Noname manuscript No.
(will be inserted by the editor)

HybridGraph: towards I/O-efficient distributed and iterative graph computing by hybrid pushing/pulling

Zhigang Wang[1] · **Yu Gu**[1] · **Yubin Bao**[1] · **Ge Yu**[1,2] · **Jeffrey Xu Yu**[3]

Received: date / Accepted: date

Abstract Graphs are rapidly growing in size in many applications. Most graph algorithms are iterative in nature and generate a large number of messages during computations. Thus, vertex-centric distributed systems usually store graph data and message data on disk to improve scalability. Currently, such systems take a push-based computation mode. This works well if few messages reside on disk. Otherwise, expensive random writes largely degrade performance. By contrast, using the existing pulling mode, messages are individually pulled for each vertex on demand and then consumed immediately. No message is spilled onto disk, potentially yielding a performance improvement when the number of messages is large. However, the random and frequent access to vertices is costly.

This paper proposes a hybrid solution to dynamically and adaptively switch modes between pushing and pulling, so as to obtain the optimal performance in different scenarios. We first use a new block-centric technique to pull messages I/O-efficiently, while, however, the iterative computation is still vertex-centric for easy use. I/O costs of data accesses are shifted from the message receiver side where messages are randomly written in pushing to the sender side where graph data are randomly read in pulling. Hence,graph data are organized by clustering vertices and edges to achieve high read-throughput. Second, for different graph algorithms, we design two seamless switching mechanisms with respective performance prediction methods to decide how and when to switch modes. Third, by fully utilizing the switching mechanism, a lightweight fault-tolerant framework is developed to strike a good balance between recovery efficiency and failure-free performance. We finally conduct extensive performance studies to confirm the effectiveness of our proposals over state-of-the-art solutions using a broad spectrum of real-world graphs.

Keywords I/O-Efficient · Distributed Graph Computing · Pushing · Pulling · Fault-tolerance

Ge Yu (✉)
yuge@cse.neu.edu.cn

Zhigang Wang
wangzhiganglab@gmail.com

Yu Gu
guyu@cse.neu.edu.cn

Yubin Bao
baoyubin@cse.neu.edu.cn

Jeffrey Xu Yu
yu@se.cuhk.edu.hk

[1]School of Computer Science and Engineering, Northeastern University, Shenyang, China
[2]Key Laboratory of Medical Image Computing of Ministry of Education, Northeastern University, Shenyang, China
[3]Department of Systems Engineering and Engineering Management, The Chinese University of Hong Kong, Hong Kong, China

1 Introduction

In order to handle iterative analysis jobs over large graphs, many systems have been developed. *Pregel* [21] by Google as one of the early distributed systems employs a Master-Slave framework (Fig. 1), where Master divides a job into several tasks running on computational nodes (Slaves) in a cluster. Typically, tasks load graph data from distributed file systems, partition data among themselves, and then start computation in parallel through a set of iterations, called "supersteps". Workloads in one superstep consist of computing each vertex (vertex-centric) and exchanging intermediate results (messages). One computational node is referred to as a sender/receiver side when the task on it is sending/receiving messages. Without loss of generality, we as-

sume that each node runs only one task. We then discuss communication among nodes, instead of tasks. Supersteps are separated by global barriers to coordinate the progress.

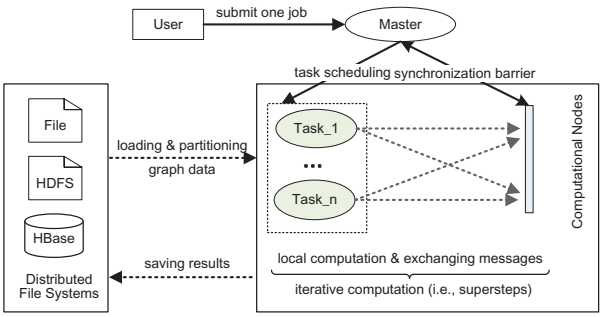
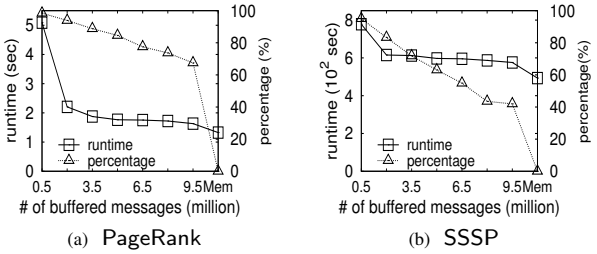


Figure 1 Illustration of iterative graph processing

Motivation: *Pregel* has been driving much of the research on enhancing its performance, including graph partitioning [12,31], communication [12,26,29], and convergence [33, 43]. This paper focuses on I/O-efficient graph analysis on a cluster, as the memory resource can be easily exhausted due to two factors. One is the drastically growing rate of real graph volumes, e.g., there are over 4.75 billion content items shared on Facebook daily. The other is the increased message scale in proportion to the graph volume. Adding new physical nodes can alleviate memory pressure, but is not always feasible. Besides, external storage provided by today's public cloud platforms is significantly cheaper (by as much as 187x on Amazon EC2) than memory.

We now demonstrate that writing/reading messages dominates the runtime of a disk-based system. Take *Giraph* [1], an open-source *Pregel* implementation, as an example. We compute PageRank (10 supersteps) and single source shortest path (SSSP, 284 supersteps) [21] over a web graph wiki with 5.7 million vertices and 130 million edges (disk storage size of 1GB) on a cluster of 5 nodes. *Giraph* needs 10 times more memory than the original graph size to guarantee that the runtime will not increase rapidly due to I/O costs of messages and resource contention. As an indication of how *Giraph* scales when messages reside on disk, Fig. 2 shows the runtime and the percentage of disk-resident messages. By varying the number of buffered messages from Mem (all are buffered) down to 0.5 million on a node, the percentage increases from 0% to 98%. The computing time increases from 130 to 510 seconds for PageRank, and from 490 to 780 seconds for SSSP. The issue we investigate thereby is to find an I/O-efficient method to process messages.

Problem analysis: Almost all existing distributed systems [1,26,2], like *Giraph*, take a pushing mode where all messages from one source vertex (svertex) as a whole are generated at the sender side and then pushed to destination vertices (dvertices) at the receiver side. Pushing is efficient if



(a) PageRank (b) SSSP

Figure 2 Impact of the number of disk-resident messages

most messages are kept in memory because a svertex is accessed only once per superstep. However, it is most likely that receivers need to store messages on disk, since data volume can be large. That is I/O-inefficient as shown in Fig. 2, because messages generated in the source-centric way exhibit the poor temporal locality among dvertices, leading to costly random writes. Another preferred mode is to pull messages from svertices on demand when dvertices are updating themselves. Messages at the receiver side can be consumed immediately after being generated, to avoid possible writes and reads. However, one svertex may be read multiple times in a random manner if it is the neighbor of several dvertices. Since the graph volume is rapidly growing, graph data usually reside on disk. Thus, the cost of random reads is still considerable, and may be larger than that of random writes in pushing, especially when the number of messages is decreasing. In fact, existing distributed pull-based systems [38,12] are designed for memory-based computations. No optimizations are designed for speeding up I/O access.

Challenges: Clearly, pushing and pulling are suitable for different scenarios, which is determined by the number of messages. However, neither of them works best in all cases. That motivates us to combine them and design a hybrid solution which supports switching between them adaptively. However, this is a non-trivial task. The reasons are twofold. First, the current pulling mode pulls desired messages for each dvertex individually, which is I/O-inefficient due to frequent and random access to svertices. Thus, directly combining them is not cost effective. Second, in order to gain optimal performance, some important issues such as the proper switching time and effective performance prediction model need to be explored, which is especially difficult when processing graph algorithms with different behaviors.

Our contributions: By solving the two challenges mentioned above, we propose a hybrid solution to I/O-efficiently handle disk-based graph computations.

First, we design a new Block-centric Pulling mode called BPull to optimize the cost of random reads. Specifically, dvertices in one block can send a block identifier to pull messages to be generated and sent for them, instead of sending requests for each one individually. BPull shifts I/O costs

of data accesses from messages to vertices. In a push-based system, messages received are from any computational nodes and are for different dvertices. Their distribution might be random and not have patterns to follow. It is very difficult to cluster messages without high overhead. The similar problem also occurs for existing pull-based methods (Pull), but data accessed are svertices. However, in BPull, we can cluster svertices and edges based on the requested block. Then a svertex will be read only once to generate messages for dvertices in one block, reducing the number of random reads.

Second, we propose a seamless switching mechanism by first decoupling existing pushing mode called Push and our BPull, and then reorganizing the decoupled functions reasonably, to efficiently switch the two modes. Further, seeking a proper switching point depends on accurately predicting the runtimes of Push and BPull, and then comparing them to select an optimal mode. We observe that graph algorithms fall into two categories based on whether or not the runtime per superstep gradually changes. For *gradual-change* algorithms, a performance metric is given to estimate the runtime difference at the t-th superstep through deeply analyzing the communication and I/O costs. We then simply use the comparison result to approximate that at the (t+1)-th superstep, so as to guide the mode selection. This is called a Basic Hybrid-Switch (BHS) solution.

However, the simple heuristic approximation rule used in BHS is not suitable for *sudden-change* algorithms as the runtime can suddenly and frequently change, even in neighboring supersteps. Instead, we propose to directly compute the performance metric of superstep-(t+1). Towards this end, we design a different but stronger prediction model to capture the real-time performance variation at the expense of providing offline knowledge and online statistics. Also, instead of directly combining Push and BPull, we introduce their variants with additional barriers, to correctly collect online statistics. Accordingly, a new switching mechanism is presented and can duly react to the mode selection, compared with that in BHS. Then a Generalized Hybrid-Switch (GHS) solution is given.

Finally, BHS, as well as GHS, enables an efficient fault-tolerant method. Existing techniques [21,37] can quickly recover failures, but write a large amount of messages in the *failure-free* execution. They expose our hybrid solution to the same inefficiency of Push that the former is trying to address. Our theoretical analysis and experiments reveal that BPull can also provide competitive recovery efficiency with nearly-zero *failure-free* performance degradation. We achieve this goal by confining pull requests to dvertices on failed tasks only. Surviving tasks do nothing but react to such requests. The behaviors of vertices are thereby different. Instead of logging heavyweight messages, we now log some lightweight metadata to restore the pulling context. Note that a missing superstep may be run with pushing before failures, but pulling in recovery. Fortunately,the hybrid solution allows us to freely select the mode as required.

The major contributions are summarized as below.

- We propose a new block-centric pulling mode (BPull) to pull messages I/O-efficiently. Also, a new data structure is designed to separate vertices and edges into blocks so as to improve the read-throughput.
- We present a basic hybrid solution (BHS) for *gradual-change* algorithms by switching Push and BPull adaptively. Due to the prominent performance prediction, it obtains optimal performance in different scenarios.
- We further generalize our hybrid solution for *sudden-change* algorithms (GHS). A suite of new polices is designed to accurately capture real-time performance variation based on the offline knowledge and online statistics, and then duly switch modes.
- We introduce a lightweight yet effective fault-tolerant method for hybrid solutions. It naturally supports fast confined recovery computations, while incurring negligible *failure-free* performance penalty.
- The resulting prototype system *HybridGraph* exposes uniform APIs to users for easily programming various graph algorithms. It hides the details of predicting performance and switching modes from users. Some system optimizations are also integrated into *HybridGraph*.
- We conduct extensive experiments over various algorithms, to confirm the effectiveness of our proposals. In comparison with up-to-date push- and pull-based systems, *HybridGraph* achieves up to 35x speedup.

This paper extends a preliminary work [36] in the following aspects. First, we detailedly analyze the boundary of BHS and then propose a generalized variant. Second, an efficient lightweight fault-tolerant method is designed. Third, we introduce simple yet expressive APIs, as well as system optimizations, to make *HybridGraph* easy and efficient to use. Fourth, to show the advantage and generality of *HybridGraph*, more graph algorithms like Maximal Independent Sets [23] and Bipartite Matching [21], are tested.

The remainder of this paper is organized as follows. Section 2 provides necessary background on existing Push and Pull. Section 3 presents our novel BPull. Sections 4 and 5 further design the basic and the generalized hybrid solutions. Section 6 elaborates the efficient fault-tolerant method. Section 7 outlines the system design issues of *HybridGraph*. Section 8 contains a thorough experimental study. Section 9 overviews related works. Section 10 concludes this paper.

2 Overview of Push and Pull

We model a graph as a directed graph $G = (V, E)$, where V is a set of vertices and E is a set of edges (pairs of vertices). For an edge (u, v), u is the source vertex denoted as svertex,

and v is the destination vertex denoted as dvertex. The in-neighbors of u is a set of vertices that have an edge linking to u, and its out-neighbors is a set of vertices that u has an edge to link. The in/out-degree of u is the number of its in/out-neighbors. We state that the memory resource is limited if it cannot store messages entirely. In this case, the limited memory resource is supposed to be allocated for more I/O-inefficient messages in a high priority instead of graph data [45]. This paper thereby assumes graph data (vertices and edges) reside on disk. In the following, we briefly introduce Push and Pull, and then analyze their performance.

Push: We use *Giraph* [1] as an example system. In one superstep, every vertex u in V is selectively scheduled to compute a user-defined function in parallel. We denote this function as compute(). For simplicity, let $M_I(u)$ and $M_O(u)$ denote the messages received and sent to/from u. The compute is shown in Eq. (1).

$$\mathsf{compute}(u^t, M_I^t(u)) \rightarrow (u^{t+1}, M_O^{t+1}(u)) \tag{1}$$

An implementation of PageRank is given in Fig. 3(a). In the t-th superstep, first, u gets a message iterator msgs of $M_I^t(u)$ which keeps messages received from in-neighbors in the (t-1)-th superstep, and then initializes a sum to zero (Lines 2-3). Second, after summing message values (Lines 4-5), the vertex value is updated from u^t to u^{t+1} (Lines 6-8). Third, u sends the new updated value divided by its out-degree as messages $M_O^{t+1}(u)$ to all its out-neighbors (Lines 9-11). M_O^{t+1} for all vertices are the basis to form M_I^{t+1} in superstep-(t+1). Finally, u will *vote to halt* to terminate computations if the maximum number of supersteps, maxNum, has been reached. We call this Push for the following two reasons. First, svertices send messages voluntarily. Second, messages have already been available on local memory/disk when they are required by dvertices to execute compute().

```
1  Vertex.compute () {
2    Messages msgs = getRecMsg();
3    double sum = 0.0;
4    for (Message m: msgs.iterator())
5      sum = sum + m.getValue();
6    double newVal =
7      0.15/getNumVertices() + 0.85*sum;
8    setValue(newVal);
9    for (Edge e: getOutEdges().iterator())
10     sendMsgTo(e.Target(),
11               getValue() / getOutDegree());
12   if (getSuperStep() > maxNum)
13     voteToHalt();
14 }
```

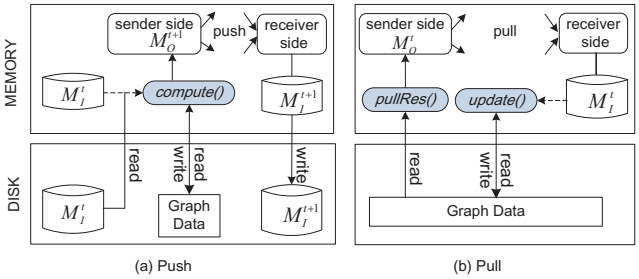
(a) PageRank in Push

```
1  Vertex.pullRes (dvertex v) {
2    if (isRespond() is True)
3      get the edge e=(u, v); //u is svertex
4        sendMsgTo(e.Target(),
5                  getValue() / getOutDegree());
6  }
7  Vertex.update () {
8    Messages msgs = getRecMsg();
9    double sum = 0.0;
10   for (Message m: msgs.iterator())
11     sum = sum + m.getValue();
12   double newVal =
13     0.15/getNumVertices() + 0.85*sum;
14   setValue(newVal);
15   if (getSuperStep() > maxNum)
16     voteToHalt();
17   else
18     voteToRespond();
19 }
```

(b) PageRank in Pull

Figure 3 Algorithm implementation in Push and Pull

Fig. 4(a) gives a brief introduction to the data flow for a typical superstep t on one computational node. In *Giraph*, $M_I^t(u)$ and $M_O^{t+1}(u)$ possibly are stored on disk when the memory is not enough to hold all of them.



(a) Push (b) Pull

Figure 4 Data flow in Push and Pull at superstep-t

Pull: It decouples compute() into two functions, namely, pullRes() and update(), as shown in Eq. (2) and Eq. (3). Compared with Push, Pull postpones the operations of generating and exchanging messages M_O^{t+1} until the (t+1)-th superstep. We explain decoupled functions at superstep-t using PageRank in Fig. 3(b). The pullRes function is called to read svertex u, and then respond the pull request from dvertex v for generating desired messages in $M_O^t(u)$/$M_I^t(v)$ (Lines 1-6). The update function is similar to the compute function in pushing but no message is produced. It updates v's value using the pulled messages in $M_I^t(v)$. In particular, at superstep-(t-1), a vertex u indicates that it will send messages to its out-neighbors by calling a voteToRespond function (Line 18). This signal is kept at u. On demand, in the t-th superstep, u will respond pull requests if the signal is true (Line 2). We outline the implementation of Pull in Fig. 4(b). Although existing pull-based systems assume that graph data reside in memory, it is most likely that we manage them on external storage for large graphs.

$$\mathsf{pullRes}(u^t) \rightarrow M_O^t(u) \tag{2}$$

$$\mathsf{update}(v^t, M_I^t(v)) \rightarrow v^{t+1} \tag{3}$$

Now we informally introduce two definitions regarding vertices in both Push and Pull, that is, ***Active Vertex*** and ***Responding Vertex***. A vertex u is an *active vertex* if and only if it is processed by compute() in Push or update() in Pull. Similarly, u is a *responding vertex* if and only if it produces messages in compute() or pullRes(). In particular, the inactive vertex becomes active again when receiving messages.

