Composing Optimization Techniques for Vertex-Centric Graph Processing via Communication Channels

Yongzhe Zhang[*†], Zhenjiang Hu[*†‡]
[*]SOKENDAI (The Graduate University for Advanced Studies), Japan
[†]National Institute of Informatics (NII), Japan
[‡]University of Tokyo, Japan
zyz915@nii.ac.jp, hu@nii.ac.jp

***Abstract*—Pregel's vertex-centric model allows us to implement many interesting graph algorithms, where optimization plays an important role in making it practically useful. Although many optimizations have been developed for dealing with different performance issues, it is hard to compose them together to optimize complex algorithms, where we have to deal with multiple performance issues at the same time. In this paper, we propose a new approach to composing optimizations, by making use of the *channel* interface, as a replacement of Pregel's message passing and aggregator mechanism, which can better structure the communication in Pregel algorithms. We demonstrate that it is convenient to optimize a Pregel program by simply using a proper channel from the channel library or composing them to deal with multiple performance issues. We intensively evaluate the approach through many nontrivial examples. By adopting the channel interface, our system achieves an all-around performance gain for various graph algorithms. In particular, the composition of different optimizations makes the S-V algorithm 3.39x faster than the current best implementation.**

***Index Terms*—Distributed Computing, Performance Evaluation, Software Architecture**

I. Introduction

Nowadays, with the increasing demand of analyzing large-scale graph data in billion or even trillion scale (e.g., the social network and world wide web), lots of research [20], [30] has been devoted to distributed systems for efficiently processing large-scale graphs. Google's Pregel system [19] is one of the most popular frameworks to handle such kind of massive graphs. It is based on the BSP model [28] and adopts the vertex-centric paradigm with explicit messages to support scalable big graph processing. Pregel's vertex-centric model has demonstrated its usefulness in implementing many interesting graph algorithms [19], [21], [24], [33], and imposed influence over the design of Pregel-like systems, such as Giraph [1], GPS [22], Mizan [13], Pregel+ [2].

While Pregel provides a friendly interface for processing massive graphs, current research shows that it is important to introduce optimizations for dealing with various performance issues such as imbalanced workload (a.k.a. skewed degree distribution) [2], [10], [32], redundancies in communication [3], [22], [32] and low convergence speed [26], [27], [31]. However, there remains one challenge: although the usefulness of these optimizations are well demonstrated in solving simple algorithms such as PageRank and single-source shortest path (SSSP)[1], it is, however, hard to combine them together to implement complex algorithms, where we may have to deal with multiple performance issues at the same time.

To see this challenge clearly, let us consider the S-V algorithm [25], [33], a known algorithm for computing connected components in undirected graph, which can be regarded as a distributed union-find algorithm [9]. Essentially, it is an iterative algorithm with two key operations — pointer jumping and tree merging. For the pointer jumping operation, the communication suffers from imbalanced workload [32], and in the meantime in the tree merging operation, the neighborhood communication (every vertex broadcasts a message to all of its own neighbors) could be potentially very heavy. Although there are techniques in separate systems [3], [22], [32] dealing with each case, there is no system capable of optimizing away both issues at the same time. The main reason is Pregel's monolithic message mechanism. When having multiple communication patterns in a single Pregel program, the system has no idea which message is for what purpose, thus it can do nothing for optimization.

In this paper, we propose a new approach to composing various optimizations together, by making use of the interface called *channel* [35] as a replacement of Pregel's message passing mechanism. Informally, a channel is responsible for sending or receiving messages of a certain pattern for some purpose (such as reading all neighbors' states, requesting data from some other vertex and so on). And by slicing the messages by their purpose and organizing them in channels, we can characterize each channel by high-level communication patterns, identify the redundancies or potential performance issues, and then apply suitable optimizations to deal with the problems.

The technical contributions of this work can be summarized as follows.

- First, we provide Pregel with a channel-based vertex-centric programming interface, which is intuitive in the sense that it is just a natural extension of Pregel's monolithic message mechanism. To demonstrate the power of the channel interface, we implement three optimizations as special channels and show how they are easily composed to optimize complex algorithms such as the above S-V algorithm.

[1]PageRank and SSSP are basically a loop executing a simple computation kernel.

- Second, we have fully implemented the system and the experiment results convincingly show the usefulness of our approach. The channel interface itself contributes to an up to 76% reduction of message size especially for complex algorithms, and the three optimized channels further improve the performance of the algorithms they are applicable (3.50x for PageRank, 4.41x for Pointer-Jumping and 5.20x for weakly connected components). Specially, the composition of different optimizations makes the S-V algorithm 3.39x faster than the best implementation available now.

The rest of the paper is organized as follows. Section II briefly reviews the basic concepts of Pregel, Section III introduces the programming interface of our channel-based system, as well as non-trivial examples showing how optimizations are composed in the same algorithm. Section IV presents the channel mechanism and the implementation of our optimizations as channels. Section V presents the experiment results, Section VI discusses the related work, and Section VII concludes the paper.

II. Pregel and its Limitations

In this section, we give a brief introduction to Pregel and illustrate the limitations of its message passing interface.

A. Pregel

The Pregel system takes a directed graph as input (an undirected graph can be treated as a directed graph that always has edges in both directions), where user-specified values can be associated on either vertices or edges. A Pregel computation consists of a series of supersteps separated by global synchronization points. In each superstep, the vertices compute in parallel executing the same user-defined function (usually the `compute()` method) that expresses the logic of a given algorithm. A vertex can read the messages sent to it in the previous superstep, mutate its state, and send messages to its neighbors or any known vertex in the graph. The termination of the algorithm is based on every vertex voting to halt. Each vertex is associated with a flag indicating whether it is active, and initially it is set to true. During the computation, a vertex can deactivate itself by invoking a `vote_to_halt()` method, and it can be reactivated externally by receiving messages

Pregel provides the message passing interface for inter-vertex communication and aggregator for global communication.

Message passing and the combiner. In Pregel, vertices communicate directly with each other by sending messages, where each message consists of a message value and a destination. The combiner optimization [19] is applicable if the receiver only needs the aggregated result (like the sum, or the minimum) of all message values, in which case the system is provided an associative binary function to combine messages for the same destination whenever possible.

Aggregator. Aggregator is a useful interface for global communication, where each active vertex provides a value, and the system aggregates them to a final result using a user-specified operation and makes it available to all vertices in the next superstep.

B. Problems in Pregel's Message Interface

Pregel is designed to support iterative computations for graphs, and it is indeed suitable for algorithms like the PageRank or SSSP. However, it is noteworthy that vertex-centric graph algorithms are in general complex. Even for some fundamental problems like connected component (CC), strongly connected component (SCC) and minimum spanning forest (MSF), their efficient vertex-centric solutions require multiple computation phases, each having different communication patterns [24], [33]. For such complex algorithms, all the computation phases have to share Pregel's message passing interface, which causes the following problems:

- When different message types are needed in different computation phases, the Pregel's message interface has to be instantiated with a type that large enough to carry all those message values.
- Usually, we can no longer optimize any of the communication patterns in these computation phases, since the system cannot distinguish which message is to be optimized.

