function* in parallel. Supersteps are executed synchronously, so that messages sent during one superstep are guaranteed to be delivered in the beginning of the next superstep. The output of a vertex-centric program is the set of vertex values at the end of the computation. If the graph is partitioned over several machines, a partition can contain several vertices and may have multiple worker-threads executing the vertex functions.

For simplicity of presentation, we define two auxiliary local data structures for each vertex. During superstep i, $inbox_v$ contains all the messages that were sent to vertex v during superstep $i-1$. $outbox_v$ stores all the messages that vertex v produces during superstep i. At the end of each superstep, the runtime handles message delivery, by removing the messages from the $outbox$ of each sender vertex and placing them to the $inbox$ of their destination vertex. Message-passing can be implemented in batch or pipelined fashion. In the first case, messages are buffered in the local outbox of each vertex and are delivered in batches at the end of each superstep. In the case of pipelining, the runtime delivers messages to destination vertices as soon as they are produced. Pipelined message-passing might improve performance and lower memory requirements, but it limits the ability of pre-aggregating the results at the outbox.

The execution semantics of the vertex-centric model are shown in Algorithm 1. Initially, all vertices are active. At the beginning of each superstep, the runtime delivers messages to vertices ($receiveMessages$ function). Then, the user-defined vertex-function $compute$ is invoked in parallel for each vertex. It receives a set of messages as input and produces one or more messages as output. At the end of a superstep, the runtime receives the messages from the $outbox$ of each vertex and computes the set of active vertices for the next superstep. The execution terminates when there are no active vertices or when a user-defined convergence condition is met.

Algorithm 1. Vertex-Centric Model Semantics

```
Input: directed graph G=(V, E)
activeVertices ← V
superstep ← 0
while activeVertices ≠ ∅ do
  for v ∈ activeVertices do
    inbox_v ← receiveMessages(v)
    outbox_v = compute(inbox_v)
  end for
  superstep ← superstep + 1
end while
```

The user-facing interface of a vertex-centric program is shown in Interface 1. The vertex function can read and update the vertex value and has access to all out-going edges. It can send a message to any vertex in the graph, addressing it by its unique ID. A vertex votes to become

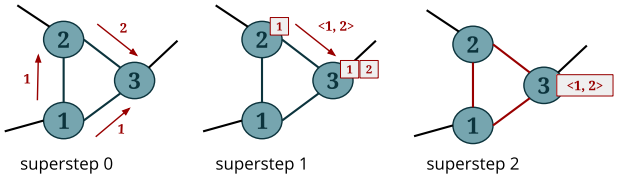

Fig. 2. A triangle counting algorithm in the vertex-centric model. Messages produced by a vertex during the current superstep are shown with red arrows. Messages received in the current superstep are shown in grey boxes.

inactive by *voting to halt*. If an inactive vertex receives a message, it becomes active again. In many implementations of the vertex-centric model, vertices can also add or remove a local edge or issue a mutation request for non-local edges or vertices.

Interface 1. Vertex-Centric Model

```
void abstract compute(Iterator[M] messages);
VV getValue();
void setValue(VV newValue);
void sendMessageTo(I target, M message);
Iterator getOutEdges();
int superstep();
void voteToHalt();
```

2.1.1 The GraphLab Variant

The vertex-centric model was introduced in Pregel [2], an implementation on top of the BSP [5] execution model. GraphLab [1] generalizes this model by introducing the notion of a vertex *scope*. The scope of a vertex contains the adjacent edges, as well as the values of adjacent vertices. The vertex function is applied over the current state of a vertex and its scope. It returns updated values for the scope (a vertex can mutate the state of its neighbors) and a set of vertices T, which will be scheduled for execution. GraphLab's programming model allows for *dynamic computation*, different consistency models and *asynchronous* execution. However, it also poses two limitations: (1) vertices can only communicate with their immediate neighbors, and (2) the graph structure has to be static, so that no mutations are allowed during execution. The shared-memory abstraction for communication is also adopted by Cyclops [65], even though its model is more restricted than the one provided by GraphLab.

2.1.2 Applicability and Expressiveness

The vertex-centric model is general enough to express a broad set of graph algorithms. The model is a good fit when the computation can be expressed as a local vertex function which only needs to access data on adjacent vertices and edges. Iterative value-propagation algorithms and fixed point methods map naturally to the vertex-centric abstraction.

A representative algorithm that can be easily expressed in the vertex-centric model is PageRank [56]. In this algorithm, each vertex iteratively updates its rank by applying a formula on the sum of the ranks of its neighbors. The pseudocode for the PageRank vertex function is shown in Algorithm 2. Initially, all ranks are set to $1/numVertices()$. In each superstep, vertices send their partial rank along their outgoing edges and use the received partial ranks from their neighbors to update their ranks, according to the PageRank formula. After a certain number of supersteps (30 in this example) all vertices vote to halt and the algorithm terminates. In this pseudocode, we see that the superstep number is used to differentiate the computation between the first and the rest of the iterations. This is a common pattern in vertex-centric applications, since during superstep 0, no messages have been received by any vertex yet.

Algorithm 2. PageRank Vertex Function

```
void compute(Iterator[double] messages):
  outE = getOutEdges().size()
  if superstep() > 0 then
    double sum = 0
    for m ∈ messages do
      sum ← sum + m
    end for
    setValue(0.15/numVertices() + 0.85 * sum)
  end if
  if superstep() < 30 then
    for e ∈ getOutEdges() do
      sendMessageTo(e.target(), getValue()/outE)
    end for
  else
    voteToHalt()
  end if
```

Non-iterative graph algorithms might be difficult to express in the vertex-centric model which heavily relies on the concept of supersteps. Furthermore, expressing a computation from the perspective of a vertex can often be challenging, as it also requires expressing all non-local state updates (further than one hop) as messages. Graph transformations and single-pass graph algorithms, like triangle counting, are not a good fit for the vertex-centric model.

Let us consider an implementation of a triangle counting algorithm in the vertex-centric model. A triangle consists of three vertices which all form edges between them. The vertex-centric abstraction provides a partitioned view that contains the value of a single vertex and its out-going edges and abstracts all communication in the form of messages to out-neighbors. However, the triangle counting algorithm requires extending the vertex-centric view to include information about the 2-hop neighborhood of a vertex and that can only be expressed with messages to out-neighbors. Such an algorithm needs to propagate a message along the edges of a triangle, so that when the message returns to the originator vertex, the triangle can be detected. In order not to count the same triangle multiple times, messages are only propagated from vertices with lower IDs to vertices with higher IDs. This algorithm is shown in Fig. 2 and proceeds in three supersteps. During the first superstep, each vertex sends its ID to all neighbors with higher ID than its own. During the second superstep, each vertex attaches its own ID to every received message and propagates the pair of IDs to neighbors with higher IDs. During the final superstep, each vertex checks the received pairs of IDs to detect whether a triangle exists.

Another non-intuitive computation pattern is sending messages to the in-neighbors of a vertex. Computing strongly connected components is an algorithm that

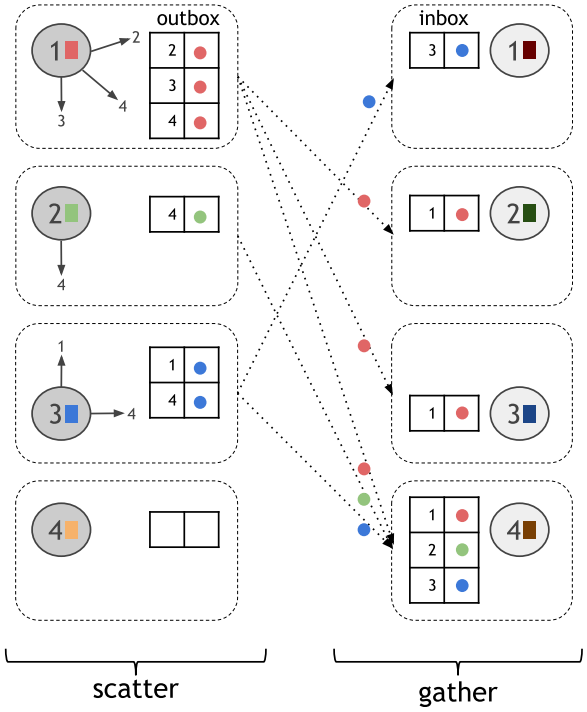

Fig. 3. One iteration in the Scatter-Gather model. In the Scatter phase, each vertex has read-access to its state, write-access to its outbox, and no access to its inbox. In the Gather phase, a vertex has read-access to its inbox, write-access to its state, but no access to its outbox.

contains this pattern [61]. Remember that each vertex only has access to its out-going edges and can only send messages to vertices with a known ID. Thus, if a vertex needs to communicate with its in-neighbors, it has to use a preprocessing superstep, during which, each vertex sends a message containing its own ID to all its out-neighbors. This way, all vertices will know the IDs of all their in-neighbors in the following superstep.

2.1.3 Performance Optimizations

The wide spread of the vertex-centric model has spawned a large body of recent research and engineering efforts to optimize its performance. In this section, we summarize some of the results of work on performance optimization for vertex-centric programs.

Communication can often become a bottleneck in the vertex-centric message-passing model. An overview of the model's limitations with regard to communication bottlenecks and worker load imbalance is presented in [47]. The authors show that high-degree vertices or custom algorithm logic can create communication and computation skew, so that a small number of vertices produce many more messages than the rest, thus also increasing the workload of the worker machines where they reside. They tackle these problems with two message reduction techniques. First, they use *mirroring*, a mechanism that creates copies of high-degree vertices on different machines, so that communication with neighbors can be local. A similar technique is also presented in [21]. Second, they introduce a *request-response* mechanism, that allows a vertex to request the value of any other vertex in the graph, even if they are not neighbors. For not neighboring vertices, such a process would require three supersteps in the vanilla vertex-centric model. High communication load can also be avoided by using sophisticated partitioning mechanisms [21], [71].

Using synchronization barriers in the vertex-centric model allows programmers to write deterministic programs and easily reason about and debug their code. Even though using barriers for synchronization facilitates parallel programming for many applications, global barriers limit concurrency and may cause unnecessary over-synchronization, and as a consequence, poor performance, especially for applications with irregular or dynamic parallelism. In fact, various graph algorithms can benefit from asynchronous [1], [38], [39] or hybrid [43], [57] execution models. In [37], the authors propose the *barrierless asynchronous parallel* BAP model, to reduce stale messages and the frequency of global synchronization. The model allows vertices to immediately access messages they have received and utilizes only barriers local to each worker, which do not require global coordination.

Several algorithm-specific optimization techniques for the vertex-centric model are proposed in [35], [61]. Specifically, the authors exploit the phenomenon of *asymmetric convergence* often encountered in graph algorithms. We say that an iterative fixpoint graph algorithm converges asymmetrically, if different parts of the graph converge at different speeds. As a result, the overall algorithm converges slowly because, during the final supersteps, only a small fraction of vertices are still active. The proposed solution is to monitor the active portion of the graph and, when it drops under some threshold, ship it to the master node, which executes the rest of the computation. Other optimizations include trading memory for communication and performing mutations (edge deletions) lazily.