Performance analysis: We analyze the total cost C in computing, for Push and Pull, as shown in Eq. (4). $\mathcal{N}$ represents the number of supersteps. In one superstep, we use C_{cpu}, C_{net}, and C_{io}, to represent the computation cost, communication cost, and I/O cost, respectively.

$$C = (C_{cpu} + C_{net} + C_{io}) \times \mathcal{N} \tag{4}$$

For Pull, the computing workload, as well as $\mathcal{N}$, is the same as that in Push. Consequently, the difference regarding C_{cpu} and $\mathcal{N}$ can be ignored. Now we find that only C_{net} and C_{io} dominate the performance difference between Push and Pull. For simplicity, we directly use network bytes and I/O bytes as the values of C_{net} and C_{io}.

From the C_{net} perspective, when the number of messages is large, the cost of sending pulling requests, can be offset by great communication gains due to concatenating/combining messages (see Section 3.2). For C_{io}, in Push, messages in M_O^i are carried across two consecutive supersteps and may end up being kept on disk if the message volume exceeds the allocated memory resource, which incurs high I/O costs. This problem is solved in Pull because messages produced in pullRes() are consumed immediately by update() in the same superstep. Nevertheless, when graph data reside on disk, Pull will frequently and randomly access svertices to respond requests, which usually costs more than accessing disk-resident messages as reported in Section 8.2.

In conclusion, first, when processing a large number of messages using limited memory resources, Pull always outperforms Push if the I/O cost of accessing svertices can be optimized. We make this condition satisfied by designing a new block-centric pulling mode (BPull) in Section 3. BPull also optimizes C_{net}. Second, the decreased number of messages inevitably narrows the gap between Push and BPull in terms of C_{net} and C_{io}, even making Push beat BPull. Thus, we design a hybrid solution in Sections 4 and 5, to adaptively choose the profitable mode for efficiency.

3 Block-centric pulling mode

This section describes our Block-centric Pulling mode, BPull, by discussing three key issues. First, we present a data structure called VE-BLOCK for sending messages. It stores a graph on disk by dividing vertices and edges into different blocks. Second, we give the details on how to pull messages in block-centric using VE-BLOCK. Third, we discuss how to determine the proper number of blocks to enhance the efficiency. Table 1 summarizes the frequent notations used in the paper.

3.1 Efficient graph storage VE-BLOCK

The VE-BLOCK data structure consists of two components: Vblocks and Eblocks. The main purpose behind VE-BLOCK is to improve the efficiency when pulling messages while allowing fast access to update vertex values. Here, consider an adjacency list to represent a graph in which for every vertex it keeps a quadruple $(id, val, |V_o|, V_o)$, where we denote by id and val the id and value of one vertex, respectively. V_o is a list of out-neighbors, and $|V_o|$ is the out-degree. In VE-BLOCK, suppose that there are $\mathcal{V}$ fixed-sized Vblocks, b_1, b_2, ..., $b_{\mathcal{V}}$, in total. We simply range-partition all vertices into $\mathcal{V}$ blocks based on the vertex ids, since graph partitioning is an NP-hard problem.[1] A Vblock keeps a list of triples $(id, val, |V_o|)$. Given b_i, we have $\mathcal{V}$ variable-sized Eblocks, g_{i1}, g_{i2}, ..., $g_{i\mathcal{V}}$, to maintain outgoing edges from any svertex in b_i. In particular g_{ij} maintains any edge (u, v) for u in b_i and v in b_j, for $1 \leq j \leq \mathcal{V}$. Furthermore, in g_{ij}, edges from the same svertex u are clustered in a fragment. The svertex id id and an integer indicating the number of clustered edges, are the auxiliary data of a fragment.

[1] Any partitioning method can be used by reordering vertices.

Table 1 List of frequent notations

Notation	Meaning
T_i	A computational node (task).
$\mathcal{T}$	Number of computational nodes.
$\mathcal{V}$	Number of vertex blocks (Vblocks).
b_i, X_i	The i-th Vblock b_i and its metadata X_i.
g_{ij}	An edge block (Eblock) with edges from b_i to b_j.
V_{act}^t/V_{res}^t	Active/responding vertices at superstep t.
$E^t/\mathcal{E}^t$	Edges read from disk by Push/BPull at superstep t.
f	Number of fragments in all Eblocks.
$\mathcal{M}$	Number of messages produced at one superstep.
M_{disk}	Messages resident on disk in Push at one superstep.
C_{io}(Push)/ C_{io}(BPull)	Number of bytes of disk-resident data accessed by Push/BPull at one superstep.
$\mathsf{BR}_i/\mathsf{BS}_i$	Message receiving/sending buffer on T_i in BPull.
B_i, B	Message receiving buffer on T_i in Push, B=$\sum B_i$.
$\mathsf{B}_\perp$	Lower bound of B making C_{io}(Push)<C_{io}(BPull).
$Q^t/\mathbb{Q}^t$	Performance metric in BHS/GHS at superstep-t.
s_{rr}/s_{rw}	Random read/write throughput of disk (MB/s).
s_{sr}	Sequential read throughput of disk (MB/s).
s_{net}	Network throughput (MB/s).

Accordingly, when a svertex needs to update its value, only a Vblock b_i it belongs to is accessed. On the other hand, messages are pulled based on b_i, which means that all vertices as dvertices in b_i will pull messages from svertices to update themselves. In this way, the cost of pull requests is minimized to a Vblock identifier. When a computational node receives the pull request, i.e., the Vblock id of b_i, it only needs to access Eblocks g_{ji} and Vblocks b_j, for $1 \leq j \leq \mathcal{V}$, to respond the request.

In order to further improve the efficiency, we also maintain ***Metadata***, including X_j for Vblock b_j, and Y_{ji} for Eblock g_{ji}. Here, X_j keeps five items: the number of svertices in b_j (#), the total in-degree (*ind*) and out-degree (*outd*) of svertices, a bitmap x_j, and a Vblock responding indicator (*res*). In particular, the i-th bit in x_j is set if there exist edges directed from vertices in b_j to vertices in b_i. *res* is "true" if at least one svertex needs to respond pull requests, i.e., sending messages. Y_{ji} records the disk file size of g_{ji} and the numbers of edges, fragments, svertices, and dvertices.

Fig. 5 shows the VE-BLOCK for an example graph with 5 vertices and 6 edges. Vertices are partitioned into three Vblocks kept by two computational nodes. On node T_1, $b_1 = \{v_1, v_2\}$ and $b_2 = \{v_3, v_4\}$; On node T_2, $b_3 = \{v_5\}$. Edges are distributed into Eblocks accordingly. For example, b_2 is with Eblocks, g_{21}, g_{22}, and g_{23}. Edge (v_3, v_2) is assigned into g_{21} because v_3 belongs to b_2 and v_2 belongs to b_1. The

bitmap in X_2 (111) indicates that vertices in b_2 have out-neighbors in the total three Eblocks g_{21}-g_{23}. Further, take g_{21} as an example. There exist two edges with two svertices $\{v_3, v_4\}$ and one dvertex $\{v_2\}$, as shown in Y_{21}.

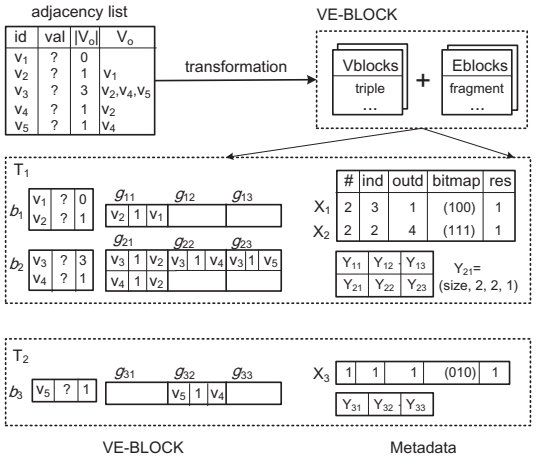


Figure 5 The VE-BLOCK structure

3.2 Pull requesting and pull responding

Two components are used to pull messages. The pull requesting operation (Pull-Request) is performed at the computational node T_x that requests messages to be consumed, and the pull responding operation (Pull-Respond) is performed at another node T_y that sends messages on demand.

We discuss Pull-Request in Algorithm 1. In a superstep, each computational node T_x will invoke Pull-Request to request messages for every Vblock b_i held by it from any computational node T_y. Here, T_x works as the receiver. All messages received for vertices in b_i are kept in the message receiving buffer BR_x. Also, an ***Active Bitmap*** is associated with vertices to indicate whether or not a specific vertex is active. A bit in *Active Bitmap* is set if the corresponding vertex originally is active or receives new messages. Like the *Active Bitmap*, a ***Responding Bitmap*** is used and the bit is set if the associated svertex is a *responding vertex*. *Active Bitmap* is also updated after invoking update(), if some vertex becomes inactive by voting to halt. Obviously, pulling messages is done in Vblocks, namely, the block-centric pulling mechanism. While, vertices are still updated in the vertex-centric manner.

In the same superstep, T_y needs to react to pull requests, as the sender. This is done by Pull-Respond (Algorithm 2). Assume T_y receives a pull request for Vblock b_i from T_x. It will check for every Vblock b_j that T_y holds, to see if there are any messages among vertices in b_j that need to be sent to vertices in b_i. Towards this end, the meta-information X_j is used. T_y first checks the Vblock responding indicator *res* in X_j. The result, if true, will make T_y further check the i-th bit in the bitmap x_j. If the i-th bit is on, Eblock g_{ji} is required. Then pullRes() is called for every *responding vertex* u as svertex in b_j, to generate messages for all dvertices in b_i. This is different from the traditional vertex-centric Pull where pullRes() returns messages for one dvertex only.

Algorithm 1: Pull-Request

```
1  foreach Vblock b_i ∈ VE-BLOCK on T_x do
2      foreach each computational node T_y do
3          send a pull request for Vblock b_i to T_y;
4          insert messages received from T_y into a buffer BR_x;
5          concatenate or combine messages in BR_x;
6          update Active Bitmap;
7      foreach vertex u in Vblock b_i do
8          u.update() if u is active;
9          update Responding Bitmap by invoking
           voteToRespond();
10         update Active Bitmap by invoking voteToHalt();
```

Algorithm 2: Pull-Respond

```
   Input : Block id i of the requested Vblock b_i
1  foreach each Vblock's metadata X_j on T_y do
2      if X_j.res is 1 and the i-th bit in X_j's bitmap is 1 then
3          foreach each fragment in Eblock g_ji do
4              if u.isRespond() is true then
5                  //u is the svertex of the fragment
6                  insert u.pullRes() into a sending buffer BS_y;
7                  concatenate or combine messages in BS_y;
8      return messages for b_i
```

At the sender side, suppose that several svertices generate messages to a dvertex v. In Algorithm 2, these message values can be concatenated to share the same id of v. The id of v is thereby transferred only once, which reduces communication costs. In addition, if the message value is commutative and associative [21], multiple values are supposed to be combined into a single one, namely, the Combiner, to further improve the performance. Similarly, in Algorithm 1, messages received at the receiver side also can be concatenated or combined to save the memory space.

3.3 Determining the number of Vblocks

In VE-BLOCK, all vertices of a graph are range-partitioned into $\mathcal{V}$ fix-sized Vblocks, and all edges are stored in $\mathcal{V} \times \mathcal{V}$ variable-sized Eblocks. The number of Vblocks, $\mathcal{V}$, becomes

critical to determine the efficiency. We discuss it from two perspectives, the memory usage and the I/O cost.

Memory usage: A computational node T_i holds two buffers for messages: BR_i for receiving messages (in Pull-Request), and BS_i for sending messages (in Pull-Respond). Note that the memory for *Metadata*, *Active Bitmap*, and *Responding Bitmap*, is negligible. Suppose T_i keeps V_i vertices and outgoing edges from V_i. *Giraph* (Push) uses B_i as the maximum number of messages in memory on T_i[2], i.e., available memory resources. Given B_i, we then analyze how to calculate a reasonable Vblock granularity $\mathcal{V}_i$ for T_i based on the size of BS_i and BR_i, and the total number of Vblocks $\mathcal{V}=\sum \mathcal{V}_i$. For simplicity, we use n_i $(= |V_i|)$ to estimate $\mathcal{V}_i$.

Suppose there exist $\mathcal{T}$ computational nodes. BS_i is divided into $\mathcal{T}$ sub-buffers, as $\mathcal{T}$ computational nodes may send pull requests to T_i simultaneously. BS_i is inversely proportional to $\mathcal{V}$, because a large $\mathcal{V}$ can decrease the number of vertices in a Vblock, and then decrease the number of messages in the sub-buffer on T_i.

Messages are sent to T_i by $\mathcal{T}$ nodes in parallel, and may not be put into BR_i in time. To handle this case at the receiver side, we have two options which are separately setting a sub-buffer for each node, like BS_i, and controlling the data-flow during sending messages. The space complexity of the former is $O(\mathcal{T}\cdot\mathsf{BR}_i)$. The memory usage increases by ($\mathcal{T}$-1) times, compared with BR_i. Thus, we follow the latter idea in this paper. Once the sender starts the sending operation, messages will be sent to T_i in packages one by one. The new package will not be delivered until the old one has been handled by T_i (i.e., messages have been put into BR_i and concatenated/combined). Like BS_i, BR_i is inversely proportional to $\mathcal{V}_i$, as the decreased number of vertices in a Vblock can reduce the number of messages received.

For algorithms that support the Combiner, when a pull request is received by T_i, messages in a sub-buffer will not be sent until all messages from T_i are produced. The goal is to thoroughly combine messages to achieve high communication gains. Obviously, the maximum number of messages after being combined for one Vblock is equal to that of vertices as dvertices, $\frac{n_i}{\mathcal{V}_i}$. Thus, $\mathsf{BS}_i = \frac{n_i}{\mathcal{V}_i}\mathcal{T}$. $\mathsf{BR}_i = 2\frac{n_i}{\mathcal{V}_i}$, because we pre-pull messages for b_{i+1} when vertices in b_i are updating their values, to reduce the blocking time of pulling messages. Finally, we set $\mathcal{V}_i$ using Eq. (5).

For algorithms that only support concatenating, buffering all messages increases the memory usage since their values cannot be combined. Thus, at the sender side, we immediately flush out messages in each sub-buffer if the message scale exceeds a given sending threshold, like *Giraph*. BS_i is thereby negligible. At the receiver side, the number of message values received is determined by the sum of the in-degree per vertex in one Vblock. The pre-pulling optimization is disabled due to the increased memory usage. In this scenario, we set $\mathcal{V}_i$ using Eq. (6).

$$\mathcal{V}_i = \frac{2n_i + n_i\mathcal{T}}{\mathsf{B}_i} \tag{5}$$

$$\mathcal{V}_i = \frac{\sum_{u\in V_i} \textit{in-degree}(u)}{\mathsf{B}_i} \tag{6}$$

I/O costs: It is worth noting that the main I/O cost is shifted from the receiver side in Push to the sender side. Pull-Respond needs I/O costs to respond pull requests as shown in Algorithm 2. Recall that we can possibly arrange edges in each Eblock g_{ij} using fragments such that edges with the same svertex are clustered together, which significantly improves the locality. Nevertheless, for each fragment, we still need I/Os to access the fragment's auxiliary data and the corresponding svertex value in Vblock. Theorem 1[3] tells us that the totally expected number of fragments is proportional to $\mathcal{V}$. Because disk I/Os in Pull-Request are independent of $\mathcal{V}$, the total I/O costs of BPull are proportional to $\mathcal{V}$.

Theorem 1 *$\forall u \in V$, the expected number of its* fragments *is proportional to $\mathcal{V}$.*

Following the analysis, we set $\mathcal{V}$ as small as possible on the prerequisite of providing sufficient memory for (BR_i+BS_i).

4 Adaptive hybrid solution

This section first gives a performance analysis of Push and BPull, and then describes a Basic Hybrid-Switch (BHS) solution to combine them to obtain optimal performance for different scenarios.

4.1 Performance analysis

Suppose that there exist four vertices, s_1, s_2, d_1, and d_2, which are distributed on two computational nodes T_1 (s_1 and s_2), and T_2 (d_1 and d_2), as shown in Fig. 6. As discussed in Section 2, we analyze C_{net} and C_{io}, the main factors affecting the performance.

Communication costs: Most push-based systems such as *Giraph* [1] and *GPS* [26] do not concatenate/combine messages at the sender side T_1, because the poor locality of messages among dvertices limits the communication gain. The gain usually cannot offset costs of concatenating/combining. By contrast, in pull-based systems, messages are generated for requested svertices on demand. The improved locality

[2] B_i indicates the message receiving buffer size. We ignore the space of the sending buffer because *Giraph* immediately flushes out messages at the sender side once a sending threshold is satisfied.

[3] All proofs appear in the appendix.

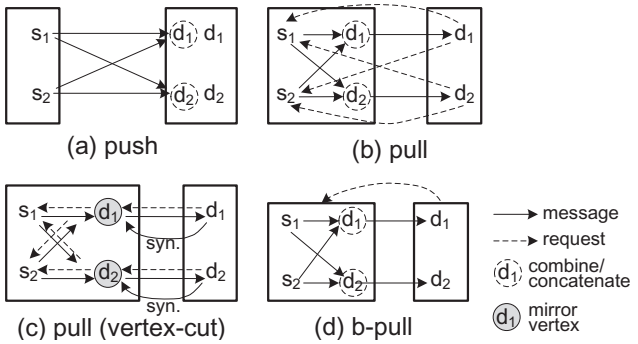


Figure 6 Comparisons of existing Push, Pull, Pull (vertex-cut), and the new BPull (block-centric)

makes concatenating/combining cost effective. However, pull requests incur additional communication costs (Fig. 6 (b)). In the worst case, pull requests will be sent up to $(|V|\mathcal{T})$ times if we do not combine requests due to the poor locality among svertices. *GraphLab PowerGraph* employs a vertex-cut mechanism for efficiency and handles the computation by a Gather-Apply-Scatter pulling mode. As shown in Fig. 6(c), d_1 is cut into a master in T_2 and a mirror in T_1. Pull requests and messages are transferred between the master and mirrors in Scatter and Gather, respectively. Note that extra messages are used to synchronize mirror values in Apply. Obviously, the number of mirrors dominates the communication cost which is proportional to $|V|$ and increases with $\mathcal{T}$ sub-linearly [12]. Finally, for BPull, the upper bound of the number of requests is $\mathcal{V}\mathcal{T} \ll |V|$. As shown in Fig. 6(d), suppose d_1 and d_2 are assigned into the same Vblock. The pull request will be sent only once. However, the number of messages is dominated by the data placement policy, and may be more than that in Pull with vertex-cut because *GraphLab PowerGraph* designs many sophisticated policies to optimize this problem. Hence, for C_{net}, BPull beats Push and Pull, and can offer a comparable performance to Pull with vertex-cut.