As mentioned before, these are the consequences of Pregel's monolithic message mechanism, which may not only increase the message size, but also prevent the possible optimizations to be applied. We have detailed discussions using S-V in Section III-C and evaluation results showing the computation and communication overhead in Section V-A. It motivates us to design a better communication interface for Pregel to efficiently handle a wide range of complex vertex-centric graph algorithms.

III. Programming with Channels

The channel mechanism is designed to help users organize the communications in vertex-centric graph algorithms. Concretely speaking, the channels are message containers equipped with a set of methods for sending/receiving messages or supporting a specific communication pattern (see Table I and Table II for the standard and optimized channels; the details are in Section IV-D). In this section, we first introduce the programming interface using the PageRank example, then we show how different optimizations can be easily composed via channels in a more complex algorithm called the S-V [25].

A. A Standard PageRank Implementation Using Channels

Writing a vertex-centric algorithm in our system using the standard channels is rather straightforward for a Pregel programmer. We present a PageRank Implementation in Fig. 1, which is basically obtained from a Pregel program (a vertex-centric `compute()` function with a parameter of received messages from the previous superstep) by replacing the sending/reading of messages by one or more user-defined message channel's send/receive methods.

In the first 30 supersteps, each vertex sends along outgoing edges (if exists) its tentative PageRank divided by

TABLE I: The APIs for standard channels.

Message-Passing Channels		Aggregator Channel
`DirectMessage(Worker<VertexT> *w);`	`CombinedMessage(Worker<VertexT> *w, Combiner<ValT> c);`	`Aggregator(Worker<VertexT> *w, Combiner<ValT> c);`
`void send_message(KeyT dst, ValT m);` `MsgIterator<KeyT, ValT> &get_iterator();`	`void send_message(KeyT dst, ValT m);` `const ValT &get_message();`	`void add(ValT v);` `const ValT &result();`

TABLE II: The APIs for optimized channels.

Scatter-Combine	Request-Respond	Propagation (Simplified)
`ScatterCombine(Worker<VertexT> *w, Combiner<ValT> c);`	`RequestRespond(Worker<VertexT> *w, function<RespT(VertexT)> f);`	`Propagation(Worker<VertexT> *w, Combiner<ValT> c);`
`void add_edge(KeyT dst);`		`void add_edge(KeyT dst);`
`void set_message(ValT m);` `const ValT &get_message();`	`void add_request(KeyT dst);` `const RespT &get_response();`	`void set_value(ValT m);` `const ValT &get_value();`

```
1   using VertexT = Vertex<int, PRValue>;
2   auto c = make_combiner(c_sum, 0.0); // a combiner
3   class PageRankWorker : public Worker<VertexT> {
4   private:
5      // two channels are defined here
6      CombinedMessage<VertexT, double> nbr;
7      Aggregator<VertexT, double> agg;
8   public:
9      PageRankWorker():nbr(this, c), agg(this, c) {}
10
11     void compute(VertexT &v) override {
12        if (step_num() == 1) {
13           value().PageRank = 1.0 / get_vnum();
14        } else {
15           // s: the pagerank of the "sink node"
16           double s = agg.result() / get_vnum();
17           value().PageRank = 0.15 / get_vnum()
18                 + 0.85 * (nbr.get_message() + s);
19        }
20        if (step_num() < 31) {
21           int numEdges = value().Edges.size();
22           if (numEdges > 0) {
23              double msg = value().PageRank / numEdges;
24              for (int e : value().Edges)
25                 nbr.send_message(e, msg);
26           } else
27              agg.add(value().PageRank);
28        } else
29           vote_to_halt();
30     }
31  };
```

Fig. 1: PageRank implementation using channels.

the number of outgoing edges (lines 21–25), over a user-defined message channel `nbr`. This channel is an instance of `CombinedMessage`, which requires a combiner to be provided in its constructor (line 9). In the next superstep, every vertex gets the sum of the message values arriving on this channel (lines 18) and calculates a new PageRank. To avoid PageRank lost in dead ends (vertices without outgoing edges), we need a *sink* node to collect the PageRank from those dead ends and redistribute it to all nodes, which is implemented by an aggregator `agg` using the addition operator (line 9). Then, in line 27, users explicitly add the PageRank of the dead ends to the aggregator, and in the next superstep the sum is returned by the aggregator's `result()` method (line 16).

All the computation logic and the channels are written in a user-defined class called `PageRankWorker` that inherits from the `Worker` class in our system. The type of vertex ID and value type are packed in to the `VertexT` type and provided to the `Worker` class. We leave the explanation of `Worker` in the next section, and users just keep in mind that programs in our system are constructed in this way.

B. Channels and Optimizations

In our channel-based system, we offer a set of optimizations as special channels (in Table II), which can be regarded as more efficient implementations (compared to the standard message passing channels) of several communication patterns. Here, we demonstrate how to enable the scatter-combine optimization (which deals with the "static messaging pattern") for PageRank. The details of this optimization will be presented in Section IV-D.

Given a channel-based PageRank implementation in Fig. 1, what we need to do is simply switching the standard message channel `msg` to the scatter-combine channel. First, we change the definition of `nbr` as follows:

```
5      // change to the scatter-combine channel
6      ScatterCombine<VertexT, double> nbr;
```

Then, in the `compute()` function, we initialize the scatter-combine channel (by invoking the `add_edge()` method) before actually sending any data, which is done in the first superstep as below:

```
12     if (step_num() == 1) {
13        value().PageRank = 1.0 / get_vnum();
14        // provide the graph topology to the channel
15        for (int e : value().Edges)
16           nbr.add_edge(e);
17     } ...
```

Finally, we switch to the scatter-channel's dedicated interface for message passing, which is `set_message()` indicating a unique message value for all neighbors:

```
22     if (numEdges > 0) {
23        double msg = value().PageRank / numEdges;
24        // no need to specify the destination
25        nbr.set_message(msg);
26     } ...
```

The rest of the program remains the same. Our experiments (subsubsection V-B1) show that, by switching to the scatter-combine channel, the PageRank immediately gets 3x faster, and all the programmer need to understand is the high-level abstraction of each channel.

C. Composition of Channels

In this part, we use a more complicated example called the Shiloach-Vishkin (S-V) algorithm [25] to show that, users can easily combine different optimizations (channels) to handle multiple performance issues in the same program.