In [46], the authors propose using special data structures, such as bit-vectors, to represent the neighborhood of each vertex. They also suggest compressing intermediate data that needs to be communicated over the network. They find that overlapping computation and communication, sophisticated partitioning, and different message-passing mechanisms have a significant impact on performance. Another issue they identify is that many algorithms have high memory requirements because of the outbox data structures of vertices growing too large. The authors propose to break each superstep into a number of smaller supersteps and processing only a subset of the vertices in each smaller superstep. Similar techniques are used in [36], [80].

2.2 Scatter-Gather

The *Scatter-Gather* abstraction, also known as *Signal-Collect* [10], is a vertex-parallel model, sharing the same *think like a vertex* philosophy as the vertex-centric model. Scatter-Gather also operates in synchronized iteration steps and uses a message-passing mechanism for communication between vertices. The main difference is that each iteration step contains two computation phases, *scatter* and *gather*. Thus, the user also has to provide two higher-order functions, one for each phase.

The model is graphically shown in Fig. 3. Scatter-Gather decouples the sending of messages from the collection of messages and state update. During the scatter phase, each vertex executes a user-defined function that sends messages or *signals* along out-going edges. During the gather phase, each vertex collects messages from neighbors and executes

a second user-defined function that uses the received messages to update the vertex state. It is important to note that, contrary to the vertex-centric model, in Scatter-Gather both message sending and receipt happen during the *same* iteration step. That is, during iteration i, the gather phase has access to the messages sent in the scatter phase of iteration i.

The execution semantics of the Scatter-Gather abstraction are shown in Algorithm 3. The input of a Scatter-Gather program is a directed graph and the output is the state of the vertices after a maximum number of iterations or some custom convergence condition has been met. Similarly to the vertex-centric model, vertices can be in an active or inactive state. Initially, all vertices are active. Vertices can either explicitly vote to halt, like in the vertex-centric model, or implicitly get deactivated, if their value does not change during an iteration step. If a vertex does not update its value during a gather phase, then it does not have to execute a scatter phase in the next iteration, because its neighbors have already received its latest information. Thus, it becomes inactive.

Algorithm 3. Scatter-Gather Model Semantics

Input: directed graph G=(V, E)
$activeVertices \leftarrow V$
$superstep \leftarrow 0$
while $activeVertices \neq \emptyset$ **do**
  $activeVertices' \leftarrow \emptyset$
  for $v \in activeVertices$ **do**
    $outbox_v \leftarrow scatter(v)$
    $S'_v \leftarrow gather(inbox_v, S_v)$
    if $S'_v \neq S_v$ **then**
      $S_v \leftarrow S'_v$
      $activeVertices' \leftarrow activeVertices' \cup v$
    end if
  end for
  $activeVertices \leftarrow activeVertices'$
  $superstep \leftarrow superstep + 1$
end while

The user-facing interfaces of Scatter and Gather are shown in Interfaces 2 and 3, respectively. Note that the scopes of the two phases are separate and each interface has different available methods. The scatter interface can retrieve the current vertex value, read the state of the neighboring edges, and send messages to neighboring vertices. The gather interface can access received messages, read and set the vertex value.

Interface 2. Scatter

void abstract **scatter**();
VV **getValue**();
void **sendMessageTo**(I target, M message);
Iterator **getOutEdges**();
int **superstep**();

Interface 3. Gather

void abstract **gather**(Iterator[M] messages);
void **setValue**(VV newValue);
VV **getValue**();
int **superstep**();

2.2.1 Applicability and Expressiveness

Scatter-Gather is a useful abstraction and can be used to express a variety of algorithms in a concise and elegant way. Similarly to the vertex-centric model, iterative, value-propagation algorithms like PageRank are a good fit for Scatter-Gather. Since the logic of producing messages is decoupled from the logic of updating vertex values based on the received messages, programs written using Scatter-Gather are sometimes easier to follow and maintain. The vertex-centric PageRank example that we saw in the previous section can be easily expressed in the Scatter-Gather model, by simply separating the sending of messages and the calculation of ranks in two phases. The pseudocode is shown in Algorithm 4. The scatter phase contains only the the rank propagation logic, while the gather phase contains the message processing and rank update logic.

Algorithm 4. PageRank Scatter and Gather Functions

void **scatter**():
  $outEdges = getOutEdges().size()$
  for $edge \in getOutEdges()$ **do**
    $sendMessageTo(edge.target(), getValue()/outEdges)$
  end for
void **gather**(Iterator[**double**] messages):
  if $superstep() < 30$ **then**
    double $sum = 0$
    for $m \in messages$ **do**
      $sum \leftarrow sum + m$
    end for
    $setValue(0.15/numVertices() + 0.85 * sum)$
  end if

Separating the messaging phase from the vertex value update logic not only makes some programs easier to follow but might also have a positive impact on performance. Scatter-Gather implementations typically have lower memory requirements, because concurrent access to the inbox (messages received) and outbox (messages to send) data structures is not required. However, this characteristic also limits expressiveness and makes some computation patterns non-intuitive. If an algorithm requires a vertex to concurrently access its inbox and outbox, then the expression of this algorithm in Scatter-Gather might be problematic. Strongly Connected Components and Approximate Maximum Weight Matching [61] are examples of such graph algorithms. A direct consequence of this restriction is that vertices cannot generate messages and update their states in the same phase. In the scatter phase, vertices have read-access to their state and adjacent edges, write-access to their outbox, and no access to their inbox. In the gather phase, vertices have read-access to their inbox, write-access to their state, but no access to their outbox or adjacent edges. Thus, deciding whether to propagate a message based on its content would require storing it in the vertex value, so that the gather phase has access to it, in the following iteration step. Similarly, if the vertex update logic includes computation over the values of the neighboring edges, these have to be included inside a special message passed from the scatter to the gather phase. Such workarounds often lead to higher

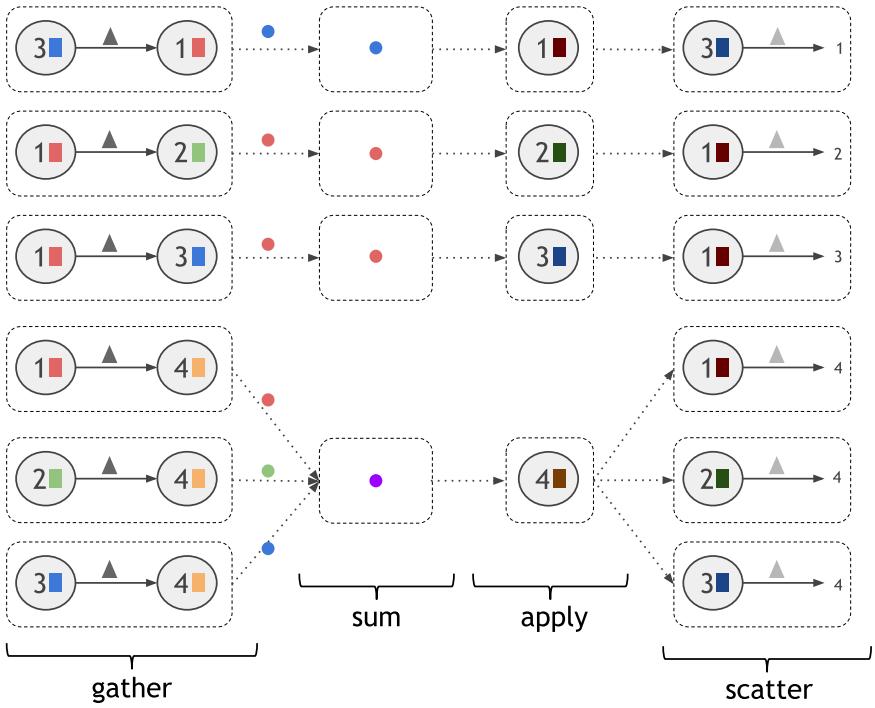

Fig. 4. The GAS computation phases. The gather phase parallelizes the computation over the edges of the input graph. The user-defined function has access to an edge, including its source and target vertex states. The sum phase combines partial values using a user-defined associative and commutative function. The vertex states are updated during the apply phase. The scatter phase is optional and can be used to update edge states.

memory requirements and non-elegant, hard to understand algorithm implementations.

For example, consider the problem of finding whether there exists a path from source vertex a to target vertex b, with total distance equal to a specified user value d. Let us assume that the input graph has edges with positive values corresponding to distances. We can solve this problem in a message-passing vertex-parallel way, by iteratively propagating messages through the graph and aggregating edge weights on the way. Messages originate from the source vertex and are routed towards the target vertex, one neighborhood hop per superstep. When vertex v receives a message, it decides whether to propagate the message to a neighbor u, based on the current distance that the message contains plus the distance of the edge that connects v with u. If the computed sum is less than or equal to d, v updates the message content and propagates it to u. If the sum exceeds the value of d, then v drops the message. In the vertex-centric model, this logic can be implemented inside the vertex compute function, since vertices receive and send messages during the same phase. However, in Gather-Scatter, vertices receive messages in the gather phase, but can only generate messages in the scatter phase. In order for a vertex to know whether to propagate a message based on its content, it needs a mechanism to allow the scatter phase access messages received in the previous superstep. One way that this can be achieved is by storing all received messages in the vertex value, so that the scatter interface can access them in the next superstep.

2.3 Gather-Sum-Apply-Scatter (GAS)

The Gather-Sum-Apply-Scatter programming abstraction (GAS), introduced by Powergraph [7], tries to address performance issues that arise when using the vertex-centric or scatter-gather model on power-law graphs. In such graphs, most vertices have relatively few neighbors, while few vertices have a very large number of neighbors. This degree *skew* causes computational imbalance in vertex-parallel programming models. The few high-degree vertices, having much more work to do during a superstep, act as stragglers, thus, slowing down the overall execution.

The GAS model addresses the bottlenecks caused by high-degree vertices by parallelizing the computation over the *edges* of the graph. The abstraction essentially decomposes a vertex-program in separate phases, which allow distributing the computation more effectively over a cluster. The computation proceeds in four phases, each executing a user-defined function. During the *gather* phase, a user-defined function is applied on each of the adjacent edges of each vertex in parallel, where an edge contains both the source vertex and the target vertex values. The transformed edges are passed to an *associative and commutative* user-defined function, which combines them to a single value during the *sum* phase. The gather and sum phases naturally correspond to a *map-reduce* step and are sometimes considered as a single phase [7]. The result of the sum phase and the current state of each vertex are passed to the *apply* user-defined function, which uses them to compute the new vertex state. During the final *scatter* phase, a user-defined function is invoked in parallel per edge, having to access the updated source and target vertex values. In some implementations of the model, the scatter phase is either optional or omitted. The four phases are graphically shown in Figure 4 and its execution semantics can be found in Algorithm 5.

The public interfaces of the GAS abstraction are shown in Interface 4. Note how these interfaces are simpler and more restrictive than the vertex-centric and gather-scatter interfaces. All four user-defined functions return a single value

TABLE 2
Comparison of the Vertex-Centric, Gather-Scatter, and GAS Programming Models

	update function properties	update function logic	communication scope	communication logic
Vertex-Centric	arbitrary	arbitrary	any vertex	arbitrary
Scatter-Gather	arbitrary	based on received messages	any vertex	based on vertex state
GAS	associative and commutative	based on neighbors' values	neighborhood	based on vertex state

and the scope of computation is restricted to local neighborhoods.