I/O costs: The cost of reading svertices in existing pull-based systems is considerable, because sending pull requests for each dvertex individually leads to frequent and random disk accesses (as shown in Table 4 and Fig. 14, in Section 8). However, BPull can decrease the upper bound of the number of I/O requests from $|E|$ in Pull to the number of fragments in VE-BLOCK. Thus, in the following, we only need to analyze C_{io} for Push and BPull.

C_{io} of a complete superstep is the I/O cost of producing messages and then updating vertices. Let $\mathcal{M}$ be the number of messages produced at the (t-1)-th superstep in Push or at the t-th superstep in BPull. Then E^{t-1} and $\mathcal{E}^t$ stand for the set of disk-resident edges used by Push and BPull, respectively. The former is decided by how many vertices are active (V_{act}^{t-1}), since an *active vertex* will update itself and possibly send messages along edges in compute(). The latter depends on the number of *responding vertices* (V_{res}^t), because only a *responding vertex* will read edges to produce messages. After completing the t-th superstep, we show the I/O costs of Push in Eq. (7) and BPull in Eq. (8), and then compare them in Theorem 2. Here, M_{disk} is the set of messages resident on disk in Push, and F^t is the set of fragments covering all edges in $\mathcal{E}^t$. We denote by $IO(.)$ the number of bytes of the given data. Specifically, $2IO(M_{disk})$ is the total number of write and read bytes regarding messages. $IO(F^t)$/$IO(V_{rr}^t)$ denotes the I/O cost of fragments' auxiliary data (F^t)/svertices values in Vblocks (i.e., V_{rr}^t) read by Pull-Respond. $IO(V^t)$ indicates the cost of updating vertex values, which is the same for both BPull and Push. Note that $IO(E^{t-1})$ depends on the specific implementation of Push [1,45]. Without loss of generality, we only consider the implementation in *Giraph* [1], and others can also be used in our hybrid solution.

$$C_{io}(\mathsf{Push}) = IO(V^t) + IO(E^{t-1}) + 2IO(M_{disk}) \quad (7)$$

$$C_{io}(\mathsf{BPull}) = IO(V^t) + IO(\mathcal{E}^t) + IO(F^t) + IO(V_{rr}^t) \quad (8)$$

The size of M_{disk}, $|M_{disk}|$, is equal to $(\mathcal{M}-B)$, if $\mathcal{M} > B = \sum_{i=1}^{\mathcal{T}} B_i$. Otherwise, it is 0 ($M_{disk}$=$\emptyset$). Let f be the number of fragments in Eblocks. Theorem 2 describes the sufficient condition of using BPull. That is, $(|E|-f)$ is B's lower bound, namely $\mathsf{B}_\perp$, which makes Push beat BPull in terms of I/O bytes. $|E|$ and f are available after building VE-BLOCK. Accordingly, we can decide whether or not to use BPull before starting to run iterative computations.

Theorem 2 *Assume that each vertex in $V^t = V$ should broadcast messages to all of its neighbors at the t-th superstep. If $B \leq (|E| - f)$, then $C_{\text{io}}(\mathsf{Push}) \geq C_{\text{io}}(\mathsf{BPull})$.*

Theorem 2 only guarantees the effectiveness if each vertex sends messages to all its out-neighbors at every superstep, such as PageRank. For other algorithms, like SSSP, the number of vertices that send messages, varies with iterations, i.e., $\mathcal{M}$ is not constant. In this scenario, the I/O comparison result is non-deterministic, even though $B \leq \mathsf{B}_\perp$.

Favorite scenarios: Pull with vertex-cut is the up-to-date existing pull-based approach, but its performance is seriously degraded when graph data reside on disk, due to frequent svertex accesses. Push also suffers from the performance degradation which is mainly caused by writing messages randomly and disabling combining/concatenating messages. By contrast, our BPull avoids disk I/O costs incurred by messages and reduces the cost of accessing svertices, while simultaneously offering a comparable communication efficiency to Pull with vertex-cut. That makes BPull outperform Pull with vertex-cut in disk scenarios. BPull also beats Push in most cases, but the decrease of $\mathcal{M}$ narrows the gap, even making Push beat BPull. This is because BPull will

pay extra costs $IO(F^t)$ and $IO(V_{rr}^t)$, compared with Push. On the other hand, the I/O and communication cost of Push is roughly proportional to $\mathcal{M}$. Not surprisingly, for SSSP-like algorithms, with the change of $\mathcal{M}$ during iterations, Push and BPull have different favorite scenarios, and using a single one can hardly achieve optimal performance.

4.2 Switching between Push and BPull

To support the switching operation between Push and BPull, BHS must accommodate them from two perspectives: the computing functions for implementing algorithms, and the data storage for consistent and efficient data accesses.

Computing functions: BPull naturally decouples compute() into pullRes() and update(). Let $\prod$ denote the concatenating or combining operation. The workload of a pull-based superstep is to first pull messages for one Vblock b_i, and then update each vertex v in b_i, until all Vblocks are done, as shown in Eq. (9). Differently, compute() in Push is separated into three functions: loadM(), update(), and pushRes(). loadM() loads messages received at the previous superstep for Vblock b_i. After that, each vertex u is first updated in update(), and then immediately broadcasts new messages in pushRes(). Eq. (10) shows the process mathematically.

The decoupling of compute() supports a seamless switching by sharing update(). As shown in Fig. 7, when switching from BPull to Push, we first invoke pullRes() and update() to update vertex values, and then immediately call pushRes() based on new values. Conversely, when switching from Push to BPull, loadM() and update() are invoked to update vertex values which will be used by pullRes() at the next superstep. Particularly, no matter which mode is run, *Active* and *Responding Bitmaps* are updated, so as to correctly update vertices and produce messages after switching operations.

$$\begin{cases} \prod\limits_{u \in V_{res}^t} \mathsf{pullRes}(u^t) \rightarrow M_I^t(b_i) \\ v \in b_i \wedge v \in V_{act}^t, \mathsf{update}(v^t, M_I^t(v)) \rightarrow v^{t+1} \end{cases} \tag{9}$$

$$\begin{cases} \prod\limits_{u \in V} \mathsf{loadM}(M_O^t(u)) \rightarrow M_I^t(b_i) \\ v \in b_i \wedge v \in V_{act}^t, \mathsf{update}(v^t, M_I^t(v)) \rightarrow v^{t+1} \textit{ and} \\ \qquad\qquad \mathsf{pushRes}(v^{t+1}) \rightarrow M_O^{t+1}(v) \end{cases} \tag{10}$$

Data storage: The shared update() makes Push and BPull can share vertex values, i.e., Vblocks in VE-BLOCK. Particularly, although pulling and pushing messages are done in a single superstep when switching from BPull to Push, update() is invoked only once for each vertex, which avoids reading/writing conflicts. On the other hand, Eblocks cannot support efficient edge accesses for Push, because all edges of u are required in pushRes(), but they are maintained in different Eblock files. We thereby store edges twice, organized in Eblocks for BPull and the adjacency list for Push, respectively. Moreover, edges in the latter are partitioned based on the distribution of svertices among Vblocks, so as to avoid redundant reads if some Vblocks are not processed.

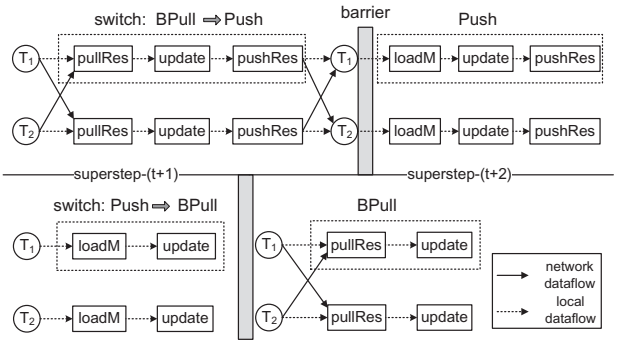


Figure 7 Switching between Push and BPull based on Q^t

4.3 Switching time

The key to gain optimal performance in BHS is deciding the right switching time, which is the focus of this section.

As we all know, many researchers have explored how to accurately predict metrics (e.g., the number of messages and active vertices) in iterations. Among them, Shang and Yu [28] present that metrics collected by the current superstep can be used to predict those of the remaining supersteps. They confirm the effectiveness for multiple algorithms over real graphs. This paper also adopts their method but the difference is that we design a metric which can characterize the performance of Push and our BPull.

Performance metric Q: The overall performance of Push and BPull is mainly dominated by the communication cost C_{net} and I/O cost C_{io}. Thus, after the t-th superstep, we use $\mathcal{M}_{co}$ (the number of concatenated or combined messages across network in BPull) and C_{io} to estimate the performance. For C_{net}, we assume that $Byte_m$ is the size of one destination vertex id if messages are concatenated, or a whole message if messages are combined. $\mathcal{M}_{co}Byte_m$ thereby denotes the extra communication volume of Push, compared with BPull. For C_{io}, recall that $IO(M_{disk})$ is the number of random-write/sequential-read bytes incurred by writing/reading messages in Push. Thus, the difference of Push and BPull is $\big(IO(E^{t-1})+IO(M_{disk})-IO(\mathcal{E}^t)-IO(F^t)\big)$ in terms of the number of sequential read bytes. On the other hand, $IO(V_{rr}^t)$ stands for the number of random read bytes incurred by pulling messages. Finally, we give a metric Q^t to evaluate the performance difference between Push

and BPull at superstep-t in Eq. (11). Here, s_{rr}/s_{rw}/s_{sr} and s_{net} stand for the random-read/random-write/sequential-read throughput, and the network throughput, respectively. Apparently, BPull has superior performance if $Q^t \geq 0$.

$$Q^t = \frac{\mathcal{M}_{co}Byte_m}{s_{net}} + \frac{IO(M_{disk})}{s_{rw}} - \frac{IO(V_{rr}^t)}{s_{rr}} + \frac{IO(E^{t-1}) + IO(M_{disk}) - IO(\mathcal{E}^t) - IO(F^t)}{s_{sr}} \quad (11)$$

Lightweight online performance prediction: Q^t is available only after the t-th superstep. Hence, it cannot affect the execution of the current superstep. However, we can use Q^t as the comparison result of Push and BPull at superstep-(t+Δt) in the near future, to select an efficient mode. Δt is the switching interval and $\Delta t \geq 1$. Now we compute Q^t by C_{io}(Push), C_{io}(BPull), and $\mathcal{M}_{co}$. Because either Push or BPull is run at a single superstep, some of such statistics can be directly collected, while others must be estimated.

Particularly, when actually running Push, we will know the set of *responding vertices*, and which Vblock has *responding vertices* (i.e., X_j.*res*=1 in Fig. 5). We then figure out the set of required Eblocks if BPull is run. Accordingly, the number of required edges, as well as fragments and svertices, is available. C_{io}(BPull) can be estimated. In addition, $\mathcal{M}_{co}$ is equal to $\mathcal{M}R_{co}$, where R_{co} is the concatenating/combining ratio in the most recent BPull execution. Note, that if BPull has not yet been run, $\mathcal{M} = |E|$ under the assumption that messages are broadcasted along all edges. In this case, the number of messages after being concatenated/combined is the sum of the numbers of dvertices in all Eblocks, and then R_{co} is known. In contrast, for estimation of Push, M_{disk} is inferred by comparing $\mathcal{M}$ against memory capacity. Further, after estimating how many edges will be read from disk based on the set of *active vertices*, we can compute C_{io}(Push). For both modes, collecting statistics and computing Q can be performed at the global barrier. That means the prediction process, as a lightweight component, can be easily integrated into existing *Pregel*-like systems without performance degradation.

Switching interval: As reported by Shang et al. [28], the accuracy of prediction is proportional to $\frac{1}{\Delta t}$, $\Delta t \geq 1$. Thus, for Δt, the smaller, the better. However, the prediction method with $\Delta t = 1$ may not work well at some supersteps. First, no message is produced when switching from Push to BPull as shown in Fig. 7. The key parameter $\mathcal{M}$ is not available and hence we fail to compute Q^t. Second, the switching operation in BHS essentially schedules decomposed functions between neighboring supersteps using different policies. Theoretically, it is cost-free since no extra computation or communication behaviors are incurred. However, when switching from BPull to Push, pullRes() and pushRes() are run in a single superstep. The potential resource contention still causes a slight performance loss. Considering both factors mentioned above, our compromised solution is setting Δt as 2. For example, if $Q^t < 0$ is detected at the current t-th superstep, both Q^{t+1} and Q^{t+2} are assumed to be negative. Thus, Push will be run in the two subsequent supersteps.

4.4 Execution of BHS

Now, we present the execution of BHS in Algorithm 3. The graph loading operation includes two tasks (Line 1): 1) storing graph data, including organizing vertices in Vblocks and storing edges twice, and 2) computing $B_\perp$ used in Theorem 2. Based on Theorem 2, we initialize the mode used in the previous superstep (*preM*) and in the current superstep (*curM*), respectively. Specifically, the initial value is BPull if $B \leq B_\perp$, and Push, otherwise (Lines 2-3). During iterations, the switching operation (Lines 15-17) is triggered when the selected mode is changed (Line 6). Otherwise, runPull() or runPush() is invoked (Lines 7-10). Meanwhile, statistics are collected so as to determine which mode is supposed to be run at the next superstep (Line 13). Note, that if $\Delta t < 2$, we skip the switching request (Line 6).

Algorithm 3: BHS-Execution

```
Input : Input graph G, buffer size B, and the maximum
        number of supersteps: maxNum
1  loading G, and computing B⊥;
2  preM ← initMode(B,B⊥);
3  curM ← preM;
4  Δt ← 0;
5  for t = 1 to maxNum do
6      if preM equals curM , or Δt < 2 then
7          if curM is BPull then
8              [M_co,C_io(Push),C_io(BPull )] ← runPull();
9          else
10             [M_co,C_io(Push),C_io(BPull )] ← runPush();
11         Δt ← Δt + 1;
12         preM ← curM;
13         curM ← evaluate(M_co,C_io(Push),C_io(BPull ));
14     else
15         runSwitch(preM, curM);
16         preM ← curM;
17         Δt ← 1;
```

5 Generalized hybrid solution

In the previous section, we have discussed the basic hybrid solution BHS. However, we observe that it may not be a good choice for some other important algorithms. The main problem is the low accuracy of its lightweight prediction

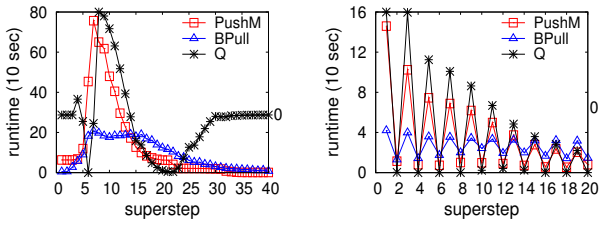
model. We now provide another alternative to Generalize the Hybrid-Switch solution (GHS).

5.1 Boundary of BHS

The prediction model in Section 4.3 is lightweight, but imposes a limitation on BHS. Based on our observations, BHS only works well for the category of algorithms where runtimes of the two modes, Push and BPull, gradually change in neighboring supersteps. The ***Gradual-Change*** property implies the mode that currently yields the best speedup is usually the optimal choice in the near future. Hence, it is reasonable to approximate the runtime difference of Push and BPull, i.e., $Q^{t+\Delta t}$, using Q^t. Many algorithms fall into this category, like SSSP, Random Walk, Connected Components, and PageRank (runtime per superstep is invariant).

In contrast, for ***Sudden-Change*** algorithms, the runtime per superstep suddenly and frequently changes during the whole iterations. The difference between Q^t and $Q^{t+\Delta t}$ can be large, even though $\Delta t = 1$. As a result, the prediction accuracy of approximating $Q^{t+\Delta t}$ using Q^t is prohibitively low, which may guide BHS to select a suboptimal mode. Such algorithms include Maximal Independent Sets (MIS) [23] or Maximal Cliques [34], Vertex Coloring [23], Bipartite Matching [21], and Edge Dominating Sets [42].

For better understanding, we now illustrate the different runtime change patterns by running two representative algorithms, SSSP and MIS. We conduct testing over a fri graph with 65M vertices and (3.7)1.8B (un)directed edges, using the Amazon EC2 cluster. Fig. 8 shows results of the up-to-date push-based method (PushM[4] [45]) and our BPull. For more details about the algorithm description and experiment settings, please refer to Section 8.1.



(a) SSSP on directed graph (b) MIS on undirected graph

Figure 8 Illustration of runtime change (fri, *amazon*). Left y-axis indicates runtime per superstep; Right y-axis indicates the value of Q

Switching for SSSP: For both modes, we observe a gradual runtime change after reaching the peak at superstep-6. When detecting $Q < 0$ at superstep-14, we can assume that the comparison result holds true in the near future, and so it does. Thus, the performance can be enhanced by switching from BPull to Push at superstep-15. Note that before runtime increases to its peak, the switching function is disabled. We do this for two reasons. First, the sudden increased runtime makes Q frequently change between positive and negative values, and then prevents BHS from boosting performance — because the current optimal mode becomes suboptimal later. Second, an algorithm typically takes only a few supersteps to reach the runtime peak, that is, the switching function still works in most supersteps. In fact, the sudden runtime increase is caused by the factor that more and more vertices are processed after receiving messages from in-neighbors. Large-scale graphs generally have the property of low-diameter, that is, the average distance between two vertices is short. Hence, messages can be quickly spread to involve the majority of vertices in computations.