1) The S-V Algorithm: The S-V algorithm is in general an adaptation of the classic union-find algorithm [9] to the distributed setting, which finds the connected components in undirected graphs with n vertices in $O(\log n)$ supersteps. In the S-V algorithm, the connectivity information is maintained by a distributed tree structure called disjoint-set [9], where each vertex holds a pointer which points to either some other vertex in the same connected component or to itself. We henceforth use $D[u]$ to represent this pointer for vertex u. Following is the high-level description of the S-V algorithm using a domain-specific language called Palgol [37], and it compiles to Pregel+ code[2].

```
1   // initially suppose we have D[u] = u for every u
2   do
3     // enter vertex-centric mode
4     for u in V
5       // whether u's parent is a root vertex
6       if (D[D[u]] == D[u])
7         // iterate over neighbors (D[e]: neighbor's pointer)
8         let t = minimum [ D[e] | e <- Nbr[u] ]
9         if (t < D[u])
10          // modify the D field of u's parent D[u]
11          remote D[D[u]] <?= t
12      else
13        // the pointer jumping (path compression)
14        D[u] := D[D[u]]
15    end
16  until fix[D] // until D stabilizes for every u
```

Starting from n root nodes, the S-V algorithm iteratively merges the trees together if crossing edges are detected. In a vertex-centric way, every vertex u simultaneously performs one of the following operations depending on whether its parent $D[u]$ is a root vertex:

- **Tree merging (lines 7–11).** If $D[u]$ is a root vertex, u sends the smallest one of its neighbors' pointer (to which we give a name t) to the root $D[u]$ and later the root points to the minimum t it receives (to guarantee the correctness of the algorithm).
- **Pointer jumping (line 14).** If $D[u]$ is not a root vertex, u modifies its pointer to its "grandfather" ($D[u]$'s current pointer). Since all the vertices below the children of root perform this operation simultaneously, it halves the distance to the current root.

The algorithm terminates when all vertices' pointers do not change after an iteration. Readers interested in the correctness of this algorithm can be found in the original paper [33] for more details.

2) Choices of Channels: In the S-V algorithm, three major performance issues are identified below by analyzing the communication patterns in the algorithm.

- The load balance issue in testing whether $D[u]$ is a root vertex or not for every u. The standard implementation is to let each u send a request to its current parent $D[u]$, then the reply message is the parent's pointer. Due to the pointer jumping, the height of the tree will decrease and the width of the tree will increase, causing a few vertices with very large degree to slow down the reply phase.
- The heavy neighborhood communication in calculating the minimum parent ID of the neighboring vertices, where all vertices need to send a unique message value to all neighbors, regardless of the vertices' local state.
- The congestion issue in the modification of parent's pointer, due to the existence of high-degree vertices.

[2]The Palgol code is presented here for easy understanding. It currently compiles to Pregel+ using only the message interface, and the the performance is close to the hand-written code [37].

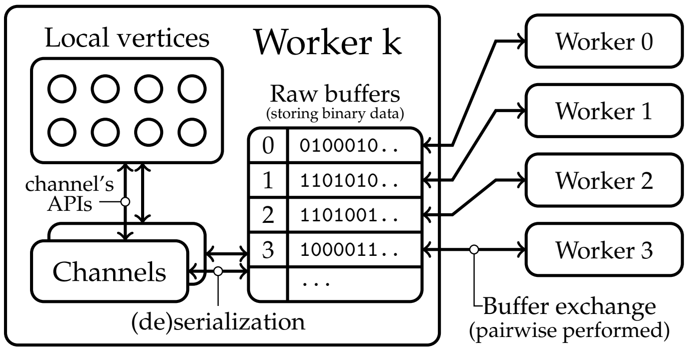


Fig. 2: The architecture of our channel-based system.

We provide the solutions to all of these issues in our system as special channels, and users just need to choose the proper channels and combine them together in the program. For the S-V algorithm, the load balance issue can be avoided by the request-respond channel, the heavy neighborhood communication is optimized by the scatter-combine channel, and a message channel with combiner solves the congestion issue.

IV. Channel Implementation

In this section, we present the design of our channel mechanism and demonstrate how three interesting channels are implemented for dealing with different performance issues.

A. Overview

Fig. 2 shows the architecture of our channel-based system. Worker is the basic computing unit in our system. When launching a graph processing task, multiple instances of workers are created, each holding a disjoint portion of the graph (a subset of vertices along with their states and adjacent lists). Workers share no memory but can communicate with each other. Such big picture is common in all Pregel systems, but ours has a unique hierarchy of the components inside the worker.

In our system, channels form an independent layer inside the worker between the vertices and the raw buffers. Each worker has $M - 1$ buffers (where M is the number of workers launched by the user) for storing binary message data for each other worker, then the channels can read or write its own address space on these buffers. The system simply exchanges the contents of the buffers pairwise using the MPI

```
class Channel {
public:
   // initialization function
   virtual void initialize() {};
   // paired (de)serialization functions
   virtual void serialize(Buffer &buff) = 0;
   virtual void deserialize(Buffer &buff) = 0;
   // return true for additional buffer exchange
   virtual bool again() { return false; };
};
```

Fig. 3: The core functions of the base class Channel.

in every superstep. Each channel independently implements a communication pattern (like messages passing or aggregator) and exposes its own interfaces (like `send_message(dst, msg) for a message channel`) to the vertex. To implement an algorithm, users should inherit the `Worker` class, override the `compute()` function and allocate the channels that are suitable for the algorithm according to the communication patterns it has.

B. Design Principle

The channel mechanism mainly targets (but not limited to) a class of optimizations that handles the redundancies in communication. For example, the standard combiner optimization [19] allows the worker to combine the messages sent to the same destination by a user-defined binary operator, and the request-respond paradigm [2] merges the requests to the same destination to avoid sending redundant copies of the same value. In these optimizations, typically, each worker processes all the messages in batch and sends a more compact but equally informative message list. After the messages are delivered, the receiver worker may further process the data and dispatch the messages to the vertices.

Having such common pattern in these communication related optimizations, our channel mechanism tries to organize them in a modular way and make them work perfectly with the Pregel abstraction. Essentially, each channel is a user-specified message handler that is invoked by the worker in every superstep. The vertices actually put messages into (or fetch messages from) the channels' local storage through each channel's dedicated APIs, and the message handler can access all the local data of the channel as well as current worker's states to implement a particular communication pattern. Channels are registered on the worker, so the composition of channels is actually trivial, which is accomplished by the worker carefully separating the messages of each channel in its message buffer.

C. The Channel Interface

Fig. 3 shows the base class `Channel` and its core functions: `initialize()`, `serialize()` for writing data to worker's raw buffer, `deserialize()` for reading data from worker's raw buffer (after the buffer exchange) and `again()` for supporting multiple rounds of communication. All the channels in our paper are implemented as derived classes of `Channel` with proper implementations of these four functions (in particular `serialize()` and `deserialize()`).

```
load_graph()
foreach channel c do c.initialize()
foreach vertex v do v.set_active(true)
while (active vertex exists) // a superstep
   foreach active vertex v do this.compute(v)
   foreach channel c do c.set_active(true)
   while (active channel exists)
      foreach active channel c do c.serialize()
      buffer_exchange()
      foreach active channel c do
         c.deserialize()
         c.set_active(c.again())
      end_for
   end_while
end_while
dump_graph()
```

Fig. 4: The computation logic of the worker for illustrating the channel mechanism.