Algorithm 5. GAS Model Semantics

Input: directed graph G=(V, E)
$a_v \leftarrow$ **empty**
for $v \in V$ **do**
  for $n \in N_v^{in}$ **do**
    $a_v \leftarrow sum(a_v, gather(S_v, S_{(v,n)}, S_n))$
  end for
  $S_v \leftarrow apply(S_v, a_v)$
  $S_{(v,n)} \leftarrow scatter(S_v, S_{(v,n)}, S_n)$
end for

Interface 4. Gather-Sum-Apply-Scatter

T abstract **gather**(VV sourceV, EV edgeV, VV targetV);
T abstract **sum**(T left, T right);
VV abstract **apply**(VV value, T sum);
EV abstract **scatter**(VV newV, EV edgeV, VV oldV);

2.3.1 Applicability and Expressiveness

The GAS abstraction imposes the restriction of an associative and commutative sum function to produce edge-parallel programs that will not suffer from computational skew. Nevertheless, the model can be used to emulate vertex-centric programs, even if the update function is not associative and commutative. To express a vertex-centric computation, the gather and sum functions can be used to combine the inbound messages (stored as edge data) and concatenate the list of neighbors needed to compute the outbound messages. The vertex compute function is then executed inside the apply phase. The apply user-defined function generates the messages, which can then be passed as vertex data to the scatter phase. Similarly, to emulate a GraphLab vertex program, the gather and sum functions can be used to concatenate all the data on adjacent vertices and edges and then run the vertex function inside the apply phase.

Executing vertex-parallel programs inside the apply function results in complexity linear in the vertex degree, thus defeating the purpose of eliminating computational skew. Moreover, manually constructing the neighborhood in the sum phase and concatenating messages in order to access them in the apply phase, are both non-intuitive and computationally expensive. Fortunately, many graph algorithms can be decomposed into a gather transformation and an associative and commutative sum function. Algorithm 6 shows a PageRank implementation using the GAS interfaces. The gather phase computes a partial rank for each neighbor. The sum phase sums up all the partial ranks into a single value, and the apply phase computes the new PageRank and updates the vertex value. The scatter phase has been omitted, since edge values do not get updated in this algorithm. Alternatively, the scatter phase can be used to selectively activate vertices for the next iteration [7].

Algorithm 6. PageRank Gather, Sum, Apply

double gather(**double** src, **double** edge, **double** trg):
  return $trg.value/trg.outNeighbors$
double sum(**double** rank1, **double** rank2):
  return $rank1 + rank2$
double apply(**double** sum, **double** currentRank):
  return $0.15 + 0.85 * sum$

If the algorithm cannot be decomposed into a gather step and a commutative-associative sum, implementation in the GAS model might require manual emulation of the vertex-centric or scatter-gather models, as described previously. For example, in the Label Propagation algorithm [68], a vertex receives labels from its neighbors and chooses the most frequent label as its vertex value. Computing the most frequent item is not an associative-commutative function. In order to express this algorithm in GAS, the sum phase needs to construct a set of all the neighbor labels. The gather user-defined function returns a set containing a single label for each neighbor. The sum user-defined function receives a pair of sets, each containing a neighbor's label and returns the sets' union. Finally, each vertex chooses the most frequent label in the apply function, which has access to all labels.

Table 2 shows a comparison among the vertex-centric, gather-scatter, and GAS programming models, with regard to update functions and communication. We notice that the vertex-centric model is the most generic of the three, allowing for arbitrary vertex-update functions and communication logic, while GAS is the most restricted.

2.3.2 Performance Optimizations

In [9], the authors find that, even though the GAS model manages to overcome the load imbalance issues caused by high-degree vertices, at the same time it poses a high memory and communication overhead for the low-degree vertices of the input graph. They propose a *differentiated vertex computation* model, where the high-degree vertices are processed using the GAS model, while the low-degree vertices are processed using a GraphLab-like vertex-centric model.

2.4 Subgraph-Centric

All the models we have seen so far, vertex-centric, gather-scatter, and GAS, operate on the scope of a single vertex or edge in the graph. While such fine-grained abstractions for distributed programming are considerably easy to use by non-experts, they might cause high communication overhead when compared to coarse-grained abstractions. In this

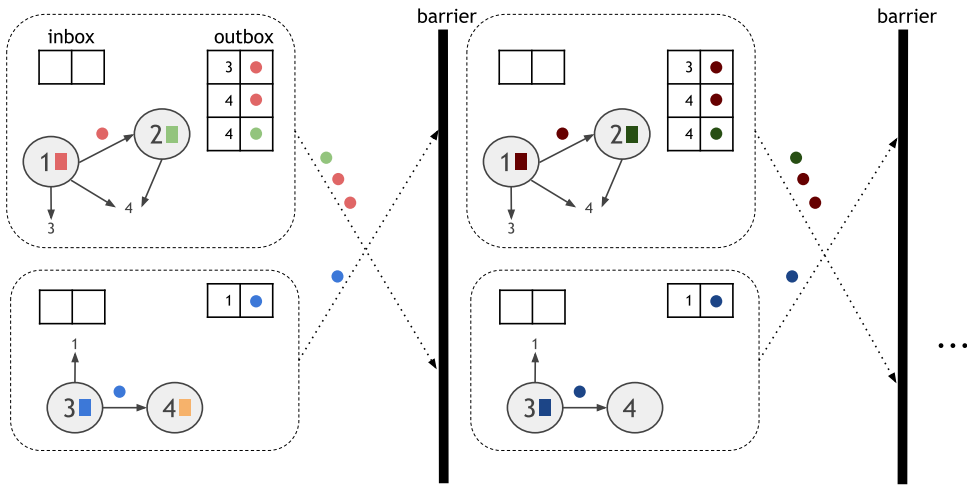

Fig. 5. Two iterations in the Partition-Centric model. Each dotted box represents a separate physical partition and dotted arrows represent communication.

section, we introduce the subgraph-centric model which extends the partitioned view into a *subgraph*. Thus, application logic and communication is scoped on subgraphs instead of single vertices or edges. In particular, we present two distributed graph processing models that operate on different types of subgraphs. The partition-centric model considers the subgraph view to be the contents of a physical partition and the neighborhood-centric model allows the programmer to define their own custom subgraphs.

2.4.1 Partition-Centric

The partition-centric model offers a lower-level abstraction than the vertex-centric, setting the whole partition as the unit of parallel computation. The model exposes the subgraph of each partition to the user-function, in order to avoid redundant communication and accelerate convergence of vertex-centric programs. The abstraction has been recently introduced by Giraph++ [11] and has been adopted and further optimized in several succeeding works [12], [13].

The main idea behind the partition-centric model relies on the perception of each partition as a proper subgraph of the input graph, instead of a collection of unassociated vertices. While in the vertex-centric model a vertex is restricted to accessing information from its immediate neighbors, in the partition-centric model information can be propagated freely inside all the vertices of the same partition. This simple property of the partition-centric model can lead to significant communication savings and faster convergence.

The partition-centric model is graphically shown in Fig. 5. Note that the whole partition becomes the parallelization unit on which the user-defined function is applied. As compared to the vertex-centric case, here, message exchange happens only between partitions, thus resulting in reduced communication costs. Inside a partition, vertices can be *internal* or *boundary*. Internal vertices are associated with their value, neighboring edges, and incoming messages. Boundary vertices only have a local *copy* of their associated value; the primary value resides in the partition where the vertex is internal. In Fig. 5, vertices 1 and 2 are internal in the upper partition, while vertices 3 and 4 are boundary. In the lower partition, vertices 3 and 4 are internal, while vertex 1 is boundary. Message exchange between internal vertices of the same partition is immediate, while messages to boundary vertices require network transfer.

The execution semantics of the partition-centric model are the same as in the vertex-centric model, with the only difference that the user-defined update function is invoked per partition and messages are distributed between partitions, not vertices. The user-facing interface of the model offers most of the methods of the vertex-centric interface, with a few additional ones, shown below. The interface needs to provide a mechanism to check whether a vertex belongs to a particular partition and to retrieve internal and boundary vertices inside a partition. Note that since the scope of the user-defined function is not a single vertex, the methods to retrieve and update vertex attributes are not part of the *compute()* interface.

Interface 5. Partition-Centric Model

```
void abstract compute();
void sendMessageTo(I target, M message);
int superstep();
void voteToHalt();
boolean containsVertex(I id);
boolean isInternalVertex(I id);
boolean isBoundaryVertex(I id);
Collection getInternalVertices();
Collection getBoundaryVertices();
Collection getAllVertices();
```

2.4.2 Neighborhood-Centric

The neighborhood-centric model [24] sets the scope of computation on *custom* subgraphs of the input graph. These subgraphs are explicitly built around vertices and their multi-hop neighborhoods, in order to facilitate the implementation of graph algorithms that operate on ego-networks; networks built around a central vertex of interest. The user specifies custom subgraphs and a program to be executed on those subgraphs. The user program might be iterative and is executed in parallel on each subgraph, following the Bulk Synchronous protocol (BSP) [5]. In contrast to the partition-centric model, in an implementation of the neighborhood-centric model, a physical partition can contain one or more

Fig. 6. A chain example graph.

custom subgraphs. Information exchange happens through shared state updates for subgraphs in the same partition and through replicas and messages for subgraphs belonging to different partitions.

2.4.3 Applicability and Expressiveness

The subgraph-centric model extends the view of the vertex-centric model to the specified subgraph. Thus, applications that require information about vertices other than the immediate out-neighbors of a vertex are typically easier to express with the model. However, it follows that the performance of this model is highly dependent on the quality of the subgraphs. In the partition-centric case, if the partitioning technique used creates well-connected subgraphs and minimizes edge cuts between partitions, it is highly probable that a partition-centric implementation will require less communication than a vertex-centric implementation and that a value-propagation algorithm will converge in fewer supersteps. As an example, consider the execution of the connected components algorithm on a chain graph, like the one shown in Figure 6. In a vertex-centric execution of the algorithm, shown in Fig. 7, the minimum value can only propagate one hop per superstep. Consequently, the algorithm requires as many supersteps as the maximum graph diameter plus one, in order to converge. In the partition-centric model instead, values can propagate asynchronously

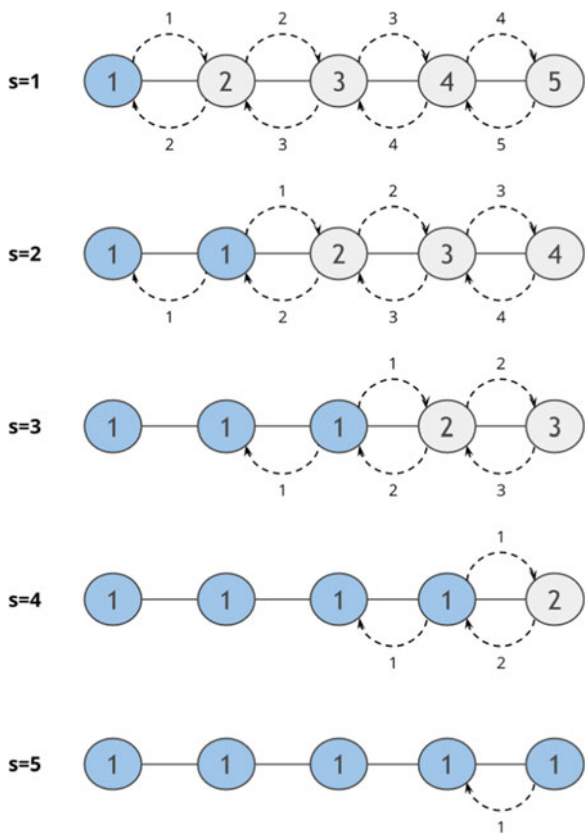

Fig. 7. Connected Components via label propagation in the vertex-centric model. The minimum value propagates to the end of the chain after 5 supersteps.