Switching for MIS: For MIS, runtimes of both modes always show the sudden and frequent change. BPull outperforms Push ($Q > 0$) at some supersteps, such as 1, 3, and 5. While, at other supersteps like 2, 4, and 6, Push beats BPull ($Q < 0$). Switching modes at the right time can significantly drop the overall runtime, but it is very difficult. For example, following BHS, the predicted Q^2 value is positive because $Q^1 > 0$, but actually $Q^2 < 0$. Then we always select the suboptimal mode BPull to run supersteps 2, 4, 6, and so on. For SSSP, to avoid the wrong selection, we can simply disable switching operations when detecting the increased runtime. It, however, is not feasible for MIS because runtime is increasing at many supersteps and then a large amount of optimization opportunities are lost. Shang et al. in [28] analyze the runtime-change cycle and then consider supersteps in a cycle as a whole. Similarly, runtimes of supersteps within a cycle cannot be optimized.

5.2 Predicting performance for *Sudden-Change* algorithms

Below, we discuss whether it is possible to effectively predict performance for *Sudden-Change* algorithms. At a given superstep-t, because $Q^{t+1} \approx Q^t$ has a low prediction accuracy, a straightforward alternative is to directly estimate Q^{t+1}. The estimation must be accomplished before processing the majority of workloads in superstep-(t+1), so that we can duly switch to the optimal mode. We will show that it is feasible by modifying the iterative framework in some way.

What factors do essentially affect Q^{t+1}? Shown in Eq. (11), both the communication gain of BPull ($\mathcal{M}_{co}Byte_m$) and the message I/O cost of Push ($IO(M_{disk})$) directly depend on how many messages in M_O^{t+1}/M_I^{t+1} are produced for updating vertices at the (t+1)-th superstep. The size of M_I^{t+1}, $\mathcal{M}$, can be obtained by summing $h^{t+1}(u)$, the number of messages produced by every *responding vertex* u. Let V_{res}^{t+1}

[4] We fail to run Push for MIS on fri. PushM is also a compared method in our experiments. It outperforms Push, especially when the message volume is large, but requires that messages are commutative.

denote the set of such *responding vertices*. Eq. (12) mathematically shows how to compute $\mathcal{M}$. Meanwhile, like the lightweight online predication model in Section 4.3, for BPull, we can infer $IO(\mathcal{E}^{t+1})$), as well as $IO(F^{t+1})$ and $IO(V_{rr}^{t+1})$, based on V_{res}^{t+1}. Also, given *active vertices* V_{act}^{t}, $IO(E^t)$ in Push is available. To sum up, Q^{t+1} can be inferred by V_{act}^{t}, V_{res}^{t+1} and $h^{t+1}(u)$.

$$\mathcal{M} = \sum_{u \in V_{res}^{t+1}} h^{t+1}(u) \tag{12}$$

When to compute Q^{t+1}? Q^{t+1} measures the runtime difference of Push and BPull by analyzing the cost of message related operations. The measurement effectiveness has been confirmed in Fig. 8. That means pushing/pulling messages in M_O^{t+1}/M_I^{t+1} is the majority of workloads of superstep-(t+1). Hence, if Q^{t+1} is known before that, we have opportunities to improve performance. Note that pushing messages actually is performed at superstep-t, but logically, it is still a big part of workloads of superstep-(t+1) because these messages are used at that superstep.

Computing Q^{t+1} by online statistics and offline knowledge. The former includes *Active Bitmap* and *Responding Bitmap*, which are updated online (see Section 3.2) and can be used to form V_{act}^{t} and V_{res}^{t+1}. The latter is the user-defined $h^{t+1}(u)$ value. $h^{t+1}(u)$ is specific to the message generation logic which has already been given by users in Push or BPull. Thus, the switching details are still hidden, and users only focus on implementing algorithms.

For BPull, the correct, complete *Active Bitmap* and *Responding Bitmap* are naturally provided at the end of the t-th superstep. The system can duly compute Q^{t+1} before superstep-(t+1). However, it is difficult for Push. The traditional implementation of Push, like *Giraph*, typically invokes pushRes() to push messages in M_O^{t+1} right after updating one vertex in update(). By reading the newly updated vertex value in memory instead of disk, high throughput is achieved. This, however, challenges the computation of Q^{t+1} since the two required bitmaps are not available until all messages are pushed, i.e., most workloads of superstep-(t+1) have been done.

Now we show another push-based implementation where update() and pushRes() are executed in two different phases, to correctly collect online statistics. As shown in Algorithm 4, at superstep-t, the variant consists of a *Vertex-updating* Phase and a *Message-pushing* Phase. The former updates vertices, as well as *active bitmap* and *responding bitmap*, based on messages from the previous superstep (Lines 3-8), and then writes vertices onto disk (Line 9). The latter reloads responding vertices to generate and push messages (Lines 12-16). Since vertices have been processed in the first phase, the correct, complete V_{act}^{t} and V_{res}^{t+1} are available after that (Line 10).

Algorithm 4: TwoPhase-Push

```
1  //Vertex-updating Phase;
2  foreach Vblock b_i ∈ VE-BLOCK do
3      ∏_{u∈V} loadM(M_O^t(u)) → M_I^t(b_i);
4      update active-bitmap[t] and V_act^t;
5      foreach v ∈ b_i ∧ v ∈ V_act^t do
6          update(v^t, M_I^t(v)) → v^{t+1};
7          update active-bitmap[t+1] and V_act^{t+1};
8          update responding-bitmap[t+1] and V_res^{t+1};
9          write v^{t+1} onto local disk;
10 form complete V_act^t and V_res^{t+1};
11 //Message-pushing Phase;
12 foreach Vblock b_i ∈ VE-BLOCK do
13     if X_j.res is 1 then
14         foreach v ∈ b_i ∧ v ∈ V_res^{t+1} do
15             loadV (v^{t+1});
16             pushRes(v^{t+1}) → M_O^{t+1}(v);
```

Compared with Push (shown in Eq. (10)), the two-phase variant needs to reload responding vertices from disk, incurring the additional cost $\frac{IO(V_{res}^{t+1})}{s_{sr}}$. On the other hand, the number of edges read in the second phase depends on *responding vertices* V_{res}^{t+1}, like BPull, instead of V_{act}^{t}. Then the I/O cost of reading edges is $IO(\mathcal{E}^{t+1})$, rather than $IO(E^t)$. Observing that a vertex processed in update() may not send messages, $V_{act}^{t} \supseteq V_{res}^{t+1}$ and hence $IO(E^t) \geq IO(\mathcal{E}^{t+1})$. The new performance metric $\mathbb{Q}^{t+1}$ is then given in Eq. (13).

$$\mathbb{Q}^{t+1} = Q^{t+1} + \frac{IO(V_{res}^{t+1}) + IO(\mathcal{E}^{t+1}) - IO(E^t)}{s_{sr}} \tag{13}$$

5.3 Switching modes

We design a *stop-to-select* policy to accomplish work of computing $\mathbb{Q}^{t+1}$ and switching modes, at the t-th superstep. The main idea is to divide one superstep into two *mini*-supersteps separated by a *mini*-barrier. The workload of the first *mini*-superstep is to update vertex values and required bitmaps. At the *mini*-barrier, all tasks stop to wait for $\mathbb{Q}^{t+1}$. Because update() is shared by both modes, even though it has been run at the first *mini*-superstep, we still can switch modes in the second *mini*-superstep based on $\mathbb{Q}^{t+1}$ to optimize the message operations. The two-phase variant of Push of course supports the new switching policy, where the two phases correspond to the two *mini*-supersteps, and the *mini*-barrier is initiated when forming V_{act}^{t} and V_{res}^{t+1} (Line 10 in Algorithm 4). Similarly, for seamless switching operations, we also propose the two-phase variant of BPull but nothing is done in the second phase since messages will be pulled at the next superstep.

Fig. 9 demonstrates the switching process. At the first *mini*-superstep, the required messages in M_I^t are collected

based on the mode used in the previous superstep. Specifically, the system invokes pullRes() for BPull or loadM() for Push. Differently, at the second *mini*-superstep, the newly selected mode guides how to produce messages in M_O^{t+1}/M_I^{t+1}. That is, new vertex values are reloaded by loadV() and then pushRes() outputs messages, if Push is a preferred mode at superstep-(t+1). Otherwise, nothing is done. Hence, the *stop-to-select* policy can duly react to the switching decision, as the decision is given before the majority of workloads of superstep-(t+1) are processed. Instead, for BHS in Fig. 7, the decision is available at the end of superstep-t. In this case, messages used in the next superstep are still pulled/pushed in the old mode. The delayed-response switching operation is acceptable when runtime gradually changes, but cannot be tolerated for *sudden-change* algorithms because of fleeting optimization opportunities.

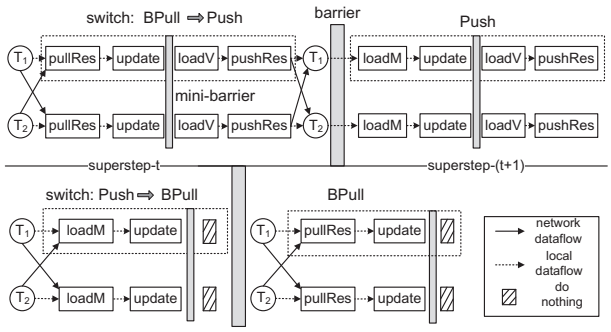


Figure 9 Switching modes between two-phase variants of Push and BPull based on $\mathbb{Q}$

5.4 BHS versus GHS: which is better?

BHS is lightweight, user-friendly, but only suitable for algorithms with the *gradual-change* property. GHS removes this limitation at the expense of adding *mini*-barriers, providing offline knowledge, and possibly increasing I/O costs. Besides, in GHS, pullRes() and pushRes() will not be invoked at the same *mini*-superstep, reducing potential resource contention. Together with the *stop-to-select* policy, GHS supports for frequently and duly switching operations.

Users can explicitly choose the preferred solution when programming algorithms. Based on our observations, a simple heuristic rule is that, if the behaviors of updating vertices and producing messages are independent of specific superstep numbers, the runtime usually gradually changes during iterations. For example, PageRank always updates vertices with the summation operation and broadcasts messages along all edges. SSSP constantly minimizes the path distance and then broadcasts messages only if a shorter path is found. Such algorithms can be efficiently supported by BHS. Otherwise, the generalized variant may be better.

6 Lightweight fault-tolerant framework

Today's graph processing systems typically tolerate failures by checkpointing [21,38], i.e., periodically making a checkpoint of data after completing a superstep. Any failure can be recovered by rolling back computations to the most recent checkpoint. Further, to avoid expensive rollback operations on surviving tasks, researchers propose to additionally log outgoing messages per superstep [21]. Upon failures, recovery computations are confined to failed/restarted tasks, while, surviving tasks send messages for recovery by local logs. However, messages accumulated over supersteps usually end up being kept on disk due to the large size, that slows down the *failure-free* (no failure happens) execution.

Our BHS as well as its variant GHS supports different message processing manners. It provides opportunities for designing a new lightweight fault-tolerant framework that can strike a good balance between recovery efficiency and *failure-free* performance. Below, we first outline our new framework and then analyze its effectiveness in theory.

6.1 Pull-confined failure recovery

BPull naturally supports a pull-confined recovery mechanism. The basic idea is that, in the presence of failures, pull requests are confined to dvertices on failed/restarted tasks. Then vertices on such restarted tasks can correctly recompute lost workloads based on messages pulled from themselves and vertices on surviving tasks. On the other hand, vertices on surviving tasks do nothing but react to confined pull requests. Thus, vertex behaviors are very different. Because BPull allows messages to be efficiently pulled on demand of dvertices, it is unnecessary to log outgoing messages in the *failure-free* execution.

To correctly re-generate required messages, all surviving tasks need to restore the context of responding pull requests using three kinds of data: svertices in Vblocks, edges in Eblocks, and the *Responding Bitmap* at the given recovery superstep. Among them, the read-only edges have already been written onto local disks at the very beginning of iterations. Vertex values have been archived per superstep. Both of the two parts are available. However, *Responding Bitmap* is dynamically updated and resides in memory. Thus, we must log it per superstep.

We next use the example graph in Fig. 5, to demonstrate how to recover failures (shown in Fig. 10). Assume that T_2 fails at superstep-(t+3). That means the value of vertex v_5 in Vblock b_3 is lost. Restarted T_2 recomputes missing supersteps (t+1) and (t+2), from the most recent checkpoint at superstep-t. During recomputations, T_2 will pull messages, if any, from itself and the surviving task T_1. On T_1 the pull context is built based on the Vblock id contained in the pull request. Thus, only Eblocks with edges linking to dvertices

on restarted tasks are actually read, while others are skipped. In this example, the id is b_3. Then v_3 and Eblock g_{23} are involved in the context due to the presence of edge (v_3, v_5).

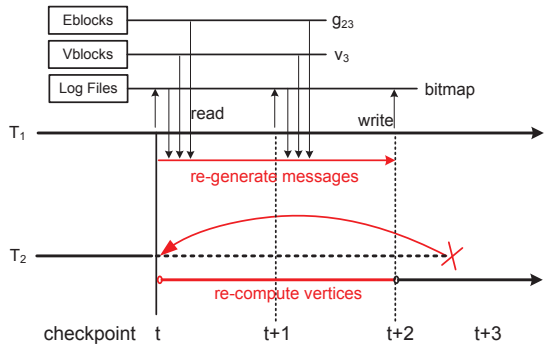


Figure 10 Illustration of the pull-confined failure recovery. To reduce clutter, read/write operations related to VE-BLOCK during the *failure-free* execution are omitted

Pull-confined recovery can be performed at any superstep by invoking decoupling functions in BHS or GHS, even though Push is originally run before failures. Algorithm 5 shows failure recovery on any task T_x before running the failed superstep t_f. At the very beginning of recovery, all of restarted tasks first load required checkpoint files (Lines 1-2). Then BPull is run continuously until the superstep right before failures (Lines 3-8). At such superstep, BPull is normally run but additionally, new messages in $M_O^{t_f}$ are pushed if superstep-t_f runs Push, i.e., switching modes from BPull to Push (Lines 9-10).

Algorithm 5: Failure-Recovery on the task T_x

Input : Checkpointing superstep t_{cp}, failed superstep t_f, selected mode at the failed superstep *fMode*

```
1  if T_x is a restarted task then
2      load checkpoint file archived at the end of superstep-t_cp;
3  for t = (t_cp+1) to (t_f-1) do
4      if T_x is a restarted task then
5          runPull();
6      else
7          restore pull context;
8          react to pull requests by invoking Algorithm 2;
9  if fMode is Push then
10     push messages in M_O^{t_f};
```

6.2 Performance analysis

We next formally compare pull-confined recovery (CpPull) against log-based recovery (CpLog). Before that, we first introduce some important symbols used in our analysis. At any superstep, given a task, C_{lgr} and C_{lgm} denote the costs of logging *Responding Bitmap* and outgoing messages, respectively. Correspondingly, C_{ldr} and C_{ldm} are the loading costs. C_{prm} means how much time a task takes to produce messages. Since checkpointing is used for both methods, we omit it in comparison. Further, some operations on restarted tasks are also the same for both, like transmitting required messages, updating vertices, and producing new messages for restarted tasks. We ignore them for brevity.

Assume that $\mathcal{T}_F$ of $\mathcal{T}$ tasks/nodes fail at superstep-(t+1). At superstep-t, we now define the overall cost of a fault-tolerant framework as the sum of data-logging cost and failure recovery cost. The former consists of two parts. First, we have logged the bitmap or messages before the current failure happens. Second, for recovering possible failures later, a restarted task also logs its bitmap or messages in recovery. That means, in CpLog, messages for $(\mathcal{T} - \mathcal{T}_F)$ surviving tasks are also produced in recovery, although such messages are not sent. The data-logging cost thereby equals $2C_{lgr}$ or $\big(2C_{lgm} + (1 - \frac{\mathcal{T}_F}{\mathcal{T}})C_{prm}\big)$. The recovery cost is dominated by getting messages required for $\mathcal{T}_F$ restarted tasks. For CpPull, such messages are produced based on re-loaded *Responding Bitmap*. Hence, the cost is equal to $\frac{\mathcal{T}_F}{\mathcal{T}}(C_{ldr} + C_{prm})$. While, for CpLog, a surviving task can directly read logged messages with cost $\frac{\mathcal{T}_F}{\mathcal{T}}C_{ldm}$. Finally, Eq. (14) and Eq. (15) mathematically show the costs of CpLog and CpPull.

$$C(\mathsf{CpLog}) = 2C_{lgm} + \frac{\mathcal{T}_F}{\mathcal{T}}C_{ldm} + (1 - \frac{\mathcal{T}_F}{\mathcal{T}})C_{prm} \tag{14}$$

$$C(\mathsf{CpPull}) = 2C_{lgr} + \frac{\mathcal{T}_F}{\mathcal{T}}(C_{ldr} + C_{prm}) \tag{15}$$

Logging *Responding Bitmap* is nearly cost-free, compared with logging messages. This is because *Responding Bitmap* associates only a bit for each vertex. Consider the largest graph used in our experiments in Table 3, uk with 106 million vertices. The bitmap requires only 12.6 MB of disk space. We thereby ignore C_{lgr} and C_{ldr}.

Let $C(\mathsf{CpLog}) \geq C(\mathsf{CpPull})$, we can infer the sufficient condition that makes CpPull outperform CpLog, as shown in Eq. (16). That is, the number of tasks failing at the same superstep must be limited. A straightforward explanation is that in CpPull, the message re-generation cost increases with the number of failed tasks, and then gradually offsets the logging cost in CpLog.

$$\mathcal{T}_F \leq \frac{C_{prm} + 2C_{lgm}}{2C_{prm} - C_{ldm}} \cdot \mathcal{T} = \lambda\mathcal{T}, \;\; s.t. \;\; 0 \leq \mathcal{T}_F < \mathcal{T} \tag{16}$$

Because sequential reads are faster than sequential writes, we have $0 < C_{ldm} < C_{lgm}$, and hence $\lambda > 50\%$. Then, if $\mathcal{T}_F \leq \frac{1}{2}\mathcal{T}$, we can guarantee that CpPull beats CpLog. In real-world distributed environments, generally, the probability that several computational nodes fail at the same time is

extremely low. Thus, CpPull is a lightweight (no message is logged), yet practical fault-tolerant framework, as validated in our experiments in Section 8.6.