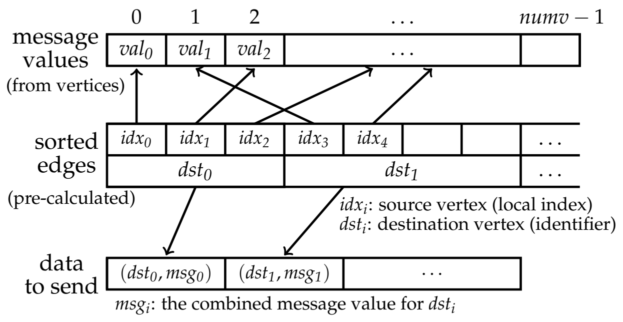
To clearly see how the workers and channels cooperate with each other, we present the computation logic of the worker in Fig. 4. The worker's computation is organized as synchronized supersteps. In each superstep, the worker first calls the `compute()` on every vertex, then it performs several rounds of buffer exchanges. In each round, the system invokes the active channels' `serialize()` and `deserialize()` methods to exchange the data between the channels and the buffers. All channels are set to active at the beginning, but they can deactivate themselves by returning `false` in the `again()` function. Channels' `initialize()` is invoked at the beginning of the computation, in which the channel can access the basic information of the graph (like graph size, number of vertices on the current worker) for initialization. While not explicitly presented in the code, the channels can activate vertices through the `Worker`'s interface by providing the vertex's ID or local index. That is how our system simulates the voting-to-halt mechanism of Pregel.

D. Case Studies

As the last part of this section, we demonstrate how to implement three optimizations, which target three important performance issues in vertex-centric graph processing.

1) Scatter-Combine Channel: The scatter-combine abstraction is a common high-level pattern appeared in many single-phase algorithms such as PageRank, single-source shortest path (SSSP) and connected component (CC). The communication in this model is captured by a `scatter()` function on each vertex to send a unique value to all neighbors, and a `combine()` function to combine the messages for each receiver. We focus on a special case where *every* vertex needs to send a value to all of its neighbors[3] regardless of its local state. An iterative algorithm having such static messaging pattern will waste time repeating the same message dispatching procedure, while a proper preprocessing can greatly reduce the computation time as well as the message size.

[3]In some algorithms like SSSP or WCC, only active vertices need to send messages, which is not the case we are targeting here.



0
1
2
...
numv − 1
message values (from vertices)
val0
val1
val2
sorted edges (pre-calculated)
idx0
idx1
idx2
idx3
idx4
dst0
dst1
idxi: source vertex (local index)
dsti: destination vertex (identifier)
data to send
(dst0, msg0)
(dst1, msg1)
msgi: the combined message value for dsti

Fig. 5: The execution logic for the scatter-combine channel.

Fig. 5 demonstrates the computation logic of the scatter-combine channel. Suppose the vertices on an worker is indexed by `0..numv-1`, then each local edge is a pair (idx, dst) where idx refers to a local vertex and the dst can be an arbitrary vertex in the graph. We sort the edges by dst in advance, then by scanning the array of the sorted edge list once, we can quickly calculate for each destination a combined message value. This is much cheaper than the normal message routine which typically requires hashing or sorting.

The APIs for the scatter-combine channel are presented in the first column of Table II. Users need to initialize the channel by adding the outgoing edges of each vertex through the `add_edge()` function before the first message sending occurs in the execution. Then, every vertex emits an initial messages using the `send_message()` interface and the combined messages for each vertex can be obtained by the `get_message()` method in the next superstep.

2) Request-Respond Channel: This is a communication pattern where two rounds of message passing (say the request phase and respond phase) together form a conversation to let every vertex request an attribute of another vertex. Typically, such computation contains vertices with high degree which causes imbalanced workload in the respond phase, and the solution is to merge the requests of the same destination on each worker. More details can be found in the original paper [32].

Our implementation of this optimization is illustrated in Fig. 6. A request is a pair (idx, dst) where idx refers to a local requester and the dst can be an arbitrary vertex in the graph. The worker sorts the requests by dst and sends exactly one message containing the worker ID to each of the unique destinations. When receiving the response values, the worker performs another scan to the sorted requests, which is sufficient to reply to all the requesters.

The middle column of Table II shows the APIs of the request-respond channel. When creating the channel, users need to provide a function that generates a response value from a vertex's state. The whole procedure is implemented in an implicit style; A vertex invokes `add_request()` with the destination vertex ID; all the requests are delivered after the request phase, and the vertices receiving any request will be automatically involved, and a response value is produced by

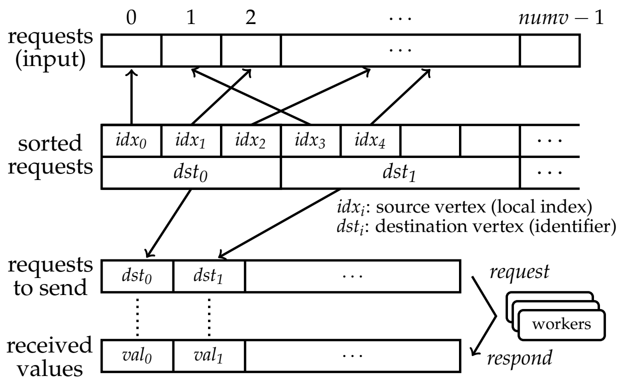


Fig. 6: The execution logic for the request-respond channel.

the user-provided function.

3) Propagation Channel: The last optimized channel is to speedup the convergence for a class of propagation-based algorithms. In these algorithms, typically, some vertices emit the initial labels, and in each of the following supersteps, vertices receiving the labels will perform some computation and may further propagate a new label to their outgoing neighbors. Since the propagation is between neighbors, such algorithms converge very slowly on graphs with large diameters.

The design of this channel is inspired by two existing techniques for improving the convergence speed. First, the GAS model [10] with an asynchronous execution mode can perform the crucial updates as early as possible without waiting for the global synchronization. Although this implementation is not feasible in our synchronous system, the high-level abstraction is suitable for describing such kind of computation. Second, the block-centric computation model [26], [27], [31] is an extension of Pregel which opens the partition to users, so that users can choose a suitable partition method and implement a block-level computation to perform the label propagation within a connected subgraph.

Our propagation channel combines the advantages of these two techniques: it provides a simplified GAS model which naturally describes such propagation-based computation, and its implementation works in a similar way as a block-level program to accelerate the label propagation. Therefore, users allocate a channel to obtain a performance gain without additional efforts on writing the block-level program.