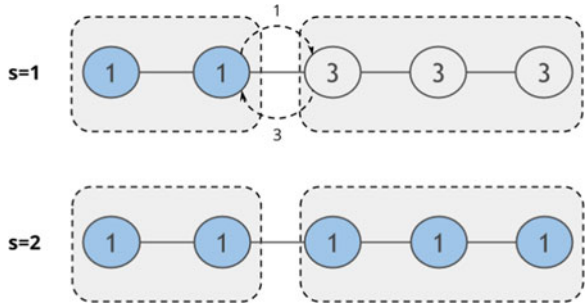

Fig. 8. Connected Components via label propagation in the partition-centric model. Vertices 1 and 2 belong to the first partition and vertices 3, 4, 5 belong to the second partition. Each partition converges asynchronously, before initiating communication with other partitions. The minimum value propagates to all vertices after 2 supersteps.

inside a partition. If the chain is partitioned in two connected subgraphs, like the ones of Fig. 8, the algorithm converges after just two supersteps. However, if the graph is partitioned poorly, there will be little to no benefit compared to the vertex-centric execution. For instance, if we partition the given chain into two partitions of odd and even vertices, the algorithm will need as many supersteps to converge as in the vertex-centric case. Sophisticated partitioning can be an expensive task and users must carefully consider the pre-processing cost that it might impose to the total job execution time.

A drawback of the partition-centric model is that users have to switch from "thinking like a vertex" to thinking in terms of partitions. In order to reason about their algorithm, users need to understand what a partition represents and how to differentiate the behavior of an internal vertex versus that of a boundary vertex. The model allows for more control on computation and communication, but at the same time exposes low-level characteristics to users. This loss of abstraction might lead to erroneous or hard-to-understand programs.

Algorithm 7 shows the pseudocode of a partition-centric PageRank implementation. It is immediately apparent that this implementation is quite longer and more complex than the ones we have seen so far. Part of the complexity is introduced from the fact that, inside each partition, PageRank computation is asynchronous. Each vertex has an additional attribute, *delta*, besides its PageRank score, where it stores intermediate updates from vertices inside the same partition. At the end of each superstep, boundary vertices with positive *delta* values produce messages to be delivered in other partitions.

The neighborhood-centric model is preferable when applications require accessing multi-hop neighborhoods or ego-centric networks of certain vertices. Such applications include personalized recommendations, social circle analysis, anomaly detection, and link prediction. While in the partition-centric model, the graph is partitioned into non-overlapping, application-independent subgraphs, in the neighborhood-centric model, subgraphs are extracted based on specific application criteria. This way, users can specify subgraphs to be k-hop neighborhoods around a set of query vertices, satisfying a particular predicate. The user program executed on these subgraphs can have arbitrary random access to the state of the whole subgraph. Note that creating these overlapping

neighborhoods requires an often expensive pre-processing step, in terms of both execution time and memory.

Algorithm 7. PageRank Partition-Centric Function

```
void compute():
  if superstep() == 0 then
    for v ∈ getAllVertices() do
      v.getValue().pr = 0
      v.getValue().delta = 0
    end for
  end if
  for iv ∈ internalVertices() do
    outEdges = iv.getNumOutEdges()
    if superstep() == 0 then
      iv.getValue().delta+ = 0.15
    end if
    iv.getValue().delta+ = iv.getMessages()
    if iv.getValue().delta > 0 then
      iv.getValue().pr+ = iv.getValue().delta
      u = 0.85 * iv.getValue().delta/outEdges
      while iv.iterator.hasNext() do
        neighbor = getVertex(iv.iterator().next())
        neighbor.getValue().delta+ = u
      end while
    end if
    iv.getValue().delta = 0
  end for
  for bv ∈ boundaryVertices() do
    bvID = bv.getVertexId()
    if bv.getValue().delta > 0 then
      sendMessageTo(bvID, bv.getValue().delta)
      bv.getValue().delta = 0
    end if
  end for
```

2.4.4 Performance Optimizations

GoFFish [12] and Blogel [13] are two subgraph-centric systems that require subgraphs to be *connected*. This restriction does not affect the expressiveness and applicability of the subgraph-centric model but enables parallelization and custom state management. For instance, a physical partition that contains multiple connected subgraphs can execute the computation on each of them in parallel. GoFFish also proposes a distributed persistent storage, optimized for subgraph access patterns. On the other hand, Blogel allows a subgraph to define and manage its own state, allowing for subgraph-level communication. Both GoFFish and Blogel are beneficial when the number of subgraphs is sufficiently larger than the number of workers, so that a balanced workload can be achieved.

2.5 Other Abstractions

While most existing programming abstractions for distributed graph processing adopt a vertex-centric or subgraph-centric view, alternative models have also been developed. Even though these other models have not yet received as much attention from the research community as the ones we have described so far, we find it important to briefly review them for completeness.

2.5.1 Filter-Process

The *filter-process* computational model, also known as or "think like an *embedding*", is proposed by Arabesque [55], a system for distributed graph mining. An embedding is a subgraph instance of the input graph that matches a user-specified pattern. The model facilitates the development of graph mining algorithms, which require subgraph enumeration and exploration. Such algorithms are challenging to express and efficiently support with a vertex-centric model, due to immense intermediate state and high computation requirements.

The programming model consists of two primary functions, *filter* and *process*. *Filter* examines whether a given embedding is eligible for processing and *process* executes some action on the embedding and may produce output. The model assumes immutable input graphs and connected graph patterns.

Computation proceeds in a sequence of *exploration steps*, using the BSP model. During each exploration step, the system explores and extends an input set of embeddings. First, a set of candidate embeddings is created, by extending the set of input embeddings. In the first exploration step the set of candidates contains all the edges or vertices of the input graph. Once the candidates have been produced, the filter function examines them and selects the ones that should be processed. The selected embeddings are then sent to the process function, which outputs user-defined values. Finally, the selected embeddings become the input set of embeddings for the next step. The computation terminates when there are no embeddings to be extended.

The filter-process model differs from the partition-centric and neighborhood-centric models, where partitions and subgraphs are generated *once*, at the beginning of the computation as a pre-processing step. In filter-process, embeddings are dynamically generated during the execution of exploration steps. The model is suitable for graph pattern mining problems, which require subgraph enumeration, such as network motif discovery, semantic data processing, and spam detection.

2.5.2 Graph Traversals

Distributed graph analysis through *graph traversals* is the programming model adopted by the Apache Tinkerpop project [51]. The system provides a graph traversal machine and language, called *Gremlin* [52], which supports distributed traversals via the Bulk Synchronous Parallel (BSP) computation model.

In the traversal model of graph databases, *traversers* walk through an input graph, following user-provided instructions. The Gremlin machine supports distributed graph traversals, by modeling traversers as messages. Vertices receive traversers, execute their traversal step, and, as a result, generate other traversers to be sent as messages to other vertices. Halted traversers are stored in a vertex attribute. The process terminates when no more traversers are being sent. The result of the computation is the aggregate of the locations of the halted traversers.

3 General-Purpose Programming Models used for Graph Processing

Except from the specialized programming models that we have reviewed so far, general-purpose distributed programming models have also been used for graph processing. We

present five such data processing abstractions, namely MapReduce, dataflow, linear algebra primitives, datalog, and shared partitioned tables. General-purpose models offer lower-level primitives, on top of which one can build domain-specific abstractions. As such, they offer wider applicability and are more expressive than the specialized models introduced in the previous section. For example, the vertex-centric model as introduced in the Pregel paper is implemented on top of MapReduce where the vertex function is executed inside the mapper. Similarly, several specialized models have been implemented on top of distributed dataflows as we discuss in Section 3.2.

3.1 MapReduce

MapReduce [4] is a distributed programming model for large-scale data processing on commodity clusters. Inspired by functional programming, it provides two operators, *map* and *reduce*, which encapsulate user-defined functions and form a static pipeline. A MapReduce application reads input in the form of *key-value* pairs from a distributed file system. Input pairs are processed by parallel map tasks, which apply the user-defined function, producing intermediate key-value pairs. The intermediate results are sorted and grouped by key, so that each group is then processed by parallel reduce tasks, applying the reduce user-defined function. Pairs sharing the same key are sent to the same reduce task. The output of each reduce task is written to the distributed file system, producing the job result.

MapReduce gained popularity because of its simplicity and scalability. However, its programming model is not suitable for graph applications, which are often iterative and require multi-step computations. Several extensions of the model have been proposed in order to support such algorithms [62], [69], [70]. The main idea of these extensions is adding a driver program that can coordinate iterations, containing one or more MapReduce jobs. The main task of the driver is to submit a new job per iteration and track convergence. Iterations are typically chained by using the output of one MapReduce-iteration as the input of the next.

Pegasus [62] implements a generalized iterative matrix-vector multiplication primitive, GIM-V, as a two-stage MapReduce algorithm. The graph is represented by two input files, corresponding to the vertices (vector) and edges (matrix). In the first stage, the map phase transforms the input edges to set the destination vertex as the key. The following reduce phase applies a user-defined *combine2* function on each group to produce partial values for each vertex. *combine2* corresponds to a multiplication of a matrix element with a vector element. In the second MapReduce stage, the mapper is an identity mapper and the reducer encapsulates two user-defined functions, *combineAll* and *assign*. *combineAll* corresponds to summing the partial multiplication results and *assign* writes the new result in the vector. GIM-V can be used to express many iterative graph algorithms, such as PageRank, diameter estimation, and connected components.

HaLoop [69] supports iterative computations on top of Hadoop [48], which it extends with a caching and indexing mechanism, to avoid reloading iteration-invariant data and reduce communication costs. It also adds methods to define loops and termination conditions to the Hadoop API.

Twister [70] extends the MapReduce API to support the development of iterative computations, including graph algorithms. It offers primitives for broadcast and scatter data transfers and implements a publish-subscribe protocol for message passing.

Scalable big graph processing in MapReduce is also discussed in [45]. The paper defines two graph join operators that can be implemented as map-reduce jobs. The first join operator propagates information from nodes to their adjacent edges and the second join operator aggregates data from adjacent edges to nodes. The input and output of the algorithms is restricted to a node table or an edge table, where each node has a unique id and an optional associated value. The work describes how algorithms like PageRank, Connected Components, and Minimum Spanning Tree can be implemented on top of these join operators.

3.2 Dataflow

Dataflow is a generalization of the MapReduce programming model, where a distributed application is represented by a Distributed Acyclic Graph (DAG) of operations. In the DAG, vertices correspond to data-parallel tasks and edges correspond to data *flowing* from one task to another. As opposed to MapReduce, dataflow execution plans are more flexible and operators can support more than one input and output. In the DAG model, iterations can be supported by loop unrolling [74], or by introducing complex iterate operators, as part of the execution DAG [57].