7 HybridGraph: a hybrid-based system

We have developed a prototype system called *HybridGraph*[5] to support our novel techniques. This section clarifies important implementation details.

7.1 System design

Fig. 11 gives the overview architecture of *HybridGraph*. Like most systems, *HybridGraph* also employs a Master-Slave framework, consisting of a master node and multiple computational nodes. The master node is in charge of computational nodes through four components. 1) Task Scheduler divides a user-submitted job into several tasks (T_x) and schedules one or more tasks to an available node. 2) Synchronous Controller coordinates the progress of tasks via a global barrier. 3) Restorer detects failures during iterations and guides the recovery process. 4) Finally, given a job, Selector always smartly chooses a preferred mode between Push and BPull.

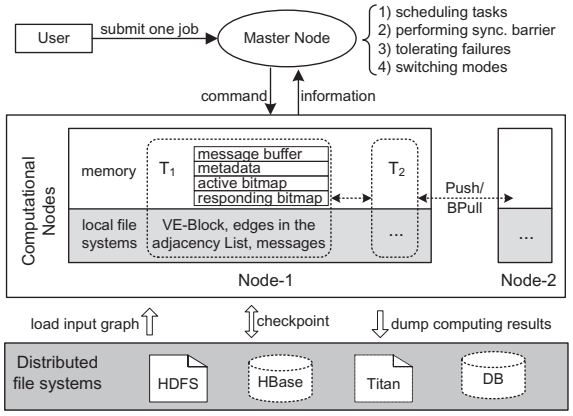


Figure 11 *HybridGraph* Architecture

HybridGraph organizes runtime data in a three-level storage hierarchy to achieve a good balance between scalability and efficiency. First, existing distributed file systems such as HDFS, are directly used to provide high availability for accessing input graph, output results, and possible checkpoints. Supporting more systems, like HBase [3] and Titan [6], is under development. Second, once iterations start, graph data as well as a few of messages under Push reside on local disks for scalable data management. Third, the limited memory resources are used to keep I/O-inefficient messages and other frequently accessed metadata and bitmaps.

[5] The source code is available at https://github.com/HybridGraph.

7.2 APIs

We now illustrate the programming APIs shown in Fig. 12. The *update* and *generateMessages* functions tell *HybridGraph* about how to update a vertex value based on old messages and then generate new messages. All programs must implement them. In particular, *generateMessages*() plays the role of pullRes() in BPull and pushRes() in Push. The difference of the two modes is how many edges are provided. In BPull, only edges linking to dvertices in some Vblock are available, while, in Push, all edges are provided. The framework will choose the correct parameter as the input automatically, according to the selected mode. That hides details from users. Besides, a vertex becomes inactive/responding by explicitly invoking *voteToHalt*()/ *voteToResponding*() in *update*().

```
//update a vertex value based on incoming messages
Void update(Vertex v, MessageIterator msgs);
//generate outgoing messages
MessageIterator generateMessages(Vertex v, EdgeIterator edges);
//a vertex will generate messages by voting to respond
Void voteToRespond();
//a vertex becomes inactive by voting to halt
Void voteToHalt();
//get the current superstep number
Void getSuperStep();
//optional, enable the generalized hybrid solution
Void useMiniBarrier();
//optional, specify how many messages will be generated
Integer h(Vertex v, Integer degree);
//optional, concatenate or combine messages
Message accumulate(Message old, Message new);
```

Figure 12 APIs provided by *HybridGraph*

Before submitting a job, by *useMiniBarrier*(), *HybridGraph* will run GHS. In this case, usually, the h function needs to be specified. By default, it returns the degree of a vertex as the number of messages.

7.3 Optimizations

HybridGraph contains two built-in optimizations.

Static data sharing: In real applications, an input graph can be loaded from HDFS repeatedly to run different jobs. Typically, the graph topology is static. That means edges in Eblocks (BPull) and the adjacency list (Push) can be preserved on local disks and shared among jobs, even though these jobs are submitted at different time points. Jobs except the first one only initialize algorithm-specific data in Vblocks. The cost of loading edges thereby can be amortized. Towards this end, Task Scheduler is modified to be aware of the distribution of static data among nodes.

Lazy edge loading: We observe that for some algorithms over undirected graph, like Bipartite Matching [21], at some supersteps, vertex u sends back new outgoing messages to

selected neighbors with ids contained by incoming messages. Theoretically, edges are not required at such supersteps. For Push, we design a lazy loading technique for efficiency. That is, edges will not be prepared before *generateMessages*() is invoked. Instead, they are read from disks only when indeed used, i.e., *getOutEdges*() is further invoked in *generateMessages*(). Currently, the lazy technique cannot work for BPull because *generateMessages*() is invoked along edges.

8 Performance studies

We compare *HybridGraph* against two push-based systems *Giraph* (Push) [1] and *MOCgraph* (PushM) [45], and a well-known pull-based system *GraphLab PowerGraph* (Pull) [12] [6]. We use BPull, BHS, and GHS to indicate our proposals in Sections 3-5, respectively. *MOCgraph* employs a message online computing technique to improve the performance of *Giraph* when messages are commutative. It is the up-to-date push-based system in terms of I/O-efficiency. *GraphLab PowerGraph* is memory-resident but we modify it to support disk-based operations, to confirm the I/O-inefficiency of existing pulling modes. *GraphLab PowerGraph* is implemented using C++, while others are based on Java. A more detailed testing report can be found in the full paper [5].

8.1 Experimental setup

Experimental cluster: We conduct testing using two clusters, a *local cluster* with HDDs (7,200 RPM) and an *amazon cluster* with SSDs. Each of them consists of 30 computational nodes with one additional master node connected by a Gigabit Ethernet switch, where one node is equipped with 4 CPUs. Other configurations are listed in Table 2. Without loss of generality, each node runs only one task.

Table 2 Configurations of two clusters

Cluster	RAM	Disk	$s_{rr}/s_{rw}/s_{sr}$ †	s_{net} ‡
local	6.0GB	500GB	1.2/1.2/2.4MB/s	112MB/s
amazon	7.5GB	30GB	18.2/18.2/18.3MB/s	116MB/s

† Reported by the disk benchmarking tool *fio-2.0.13*, using the mixed I/O pattern "random/sequential mixed reads and writes, and 50% of the mix should be reads". This is because reads and writes are performed at the same time in a real system.
‡ Reported by the network benchmarking tool *iperf-2.0.5*.

Algorithms: We test six graph algorithms, namely, PageRank [21], Single Source Shortest Path (SSSP) [21], Label Propagation Algorithm (LPA) [25], Simulating Advertisement (SA) [19], Maximal Independent Sets (MIS) [23] and Bipartite Matching (BM) [21]. Because Fig. 3(b) has already introduced PageRank, we simply review other algorithms.

[6] All tests are run in the synchronous manner.

SSSP: It finds the shortest distance between a given source vertex to any other one. Initially, the given source vertex is active and has the shortest distance 0 as its value. In every superstep, a vertex minimizes its distance based on the values of in-neighbors. A newly found distance will be broadcasted to all out-neighbors.

LPA: It is a near linear community detection algorithm based on label propagation. The label value of each vertex is initialized by its own unique vertex id. In the following supersteps, the value is updated by the label that a maximum number of its in-neighbors have. All vertices must broadcast their labels per superstep, in order to collect the whole information from in-neighbors.

SA: It is to simulate advertisements on social networks. Each vertex represents a person with a list of favorite advertisements as its value. A selected vertex is identified as the source and broadcasts its value to out-neighbors. For one vertex, any received advertisement is either further forwarded to out-neighbors or ignored, which is decided by his/her interests. Here we assume that a vertex updates its value by the advertisement that a maximum number of its responding in-neighbors have.

MIS: An independent set is a set of vertices, no two of which are adjacent. Such a set is maximal, denoted by V_{MIS}, if it is not a subset of any other independent set. Let V_{Not} be the set of vertices that are not in V_{MIS}. MIS aims to assign any vertex $u \in V$ to either V_{MIS} or V_{Not}. Assume that V_{UN} is the set of vertices not yet assigned. Initially, both V_{MIS} and V_{Not} are empty, while $V_{UN} = V$. MIS proceeds by alternatively running two kinds of supersteps S_0 and S_1. In S_0, $\forall u \in V_{UN}$, it is moved into V_{Not} if any one of its neighbors is in V_{MIS}. Meanwhile, other vertices in V_{UN} broadcast their ids to indicate they are still unassigned. In S_1, a vertex in V_{UN} is moved into V_{MIS}, if it has the smallest id among all adjacent vertices in V_{UN} (inferred by messages from S_0). Then, new messages are broadcasted to put neighbors into V_{Not} in S_0 of the next round. MIS converges until V_{UN} becomes empty.

BM: The goal of BM is to find a maximal matching on a bipartite graph with two distinct sets of vertices, labelled *Left* and *Right*. Such a matching is a subset of edges without common endpoints, termed E_{BM}. No additional edge can be added to E_{BM}. Computing E_{BM} requires a series of phases. One phase P_i is further divided into four supersteps (P_{i0}-P_{i3}). Specifically, in P_{i0}, each left vertex u broadcasts its id as invitation messages to its neighbors. In P_{i1}, a right vertex v randomly accepts one of invitations and sends back v.id as an acceptance notification. In P_{i2}, several acceptances may be received for any selected inviter, but only one is confirmed by replying a new message. In P_{i3}, a right vertex will receive one confirmation at most. The selected inviter in P_{i2} and the confirmed accepter in P_{i3} mark themselves as matched and then a newly found edge is added to E_{BM}.

A matched vertex will not be processed in remaining supersteps. BM terminates when E_{BM} keeps invariant.

Clearly, in LPA, SA, and BM, messages for one vertex u must be processed as a whole to update u. They are non-commutative and non-associative. Hence, PushM lacks built-in support for the three algorithms. Further, Pull and BPull, can only concatenate message values. While, messages in PageRank, SSSP, and MIS, are commutative and associative due to the summing or minimizing update policy. As a result, all tested solutions can run these algorithms and combine messages if possible. From the perspective of hybrid solutions, the periodic vertex behaviors in MIS and BM will suddenly and frequently change the runtime per superstep. We thereby test the two *Sudden Change* algorithms using GHS, while other algorithms are run under BHS. In particular, during P_{i1}-P_{i3} of BM, new messages are generated based on received messages. The lazy edge loading technique in Section 7.3 can be used to enhance the performance of the built-in Push mode in *HybridGraph*.

Graph datasets: All tests are run over 6 real graphs listed in Table 3. We convert these graphs into undirected ones as inputs when running MIS and BM. Further, to construct a bipartite graph for BM, vertices are hashed into two distinct sets by the modulo operator (%2) with ids. Only edges between the sets are preserved. Besides, each edge is assigned a weight randomized between 0 and 1 when generating messages in SSSP. By default, we use 5 nodes for small graphs livej, wiki, and ork, and 30 nodes for other large graphs. A graph is partitioned by the range method [1] for *Giraph*, *MOCgraph*, and *HybridGraph*. For *GraphLab PowerGraph*, many intelligent methods are provided, but only Oblivious is used since others exhibit the similar I/O-performance.

Table 3 Real graph datasets (M: million)

Graph	# Vertices	# Directed or undirected edges	Disk size
livej[7]	4.8M	68/86M	0.5/0.6 GB
wiki[8]	5.7M	130/209M	1.0/1.6 GB
ork[9]	3.1M	234/234M	1.6/1.7 GB
twi[10]	41.7M	1,470/2,426M	12.9/20.6 GB
fri[11]	65.6M	1,810/3,699M	17.0/32.0 GB
uk[12]	105.9M	3,740/6,638M	33.0/56.2 GB

[7] http://snap.stanford.edu/data/soc-LiveJournal1.html
[8] http://haselgrove.id.au/wikipedia.htm
[9] http://socialnetworks.mpi-sws.org/data-imc2007.html
[10] http://an.kaist.ac.kr/traces/WWW2010.html
[11] http://snap.stanford.edu/data/com-Friendster.html
[12] http://law.di.unimi.it/webdata/uk-2007-05/

Experiments design: Section 8.2 evaluates memory demands in particular when messages are far larger than the memory allowed to hold messages, to validate the effectiveness of BPull and BHS. Section 8.3 gives the detailed performance analysis of BHS. Section 8.4 further tests GHS using MIS and BM. Section 8.5 reports the effectiveness of system optimizations introduced in Section 7.3. Section 8.6 explores the performance of different fault-tolerant frameworks. Finally, we study scalability in Section 8.7.

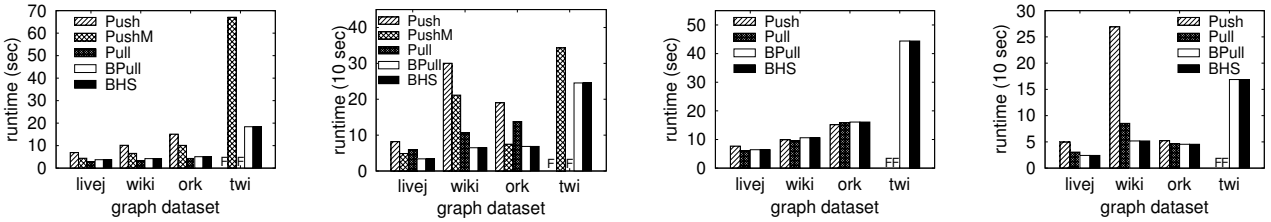
We state two testing scenarios: 1) *sufficient memory*. All systems tested manage data in memory. 2) *limited memory*. *Giraph* has multiple data management policies, and we take the one chosen by Zhou et al. [45] for efficiency. That is, graph data reside on disk, and, also, partial message data are spilled onto disk if the message buffer of one computational node B_i is full. We set the buffer as B_i=0.5 million for small graphs livej, wiki and ork, B_i=1 million for the large graph twi, and B_i=2 million for larger graphs fri and uk. Correspondingly, *MOCgraph* and *GraphLab PowerGraph* use the buffer to cache vertices. Specially, we set a larger buffer for the latter (B_i=2.5 million) to guarantee that most vertices ($> 70\%$) reside in memory. Otherwise the runtime is unacceptable. The LRU replacing strategy is used to manage vertices in *GraphLab PowerGraph*. *HybridGraph* always stores data in VE-BLOCK on the external storage. However, B_i affects the number of Vblocks based on Eq. (5) and Eq. (6).

In the *failure-free* execution of all experiments except fault tolerance, for LPA and PageRank, the average metrics (e.g., runtime and I/O bytes) of one superstep are reported, by totally running 5 supersteps, as the workload per superstep is constant. Other algorithms are run until they converge and we report the metrics of the whole iterations. Settings for fault tolerance are given in Section 8.6. In addition, missing bars labelled with 'F' indicate unsuccessful runs.

8.2 Overall performance evaluation

We test algorithms except MIS and BM in two scenarios: one is *sufficient memory* where push-based and pull-based modes have different favorites (Fig. 13), and the other is *limited memory* where BHS is better (Table 4 and Table 5).

Runtime using *sufficient memory* on the *local cluster*: In general, BPull has superior performance to Push, especially for SSSP and SA over the large diameter graph wiki, which has a long convergent stage where few vertices are updated. This is because BPull reads edges based on V_{res}, instead of V_{act} as Push/PushM. Since $V_{res} \subseteq V_{act}$, BPull reads fewer edges than Push. PushM beats Push, because its message online computing can greatly alleviate the memory pressure to avoid starting Java garbage collection frequently. Besides, with *sufficient memory*, communication costs dominate the sign of Q^t. BHS runs BPull since the latter can efficiently concatenate/combine messages. That is, BHS and BPull are identical, yielding the same runtime. Another observation we can make is that BHS and BPull even outperform Pull in some cases, as they send fewer pull requests than Pull and can offer a comparable message transfer efficiency by



(a) Runtime of PageRank (b) Runtime of SSSP (c) Runtime of LPA (d) Runtime of SA

Figure 13 Runtime with *sufficient memory* on the *local cluster*. Runtime of BPull/BHS is comparable to or even better than that of competitors

Table 4 Runtime with *limited memory* on the *local cluster* (s: second, m: minute)

	PageRank					SSSP					LPA				SA			
	Push	PushM	Pull	BPull	BHS	Push	PushM	Pull	BPull	BHS	Push	Pull	BPull	BHS	Push	Pull	BPull	BHS
livej	40s	20s	327s	3s	3s	220s	198s	39m	40s	39s	27s	387s	7s	7s	103s	31m	29s	28s
wiki	59s	36s	960s	4s	4s	778s	612s	87m	84s	81s	50s	17m	12s	12s	666s	69m	77s	76s
ork	118s	34s	20m	6s	7s	922s	504s	166m	81s	65s	83s	18m	20s	20s	380s	97m	54s	54s
twi	146s	98s	F	34s	34s	722s	607s	F	349s	218s	214s	F	56s	56s	492s	F	162s	146s
fri	930s	52s	F	27s	27s	975s	930s	F	413s	324s	868s	F	77s	77s	600s	F	252s	244s
uk	436s	202s	F	12s	12s	F	F	F	489s	489s	364s	F	67s	67s	F	F	313s	314s

Table 5 Runtime with *limited memory* on the *amazon cluster* (s: second, m: minute)

	PageRank					SSSP					LPA				SA			
	Push	PushM	Pull	BPull	BHS	Push	PushM	Pull	BPull	BHS	Push	Pull	BPull	BHS	Push	Pull	BPull	BHS
livej	47s	15s	305s	2s	2s	223s	139s	31m	33s	30s	43s	335s	6s	6s	122s	28m	22s	21s
wiki	71s	24s	841s	4s	4s	980s	585s	73m	73s	65s	67s	918s	10s	10s	840s	68m	64s	64s
ork	118s	34s	22m	3s	3s	22m	325s	214m	75s	57s	115s	21m	15s	15s	527s	84m	42s	40s
twi	170s	63s	F	34s	34s	678s	447s	F	389s	251s	217s	F	37s	37s	437s	F	143s	135s
fri	627s	35s	F	23s	27s	18m	586s	F	364s	243s	858s	F	56s	56s	518s	F	204s	191s
uk	436s	115s	F	8s	8s	41m	28m	F	289s	289s	471s	F	53s	53s	835s	F	141s	140s

combining messages. However, compared with PushM, the gains of BHS and BPull may be offset by the cost of accessing svertices (e.g., SSSP over ork).