Fig. 7 describes the high-level model for the propagation channel as well as the execution logic in our implementation. Initially, each vertex is associated with a value and is set to active. Whenever having an active vertex u in the graph, it reads each incoming neighbors and the corresponding edges (if exists), and calculate a value a_i by a user-provided function f. Then, a combiner h updates the original vertex value u by each neighbor's a_i and returns a new vertex value u'. If the new value u' is different from the original value u, we activate all outgoing neighbors of u to propagate the update, and finally u is deactivated after being processed. The computation stops when all the vertices are inactive. Note that we require h to be

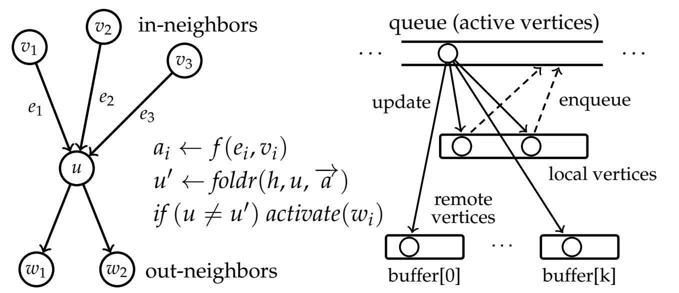


Fig. 7: The propagation channel's high-level model and computation logic.

commutative, so that the order of combining a_i does not affect the result. Moreover, when any of the incoming neighbors of u is modified, u needs to read the modified vertex to update its own value, instead of recomputing the *foldr* by reading all its incoming neighbors' values.

This computation model is implemented by each worker performing a BFS-like traversal on the subgraph it holds. Starting from the initial setting, each worker propagates the values along the edges as far as possible. It updates the local vertices directly, but records the changes on remote vertices as messages. The buffer exchange is performed after no update is viable on any worker. After the remote updates triggered by messages, a new round of local propagation is performed. It terminates when all vertices have converged.

The last column of Table II shows the APIs of a simplified propagation channel without considering the edge weights (for saving space), so users provide a combiner to calculate the new vertex value. Each vertex adds its adjacent list to the channel via `add_edge()` and sets the initial value by `set_value()`, and in the next superstep, a vertex invokes `get_value()` to get the final value after the propagation converges. To make the best use of the propagation channel, users should properly partition the graph and attach the partition IDs to the vertex IDs.

V. Experiments

The experiments are conducted on an Amazon EC2 cluster of 16 nodes (with instance type m5.2xlarge), each having 8 vCPUs and 32G memory. The connectivity between any pair of nodes in the cluster is 10Gb. The datasets are listed in Table III including both real-world graphs (Wikipedia[4], Twitter[5] and SubDomain[6]) and synthesized graphs (Chain, Tree and RMAT [14]). Graphs are converted to the required type (directed, undirected or weighted) for each algorithm in the experiments.

We select six representative algorithms in our evaluation, including PageRank (PR), Pointer-Jumping (PJ), Weakly Connected Component (WCC), S-V algorithm (S-V), Strongly Connected Component (SCC) and Minimum Spanning Forest (MSF). For comparison, we also present the results of our best-effort implementations in Pregel+ [2] and Blogel [31]. Both of them are typical Pregel implementations, where Pregel+ supports the request-respond paradigm and mirroring technique in two special modes (*reqresp* mode and *ghost* mode respectively) and Blogel supports the block-centric computation. All of these systems mentioned above as well as our channel-based system are implemented in C++ on top of the Hadoop Distributed File System (HDFS). The source code of our system can be accessed at `https://bitbucket.org/zyz915/pregel-channel`.

TABLE III: Datasets used in our evaluation

Dataset	$\lvert V\rvert$	$\lvert E\rvert$	avg. Deg
SubDomain	99.41M	1.94B	19.52
Twitter	41.65M	1.47B	70.51
Tree*	1.00B	1.00B	1.00
Chain*	1.00B	1.00B	1.00
RMAT*	400M	2.00B	5.00
Wikipedia	18.27M	172.31M	9.43

datasets marked with $*$ are synthetic.

A. The Channel Mechanism

First, we evaluate the standard channels (the message passing channels and aggregator) in our system. Basically, rewriting a Pregel program into a channel-based version is just about replacing the matched send-receive pairs into the same channel's send/receive function. The message is chosen as small as possible, and we always use a combiner if applicable. We compare both implementations to see whether there is any overhead or benefits introduced by our channel mechanism.

The experiment results are presented in Table IV, where a straightforward rewriting achieves a speedup ranging from 1.16x to 4.16x among all the five algorithms on those datasets. For SCC and S-V, we also observe a significant reduction on message size ranging from 23% to 62%.

Analysis. The channel mechanism itself can improve the performance, due to the following two reasons. First, our system allows users to specify a combiner to a channel whenever applicable, while in Pregel, we can specify a global combiner only when all the messages in the algorithm can use that combiner. This difference makes our S-V and SCC more message-efficient, where the inapplicability of combiner in Pregel+ causes a 4.16x and 2.10x message size for S-V and SCC respectively on Twitter.

Second, our channel-based system allows users to choose different message types for different channels, while in Pregel+, a global message type is chosen to serve all communication in the program. Then, the MSF (we refer to a particular version here [8]) is a typical example that uses heterogeneous messages in different phases of the algorithm. The largest message type is a 4-tuple of integer values for storing an edge, but the smallest one is just an `int`.

For the rest algorithms, there is no significant difference when implemented in two systems. Still, our system reduces

[4]http://konect.uni-koblenz.de/networks/dbpedia-link
[5]http://konect.uni-koblenz.de/networks/twitter_mpi
[6]http://webdatacommons.org/hyperlinkgraph/2012-08/download.html

TABLE IV: Comparison of the basic implementation of graph algorithms in Pregel+ and channel-based system.

PJ	Chain		Tree			
	pregel	channel	pregel	channel		
runtime (s)	596.99	327.86	221.76	105.70		
msg (GB)	463.86	463.86	89.17	89.17		

S-V	RMAT		SubDomain		Twitter	
	pregel	channel	pregel	channel	pregel	channel
runtime (s)	465.98	232.91	411.67	155.36	143.03	59.71
msg (GB)	212.64	112.49	298.62	71.82	115.96	28.53

PR	RMAT		SubDomain		Twitter	
	pregel	channel	pregel	channel	pregel	channel
runtime (s)	509.93	417.70	308.83	236.36	235.37	194.00
msg (GB)	413.03	413.03	160.99	160.99	128.81	128.81

WCC	RMAT		SubDomain		Twitter	
	pregel	channel	pregel	channel	pregel	channel
runtime (s)	65.36	42.31	54.61	41.02	22.58	17.47
msg (GB)	37.83	37.83	19.65	19.65	9.26	9.26

SCC	RMAT		SubDomain		Twitter	
	pregel	channel	pregel	channel	pregel	channel
runtime (s)	266.52	116.24	345.38	382.68	99.99	56.64
msg (GB)	118.56	91.09	132.07	62.99	77.53	45.79

MSF	RMAT		SubDomain		Twitter	
	pregel	channel	pregel	channel	pregel	channel
runtime (s)	547.98	319.86	200.07	138.71	138.92	119.27
msg (GB)	438.41	400.11	173.18	161.53	123.36	117.50

TABLE V: Experiment results for each optimized channel.