Spark [74], Stratosphere [76], Apache Flink [49], Hyracks [77], Asterix [78], and Dryad [79] are some of the general-purpose distributed execution engines implementing the DAG model for data-parallel analysis. Naiad [60] enriches the dataflow model with timestamps, representing logical points in computation. Using this timestamps, they build the *timely dataflow* computational model, which supports efficient incremental computation and nested loops.

Dataflow systems offer different levels of abstraction for writing distributed applications. In Dryad and early versions of Stratosphere, the user describes the DAG by explicitly creating task vertices and communication edges. User-defined functions are encapsulated in the vertices of the graph. Modern dataflow systems, like Apache Spark and Apache Flink, offer declarative APIs for expressing distributed data analysis applications. Data sets are represented by an abstraction that handles partitioning across machines, like RDDs [75], and operators define data transformations, like map, group, join, sort.

It has been shown that the vertex-centric, scatter-gather, and other iterative models can be mapped to relational operations [14], [25], [57], [58]. For example, by representing the graph as two data sets corresponding to the vertices and edges, the vertex-centric model can be emulated by a join followed by a group-by operation. Algorithm 8 shows an implementation of the PageRank algorithm in the Spark Scala API. The input edges ($links$) are grouped by the source ID to create an adjacency list per vertex and $ranks$ are initialized to 1.0. Then, the $links$ are iteratively joined with the current $ranks$ to retrieve each node's neighbors' rank values. A reduce operation is applied on the neighbor ranks to compute the updated PageRank for each node. Spark, Flink, and AsterixDB, all currently offer high-level APIs and

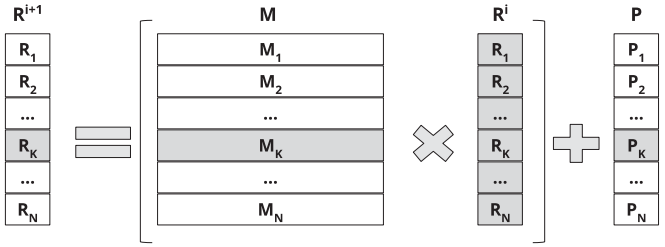

Fig. 9. PageRank computation as a vector-matrix multiplication. Vector R_i represents the computed ranks at iteration i, M is the input graph's adjacency matrix, with values scaled to account for the damping factor, and P contains the transition probabilities.

libraries for graph processing on their dataflow engines. Differential dataflow [59] also exposes a set of programming primitives, on top of which, higher-level programming models, such as vertex-centric, can be implemented.

Interestingly, Apache Flink's graph processing API, Gelly, demonstrates how the dataflow model can be used to implement several programming abstractions for distributed graph processing in the same system. Apache Flink has native support for iterative computations and contains an operator for performing delta iterations [57]. This operator receives two input data sets and maintains state in the form of a distributed hash table. The first input, the *solution set*, serves as the initial value of the state and feeds the step function together with the second input, the *workset*. During an iteration, the step function is evaluated and produces two outputs. The first output corresponds to updates that are applied on the solution set and the second output becomes the next workset. The iterative computation finishes when the workset is empty or when a custom convergence criterion is met. Then, the solution set becomes the output of the delta iterate operator. Gelly maps the vertex-centric, scatter-gather, and gather-sum-apply models to iterative dataflows built using the delta iteration operator.

Algorithm 8. PageRank in Apache Spark Scala API

```
Input lines: a list of space-separated ID node pairs
  val links = lines.map{ s =>
    val parts = s.split(" ")(parts(0), parts(1))
  }.distinct().groupByKey().cache()
  var ranks = links.mapValues(v => 1.0)
  for (i < - 1 to iters) {
    val contribs = links.join(ranks).values.flatMap{ case
      (urls, rank) =>
        val size = urls.size
        urls.map(url => (url, rank / size))
    }
    ranks = contribs.reduceByKey(_ + _)
      .mapValues(0.15 + 0.85 * _)
  }
  val output = ranks.collect()
```

3.3 Linear Algebra Primitives

Linear algebra primitives and operations have been long used to express graph algorithms. This model leverages the duality between a graph and its adjacency matrix representation [40]. A graph $G = (V, E)$ with N vertices can be represented by a NxN matrix M, where $M_{ij} = 1$ if there is an edge e_{ij} from node i to node j and 0 otherwise. Using this representation, many graph analysis algorithms can be expressed as a series of linear algebra operations. For example, a breadth-first search (BFS) starting from node i can be easily expressed as matrix multiplication. Starting from an initial $1xN$ vector y_0, where only the ith element is non-zero, the multiplication $y_1 = y_0 * M$ will give the immediate neighbors of node i, $y_2 = y_1 * M$ will give the 2-hop neighbors of i, and so on. Figure 9 shows the computation of the PageRank algorithm in this model. Essentially, PageRank corresponds to the dominant eigenvector of the input graph's adjacency matrix. Starting from an initial vector of ranks R_0, PageRank can be computed by iteratively multiplying R_i with the adjacency matrix and adding the transition probabilities to get R_{i+1}, until convergence.

Combinatorial BLAS [31] and Presto [73] are two seminal works in porting the linear algebra model for graph processing to a distributed setting. In Combinatorial BLAS, the graph adjacency matrix is represented as a distributed sparse matrix and graph operations are mapped to linear algebra operations between sparse matrices and vectors. The main operations perform matrix-matrix multiplication and matrix-vector multiplication and require user-defined functions for addition and multiplication. Presto extends the R language to support a distributed array abstraction, which can represent both dense and sparse matrices. Computation is distributed automatically based on the data partitioning. For instance, in Fig. 9, the shaded areas correspond to the data that is required for the computation of R_K. We notice that only a single row of the adjacency matrix is accessed and only one element of the transition probability vector. Thus, the computation task of R_K should be sent to the partition that contains M_K and P_K. Extracting such access patterns is essential for partitioning data and computation in the linear algebra model.

3.4 Datalog Extensions

Datalog is a declarative logic programming language used as query language for deductive databases. Datalog programs consist of a set of *rules*, which can be recursive. Datalog's support for recursions makes it suitable for expressing iterative graph algorithms. In [29], Datalog is enhanced with a set of extensions that allow users to define data distribution, efficient data structures for representing adjacency lists, and recursive aggregate functions, which can be efficiently evaluated using semi-naive evaluation [41]. Algorithm 9 shows the set of rules to compute an iteration of PageRank, using these extensions.

Algorithm 9. A PageRank iteration in SociaLite [29]

PageRank (iteration i+1) Input N: number of vertices
EDGE (int src: 0..N, (int $sink$)).
EDGECOUNT (int src: 0..N, int cnt).
NODES (int n: 0..N).
RANK (int $iter$: 0..10, (int $node$: 0..N, int $rank$)).
RANK(i+1, n, \$SUM(r)) :- NODES(n), $r = 0.15/N$;
:- RANK(i, p, r_1), EDGE(p, n),
EDGECOUNT(p, cnt),
$cnt > 0$, $r = 0.85 * r_1/cnt$.

A Datalog program is called *stratifiable* if it contains no negation operations within a recursive cycle. For such a

TABLE 3
Distributed Graph Processing Systems Comparison in Terms of Supported Programming Models, Execution Model, and Communication Mechanisms

System	Year	Programming Model	Execution	Communication
Pegasus [62]	2009	MapReduce	S	Dataflow
Pregel [2]	2010	Vertex-Centric	S	Message-Passing
Signal/Collect [10]	2010	Scatter-Gather	A, S	Message-Passing
HaLoop [69]	2010	MapReduce	S	Dataflow
Twister [70]	2010	MapReduce	S	Dataflow
Piccolo [72]	2010	Partitioned Tables	S	Shared global state
Apache Giraph [36]	2011	Vertex-Centric	S	Message-Passing
Comb. BLAS [31]	2011	Linear Algebra	S	Message-Passing
Apache Hama [53]	2012	Vertex-Centric	S	Message-Passing
GraphLab [1]	2012	Vertex-Centric	A, S	Shared Memory
PowerGraph [7]	2012	GAS	A, S	Shared Memory
Giraph++ [11]	2013	Subgraph-Centric	H	Message-Passing
Naiad [60]	2013	Dataflow	A, S, I	Dataflow
GPS [21]	2013	Vertex-Centric	S	Message-Passing
Mizan [6]	2013	Vertex-Centric	S	Message-Passing
Presto [73]	2013	Linear Algebra	S	Dataflow
Giraphx [64]	2013	Vertex-Centric	A	Shared Memory
X-Pregel [66]	2013	Vertex-Centric	S	Message-Passing
LFGraph [32]	2013	Vertex-Centric	S	Shared Memory
SociaLite [29]	2013	Datalog Extensions	S	Message-Passing
Trinity [28]	2013	Vertex-Centric	A, S	Message-Passing, Shared Memory
Graphx [25]	2014	GAS	S	Dataflow
GoFFish [12]	2014	Subgraph-Centric	H	Message-Passing
Blogel [13]	2014	Vertex-Centric, Subgraph-Centric	H	Message-Passing
Seraph [67]	2014	Vertex-Centric	S	Message-Passing
Cyclops [65]	2014	Vertex-Centric	S	Message-Passing
GiraphUC [37]	2015	Vertex-Centric	A, H	Message-Passing
Gelly [50]	2015	Vertex-Centric, Scatter-Gather, GAS	S	Dataflow
Pregelix [14]	2015	Vertex-Centric	S	Dataflow
Apache Tinkerpop [51]	2015	Graph Traversals	S	Message-Passing
PowerLyra [9]	2015	GAS	S	Shared Memory
NScale [24]	2015	Neighborhood-Centric	S	Dataflow
Arabesque [55]	2015	Filter-Process	S	Message-Passing
Pregel+ [47]	2015	Vertex-Centric	S	Message-Passing

S stands for synchronous*, A for* asynchronous*, H for* hybrid*, and I for* incremental.

program, there exists a unique greatest fixed point for its rules. In order to parallelize Datalog programs, it is assumed that all input programs are stratifiable. To parallelize the recursive aggregate functions, these have to be *monotone*, i.e., idempotent, commutative, and associative. Under these circumstances, it is shown in [29] that delta stepping [42] can be used to parallelize monotone recursive aggregate functions.

3.5 Shared Partitioned Tables

Distributed graph processing through shared partitioned tables is an idea implemented by Piccolo [72]. The computation is expressed as a series of user-provided *kernel* functions, which are executed in parallel and *control* functions, which are executed on a single machine. Kernel instances communicate through shared distributed, mutable state. This state is represented as in-memory tables whose elements are stored in the memory of different compute nodes. Kernels can use a key-value table interface to read and write entries to these tables. The tables are partitioned across machines, according to a user-provided partitioning function. Users are responsible for handling synchronization and resolution functions for concurrent writes in the shared tables.

4 Categorization of Distributed Graph Processing Systems

In this section, we present a taxonomy of recent distributed graph processing systems. We use the following parameters to categorize the systems:

Programming Model. This parameter refers to the programming abstraction that the system offers and is the main focus of this survey. We have described the different programming models in Sections 2 and 3.