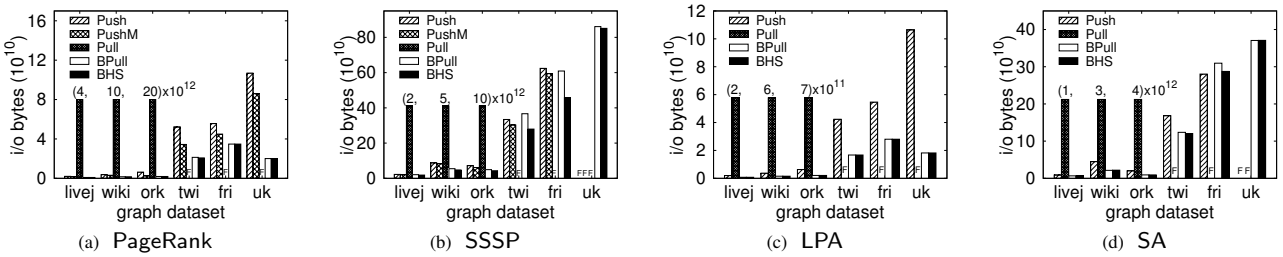
Runtime using *limited memory* on the *local cluster*: Since pull-based modes eliminate expensive message disk I/Os, the speedup of BPull/BHS compared with Push is up to a factor of 35 (PageRank over uk). Compared with PushM, BPull/BHS can still offer roughly 6x speedup for SSSP over wiki and 16x speedup for PageRank over uk. Note that for SSSP over twi, BPull does not work well as expected (only 1.7x faster than PushM). This is because the skewed power-law degree distribution increases the number of fragments, and then the increasing costs of accessing svertices and auxiliary data of fragments are difficult to be offset, especially when the number of messages decreases. The optimal solution is to switch Push and BPull dynamically, like BHS. For SSSP, BHS decreases the runtime of BPull by up to 37.5% over twi. For SA, the gain is roughly 10% over twi.

Runtime using *limited memory* on the *amazon cluster*: This suite of experiments is run on the *amazon cluster* to show the impact of SSDs (Table 5). As expected, all solutions except Push generally benefit from the fast random reads/writes. In particular, for SSSP over twi, BPull is only 1.1 times faster than PushM, instead of 1.7 times on the *local cluster*. The runtime gap is narrowed on SSDs. However, BHS is still 1.8 times faster than PushM. In conclusion, BPull and BHS still perform best. On the other hand, an interesting observation is that the performance of Push is not improved, and even worse in many cases. This is because *Giraph* employs a sort-merge mechanism to handle disk-resident messages, where sorting is computation-intensive. However, each node in the *amazon cluster* is quipped with virtual CPUs. Its computing power is not so strong as that of the node with physical CPUs in the *local cluster*.

I/O costs using *limited memory* on the *local cluster*: Fig. 14 reports I/O costs, in terms of the total number of read and write bytes. Obviously, Pull exhibits an extremely expensive I/O cost due to random and frequent access to svertices. PushM beats Push, as a lot of messages are consumed online by memory-resident dvertices. Finally, when running SSSP over twi, the I/O cost of BPull is usually larger than that of Push and PushM, since the gain achieved by eliminating message I/Os cannot offset the cost incurred by accessing svertices and auxiliary data of fragments. However, BHS can optimize it by switching modes adaptively.

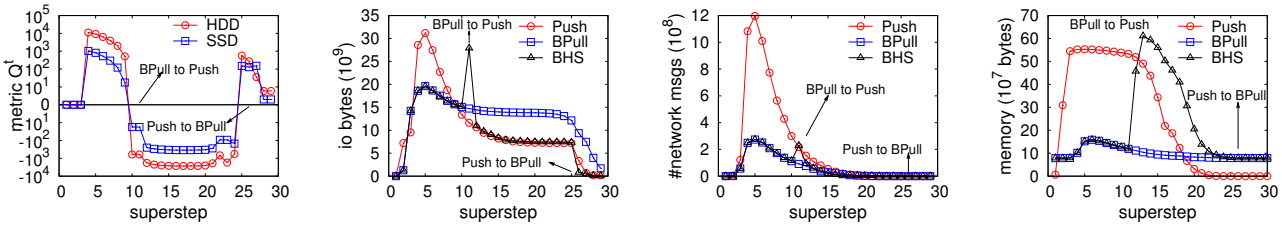
8.3 Analysis of BHS

We validate the effectiveness of BHS using the same setting in Tables 4 and 5. We first explore the impact of hard-



(a) PageRank (b) SSSP (c) LPA (d) SA

Figure 14 I/O costs with *limited memory* on the *local cluster*



(a) Performance metric (b) Disk I/O costs (c) Network costs (d) Memory usage

Figure 15 Analysis of Q^t, I/O costs, network costs, and memory usage for BHS (SSSP over twi, *limited memory*, *local cluster*)

ware characteristics on the performance metric Q, on *local* and *amazon* clusters. After that, a detailed analysis is given to show the resource requirements when switching between Push and BPull, on the *local* cluster. Here, SSSP over twi is tested, since BHS achieves the most gain in this case, and other cases exhibit the similar phenomenon.

Impact of hardware characteristics on Q^t: Fig. 15 (a) shows values of Q^t on the *local cluster* (HDDs) and the *amazon cluster* (SSDs). There obviously exist two switching points: one happens at the 10th superstep, and the other happens at the 25th superstep. For the two clusters, we observe that the switching points do not change. A straightforward explanation is that both BPull and Push can benefit from the fast random read/write performance of SSDs. We then analyze this observation in details. When the sign of Q^t changes, the number of messages is usually small. As listed in Table 2, s_{net} is significantly larger than $s_{sr}/s_{rd}/s_{rw}$. Thus, the impact of communication costs can be ignored. Further, s_{rr}, s_{rw}, and s_{sr}, are close in values. Based on Eq. (11), we can find that the final sign of Q^t is mainly dominated by $\big(C_{io}(\text{Push}) - C_{io}(\text{BPull})\big)$. The latter is orthogonal to the hardware characteristics. Instead, it only relies on the graph topology, the specific algorithm, and the memory capacity.

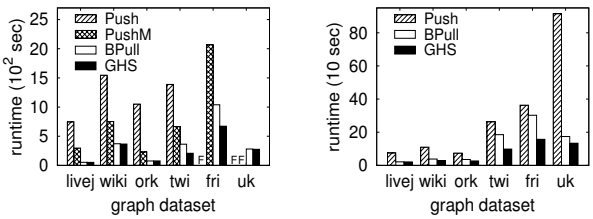
Resource requirements of switching operations: In this suite of experiments, we use Push and BPull as compared solutions, as BHS always chooses one of them to execute iterative computations. Figs 15 (b)-(d) report the change of I/O-pressure, network communication costs, and memory usage. Generally, compared with Push and BPull, BHS does not incur additional resource requirements if the switching operation is not triggered, even though it maintains two replicas of edges. This is because when executing Push/BPull, BHS only accesses edges in the adjacency list/Eblocks. However, in the case of switching from BPull to Push (at the 11th superstep), the resource requirements increase, because BHS must pull messages from svertices, while simultaneously pushing new messages to dvertices. That means, at the switching superstep, messages which will be handled at the 12th superstep in Push are processed in advance. Although shifting the message processing does not incur additional work, the sudden increase of I/O-pressure, network costs, and memory usage, may slightly slow down the performance due to resource contention. Based on our test, BHS takes 22.944s to accomplish the computation at superstep-11. Compared with the sum of 17.512s (BPull at superstep 11) and 5.01s (Push at superstep-12), the performance degradation is less than 2%. This can be easily offset by the switching gains (69.8% for Push and 37.5% for BPull). By contrast, when switching from Push to BPull at superstep-26, messages that should have been generated and pushed at this superstep will be pulled at the next superstep. The resource requirements will not increase. Note that the memory usage of BHS is always more than that of Push, even though switching from BPull to Push has been done at superstep-11. The is because BHS needs to maintain *Metadata* in VE-BLOCK for possibly running BPull.

8.4 Analysis of GHS

In the following, we compare GHS and existing solutions over two representative *sudden-change* algorithms, MIS and BM, to show GHS could get superior performance. All tests are run with *limited memory*. Because the I/O-inefficiency of Pull has already been validated in the previous experiment

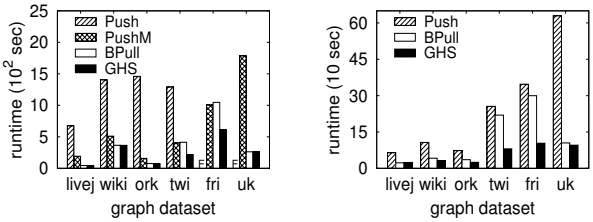
(Tables 4 and 5), now we simply remove it from the tests to reduce clutter in figures.

Runtime: The runtime comparison results are presented in Figs. 16 and 17. Overall, BPull lowers the runtime in all cases except MIS on fri in Fig. 17. The switching mechanism creates an additional gap between GHS and other competitors, and makes GHS perform best. Taking MIS as an example, GHS is approximately 48% faster than BPull over twi (*amazon cluster*). For BM, the improvement is even up to 65% over fri (*amazon cluster*). This is not surprising, given that GHS always smartly, and duly, selects the optimal mode to run the coming superstep.



(a) MIS (b) BM

Figure 16 Runtime of GHS on the *local cluster*



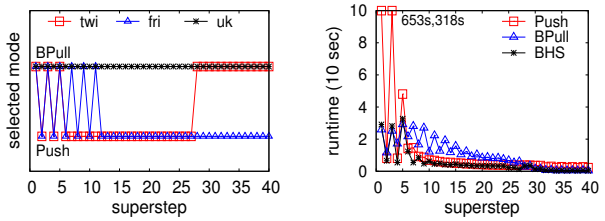
(a) MIS (b) BM

Figure 17 Runtime of GHS on the *amazon cluster*

Mode decision: To better understand GHS's improvements to each superstep, we design experiments to answer the following questions:

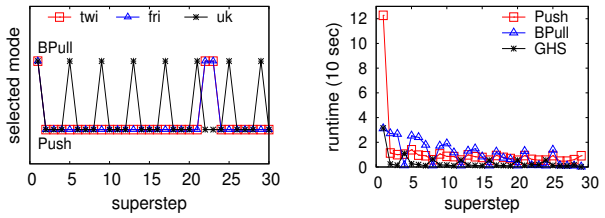
- Effectiveness: how likely can it seek the optimal mode?
- Efficiency: how much time does it spend on switching?

We first analyze the effectiveness. As shown in Figs. 16 and 17, GHS works especially well on large graphs twi, fri, and uk. We thereby show the mode decision per superstep on these graphs (Fig. 18 (a) and Fig. 19 (a)). We further demonstrate the change of runtimes during iterations on the graph with the most significant performance improvement (Fig. 18 (b) and Fig. 19 (b)). After comparing sub-figures (a) and (b), we observe that GHS can exactly capture the variation of runtime difference between Push and BPull, and then effectively select the optimal mode to run a given superstep.



(a) Mode decision (b) Runtime per superstep on twi

Figure 18 Performance analysis for MIS (*amazon cluster*)



(a) Mode decision (b) Runtime per superstep on fri

Figure 19 Performance analysis for BM (*amazon cluster*)

Second, the performance penalty of GHS is mainly dominated by the *mini*-barriers when Push is run. The reason is that a *mini*-barrier separates vertex updating and message sending into two different *mini*-supersteps. That means the node accomplishing updates in the first *mini*-superstep in advance cannot proceed to generate and exchange messages. Instead, it blocks itself to wait for stragglers. Also, new vertex values are reloaded at the second *mini*-superstep. To explore the impact of the two factors, we construct three solutions by removing *mini*-barriers:

- S-0, as a baseline, is equivalent to GHS.
- S-1, removes *mini*-barriers when BPull is run.
- S-2, removes *mini*-barriers when Push is run.

To perform an end-to-end comparison, switching operations in the three solutions must be the same. Towards this end, we manually set the switching points prior to iterations for S-1 and S-2, based on the mode decision of S-0. Table 6 summarizes the runtime comparison. Clearly, the impact of removing *mini*-barriers is less significant for BPull, since nothing is done in the second *mini*-superstep. While, however, it is important for Push in some cases as analyzed above. Generally, the performance degradation is still acceptable when compared with the switching gains.

8.5 Effectiveness of system optimizations

We next show the effectiveness of the static data sharing and lazy edge loading techniques presented in Section 7.3.

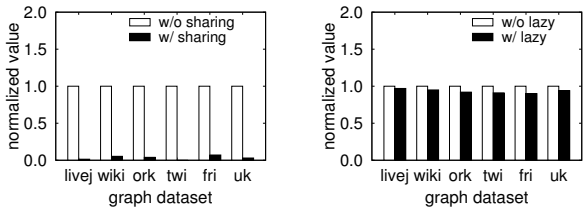
Fig. 20 (a) compares the runtimes of loading input graphs w/ and w/o sharing static data (typically edges). All tests are

Table 6 Impact of *mini*-barriers (seconds, *amazon cluster*)

graph	MIS			BM		
	S-0	**S-1**	**S-2**	**S-0**	**S-1**	**S-2**
livej	42.5	41.9	42.4	24.2	23.3	22.7
wiki	364.6	362.8	364.5	31.6	30.6	31.2
ork	74.1	73.7	74.0	24.6	23.9	23.9
twi	216.8	215.1	193.8	80.4	79.7	72.0
fri	606.8	605.7	593.3	103.5	102.9	90.6
uk	264.9	263.7	274.8	264.0	95.5	94.0

performed by running PageRank under BHS. The loading runtimes are normalized to the case that the sharing function is disabled. Overall, this technique can lower the runtime by 90% on average. The reasons are twofold. First, the number of edges is greatly larger than that of vertices. Reading so many edges from HDFS is costly. Second, edges with the same svertex are parsed into several fragments to build VE-BLOCK, which is computation-intensive. Sharing edges completed avoids the reading and parsing operations.

Fig. 20 (b) compares the computation runtimes of GHS for BM w/ and w/o the lazy edge loading optimization. Testing results are also normalized to the case w/o optimization. The runtime is decreased by roughly 6%.



(a) Static data sharing (b) Lazy edge loading

Figure 20 Impact of static data sharing and lazy edge loading optimizations (*limited memory*, *amazon cluster*)

8.6 Fault tolerance

We next test three fault-tolerant frameworks on *HybridGraph*, including: 1) Scratch, recomputes from scratch upon failures, since nothing is prepared during the *failure-free* execution; 2) Cp, periodically makes a checkpoint of vertex values and hence failed tasks can recover from failures based on the most recent checkpoint [38]; 3) CpLog, additionally logs outgoing messages per superstep to perform a confined recovery [21,37]; 4) CpPull, is our method in Section 6.

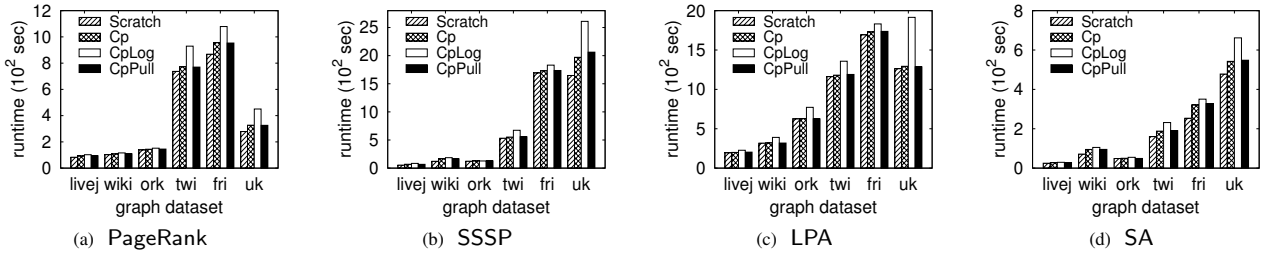
Because GHS performs the similar switching operations as BHS from the fault-tolerance perspective, we focus on the latter. By offline analyzing the checkpointing cost, we observe that a proper interval is 10. Further, unless otherwise specified, we test fault-tolerance with two settings. 1) We run PageRank and LPA with 30 supersteps and kill one task manually at superstep-29 to simulate a failure; 2) SSSP and SA are run until they converge, and we set the failure point at the end of iterations, i.e., $0.9\mathcal{N}$ where $\mathcal{N}$ is the total number of supersteps in the *failure-free* execution. To compute a deterministic SSSP, edge weights are added into input graphs on HDFS (the disk size is thereby increased, e.g., for uk, it is up to 99GB), instead of being generated when producing messages. Besides, all tests are run on the *local cluster* using *limited memory*.

Below, we first report the overall runtime, i.e., the elapsed time from the point where computations start to the point where computations terminate. We then report the λ value in Eq. (16) to show that CpPull can work well in practice. Because failure recovery includes reloading input graph and checkpoint, and recomputing missing supersteps, we finally analyze the performance from the two perspectives, by varying the number of failed tasks ($\mathcal{T}_F$) and failed supersteps.

Overall runtime: Figs. 21 and 22 show the overall runtimes w/ and w/o failures, respectively. Compared with Cp, CpLog greatly drops the runtime when encountering failures, at the expense of degrading the *failure-free* execution. For example, the *failure-free* performance loss is up to 48% for LPA over uk. While, however, the impact of our CpPull is less significant because of nearly-zero logging costs. Even so, CpPull can recover failures quickly, due to the on-demand message generation in BPull. The lightweight logging policy and the fast recovery mechanism yield up to 32% runtime improvement for LPA over uk, compared with CpLog.