Scatter-Combine channel using PR				
Program	SubDomain		Twitter	
	runtime	message	runtime	message
pregel+ (basic)	308.83	160.99	235.37	128.81
pregel+ (ghost)	353.77	152.50	256.95	111.28
channel (basic)	236.36	160.99	194.00	128.81
channel (scatter)	88.18	109.12	69.46	87.31

Request-Respond channel using PJ				
Program	Tree		Chain	
	runtime	message	runtime	message
pregel+ (basic)	221.76	89.17	596.99	463.86
pregel+ (reqresp)	342.44	29.88	4279.06	336.27
channel (basic)	105.70	89.17	327.86	463.86
channel (reqresp)	50.29	19.92	328.87	224.18

Propagation channel using WCC				
Program	Wikipedia		Wikipedia (P)	
	runtime	message	runtime	message
pregel+ (basic)	16.96	2.85	15.31	0.49
blogel	20.39	1.11	5.10	0.11
channel (basic)	15.67	2.85	15.85	0.49
channel (prop.)	8.64	1.66	3.05	0.17

the runtime of PR and WCC by up to 26% and 35% (using the `CombinedMessage` class), and for PJ (using the `DirectMessage` class) the number is 52%. We believe that the improvement is due to the choice of message interface (in particular the message iterator in `DirectMessage` instead of nested C++ vectors in Pregel+). Nevertheless, we show that our system implementation is reasonably efficient.

B. Effectiveness of Optimized Channels

Here, we evaluate the efficiency of our optimized channels against the message passing channels using the applications that each kind of channel is applicable. In this part, we choose rather simple algorithms, so that we can clearly see how optimized channels can improve the performance in different scenarios.

1) Scatter-Combine Channel: PageRank is a typical graph algorithm that can be optimized by the scatter-combine channel. We test Pregel+'s basic implementation, Pregel+'s *ghost* mode (a.k.a. the mirroring technique [32]), the standard channel version (Fig. 1) and the scatter-combine channel version. For Pregel+'s mirroring technique, we set the threshold to 16 in all cases.

The experiment results are presented in the upper part of Table V. The basic mode of Pregel+ and our standard version are close in both execution time and message size, while the scatter-combine channel achieves a speedup ranging from 3.39x to 3.50x and reduces roughly one third of the message size. Pregel+'s ghost mode use less messages, but the execution time (including the preprocessing time) is not reduced significantly.

Analysis. The improvement on execution time clearly shows the effectiveness of the scatter-combine channel. As explained in subsubsection IV-D1, it can generate the combined messages by a linear scan of the edges, while Pregel+'s basic mode and the `CombinedMessages` have to use hash table or sorting in every superstep. The reduction on total message size is explained by the removal of redundant transmission of vertices' identifiers.

All these three programs use the *receiver-centric* message combining (for high-degree receiver), while Pregel+'s mirroring technique has the *sender-centric* message combining to further reduce the messages. However, such method is computational intensive and the overall computational cost is higher. We show that the computational cost in message processing is a major problem in some algorithms, and our scatter-combine achieves better performance than existing approaches.

2) Request-Respond Channel: We consider the pointer-jumping algorithm (which is also part of the S-V algorithm) as a minimum example that uses the request-respond paradigm. Given a (forest of) rooted tree, each vertex initially knows its parent and tries to find the root of the tree it belongs to. We test Pregel+'s basic implementation, Pregel+'s *reqresp* mode (which is the original implementation of the request-respond paradigm [32]), the standard channel version and the scatter-combine channel version. We use two types of graphs, a randomly generated tree and a chain. Vertices are randomly assigned to workers.

The middle part of Table V summarizes the results on the two graphs. Without the request-respond optimization, the standard implementations in the two systems use exactly the same number of messages, but ours runs 2.10x faster on a chain 1.82x faster on a randomly generated tree. Contrary to our expectation, Pregel+'s *reqresp* mode has a negative effect on the execution time, although the message size indeed decreases. Our implementation of the request-respond paradigm shows reasonable results, which runs faster on a randomly generated tree, and is as good as an ordinary implementation when tree degrades to a chain. Compared to Pregel+'s *reqresp* mode, our implementation constantly reduces the message size by 33%, and achieves a significant performance gain (up to 13.01x) on the Chain.

Analysis. Although sharing the same idea, the implementations of the request-respond paradigm in our system and Pregel+ are different, which we believe is the main reason that makes our implementation better in both runtime and message size. The request-respond channel works better on Tree, since it easily generates high-degree vertices during the computation. For Chain, there is actually no high-degree vertex until the final stage of the algorithm, but it does not compensate the computational overhead in the channel implementation.

We also observe that, in real algorithms like S-V (Section III-C), we are actually dealing with a dynamic forest, where the finding of the root vertex root is fused with the tree merging. In this special case, Pregel+'s *reqresp* mode can still make an improvement (see Table VI). Nevertheless, we verify that our implementation of the request-respond technique is reasonably effective, and is faster than the one in Pregel+.

3) Propagation Channel: We consider the HCC algorithm [11] as a suitable example for using this optimization, which finds the weakly connected component (WCC) of a directed graph. In this experiment, we present both the results on the original Wikipedia graph and the partitioned graph by METIS [12]. We also add the Blogel version here since the block-centric model is applicable [31]. We choose METIS since it requires no additional knowledge of the graph.

The experiment results are presented in the bottom part of Table V. First, the Pregel+ program and a standard channel version in our system are very close in both execution time and message size. The block-centric version in Blogel works slightly worse on the original graph, but achieves roughly 3x faster when the input graph is properly partitioned. Our propagation channel version works consistently better than all other implementations in terms of execution time on both graphs (1.67x faster than Blogel). The number of messages used in the propagation channel version is the same as the Blogel version, but the message size in Blogel is 33% less due to its special treatment of partition information. Nevertheless, running WCC on partitioned graph is not message intensive.

Analysis. A partitioner reduces the communication cost between the workers, but for the standard WCCs (program 1 and 3), it still takes a large number of supersteps to converge, so the execution time is not reduced. Both of Blogel and our propagation channel use a block-level program to speedup the convergence and our system outperforms Blogel slightly.

It is also noteworthy that, WCC's standard implementation is simply an iterative neighborhood communication that needs around 10 lines of code for the `compute()` function. While the Blogel version requires users to additionally write a block-level computation of more than 100 lines of code[7], switching to the propagation channel in our system is much easier. It is clear that our system achieves both conciseness and efficiency compared to the block-centric model.