Execution Model. The execution model refers to a specific implementation of a programming model. We categorize execution models into synchronous, asynchronous, hybrid, and incremental, which we describe in Section 4.2.

Communication Mechanism. The communication mechanism defines how the partitioned views of a given abstraction communicate with each other. We categorize the mechanisms into message-passing, shared-memory, and dataflow, which we explain in Section 4.3.

Applications. To further assess distributed graph programming model expressiveness and systems usability, we survey the applications which appear in recent distributed graph processing systems papers. We group graph algorithms used in the papers describing the systems of Table 3. We collect

TABLE 4
Applications Used in Distributed Graph Processing Systems Papers to Demonstrate Programming Models and Evaluate Performance

Application	#Appearances	Programming Models
PageRank (and variations)	31	Vertex-Centric, Scatter-Gather, GAS, Dataflow, Linear Algebra, Subgraph-Centric, Partitioned Tables, MapReduce, Datalog Extensions
Shortest Paths (and variations)	15	Vertex-Centric, Scatter-Gather, Dataflow, Linear Algebra, Subgraph-Centric, Datalog Extensions, GAS
Weakly Connected Components	12	Vertex-Centric, GAS, Dataflow, Subgraph-Centric, MapReduce
Graph Coloring	5	Vertex-Centric, Scatter-Gather, Linear Algebra, GAS
Alternate Least Squares (ALS)	4	Vertex-Centric, GAS, Linear Algebra
Triangle count	4	Vertex-Centric, Linear Algebra, Subgraph-Centric, Datalog Extensions
K-Means	4	Vertex-Centric, Linear Algebra, Subgraph-Centric, Partitioned Tables
Minimum Spanning Forest / Tree	4	Vertex-Centric
Belief Propagation (and variations)	2	Vertex-Centric
Strongly Connected Components	2	Vertex-Centric, Dataflow
Label Propagation	2	Vertex-Centric
Diameter Estimation	2	GAS, MapReduce
Clustering Coefficient	2	Subgraph-Centric, Datalog Extensions
Motif Counting	2	Subgraph-Centric, Filter-Process
BFS	2	Vertex-Centric, Subgraph-Centric
Centrality Measures	1	Linear Algebra
Markov Clustering	1	Linear Algebra
Find Mutual Neighbors	1	Datalog Extensions
SALSA	1	Dataflow
n-body	1	Partitioned Tables
Bipartite Matching	1	Vertex-Centric
Semi-Clustering	1	Vertex-Centric
Random Walk	1	Vertex-Centric
K-Core	1	Vertex-Centric
Approximate Max. Weight Matching	1	Vertex-Centric
Graph Coarsening	1	Subgraph-Centric
Identifying Weak Ties	1	Subgraph-Centric
Frequent Subgraph Mining	1	Filter-Process
Finding Cliques	1	Filter-Process

applications appearing as examples to demonstrate APIs and programming model interfaces, and applications used for performance evaluation. We aim to provide insights into systems' applicability and efficiency based on the assumption that systems authors choose *representative* algorithms to include in their papers.

Table 3 contains 34 such systems and compares them in terms of programming model, execution model, and communication mechanisms. We present the results regarding applications in Table 4 and discuss them in Section 4.4.

4.1 Programming Model

The vertex-centric model appears to be the most commonly implemented abstraction among the systems that we consider. Pregel, Apache Giraph, Apache Hama, GPS, Mizan, Giraphx, Seraph, GiraphUC, Pregel+, and Pregelix (Apache AsterixDB) implement the full semantics of the model. GraphLab, LFGraph, Cyclops, Gelly, and Trinity do not support graph mutations, while the former three do not support communication with vertices outside of the defined scope / neighborhood either. GraphX provides an operator called *pregel* for iterative graph processing, but, in fact, this operator implements the GAS paradigm. Apache Tinkerpop provides connectors to Giraph and Spark, allowing the execution of graph traversals on top of their computation engines.

4.2 Execution Model

We encounter four execution techniques in the graph systems considered in this survey: synchronous (S), asynchronous (A), hybrid (H), and incremental (I). Synchronous execution refers to implementations where a global barrier separates one iteration from the next. In such a model, during iteration i, vertices perform updates based on values computed in iteration $i-1$. On the other hand, in the asynchronous execution model, computation is performed on the most recent state of the graph. Synchronization can happen either through shared memory or through local barriers and distributed coordination. In a hybrid execution model, synchronous and asynchronous modes can coexist. For example, in Giraph++, computation and communication inside each partition happens asynchronously, while cross-partition computation requires global synchronization points. Incremental execution refers to the ability of a system to efficiently update the computation when its input changes, without halting and re-computing everything from scratch.

Synchronous execution is the most common design choice. It simplifies application development and facilitates debugging. Asynchronous execution is usually supported together with synchronous or hybrid. The incremental execution model is only supported in Naiad.

4.3 Communication Mechanism

We come across different communication mechanisms. In the Message-Passing model, the state is partitioned across worker tasks and updates to non-local state happen by sending and receiving messages. Worker tasks have read-write access to local state but they cannot directly access and modify state of a different worker. On the contrary, the shared memory mechanism allows tasks in different machines to communicate by mutating shared state. Systems that employ this mechanism need to account for race conditions and data consistency. In the dataflow model, operators are usually stateless and data flows from one stage of computation to the next. In order to efficiently support graph computations in this model, dataflow systems offer explicit or automatic caching mechanisms. For example, Spark has a *cache()* method, which can be used to cache the graph structure, which is static. In Stratosphere and Flink, the optimizer will detect loop-invariant data and cache them automatically.

4.4 Applications

Table 4 shows the most commonly encountered applications in recent distributed graph processing systems papers, sorted by appearance frequency. For each application, we also list the programming models in which it has been implemented. Before discussing the data of this table, it is important to clarify the goals of this categorization. Table 3 does not demonstrate the capabilities of each abstraction and the absence of an abstraction from a row does not necessarily reflect its inability to express that application. The table rather presents an aggregation of applications used in recent distributed graph processing systems per programming model. We hope that this data provides an insight into the intentions of the system designers regarding their offered abstractions and reveals open issues in the research area.

Unsurprisingly, we find the PageRank algorithm to be extremely popular. This algorithm appears in 31 out of the 34 examined systems and we encounter implementations in 9 out of the 11 programming abstractions that we present in Sections 2 and 3. Shortest paths calculation and weakly connected components appear in more than one third of the papers. Graph coloring, ALS, k-means clustering, and minimum spanning forest also appear to be quite commonly used. However, we notice that the majority of applications are only encountered once or twice. This is partially explained by the fact that some of them, like finding cliques, serve the purpose of introducing a specialized programming model, like filter-process. Nevertheless, it appears that the vertex-centric model has been used to implement most of the applications in the table.

Based on the findings of Table 4 and our analysis in the rest of this paper, we can categorize graph analysis applications and provide guidelines regarding suitable programming models as follows.

Value-Propagation. Value-propagation algorithms are fixpoint algorithms that iteratively refine the values of the vertices, until they all converge to their final values. Such applications include PageRank, Single-Source Shortest Paths, Weakly Connected Components, and Label Propagation. In these applications, vertex values are typically computed by applying a function on the values of their first-hop neighbors. Value-propagation algorithms can be easily expressed using the vertex-centric, gather-sum-apply, and scatter-gather models.

Traversals. Graph traversals are algorithms that visit the vertices of a graph with the objective to find a particular value or pattern. Breadth-first search and depth-first search are graph traversal algorithms. Even though the vertex-centric model has been used to implement graph traversals, its limited view of the first-hop neighborhood for each vertex might incur unecessary overheads for such computations. Subgraph-centric models are preferable in this case, as they allow for extended subgraph views.

Ego-Network Analysis. Ego-network applications include algorithms that compute personalized metrics using neighborhood information for each vertex. Among the models that we consider in this chapter, the neighborhood-centric model is the most suitable model for such applications, as it allows creating neighborhood-based views of the graph.

Pattern Matching. Graph pattern matching applications include graph mining algorithms, like identifying cliques and frequent subgraphs. Among the models that we consider in this chapter, the filter-process and the subgraph-centric models appears to be best suitable for such applications.

Machine Learning. Several machine learning tasks can be expressed using graph models. Table 4 includes machine learning algorithms, such as k-means clustering, Alternate Least Squares, and Markov Clustering. Such applications are most commonly implemented using linear algebra models.

5 Discussion and Open Issues

Similarly to how the introduction of the map-reduce programming model simplified large-scale data analysis to a large extend, the invention of the vertex-centric model revolutionized the area of distributed graph processing. *Thinking like a vertex* has proved to be a valuable abstraction that allows writing comprehensive programs for a variety of graph problems, as suggested by the data in Tables 3 and 4.

Table 3 also suggests that the majority of the existing distributed graph processing systems offer a vertex-centric or subgraph-centric view. However, as our analysis in Section 2 reveals, there exist important differences among the models that make some algorithmic patterns hard to express with certain abstractions. According to Table 2, among the vertex-oriented abstractions, the vertex-centric model is the most flexible one, allowing for arbitrary update functions and communication logic, while GAS is the most restricted, requiring associative and commutative updates. On the other hand, subgraph-centric models extend the partition view to multiple hops around a vertex, allowing for easier expression of algorithms that do not only depend on the state of immediate neighbors. Our study suggests that extending the view from a vertex to a subgraph comes at the cost of having to handle internal and external vertices separately and might result into verbose programs.

Table 4 and the analysis of Section 4.4 reveal that no model is suitable for all kinds of applications. It is an open challenge for researchers and systems designers to either invent a more expressive and flexible programming model or explore the possibility of supporting multiple programming abstractions on top of the same platform. To that end,

general-purpose dataflow systems look promising. It has already been shown how the vertex-centric, scatter-gather, and GAS abstractions can be mapped to relational execution plans, and how distributed dataflow frameworks can efficiently support them [14], [25], [57]. Nevertheless, current systems still rely on the user for efficient implementations. The research community could investigate improving existing relational optimizers for handling graph tasks or building graph processing optimizers that could choose the most suitable model based on the application characteristics and input graph properties.

Even though graph algorithms exhibit a variety of data access and communication patterns, we find that the applications used for evaluating usability and performance of graph programming models and systems do not reflect this diversity. Indeed, only very small set of graph algorithms (PageRank, Connected Components, Shortest Paths) is being repeatedly used and promoted as a representative benchmark, when, in essence, these algorithms are quite similar; single-stage, value-propagation iterative algorithms. We believe that the research community would benefit immensely from exploring more complex, multi-stage algorithms to further challenge the expressiveness of existing abstractions and the performance of current systems implementations.

Another open question in the area of distributed graph processing is clarifying the conditions that make distribution necessary. It has been recently shown that single-threaded implementations can outperform some distributed graph processing systems [33]. At the same time, significant progress has been made in building efficient single-machine graph processors that can handle billion-edge graphs [26], [27], [34]. Thus, it is critical to quantify parallelization and communication overheads in distributed graph processing systems and clarify the circumstances under which a distributed implementation would be preferable over a centralized solution.