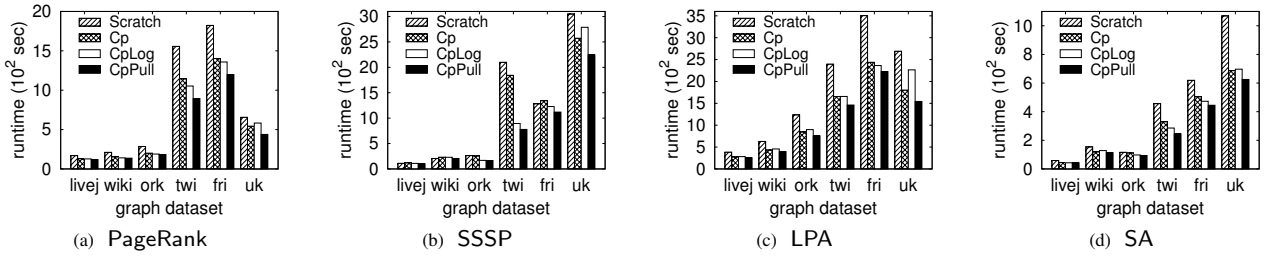
Practicability of CpPull: Besides a formal analysis in Section 6.2, now we empirically validate that CpPull is a practical fault-tolerant method. As shown in Table 7, in all cases, $\lambda > 50\%$. Generally, LPA and SA have a larger λ than PageRank and SSSP, since messages in the latter can be combined to reduce the volume of logged data in CpLog. Particularly, when $\lambda \geq 1$, $\mathcal{T}_F \leq \min\{\lambda\mathcal{T}, \mathcal{T}\} = \mathcal{T}$. That means, in any cases, we can guarantee CpPull outperforms CpLog in theory.

Impact of the number of failed nodes: We then explore the performance features by varying the number of failed tasks. Shown in Fig. 23 (a), both CpLog and CpPull scale with failed tasks due to increased recovery workloads, while Cp is not sensitive to the variation because it always rolls back computations on all tasks. We further distinguish CpPull from CpLog in recovery. First, both of them benefit from reloading input data for restarted tasks only (see Fig. 23 (b)). Second, as analyzed in Section 6.2, CpPull recovers failures more efficiently than CpLog when $\mathcal{T}_F < \lambda\mathcal{T} = 0.59 \times 30 = 17.7$ (Fig. 23 (c), $\lambda = 0.59$ is listed in Table 7). Fig. 23 (c) also reveals that even though $\mathcal{T}_F > 0.59\mathcal{T}$, CpPull still has a superior performance. This is because when more restarted tasks log messages, the straggler problem is considerable and then slows down the practical performance of CpLog.



(a) PageRank (b) SSSP (c) LPA (d) SA

Figure 21 Overall runtime in the failure-free execution



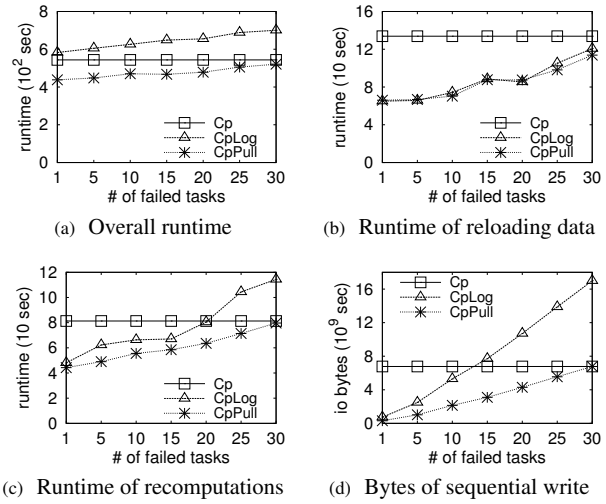
(a) PageRank (b) SSSP (c) LPA (d) SA

Figure 22 Overall runtime when one task fails

Table 7 λ in CpPull

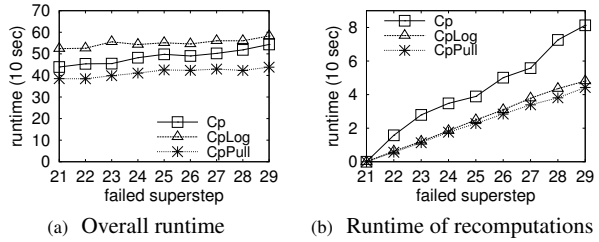
	PageRank	SSSP	LPA	SA
livej	0.76	0.50-0.64	2.02	0.50-1.26
wiki	0.62	0.50-0.57	1.46	0.50-1.10
ork	0.61	0.50-0.55	2.20	0.50-1.16
twi	0.78	0.50-0.65	1.43	0.50-1.10
fri	0.75	0.50-0.65	1.20	0.50-0.94
uk	0.59	0.50-0.53	2.73	0.50-0.86

Impact of the failed superstep: We finally study the scalability of checkpoint-based frameworks by fixing $\mathcal{T}_F$ as 1 and then varying the failed supersteps. All frameworks scale with the increased failed superstep (Fig. 24 (a)) because more missing supersteps are recomputed. Because reloading data does not care about where the failure happens, we then report the recomputation cost in Fig. 24 (b). CpLog and CpPull have the similar performance because the cost of logging messages on a single restarted task is small. Also, straggler problem in the former can be ingored.



(a) Overall runtime (b) Runtime of reloading data
(c) Runtime of recomputations (d) Bytes of sequential write

Figure 23 Performance analysis when varying the number of failed tasks (PageRank over uk)



(a) Overall runtime (b) Runtime of recomputations

Figure 24 Performance analysis when varying the failed superstep (PageRank over uk).

8.7 Scalability

We study the scalability of PageRank in Fig. 25 using the-state-of-the-art push-based and pull-based solutions: PushM and BHS. All tests are run using *limited memory*. Obviously, decreasing the number of nodes increases the volume of data on each node, and then leads to more disk I/Os, i.e., writing/reading more messages for PushM, and reading more data in VE-BLOCK for BPull in BHS. The former is much more expensive than the latter. Thus, we observe a super-linear performance degradation for PushM. By contrast, the runtime increases sub-linearly for BHS.

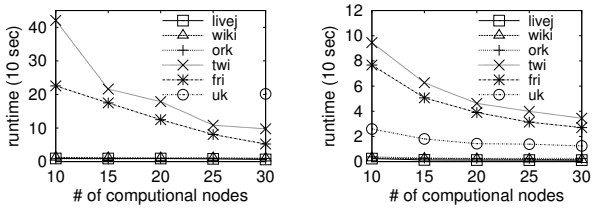


Figure 25 Scalability of computations (*limited memory*, *local cluster*)

9 Related Work

We summarize representative distributed graph systems in Table 8. Also, existing fault-tolerant techniques are discussed to highlight our contributions.

Push-based systems: As we all know, iterative graph algorithms generate a large number of messages [45], and it is a non-trivial task to efficiently manage them on external storage. Among existing distributed systems with disk-resident data, one policy employed by *PEGASUS* and *Gbase*, is to directly access messages using a distributed file system underneath, e.g., HDFS, for fault tolerance, leading to an expensive I/O-cost. In *Haloop*, a Mapper Input Cache is used to keep data on local disk instead of HDFS, which reduces the runtime by nearly 20% [8]. Besides, *Giraph* also manages messages on local disk if it cannot hold all of them in memory. Although the local disk has a superior performance, it is still far from ideal. As reported by Zhou et al. [45], for PageRank over their largest graph dataset, the runtime of *Giraph* with memory capacity of 1.5GB$\times$46 is roughly 6 times more than that with memory capacity of 4GB$\times$46, due to local I/O costs. In addition, *Pregelix* exploits "join" as used in the database community to model the matching operation between messages and vertices, and utilizes a B-tree to improve the disk performance, but the effect is limited for message-intensive algorithms, like PageRank.

Another preferred solution is to reduce the number of messages resident on disk. *Giraph* uses a Combiner to combine messages sent to the same vertex into a single one, which decreases the storage requirements at the receiver side. Furthermore, *MOCgraph* online processes messages received in a streaming manner, instead of keeping them until the next superstep. Combiner inherently requires messages involved in combining to be resident in memory. For online computing in *MOCgraph*, vertices also must be kept in memory. However, it is difficult to satisfy these requirements when processing extremely large graphs. In addition, neither of them works when messages are not commutative.

Many push-based systems also keep graph data on disk during iterations to improve the scalability. *Giraph* exchanges data between memory and disk using the LRU replacement strategy, but the poor locality of data accesses limits the effectiveness. *Hama* records the edge offset of each vertex to avoid loading unnecessary edges. It is suboptimal when all edges are required, such as PageRank. *GraphX* partitions vertices and edges into collections independently to process them in parallel, leading to many-to-many associations among collections. This data structure works well in memory, but is I/O-inefficient if collections reside on disk. The reason is that when performing "join" at an edge collection ("join site"), vertices from many vertex collections may be written/read on disk when the memory cannot hold them. *MOCgraph* organizes graph data by a hot-aware repartitioning method, to keep more high in-degree vertices in memory for its online computing technique. Apparently, all of them ignore the cost of accessing vertex values, since vertex-centric push-based systems access each vertex once in a superstep. However, that is a problem in pull-based systems, and we design a new block-centric data structure to address it. Apart from the vertex-centric model, *Giraph++* and *Bolgel* propose a sub-graph centric model to accelerate graph analysis, as vertices in the same subgraph directly communicate with each other and can be updated by existing sequential algorithms. This is complement to our work and can be implemented on *HybridGraph*.

Table 8 Distributed graph systems

Name	PUSH	PULL	DISK
Giraph++ [33]	✓		
Blogel [39]	✓		
GiraphX [32]	✓		
GPS [26]	✓		
GRE [41]	✓		
Mizan [19]	✓		
Naiad [22]	✓		
Pregel [21]	✓		
Trinity [29]	✓		
Faunus/Titan [6]	✓		✓
PEGASUS [18]	✓		✓
Gbase [17]	✓		✓
Haloop [8]	✓		✓
Giraph [1]	✓		✓
GraphX [13]	✓		✓
MOCgraph [45]	✓		✓
Hama [2]	✓		✓
Pregelix [7]	✓		✓
Surfer [9]	✓		✓
Chronos [14]	✓	✓	
Kineograph [11]	✓	✓	
Pregel+ [40]	✓	✓	
Kylin [15]		✓	
Seraph [38]		✓	
GraphLab PowerGraph [20,12]		✓	
LFGraph [16]		✓	

Pull-based systems: As listed in Table 8, there exist several pull-based systems only designed for memory-resident computations. In these systems, destination vertices need to send pull requests to source vertices individually, which

consumes much traffic even though some requests can be combined [11,38,40,15]. *GraphLab PowerGraph* employs a vertex-cut mechanism to reduce the network cost of sending requests and transferring messages at the expense of replicating vertices. Also, in these pull-based systems, reading a source vertex may be performed multiple times if it is the neighbor of different destination vertices, which is not free [14]. Since none of these systems considers the I/O cost, adapting their techniques for computations of disk-resident vertices may cause serious I/O-inefficiency. Finally, *Chronos*, *Kineograph*, and *Pregel+* support Push and Pull meanwhile, but either Push or Pull is used for a given algorithm. However, BHS and GHS presented in this paper can switch between Push and our BPull adaptively during iterations to obtain optimal performance. Our BPull focuses on improving the I/O efficiency of reading vertices, and optimizing the communication cost of sending pull requests and exchanging messages. We stress that the push-based system *Blogel* supports block-level communication to save network resources, but only for specific algorithms like connected components. This is because a block is modeled as a sub-graph and vertices within it must share the same value, which makes it unsuitable for many algorithms, like PageRank and SSSP. *Blogel* and BPull are technically orthogonal and have totally different implementation mechanisms.

Fault tolerance: Today's graph processing systems typically use checkpointing [21] to tolerate failures. Further, researchers propose to log messages [21,30] to confine recovery to failed tasks only. However, this approach slows down the *failure-free* execution. Instead, our solution can balance the tradeoff between *failure-free* execution and failure recovery. We are aware that Spark [4] divides data into several partitions and then logs the lineage of operations among partitions [44]. Upon failures, if the lost partition depends on a limited number of others ("narrow dependency"), lineage can confine recomputations to such partitions only. However, in graph systems there exist many costly "wide dependency" operations where each partition depends on all of others, such as joining messages from neighbors with vertex values. In this case, full recomputations as used in checkpointing are required. Note that many techniques have been proposed to improve the checkpointing performance, like archiving data in a streaming manner [37], reducing the data volume [38], parallelizing the recovery computations [30], dynamically adjusting the checkpointing interval and prioritizing data [35]. Because we focus on optimizing the logging cost, all of these techniques are complement to our work.

We are aware that some reactive solutions can directly recover failures without checkpointing or logging messages. This can be achieved by replicating data [24,10], but memory resources may be quickly exhausted [45]. Another implementation is to carefully design an algorithm-specified compensation function [27], which is a nontrivial task [37].

10 Conclusion

This paper proposes a new adaptive and I/O-efficient message processing mechanism for vertex-centric graph computing on cloud. First, we show that the way of consuming messages received immediately will reduce the I/O cost at the receiver side to zero. The I/O reduction shifts to the sender side, and becomes the cost of reading a graph. Therefore, by effectively organizing a graph beforehand, we design a new pulling mode BPull. Further, to obtain optimal performance, Push and BPull are seamlessly combined in our BHS/GHS framework and adaptively switched according to the variation in message scale. Also, an efficient fault-tolerant framework is designed by utilizing the properties of BHS/GHS. Experiments show that our proposals can significantly outperform the up-to-date methods.

References

1. Apache giraph. `http://giraph.apache.org/`.
2. Apache hama. `https://hama.apache.org/`.
3. Apache hbase. `https://hbase.apache.org/`.
4. Apache spark. `http://spark.apache.org/`.
5. Hybridgraph technical report. `https://sites.google.com/site/vldbj17/`.
6. Titan. `http://titan.thinkaurelius.com/`.
7. Y. Bu, V. Borkar, J. Jia, M. J. Carey, and T. Condie. Pregelix: Big (ger) graph analytics on a dataflow engine. *In Proc. of the VLDB Endowment*, 8(2):161–172, 2014.
8. Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst. Haloop: efficient iterative data processing on large clusters. *In Proc. of the VLDB Endowment*, 3(1-2):285–296, 2010.
9. R. Chen, X. Weng, B. He, and M. Yang. Large graph processing in the cloud. In *Proc. of SIGMOD*, pages 1123–1126. ACM, 2010.
10. Z. Chen. Algorithm-based recovery for iterative methods without checkpointing. In *Proc. of HPDC*, pages 73–84. ACM, 2011.
11. R. Cheng, J. Hong, A. Kyrola, Y. Miao, X. Weng, M. Wu, F. Yang, L. Zhou, F. Zhao, and E. Chen. Kineograph: taking the pulse of a fast-changing and connected world. In *Proc. of EuroSys*, pages 85–98. ACM, 2012.
12. J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *Proc. of OSDI*, volume 12, page 2, 2012.
13. J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica. Graphx: Graph processing in a distributed dataflow framework. In *Proc. of OSDI*, pages 599–613, 2014.
14. W. Hant, Y. Miao, K. Li, M. Wu, F. Yang, L. Zhou, V. Prabhakaran, W. Chen, and E. Chen. Chronos: a graph engine for temporal graph analysis. In *Proc. of EuroSys*, page 1. ACM, 2014.
15. L.-Y. Ho, T.-H. Li, J.-J. Wu, and P. Liu. Kylin: An efficient and scalable graph data processing system. In *Proc. of IEEE BigData*, pages 193–198. IEEE, 2013.
16. I. Hoque and I. Gupta. Lfgraph: Simple and fast distributed graph analytics. In *Proc. of the First ACM SIGOPS Conference on Timely Results in Operating Systems*, page 9. ACM, 2013.
17. U. Kang, H. Tong, J. Sun, C.-Y. Lin, and C. Faloutsos. Gbase: a scalable and general graph management system. In *Proc. of SIGKDD*, pages 1091–1099. ACM, 2011.
18. U. Kang, C. E. Tsourakakis, and C. Faloutsos. Pegasus: A peta-scale graph mining system implementation and observations. In *Proc. of ICDM*, pages 229–238. IEEE, 2009.

