Hybrid Pulling/Pushing for I/O-Efficient Distributed and Iterative Graph Computing

Zhigang Wang[†], Yu Gu[†], Yubin Bao[†], Ge Yu[†], Jeffrey Xu Yu[‡]
[†]Northeastern University, China
[‡]The Chinese University of Hong Kong, China
wangzhiganglab@gmail.com, {baoyubin, guyu, yuge}@ise.neu.edu.cn, yu@se.cuhk.edu.hk

ABSTRACT

Billion-node graphs are rapidly growing in size in many applications such as online social networks. Most graph algorithms generate a large number of messages during iterative computations. Vertex-centric distributed systems usually store graph data and message data on disk to improve scalability. Currently, these distributed systems with disk-resident data take a push-based approach to handle messages. This works well if few messages reside on disk. Otherwise, it is I/O-inefficient due to expensive random writes. By contrast, the existing memory-resident pull-based approach individually pulls messages for each vertex on demand. Although it can be used to avoid disk operations regarding messages, expensive I/O costs are incurred by random and frequent access to vertices.

This paper proposes a hybrid solution to support switching between push and pull adaptively, to obtain optimal performance for distributed systems with disk-resident data in different scenarios. We first employ a new block-centric technique (b-pull) to improve the I/O-performance of pulling messages, although the iterative computation is vertex-centric. I/O costs of data accesses are shifted from the receiver side where messages are written/read by push to the sender side where graph data are read by b-pull. Graph data are organized by clustering vertices and edges to achieve high I/O-efficiency in b-pull. Second, we design a seamless switching mechanism and a prominent performance prediction method to guarantee efficiency when switching between push and b-pull. We conduct extensive performance studies to confirm the effectiveness of our proposals over existing up-to-date solutions using a broad spectrum of real-world graphs.

Keywords

I/O-Efficient; Distributed Graph Computing; Push; Pull

1. INTRODUCTION

The proliferation of popular online social networks such as Facebook and Twitter largely depends on graph analysis over billion-node graphs. In order to handle large graphs, many systems have been developed. *Pregel* [17] by Google as one of the early distributed systems takes a vertex-centric method based on the Bulk-Synchronous-Parallel [26] model (*BSP*) to process graph analysis jobs. *Pregel* is a Master-Slave framework (Fig. 1), where Master divides a job into several tasks assigned onto computational nodes, namely Slaves, in a cluster. Typically, tasks load graph data from a distributed file system and then partition data among themselves. Afterwards, they start computation in parallel through a set of iterations, called "supersteps". The workload in one superstep consists of local computation (i.e., invoking a user-defined function for each vertex) and exchanging intermediate results (i.e., 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 assume that each node runs only one task. This paper thereby discusses communication among nodes, instead of tasks. Two consecutive supersteps are separated by a synchronization barrier to guarantee that the message exchange work is finished.

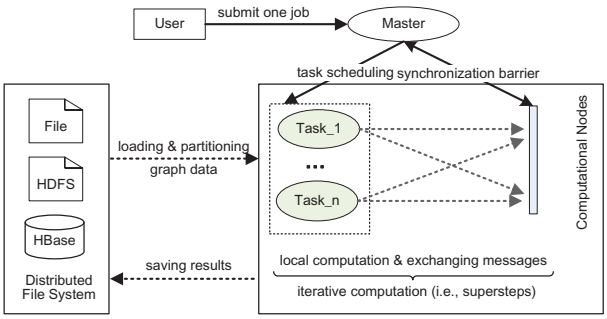


Figure 1: Illustration of iterative graph processing

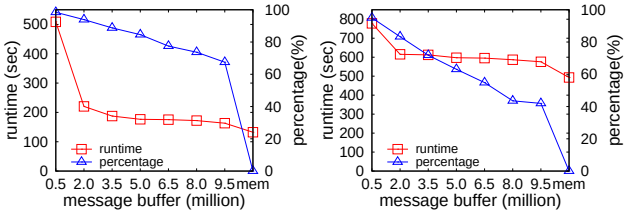
The framework of *Pregel* has been driving much of the research on enhancing its performance, including graph partitioning [8, 23], communication [8, 20, 22], and convergence [25, 31]. This paper focuses on performing graph analysis on a cluster I/O-efficiently, as the memory resource of a given cluster can be easily exhausted due to two main factors. One is the drastically growing rate of real-world graph volumes, e.g., there are over 4.75 billion content items shared on Facebook daily. The other one is the increased message scale in proportion to the graph volume. Adding new physical nodes can alleviate memory pressure, but is not always feasible. The issue we investigate is to find an I/O-efficient way to handle a large number of messages received on every computational node.

Motivation: The fastest way to handle messages received is to keep them in memory. Take *Giraph* [2], an open-source *Pregel* implementation, as an example. In computing PageRank and the single source shortest path (SSSP) [17] over a web graph wiki with 5.7 million vertices and 130 million edges (disk storage size of 1GB) on a cluster of 5 computational nodes, it needs 10 times more memory than the original graph size to guarantee that the overall runtime will not increase rapidly due to disk I/O costs of messages and

SIGMOD'16, June 26-July 01, 2016, San Francisco, CA, USA

DOI: http://dx.doi.org/10.1145/2882903.2882938

memory contention. Otherwise, the performance will drop significantly. As an indication of how *Giraph* scales when graph data and partial message data reside on disk, Fig. 2 shows the overall runtime and the percentage of messages on disk for PageRank (10 supersteps) and SSSP (converging after 284 supersteps). We vary the message buffer size from mem (all messages are kept in memory) down to 0.5 million on a computational node, while the percentage of messages that need to be stored on disk increases from 0% to 98%. As illustrated, the computing time increases from 130 to 510 seconds for PageRank, and from 490 to 780 seconds for SSSP, which confirms the expensive I/O cost incurred by messages. In particular the runtime of PageRank has largely increased from 130 to 160 seconds even though only 4% of messages reside on disk.