Finally, a largely unexplored direction in the area of distributed graph processing is support for dynamic graphs and temporal analytics. Most graphs are highly dynamic in nature or are generated in real-time from streaming sources. However, most of the existing graph processors assume a static immutable graph structure. Systems and models need to evolve in order to facilitate real-time graph analytics and incremental processing of changing graph inputs. Kineograph [3] and Chronos [63] are two pioneering works in this direction. Kineograph supports incremental graph processing through consistent periodical snapshots, while Chronos proposes a novel in-memory layout of temporal graphs that allows for efficient computation across a series of snapshots.

6 Related Work

The work that is closest to ours is a recent survey of vertex-centric frameworks for graph processing [20]. It presents a extensive study of frameworks that implement the vertex-centric programming model and compares them in terms of system design characteristics, such as scheduling, partitioning, fault-tolerance, and scalability. Moreover, it briefly introduces subgraph-centric frameworks, as an optimization to vertex-centric implementations. While there is some overlap between this work and ours, the objective of our study is to present the first comprehensive comparison of distributed graph processing abstractions, regardless of the specifics of their implementations. While in [20] the discussion revolves around certain frameworks, in our work, we first consider the programming models decoupled from the systems, then build a taxonomy of systems based on their implemented model.

A notable study of parallel graph processing systems is presented in [23]. The work surveys over 80 systems, spanning single-machine, shared-memory, and distributed architectures. The scope of their study is much wider than ours, yet their results are driven by system implementations and platform design choices. With regard to programming models, they consider general-purpose processing systems, vertex-centric, and subgraph-centric, but do not further expand on execution semantics, user-facing interfaces or limitations.

While the focus of our study is to describe and compare available programming abstractions for distributed graph processing, several recent studies focus on the performance of modern distributed graph processing platforms. In [22], the authors compare the performance of six systems, including general-purpose data processing frameworks, specialized graph processors and a non-distributed graph database. The study evaluates these systems across a number of performance metrics, such as execution time, CPU and memory utilization, and scalability, using real-world datasets and a diverse set of graph algorithms. Similarly, [44] presents an experimental evaluation of distributed graph processing platforms, but covering only specialized graph processors. The also study the effectiveness of implemented optimization techniques and compare performance with a single machine system as baseline. A similar but more extensive study can be found in [18]. Both these studies conclude that no single system performs best in all cases and also highlight the need for a standard benchmark solution, that could simplify the performance comparison of graph processing platforms. Even though a standard benchmarking solution for graph processing systems does not exist yet, [16], [17] are two significant steps towards that direction. They propose a clear vision for a benchmark, listing challenges that need to be addressed and share early results in designing a graph processing benchmark, called Graphalytics. Graphalytics supports graph generation with custom degree distributions and structural characteristics and already supports several systems and algorithms.

7 Conclusion

Graphs are practical data structures that can elegantly capture relationships between data items. Efficiently analyzing large-scale graphs is gradually becoming an essential requirement for business intelligence, social networks, web applications, and modern science. In order to cope with the rapid increase of graph data sizes, efficient distributed graph processing solutions are essential. To facilitate the development of graph algorithms in a distributed environment, several high-level abstractions for graph processing have been proposed in recent literature.

In this survey, we present a comprehensive review of the most prevalent high-level abstractions for distributed graph

processing. We analyze and compare their semantics and user-facing interfaces. We comment on their usability and expressiveness, identifying representative applications and computation patterns that are hard to express. We further survey proposed performance optimizations and model variations. We then look at recent distributed graph processing systems and categorize them in terms of programming model, execution model, and communication mechanisms. We also survey graph analysis applications that are used in recent papers to demonstrate systems usability and performance. Finally, we discuss open challenges in the area of distributed graph processing and identify future research directions. We believe that the community would benefit from experimenting with more complex applications and exploring the limits of current abstractions. We see a promising direction in supporting multiple graph programming models on top of general-purpose dataflow systems and we anticipate a great research potential in the area of real-time and temporal graph analytics.

Acknowledgments

This work was done while Ms. Vasiliki Kalavri was at KTH Royal Institute of Technology.

References

[1] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein, "Distributed GraphLab: A framework for machine learning and data mining in the cloud," *Proc. VLDB Endow.*, vol. 5, no. 8, pp. 716–727, April 2012. doi: http://dx.doi.org/10.14778/2212351.2212354

[2] G. Malewicz, et al., "Pregel: A system for large-scale graph processing," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2010, pp. 135–146. doi: http://dx.doi.org/10.1145/1807167.1807184

[3] R. Cheng, et al., "Kineograph: Taking the pulse of a fast-changing and connected world," in *Proc. 7th ACM Eur. Conf. Comput. Syst.*, 2012, pp. 85–98.

[4] J. Dean, and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[5] L. G. Valiant, "A bridging model for parallel computation," *Commun. ACM*, vol. 33, no. 8, pp. 103–111, 1990.

[6] Z. Khayyat, K. Awara, A. Alonazi, H. Jamjoom, D. Williams, and P. Kalnis, "Mizan: A system for dynamic load balancing in large-scale graph processing," in *Proc. 8th ACM Eur. Conf. Comput. Syst.*, 2013, pp. 169–182.

[7] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "PowerGraph: Distributed graph-parallel computation on natural graphs," in *Proc. 10th USENIX Conf. Operating Syst. Des. Implementation*, 2012, pp. 17–30.

[8] D. Gregor and A. Lumsdaine, "The Parallel BGL: A generic library for distributed graph computations," *Parallel Object-Oriented Scientific Comput.*, vol. 2, pp. 1–18, 2005.

[9] R. Chen, J. Shi, Y. Chen, and H. Chen, "Powerlyra: Differentiated graph computation and partitioning on skewed graphs," in *Proc. 10th Eur. Conf. Comput. Syst.*, 2015, Art. no. 1.

[10] P. Stutz, A. Bernstein, and W. Cohen, "Signal/collect: Graph algorithms for the (Semantic) Web," in *Proc. 9th Int. Semantic Web Conf.*2010, pp. 764–780, doi: http://dx.doi.org/10.1007/978–3-642-17746-0_48

[11] Y. Tian, A. Balmin, S. A. Corsten, S. Tatikonda, and J. McPherson, ,"From think like a vertex to think like a graph," *Proc. VLDB Endowment*, vol. 7, no. 3, pp. 193–204, 2013.

[12] Y. Simmhan, et al., "Goffish: A sub-graph centric framework for large-scale graph analytics," in *Proc. Euro-Par Parallel Processing*, 2014, pp. 451–462.

[13] D. Yan, J. Cheng, Y. Lu, and W. Ng, "Blogel: A block-centric framework for distributed computation on real-world graphs," *Proc. VLDB Endowment*, vol. 7, no. 14, pp. 1981–1992, 2014.

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

[15] J. Shun, and G. E. Blelloch, "Ligra: A lightweight graph processing framework for shared memory," *ACM SIGPLAN Notices*, vol. 48, no. 8, pp. 135–146, 2013.

[16] Y. Guo, A. L. Varbanescu, A. Iosup, C. Martella, and T. L. Willke, "Benchmarking graph-processing platforms: A vision," in *Proc. 5th ACM/SPEC Int. Conf. Performance Eng.*, 2014, pp. 289–292.

[17] A. Iosup, T. Hegeman, W. L. Ngai, S. Heldens, A. Prat-Prez, T. Manhardto, H. Chafio, M. Capotä, N. Sundaram, M. Anderson, I. G. Tänase, Y. Xia, L. Nai, and P. Boncz, "LDBC Graphalytics: A benchmark for large-scale graph analysis on parallel and distributed platforms," in *Proc. VLDB Endow*, Sep. 2016, vol. 9, no. 13, pp. 1317–1328.

[18] O. Batarfi, R. El Shawi , A. G. Fayoumi, R. Nouri, A. Barnawi, and S. Sakr, "Large scale graph processing systems: Survey and an experimental evaluation," *Cluster Computing*, vol. 18, no. 3, pp. 1189–1213, 2015.

[19] R. Elshawi, O. Batarfi, A. Fayoumi, A. Barnawi, and S. Sakr, "Big graph processing systems: State-of-the-art and open challenges," in *Proc. IEEE 1st Int. Conf. Big Data Comput. Service Appl.*, 2015, pp. 24–33.

[20] R. R. McCune, T. Weninger, and G. Madey, "Thinking like a vertex: A survey of vertex-centric frameworks for large-scale distributed graph processing," *ACM Comput. Surv.*, vol. 48, no. 2, Oct. 2015, Art. no. 25, doi: http://dx.doi.org/10.1145/2818185

[21] S. Salihoglu and J. Widom, "Gps: A graph processing system," in *Proc. 25th Int. Conf. Scientific Statistical Database Manage.*, 2013, Art. no. 22.

[22] Y. Guo, M. Biczak, A. L. Varbanescu, A. Iosup, C. Martella, and T. L. Willke, "How well do graph-processing platforms perform? an empirical performance evaluation and analysis," in *Proc. IEEE 28th Int. Parallel Distrib. Process. Symp.*, 2014, pp. 395–404.

[23] N. Doekemeijer and A. L. Varbanescu, "A survey of parallel graph processing frameworks," Delft University of Technology, Delft, The Netherlands, Tech. Rep. PDS-2014–003, 2014.

[24] A. Quamar, A. Deshpande, and J. Lin, "NScale: Neighborhood-centric large-scale graph analytics in the cloud," *The VLDB Journal*, vol. 25, no. 2, pp. 125–150, Apr. 2016.

[25] 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. USENIX Symp. Operating Syst. Des. Implementation*, 2014, pp. 599–613.

[26] A. Roy, I. Mihailovic, and W. Zwaenepoel . "X-stream: Edge-centric graph processing using streaming partitions," in *Proc. 24th ACM Symp. Operating Syst. Principles*, 2013, pp. 472–488.

[27] A. Roy, L. Bindschaedler, J. Malicevic, and W. Zwaenepoel, "Chaos: Scale-out graph processing from secondary storage," in *Proc. 25th Symp. Operating Syst. Principles*, 2015, pp. 410–424.

[28] B. Shao, H. Wang, and Y. Li, "Trinity: A distributed graph engine on a memory cloud," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2013, pp. 505–516.

[29] J. Seo, J. Park, J. Shin, and M. S. Lam, "Distributed SociaLite: A datalog-based language for large-scale graph analysis," *Proc. VLDB Endowment*, vol. 6, no. 14, pp. 1906–1917, 2013.

[30] D. Nguyen, A. Lenharth, and K. Pingali, "A lightweight infrastructure for graph analytics," in *Proc. 24th ACM Symp. Operating Syst. Principles*, 2013, pp. 456–471.

[31] A. Buluc and J. R. Gilbert, "The Combinatorial BLAS: Design, implementation, and applications," *Int. J. High Performance Comput. Appl.*, vol. 25, no. 4, pp. 496–509, 2011.

[32] I. Hoque and I. Gupta, "LFGraph: Simple and fast distributed graph analytics," in *Proc. 1st ACM SIGOPS Conf. Timely Results Operating Syst.*, 2013, Art. no. 9.

[33] F. McSherry, M. Isard, and D. G. Murray, "Scalability! But at what COST," in *Proc. 15th Workshop Hot Topics Operating Syst.*, 2015, pp. 14–14.

[34] A. Kyrola, G. E. Blelloch, and C. Guestrin, "GraphChi: Large-Scale Graph Computation on Just a PC," in *Proc. USENIX Symp. Operating Syst. Des. Implementation*, vol. 12, pp. 31–46. 2012.