[7]http://www.cse.cuhk.edu.hk/blogel/code/apps/block/hashmin/block.zip

TABLE VI: Experiment results of the S-V implementations using different combinations of channels.

Program	SubDomain		Twitter	
	runtime	message	runtime	message
1-pregel+ (basic)	411.67	298.62	143.03	115.96
2-pregel+ (reqresp)	174.67	66.17	74.20	29.12
3-channel (basic)	155.36	71.82	59.71	28.53
4-channel (reqresp)	128.96	59.74	53.16	24.86
5-channel (scatter)	75.45	44.69	31.86	18.15
6-channel (both)	51.59	32.60	24.94	14.49

TABLE VII: Experiment results of the Min-Label algorithm

Program	Wikipedia		Wikipedia (P)	
	runtime	message	runtime	message
1-pregel+ (basic)	52.15	9.85	50.51	2.70
2-channel (basic)	61.89	4.98	67.84	1.29
3-channel (prop.)	31.37	4.42	13.96	1.12

C. Combination of Channels

In this part, we verify the multiple performance issues in the S-V (see discussions in Section III-C) by running the programs using different combination of channels in our system. We show that a combination of properly chosen channels can finally lead to much better performance. To cover all the special channels we have, we also present the experiment results of the Min-Label algorithm [33] for finding Strongly Connected Components (SCCs).

1) The S-V Algorithm: According to the previous discussion, the request-respond channel and the scatter-combine channel are applicable in the algorithm implementation. We thus have four S-V programs in our system covering all the combination of the two optimized channels. For comparison, we also give the result of our best-effort implementation in Pregel+'s *basic* and *reqresp* modes.

The results are presented in Table VI. As expected, the basic version (program 3) without using any specialized channel is the slowest, and the fully optimized version (program 6) takes only one third of the execution time. Furthermore, using either of the request-respond channel (program 4) or the scatter-combine channel (program 5) can lead to a decent improvement on both graphs. Pregel+'s basic mode runs extremely slowly, which is mainly due to the inapplicability of the combiner optimization. Then, even with the request-respond paradigm (in which the combiner optimization is enabled), Pregel+ is still slower than our unoptimized version on both graphs.

Analysis. The experiment clearly verifies the multiple performance issues in the S-V implementation. Even with the request-respond optimization, the S-V algorithm still suffers the heavy communication cost, since the redundancies in the neighborhood communication become the major problem. Our system combines all the optimizations and makes the algorithm work consistently well.

2) Min-Label Algorithm: Strongly connected component (SCC) is a fundamental problem in graph theory and it is widely used in practice to reveal the properties of the graphs. A typical Min-Label algorithm [33] for finding SCCs in Pregel is already complex which is an iterative algorithm where the main iteration contains four subroutines, including the removal of trivial SCCs, forward/backward label propagation, SCC recognization and relabeling. The algorithm suffers the problem of low convergence speed.

Our system offers the `Propagation` channel for the forward/backward label propagation, which achieves a 2x speedup on Wikipedia, and a nearly 4x faster on partitioned Wikipedia (see Table VII). This optimization is not possible in any of the existing system.

VI. Related Work

Google's Pregel [19] is the first specific in-memory system for distributed graph processing. It adopts the Bulk-Synchronous Parallel (BSP) model [28] with explicit messages to let users implement graph algorithms in a vertex-centric way. The core design of Pregel has been widely adopted by many open-source frameworks [20], [30], and most of them inherit the monolithic message passing interface, meaning that the messages of different purposes are mixed and indistinguishable for the system. As an attempt for optimizing communication patterns, Pregel+ extends Pregel with additional interfaces (in particular, the *reqresp* and the *ghost* mode), but it is less flexible since the two modes cannot be composed and adding optimizations is inconvenient.

To support intuitive message slicing in Pregel-like systems, Telos [18] proposes a layered architecture where interleaving tasks are implemented as separate *Protocols*, each having a user-defined `compute()` function with a dedicated message buffer. However, it lacks an essential feature for optimization that users cannot modify the implementation of the message buffer. Husky [35] is a general-purpose distributed framework with the channel interface, and it supports primitives like *pull*, *push* and *migrate* and *asynchronous updates* to combine the strength of graph-parallel and machine learning systems. We extend this idea for composing optimizations in graph-parallel system and propose our optimized channels for three common performance issues.

There has been much research studying the optimizations on Pregel-like systems, and our optimized channels draw inspiration from this line of research, such as the sender-side message combining (a.k.a. vertex-replication, mirroring) [2], [3], [15], [22], the request-respond paradigm [32], the block-centric model [26], [27], [31] and so on. In particular, our scatter-combine channel recognizes the static messaging pattern and reduces the computational cost as well as message size by preprocessing, which is novel and turns out to be effective for communication-intensive algorithms like PageRank and S-V. We also demonstrate how complex algorithms like S-V and SCC can be optimized by such technique, while most existing systems only focus on rather simple algorithms.

Apart from Pregel, there are graph-parallel systems that use high-level models to organize the computation and communication, which brings more opportunities for optimization. For example, the Gather-Apply-Scatter (GAS) model (used by GraphLab [16], PowerGraph [10] and PowerLyra [6]) is a typical one that describes a vertex-program by three functions, and the scatter-combine model (used by Graphine [34]) fuses the scatter and gather operations, resulting a more compact two-phase model. Our channel mechanism shares the same spirit; through the channels, we can equip a system with even more abstractions, so that users can choose whatever suitable for their algorithms.

There are also graph systems using a functional interface with high-level primitives to manipulate the entire graph, such as GraphX [29] (a library on top of Apache Spark [36]) and its extension HelP [23]. However, their primitives are hard to compose. Furthermore, experiment results [35] show that they are less efficient than other systems even on simple algorithms like PageRank. Sparse-matrix based frameworks (e.g. the CombBLAS [5] and PEGASUS [11]) are also popular for handling graphs which provide linear algebra primitives, but the lack of graph semantics makes it hard for deep optimizations.

VII. Conclusion

In this paper, we propose to use the channel interface as a replacement of Pregel's message passing and aggregator mechanism, which not only structures the communication in Pregel algorithms in an intuitive way, but also enables different kinds of optimizations to work together for dealing with different performance issues in the same program. We also demonstrate how three optimized channels are implemented in this manner. Experiments show that, our channel-based system along with the current three optimized channels can achieve significantly better performance for a wide spectrum of graph algorithms.

We believe that, having such system with various modular optimizations available can help users quickly build efficient graph applications. In particular, our methodology of "optimizing a program by choosing suitable channels" hopes to change the current situation that different performance issues are addressed by separate systems, so that users can get rid of the steep cost of learning different systems' interfaces or strengths and enjoy using a single all-around system.