19. 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. of Eurosys*, pages 169–182. ACM, 2013.
20. 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. *In Proc. of the VLDB Endowment*, 5(8):716–727, 2012.
21. G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *Proc. of SIGMOD*, pages 135–146. ACM, 2010.
22. D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: a timely dataflow system. In *Proc. of SOSP*, pages 439–455. ACM, 2013.
23. D. Peleg. *Distributed computing: a locality-sensitive approach*. SIAM, 2000.
24. M. Pundir, L. M. Leslie, I. Gupta, and R. H. Campbell. Zorro: Zero-cost reactive failure recovery in distributed graph processing. In *Proc. of SoCC*, pages 195–208. ACM, 2015.
25. U. N. Raghavan, R. Albert, and S. Kumara. Near linear time algorithm to detect community structures in large-scale networks. *Physical Review E*, 76(3):036106, 2007.
26. S. Salihoglu and J. Widom. Gps: A graph processing system. In *Proc. of SSDBM*, page 22. ACM, 2013.
27. S. Schelter, S. Ewen, K. Tzoumas, and V. Markl. All roads lead to rome: optimistic recovery for distributed iterative data processing. In *Proc. of CIKM*, pages 1919–1928. ACM, 2013.
28. Z. Shang and J. X. Yu. Catch the wind: Graph workload balancing on cloud. In *Proc. of ICDE*, pages 553–564. IEEE, 2013.
29. B. Shao, H. Wang, and Y. Li. Trinity: A distributed graph engine on a memory cloud. In *Proc. of SIGMOD*, pages 505–516. ACM, 2013.
30. Y. Shen, G. Chen, H. Jagadish, W. Lu, B. C. Ooi, and B. M. Tudor. Fast failure recovery in distributed graph processing systems. *In Proc. of the VLDB Endowment*, 8(4):437–448, 2014.
31. I. Stanton and G. Kliot. Streaming graph partitioning for large distributed graphs. In *Proc. of SIGKDD*, pages 1222–1230. ACM, 2012.
32. S. Tasci and M. Demirbas. Giraphx: parallel yet serializable large-scale graph processing. In *Euro-Par 2013 Parallel Processing*, pages 458–469. Springer, 2013.
33. Y. Tian, A. Balmin, S. A. Corsten, S. Tatikonda, and J. McPherson. From" think like a vertex" to" think like a graph. *In Proc. of the VLDB Endowment*, 7(3):193–204, 2013.
34. S. Tsukiyama, M. Ide, H. Ariyoshi, and I. Shirakawa. A new algorithm for generating all the maximal independent sets. *SIAM Journal on Computing*, 6(3):505–517, 1977.
35. Z. Wang, Y. Gu, Y. Bao, G. Yu, and L. Gao. An i/o-efficient and adaptive fault-tolerant framework for distributed graph computations. *Distributed and Parallel Databases*, 35(2):177–196, 2017.
36. Z. Wang, Y. Gu, Y. Bao, G. Yu, and J. X. Yu. Hybrid pulling/pushing for i/o-efficient distributed and iterative graph computing. In *Proc. of SIGMOD*, pages 479–494. ACM, 2016.
37. C. Xu, M. Holzemer, M. Kaul, and V. Markl. Efficient fault-tolerance for iterative graph processing on distributed dataflow systems. In *In Proc. of ICDE*, pages 613–624. IEEE, 2016.
38. J. Xue, Z. Yang, Z. Qu, S. Hou, and Y. Dai. Seraph: an efficient, low-cost system for concurrent graph processing. In *Proc. of HPDC*, pages 227–238. ACM, 2014.
39. D. Yan, J. Cheng, Y. Lu, and W. Ng. Blogel: A block-centric framework for distributed computation on real-world graphs. *Proc. of the VLDB Endowment*, 7(14):1981–1992, 2014.
40. D. Yan, J. Cheng, Y. Lu, and W. Ng. Effective techniques for message reduction and load balancing in distributed graph computation. In *Proc. of WWW*, pages 1307–1317, 2015.
41. J. Yan, G. Tan, and N. Sun. Gre: A graph runtime engine for large-scale distributed graph-parallel applications. *arXiv preprint arXiv:1310.5603*, 2013.
42. M. Yannakakis and F. Gavril. Edge dominating sets in graphs. *SIAM Journal on Applied Mathematics*, 38(3):364–372, 1980.
43. J. Yin and L. Gao. Scalable distributed belief propagation with prioritized block updates. In *Proc. of CIKM*, pages 1209–1218, 2014.
44. M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 2–2. USENIX Association, 2012.
45. C. Zhou, J. Gao, B. Sun, and J. X. Yu. Mocgraph: Scalable distributed graph processing using message online computing. *Proc. of the VLDB Endowment*, 8(4):377–388, 2014.

Appendix

A Proof of Theorem 1

Proof Let $F[\mathcal{V}]$ denote the number of fragments associated with u as svertex in Vblock. $\mathbb{E}(F[\mathcal{V}])$ is the expected $F[\mathcal{V}]$. Given a vertex u in Vblock b_i, let the indicator Z_j denote the event that there exists at least one of its outgoing edges in Eblock g_{ij}, i.e., one fragment exists in g_{ij}. The expectation Z_j is $\mathbb{E}(Z_j) = 1 - (1 - P[\mathcal{V}])^{d[u]}$, where $d[u]$ stands for the out-degree of u. Here, $P[\mathcal{V}]$ is the probability of putting an edge into Eblock g_{ij} among $\mathcal{V}$ Eblocks, and $P[\mathcal{V}] \propto \frac{1}{\mathcal{V}}$. The expected number of fragments is:

$$g(\mathcal{V}) = \mathbb{E}(F[\mathcal{V}]) = \sum_{j=1}^{\mathcal{V}} \mathbb{E}(Z_j) = \mathcal{V}\big(1 - (1 - P[\mathcal{V}])^{d[u]}\big).$$

As can be inferred, the first derivative is:

$$g^{'}(\mathcal{V}) = 1 - (1 + \frac{d[u] - 1}{\mathcal{V}})(1 - \frac{1}{\mathcal{V}})^{d[u]-1}.$$

Considering that $d[u] \geq 1$ for most graphs, the second derivative is:

$$g^{''}(\mathcal{V}) = -\frac{d[u](d[u] - 1)}{\mathcal{V}}(1 - \frac{1}{\mathcal{V}})^{d[u]-2} \leq 0.$$

Thus, we have: $g^{'}(\mathcal{V}) \geq g^{'}(\mathcal{V} \to +\infty) = 0$. Suppose $\mathcal{V}_1 \leq \mathcal{V}_2$. Obviously, $\mathbb{E}(F[\mathcal{V}_1]) \leq \mathbb{E}(F[\mathcal{V}_2])$, which means $\mathbb{E}(F[\mathcal{V}]) \propto \mathcal{V}$.

B Proof of Theorem 2

Proof We use S_m, S_v, S_e, S_f represent the average size of per message, per vertex value, per edge, and auxiliary data of each fragment, respectively. We first analyze the features of accessing edges. For BPull, at each superstep, edges are involved in pullRes() to broadcast messages. By contrast, they are accessed in compute() in Push. When all vertices broadcast messages to their neighbors along outgoing edges, $|E^{t-1}| = |E| = \mathcal{M}$. Accordingly, we can compute the I/O cost of edges in Push and BPull, $IO(\mathcal{E}^t) = IO(E^{t-1}) = S_e|E| = S_e\mathcal{M}$. Further, for Push, the I/O bytes of accessing disk-resident messages is: $2(|E|-B)S_m$. For BPull, because $S_m \geq S_v$ and $S_m \geq S_f$, then:

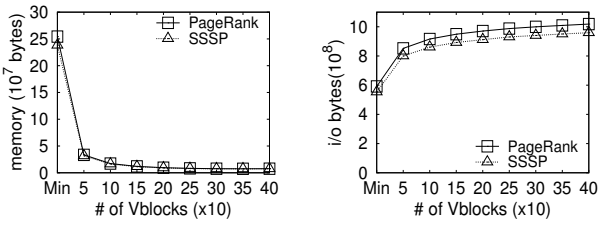
$$IO(F^t) + IO(V_{rr}^t) \leq f(S_f + S_v) \leq 2fS_m.$$

Finally, suppose $B \leq (|E| - f)$, we can infer that:

$$C_{io}(\mathsf{Push}) - C_{io}(\mathsf{BPull}) \geq 2S_m(|E| - B - f) \geq 0.$$

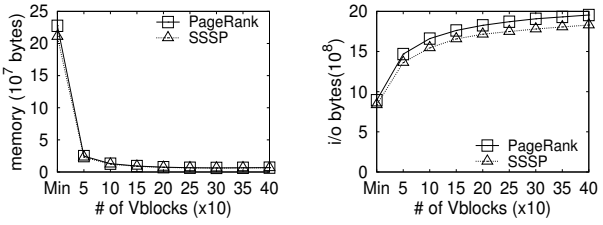
C Impact of the number of Vblocks

We conduct testing to explore the impact of $\mathcal{V}$ using PageRank and SSSP over graphs livej and wiki, on 5 nodes of the *local cluster* (described in Section 8). Fig. 26 and Fig. 27 show the curve of the memory requirement of buffered messages and metadata, and I/O bytes, respectively, when varying the number of Vblocks. The x-axis is the number of Vblocks ($\mathcal{V}$) where x indicates that $\mathcal{V} = x \times 10$ Vblocks are used, and min is the minimum number 5 of Vblocks used, which means that each node has 1 Vblock. For PageRank, we set the number of supersteps as 10 and report the average. For SSSP, we run it until the algorithm converges, and report the maximum value among supersteps. With the increase of $\mathcal{V}$, the memory requirement rapidly drops, while the cost of I/O bytes significantly increases, as more fragments are generated (Theorem 1). We also show the overall runtime when varying $\mathcal{V}$ in Fig. 28. In particular, we find that for SSSP there exists a turning point between $\mathcal{V}$=5 (min) and $\mathcal{V}$=100 ($x = 10$). This is because the iterative computation of SSSP exhibits a gradual convergence stage where fewer edges are required. In this scenario, although a smaller $\mathcal{V}$, such as min, can decrease the number of fragments (Theorem 1), one Eblock stores more edges on average. Since data are scanned in Eblocks, many useless edges are accessed, which wastes I/O bandwith. This algorithm-specific characteristic does not violate Theorem 1. Besides, the prior knowledge about the number of required edges and the corresponding variation during iterations, is not easily achieved. Thus, we still calculate $\mathcal{V}$ using the rules described in Section 3.3 under the assumption that all edges are required.



(a) Memory requirement (b) I/O bytes

Figure 26 Memory requirements and I/O bytes (over livej)

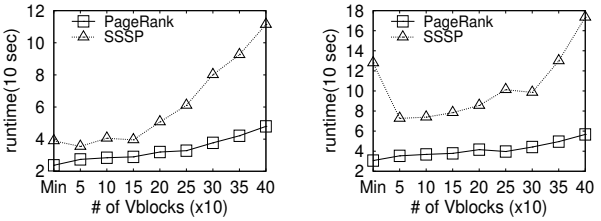


(a) Memory requirement (b) I/O bytes

Figure 27 Memory requirements and I/O bytes (over wiki)

D Impact of combining messages

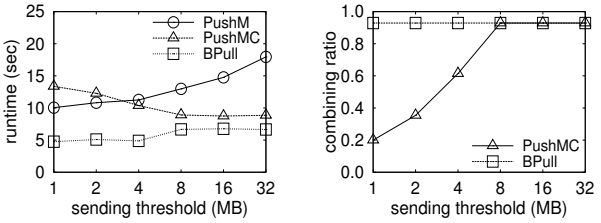
Distributed systems usually set a sending threshold to control the communication behavior of sending messages, in order to make full use of the network idle time and reduce the overhead of building connections. Assume that the threshold is 2 pieces of messages. As shown in Fig. 6 (a), for Push, s_1 generates two messages $m(s_1, d_1)$ and $m(s_1, d_2)$ for d_1 and d_2. Then the system starts the sending operation because a buffer overflow occurs. After that, s_2 also generates a message $m(s_2, d_1)$ for d_1. However, it cannot be combined with $m(s_1, d_1)$ since the latter is unavailable. Thus, the communication gain is limited and usually cannot offset the cost of executing the combining operation. By contrast, for Pull (including BPull), messages are generated based on the demand of the destination vertex. For example, when d_1 sends a pull request, s_1 and s_2 will generate messages for it. This mechanism generally allows all messages to be combined, even though the sending threshold is small.



(a) livej (b) wiki

Figure 28 Overall runtime (PageRank and SSSP)

We modify *MOCgraph* to support combining messages at the sender side. The modified version is identified as PushMC. We define the combining ratio as $\frac{\#\_of\_combined\_messages}{\#\_of\_total\_messages}$. Not surprisingly, as shown in Fig. 29 (a) (in the same setting used in Fig. 13 (a)), when varying the sending threshold (in bytes) from 1MB to 32MB, the runtime of PushM increases because a large threshold cannot make full use of the network idle time. By contrast, PushMC works well, as many messages can be combined (i.e., a large combining ratio shown in Fig. 29 (b)), leading to a communication gain. However, the gain is easily offset by the cost of combining if the threshold is small. On the other hand, for BPull, the communication gain is orthogonal to the threshold. Consequently, the only challenge for BPull is to set a reasonable threshold to make full use of network resources. Fortunately, this constraint is relatively loose. As shown in Fig. 29, BPull works well from 1MB to 4MB. This paper thereby uses 4MB as the default sending threshold.

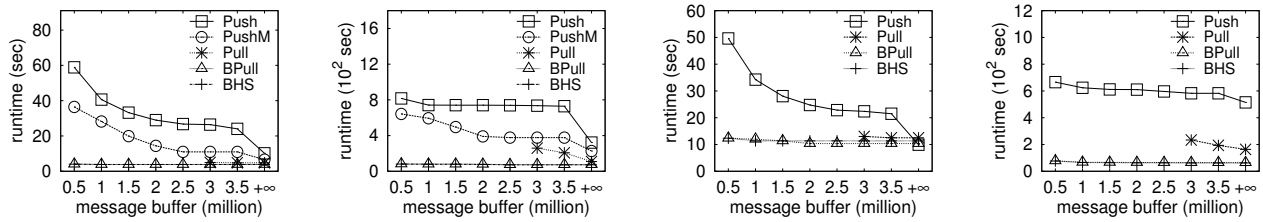


(a) Runtime (b) Combining ratio

Figure 29 Testing the effectiveness of combining messages for PushM and BPull (PageRank over ork)

E Impact of message buffer size

We run PageRank, SSSP, LPA, and SA over all graphs to explore the impact of message buffer size. We vary the buffer size from $+\infty$ (all messages are kept in memory for Push and PushM; all vertices are kept in memory for Pull) to *limited memory*. Fig. 30 shows the performance vs. buffer size plots. The runtime of Push significantly



(a) PageRank (b) SSSP (c) LPA (d) SA

Figure 30 Performance analysis when varying the memory size (wiki, *local cluster*)

Table 9 Comparisons of runtime in five scenarios about *GraphLab PowerGraph* (seconds)

scenarios	PageRank			SSSP			LPA			SA		
	livej	wiki	ork	livej	wiki	ork	livej	wiki	ork	livej	wiki	ork
original	3.0	3.6	5.0	58.8	105.7	120.8	7.4	11.1	18.6	30.4	80.5	47.5
ext-mem	3.1	4.1	5.9	60.2	108.8	129.6	7.6	11.7	19.2	31.6	85.5	49.0
ext-edge	3.9	4.8	7.1	93.5	207.4	220.6	8.6	12.5	21.6	41.6	163.7	64.6
ext-edge-v3	4.5	5.0	7.1	137.2	259.2	222.8	8.8	13.0	21.1	80.4	233.5	64.6
ext-edge-v2.5	654.7	960.4	1187.8	2318.4	5219.5	9956.5	387.0	1024.2	1103.4	1869.5	4160.3	5841.7

increases with the decrease of available buffer size, since accessing messages on disk is extremely expensive. PushM can alleviate the performance degradation by online processing messages sent to vertices resident in memory instead of spilling them onto disk. However, the gain is limited when the buffer further decreases (such as 0.5 million). This is because more vertices are resident on disk, then each message received has less probabilities to be computed online. Finally, when B_i decreases, the performance of Pull drastically degenerates, which validates the I/O-inefficiency of existing pull-based approaches. By contrast, our VE-BLOCK largely alleviates this problem in our BPull and BHS. Hence, BPull and BHS perform the best.

F Performance analysis of *GraphLab PowerGraph*

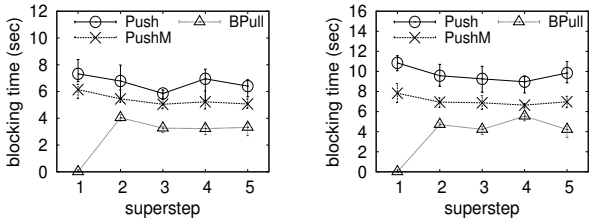
In this suite of experiments, we use five testing scenarios to confirm that our extended *GraphLab PowerGraph* does not incur extra costs compared with the original version. The first one is 1) *original*: using the original *GraphLab PowerGraph* to process data in memory. The other four scenarios use our disk extension: 2) *ext-mem*: all data are memory-resident, like *original*; 3) *ext-edge*: edges reside on disk and vertices are kept in memory; 4) *ext-edge-v3*: edges reside on disk and each task caches 3 million vertices at most in memory; 5) *ext-edge-v2.5*: the number of cached vertices is reduced to 2.5 million.

Table 9 reports the performance in five scenarios using 5 computational nodes of the *local cluster*. Clearly, *ext-mem* achieves a comparable performance with the original *GraphLab PowerGraph*, which validates that our extension is reasonable. Further, the runtime of *ext-edge* slightly increases, because edges are read only once per superstep. Finally, with the increase of the number of disk-resident vertices, the performance seriously degrades, due to random access to vertices.

G Blocking time and network traffic

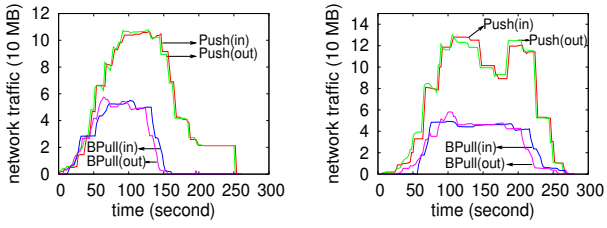
This section analyzes the efficiency of Push and BPull from the perspectives of blocking time and network traffic. Here, blocking time is the time of exchanging messages among computational nodes. We run PageRank in the same setting used in Fig. 13 (a). Fig. 31 shows the average value and fluctuant range (min-max) for blocking time over wiki and ork. BPull starts exchanging messages from the 2nd superstep.

Fig. 32 shows the network traffic for Push and BPull. The network traffic includes all input and output on bytes, and is extracted by GANGLIA [13], a cluster monitoring tool, where the monitoring interval is for every 2 seconds. In particular, we disable the combining function of BPull, to reduce the impact on network traffic and then make a relatively fair comparison with Push. Even so, the almost 50% reduction of network traffic is still achieved due to concatenating messages to the same destination vertices. In Push, both concatenating and combining are disabled, as they are not cost-effective. The traffic of PushM is the same as Push's, since it cannot optimize communication costs.



(a) Blocking time of wiki (b) Blocking time of ork

Figure 31 Blocking time: Push vs. BPull



(a) Network traffic of wiki (b) Network traffic of ork

Figure 32 Network traffic: Push vs. BPull

[13] Ganglia. `http://ganglia.sourceforge.net/`

It is worth noting that in Push and PushM, all distributed tasks will produce and send messages in parallel, whereas in BPull, the message operation is triggered by pulling requests from destination vertex blocks. Our tests show BPull offers a comparable parallelism and superior communication efficiency to Push.