[35] D. Yan, J. Cheng, K. Xing, Y. Lu, W. Ng, and Y. Bu, "Pregel algorithms for graph connectivity problems with performance guarantees," *Proc. VLDB Endowment*, vol. 7, no. 14, pp. 1821–1832, 2014.

[36] A. Ching, S. Edunov, M. Kabiljo, D. Logothetis, and S. Muthukrishnan, "One trillion edges: Graph processing at Facebook-scale," *Proc. VLDB Endowment*, vol. 8, no. 12, pp. 1804–1815, 2015.

[37] M. Han and K. Daudjee, "Giraph unchained: Barrierless asynchronous parallel execution in pregel-like graph processing systems," *Proc. VLDB Endowment*, vol. 8, no. 9, pp. 950–961, 2015.
[38] C. Xie, R. Chen, H. Guan, B. Zang, and H. Chen, "Sync or async: Time to fuse for distributed graph-parallel computation," in *Proc. 20th ACM SIGPLAN Symp. Principles Practice Parallel Program.*, 2015, pp. 194–204.
[39] D. P. Bertsekas and J. N. Tsitsiklis, *Parallel and Distributed Computation: Numerical Methods*. Upper Saddle River, NJ, USA: Prentice-Hall, 1989.
[40] J. Kepner and J. Gilbert, Eds. *Graph Algorithms in the Language of Linear Algebra*, vol. 22. Philadelphia, PA, USA: SIAM, 2011.
[41] F. Bancilhon, *Naive Evaluation of Recursively Defined Relations*. New York, NY, USA: Springer, 1986.
[42] Meyer, Ulrich, and Peter Sanders, "Δ-stepping: A parallelizable shortest path algorithm," *J. Algorithms*, vol. 49, no. 1, pp. 114–152, 2003.
[43] G. Wang, W. Xie, A. J. Demers, and J. Gehrke, "Asynchronous Large-Scale Graph Processing Made Easy," in *Proc. 6th Biennial Conf. Innovative Data Syst. Res. (CIDR'13)*, 2013.
[44] Y. Lu, J. Cheng, D. Yan, and H. Wu, "Large-scale distributed graph computing systems: An experimental evaluation," *Proc. VLDB Endowment*, vol. 8, no. 3, pp. 281–292, Nov. 2014. doi: http://dx.doi.org/10.14778/2735508.2735517
[45] L. Qin, J. X. Yu, L. Chang, H. Cheng, C. Zhang, and X. Lin, "Scalable big graph processing in MapReduce," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2014, pp. 827–838.
[46] N. Satish, et al., "Navigating the maze of graph analytics frameworks using massive graph datasets," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2014, pp. 979–990.
[47] D. Yan, J. Cheng, Y. Lu, and W. Ng, "Effective techniques for message reduction and load balancing in distributed graph computation," in *Proc. 24th Int. Conf. World Wide Web*, 2015, pp. 1307–1317.
[48] Apache Hadoop. [Online]. Available: http://hadoop.apache.org/, Last Accessed: Jan. 2016.
[49] Apache Flink. [Online]. Available: http://flink.apache.org/, Last Accessed: Jan. 2016.
[50] Introducing Gelly: Graph Processing with Apache Flink. [Online]. Available: http://flink.apache.org/news/2015/08/24/introducing-flink-gelly.html, Last Accessed: Dec. 2015.
[51] Apache Tinkerpop (incubating). [Online]. Available: http://tinkerpop.incubator.apache.org/, Last Accessed: Dec. 2015.
[52] M. A. Rodriguez, "The Gremlin graph traversal machine and language (invited talk)," in *Proc. 15th Symp. Database Program. Languages*, 2015, pp. 1–10.
[53] Apache Hama. [Online]. Available: http://hama.apache.org/, Last Accessed: Dec. 2015.
[54] K. Munagala and A. Ranade, "I/O-complexity of graph algorithms," in *Proc. 10th Annu. ACM-SIAM Symp. Discrete Algorithms*, 1999, pp. 687–694.
[55] C. H. C. Teixeira, et al., "Arabesque: A system for distributed graph mining," in *Proc. 25th Symp. Operating Syst. Principles*, 2015, pp. 425–440, doi: http://dx.doi.org/10.1145/2815400.2815410
[56] S. Brin and L. Page, "The anatomy of a large-scale hypertextual web search engine, 1998, " in *Proc. 7th World Wide Web Conf.*, 2007, pp. 107–117.
[57] S. Ewen, K. Tzoumas, M. Kaufmann, and V. Markl, ,"Spinning fast iterative data flows," *Proc. VLDB Endowment*, vol. 5, no. 11, pp. 1268–1279, 2012.
[58] V. Kalavri, S. Ewen, K. Tzoumas, V. Vlassov, V. Markl, and S. Haridi, "Asymmetry in large-scale graph analysis, explained," in *Proc. Workshop GRAph Data Manage. Experiences Syst.*, 2014, pp. 1–7.
[59] F. D. McSherry, R. Isaacs, M. A. Isard, and D. G. Murray, "Differential dataflow," in *Proc. Conf. Innovative Data Syst. Res. (CIDR)*, 2013.
[60] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi, "Naiad: A timely dataflow system," in *Proc. 24th ACM Symp. Operating Syst. Principles*, 2013, pp. 439–455.
[61] S. Salihoglu and J. Widom, "Optimizing graph algorithms on pregel-like systems," *Proc. VLDB Endow.*, vol. 7, no. 7, pp. 577–588, Mar. 2014, doi: http://dx.doi.org/10.14778/2732286.2732294
[62] U. Kang, C. E. Tsourakakis, and C. Faloutsos, "PEGASUS: A peta-scale graph mining system implementation and observations," in *Proc. 9th IEEE Int. Conf. Data Mining*, 2009, pp. 229–238, doi: http://dx.doi.org/10.1109/ICDM.2009.14
[63] W. Han, et al., "Chronos: A graph engine for temporal graph analysis," in *Proc. 9th Eur. Conf. Comput. Syst.*, 2014, Art. no. 1.
[64] S. Tasci and M. Demirbas, "Giraphx: Parallel yet serializable large-scale graph processing," in *Proc. Euro-Par Parallel Process.*, 2013, pp. 458–469.
[65] R. Chen, X. Ding, P. Wang, H. Chen, B. Zang, and H. Guan, "Computation and communication efficient graph processing with distributed immutable view," in *Proc. 23rd Int. Symp. High-Performance Parallel Distrib. Comput.*, 2014, pp. 215–226.
[66] N. T. Bao and T. Suzumura, "Towards highly scalable pregel-based graph processing platform with x10," in *Proc. 22nd Int. Conf. World Wide Web Companion*, 2013, pp. 501–508.
[67] J. Xue, Z. Yang, Z. Qu, S. Hou, and Y. Dai, "Seraph: An efficient, low-cost system for concurrent graph processing," in *Proc. 23rd Int. Symp. High-Performance Parallel Distrib. Comput.*, 2014, pp. 227–238.
[68] U. N. Raghavan, R. Albert, and S. Kumara, "Near linear time algorithm to detect community structures in large-scale networks," *Physical Rev.*, vol. E 76, no. 3, 2007, Art. no. 036106.
[69] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst, "HaLoop: Efficient iterative data processing on large clusters," *Proc. VLDB Endowment*, vol. 3, no. 1–2, pp. 285–296, Sep. 2010. doi: http://dx.doi.org/10.14778/1920841.1920881
[70] J. Ekanayake, et al., "Twister: A runtime for iterative MapReduce," in *Proc. 19th ACM Int. Symp. High Performance Distrib. Comput.*, 2010, pp. 810–818, doi: 10.1145/1851476.1851593 http://doi.acm.org/10.1145/1851476.1851593
[71] R. Chen, M. Yang, X. Weng, B. Choi, B. He, and X. Li, "Improving large graph processing on partitioned graphs in the cloud," in *Proc. 3rd ACM Symp. Cloud Comput.*, 2012, Art. no. 3, doi: http://dx.doi.org/10.1145/2391229.2391232
[72] R. Power and J. Li, "Piccolo: Building fast, distributed programs with partitioned tables," in *Proc. USENIX Symp. Operating Syst. Des. Implementation*, 2010, pp. 1–14.
[73] S. Venkataraman, E. Bodzsar, I. Roy, A. AuYoung, and R. S. Schreiber, "Presto: Distributed machine learning and graph processing with sparse matrices," in *Proc. 8th ACM Eur. Conf. Comput. Syst.*, 2013, pp. 197–210.
[74] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," in *Proc. 2nd USENIX Conf. Hot Topics Cloud Comput.*, 2010, Art. no. 10.
[75] M. Zaharia, et al., "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proc. 9th USENIX Conf. Netw. Syst. Design Implementation*, 2012, pp. 2–2.
[76] A. Alexandrov, et al., "The Stratosphere platform for big data analytics," *VLDB J.*, vol. 23, no. 6, pp. 939–964, 2014.
[77] V. Borkar, M. Carey, R. Grover, N. Onose, and R. Vernica, "Hyracks: A flexible and extensible foundation for data-intensive computing," in *Proc. IEEE 27th Int. Conf. Data Eng.*, 2011, pp. 1151–1162.
[78] S. Alsubaiee, et al., "ASTERIX: An open source system for Big Data management and analysis," *Proc. VLDB Endowment*, vol. 5, no. 12, pp. 1898–1901, 2012.
[79] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: Distributed data-parallel programs from sequential building blocks," *ACM SIGOPS Operating Syst. Rev.*, vol. 41, no. 3, pp. 59–72, 2007.
[80] V. Kalavri, T. Simas, and D. Logothetis, "The shortest path is not always a straight line: Leveraging semi-metricity in graph analysis," *Proc. VLDB Endowment*, vol. 9, no. 9, pp. 672–683, 2016.
[81] D. Yan, et al., "A general-purpose query-centric framework for querying big graphs," *Proc. VLDB Endowment*, vol. 9, no. 7, pp. 564–575, 2016.

Vasiliki Kalavri received the PhD degree in distributed computing from KTH, Stockholm and UCLouvain, Belgium. She is a postdoctoral fellow in the Systems group, Department of Computer Science, ETH Zurich, Switzerland. Her research interests include distributed data processing, streaming systems and algorithms, data center management and performance analysis, and distributed graph processing. She is a PMC member of Apache Flink and a core developer of its graph processing API, Gelly. Prior to joining ETH, She was an Erasmus Mundus Joint Doctorate fellow.

Vladimir Vlassov is an associate professor of computer systems in the Department of Software and Computer Systems, School of Information and Communication Technology, KTH Royal Institute of Technology in Stockholm, Sweden. His current research focus is on data-intensive computing, data mining, and autonomic computing. He was visiting scientist with Massachusetts Institute of Technology (1998), and the University of Massachusetts Amherst (2004). He has participated in number of EU research projects and projects funded by Swedish funding agencies and US National Science Foundation USA. He is one of the coordinators of the Erasmus Mundus Joint Doctorate in distributed computing.

Seif Haridi is the chair-professor of computer systems with KTH, the Royal Institute of Technology, Sweden. He is also the chief scientist of the research institute RISE SICS. His research covers several areas in computer science, including distributed systems, programming systems, distributed algorithms and data-intensive computing.