As a remaining issue at the current stage, users have to choose the proper channels for their algorithms, which may require some extent of expertise. To further make the system easy to use for non-expert users, in the future, we are interested in studying the compilation from a high-level declarative domain-specific language (such as Palgol [37]) to our system.

Acknowledgment

We thank anonymous reviewers for their insightful comments to improve this paper. This work is partially supported by the Japan Society for the Promotion of Science (JSPS) Grant-in-Aid for Scientific (S) No. 17H06099.

References

[1] Apache Giraph. http://giraph.apache.org/.

[2] Pregel+. http://www.cse.cuhk.edu.hk/pregelplus/.

[3] N. T. Bao and T. Suzumura. Towards highly scalable pregel-based graph processing platform with x10. In *Proceedings of the 22nd International Conference on World Wide Web*, pages 501–508. ACM, 2013.

[4] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. In *Proceedings of the 7th International Conference on World Wide Web*, pages 107–117. Elsevier, 1998.

[5] A. Buluç and J. R. Gilbert. The combinatorial BLAS: Design, implementation, and applications. *The International Journal of High Performance Computing Applications*, 25(4):496–509, 2011.

[6] R. Chen, J. Shi, Y. Chen, and H. Chen. PowerLyra: Differentiated graph computation and partitioning on skewed graphs. In *Proceedings of the Tenth European Conference on Computer Systems*, page 1. ACM, 2015.

[7] Y. Cheng, F. Wang, H. Jiang, Y. Hua, D. Feng, and X. Wang. LCC-Graph: A high-performance graph-processing framework with low communication costs. In *2016 IEEE/ACM 24th International Symposium on Quality of Service (IWQoS)*, pages 1–10. IEEE, 2016.

[8] S. Chung and A. Condon. Parallel implementation of Borůvka's minimum spanning tree algorithm. In *IPPS*, pages 302–308. IEEE, 1996.

[9] H. N. Gabow and R. E. Tarjan. A linear-time algorithm for a special case of disjoint set union. *J. Comput. System Sci.*, 30(2):209–221, 1985.

[10] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. PowerGraph: distributed graph-parallel computation on natural graphs. In *OSDI*, pages 17–30, 2012.

[11] U. Kang, C. E. Tsourakakis, and C. Faloutsos. Pegasus: A peta-scale graph mining system implementation and observations. In *Proceedings of the Ninth IEEE International Conference on Data Mining*, pages 229–238. IEEE, 2009.

[12] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on scientific Computing*, 20(1):359–392, 1998.

[13] 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 *EuroSys*, pages 169–182. ACM, 2013.

[14] F. Khorasani, R. Gupta, and L. N. Bhuyan. Scalable simd-efficient graph processing on gpus. In *Proceedings of the 24th International Conference on Parallel Architectures and Compilation Techniques*, PACT '15, pages 39–50, 2015.

[15] 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. *Proceedings of the VLDB Endowment*, 5(8):716–727, 2012.

[16] 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. *Proceedings of the VLDB Endowment*, 5(8):716–727, 2012.

[17] Y. Lu, J. Cheng, D. Yan, and H. Wu. Large-scale distributed graph computing systems: An experimental evaluation. *Proceedings of the VLDB Endowment*, 8(3):281–292, 2014.

[18] A. Lulli, P. Dazzi, L. Ricci, and E. Carlini. A multi-layer framework for graph processing via overlay composition. In *European Conference on Parallel Processing*, pages 515–527. Springer, 2015.

[19] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *SIGMOD*, pages 135–146. ACM, 2010.

[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 Computing Surveys (CSUR)*, 48(2):25, 2015.

[21] L. Quick, P. Wilkinson, and D. Hardcastle. Using Pregel-like large scale graph processing frameworks for social network analysis. In *ASONAM*, pages 457–463. IEEE, 2012.

[22] S. Salihoglu and J. Widom. GPS: a graph processing system. In *SSDBM*, page 22. ACM, 2013.

[23] S. Salihoglu and J. Widom. HelP: High-level primitives for large-scale graph processing. In *Proceedings of Workshop on GRAph Data management Experiences and Systems*, pages 1–6. ACM, 2014.

[24] S. Salihoglu and J. Widom. Optimizing graph algorithms on Pregel-like systems. *Proceedings of the VLDB Endowment*, 7(7):577–588, 2014.

[25] Y. Shiloach and U. Vishkin. An $O(\log n)$ parallel connectivity algorithm. *Journal of Algorithms*, 3:57–67, 1982.

[26] Y. Simmhan, A. Kumbhare, C. Wickramaarachchi, S. Nagarkar, S. Ravi, C. Raghavendra, and V. Prasanna. Goffish: A sub-graph centric framework for large-scale graph analytics. In *European Conference on Parallel Processing*, pages 451–462. Springer, 2014.

[27] Y. Tian, A. Balmin, S. A. Corsten, S. Tatikonda, and J. McPherson. From think like a vertex to think like a graph. *Proceedings of the VLDB Endowment*, 7(3):193–204, 2013.

[28] L. G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, 1990.

[29] R. S. Xin, J. E. Gonzalez, M. J. Franklin, and I. Stoica. GraphX: A resilient distributed graph system on Spark. In *First International Workshop on Graph Data Management Experiences and Systems*, page 2. ACM, 2013.

[30] D. Yan, Y. Bu, Y. Tian, A. Deshpande, et al. Big graph analytics platforms. *Foundations and Trends® in Databases*, 7(1-2):1–195, 2017.

[31] D. Yan, J. Cheng, Y. Lu, and W. Ng. Blogel: A block-centric framework for distributed computation on real-world graphs. *Proceedings of the VLDB Endowment*, 7(14):1981–1992, 2014.

[32] D. Yan, J. Cheng, Y. Lu, and W. Ng. Effective techniques for message reduction and load balancing in distributed graph computation. In *International World Wide Web Conference*, pages 1307–1317. ACM, 2015.

[33] D. Yan, J. Cheng, K. Xing, Y. Lu, W. Ng, and Y. Bu. Pregel algorithms for graph connectivity problems with performance guarantees. *Proceedings of the VLDB Endowment*, 7(14):1821–1832, 2014.

[34] J. Yan, G. Tan, Z. Mo, and N. Sun. Graphine: programming graph-parallel computation of large natural graphs for multicore clusters. *IEEE Transactions on Parallel and Distributed Systems*, 27(6):1647–1659, 2016.

[35] F. Yang, J. Li, and J. Cheng. Husky: Towards a more efficient and expressive distributed computing framework. *Proceedings of the VLDB Endowment*, 9(5):420–431, 2016.

[36] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. *HotCloud*, 10(10-10):95, 2010.

[37] Y. Zhang, H.-S. Ko, and Z. Hu. Palgol: A high-level DSL for vertex-centric graph processing with remote data access. In *Proceedings of the 15th Asian Symposium on Programming Languages and Systems*, pages 301–320. Springer, 2017.