(a) PageRank (b) SSSP

Figure 2: The impact of messages (over wiki)

Problem Analysis: Almost all existing distributed graph systems [2, 20, 3], like *Giraph*, take a push-based approach to handle messages by pushing them from source vertices at the sender side to destination vertices at the receiver side. The push-based approach is efficient if most messages are kept in memory because every source vertex will be accessed only once in one superstep. However, it is most likely that the receivers need to store messages on disk, since the message volume can be large. That is I/O-inefficient as shown in Fig. 2, due to the poor temporal locality of messages among destination vertices, caused by writing data randomly. Another preferred approach is to pull messages from source vertices on demand when destination vertices are updating themselves. A receiver avoids writing/reading messages, as they can be consumed immediately after being generated. However, one source vertex may be read multiple times if it is the neighbor of several destination vertices. Since the graph volume is rapidly growing, graph data usually reside on disk. Thus, the I/O cost of reading vertices is still considerable, and may offset that of accessing messages in push, especially when the number of messages is decreasing during iterations. In fact, existing distributed pull-based systems such as *Seraph* [30] and *GraphLab PowerGraph* [8] are designed for memory-resident computation, and no special optimizations are considered for speeding up I/O access.

Obviously, push and pull 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 the two approaches and design a hybrid solution which supports switching between them adaptively. However, this is a non-trivial task due to the following two reasons. First, for current pull-based approaches, each destination vertex individually sends pull requests for desired messages. The frequent and random access to source vertices may cost more than writing/reading messages in push based on our test. 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 are especially difficult in a complex distributed system.

Our Contributions: In order to solve the above two challenges, we propose a novel hybrid solution, called hybrid, to implement an I/O-efficient and adaptive message exchange mechanism for disk-resident graph computations.

First, we design a block-centric pull approach called b-pull to optimize the cost of accessing source vertices when pulling messages. Specifically, in b-pull, vertices 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. b-pull shifts I/O costs of data accesses from messages to vertices. Recall that in a push-based system, messages received are from other computational nodes and are for different vertices. Their distribution might be random and not have patterns to follow. It is difficult to cluster messages received without high overhead. A similar problem also occurs for existing pull-based approaches, but data accessed are source vertices. However, in b-pull, graph data can be organized in an efficient way to reduce the I/O cost by clustering source vertices and edges based on the requested block.

Second, we propose a seamless switching mechanism by first decoupling push and b-pull, and then reorganizing the decoupled functions reasonably, to reduce extra switching costs. Additionally, a prominent performance metric is given through deeply analyzing the total communication and I/O costs for push and b-pull. We should stress that hybrid is not a replacement for our b-pull and existing push approaches. Instead, it always tries to choose a profitable one between them to run graph algorithms during iterations, although b-pull outperforms push in many cases. To this end, it employs b-pull and push as core components and builds a feasible switching scheme on top of them.

The major contributions of our work are summarized as below.

- We propose a new pull-based approach called b-pull which uses a block-centric mechanism to pull messages, in order to reduce the high I/O cost of dealing with messages received in iterations when the memory is not sufficient. A new data structure is designed to separate vertices and edges into different blocks. Furthermore, edges in one block can be clustered well to improve the locality when accessing vertices.

- We present a hybrid solution for handling iterative computations I/O-efficiently, by combining push and b-pull and switching between them adaptively. hybrid obtains optimal performance in different scenarios due to the seamless switching mechanism and the prominent performance prediction.

- We implement our proposals in a prototype system called *HybridGraph*, and then conduct experimental studies to confirm the efficiency of hybrid, in comparison with two typical push-based systems called *Giraph* and *MOCgraph*, and a well-known pull-based system *GraphLab PowerGraph*.

Paper Organization: The remainder of this paper is organized as follows. Section 2 discusses the related work. Section 3 introduces the overview of push, pull, and hybrid. Section 4 describes the b-pull technique and Section 5 presents the hybrid solution after analyzing the performance of push and b-pull. Section 6 reports our experimental studies. We conclude this work in Section 7.

2. RELATED WORK

A lot of representative distributed graph systems have been developed and we summarize them in Table 1.

Push-based Systems: As we all know, iterative graph algorithms generate a large number of messages [32], 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*, *Gbase*, and *Faunus*, is to directly access messages using a distributed file system underneath, e.g., HDFS, for fault tolerance, leading to an expensive I/O-cost. Bu et al. design a Mapper Input Cache to keep data on local disks instead of HDFS for optimization, which reduces the runtime by nearly 20% based on their experiments [5]. Besides, *Giraph* manages messages on local disks 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. [32], 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.

Table 1: Distributed Graph Systems

Name	PUSH	PULL	DISK
Giraph++ [25]	✓		
Blogel [27]	✓		
GiraphX [24]	✓		
GPS [20]	✓		
GRE [29]	✓		
LFGraph [12]	✓		
Mizan [15]	✓		
Naiad [18]	✓		
Pregel [17]	✓		
Pregel+ [28]	✓		
Trinity [22]	✓		
Faunus [1]	✓		✓
PEGASUS [14]	✓		✓
Gbase [13]	✓		✓
Giraph [2]	✓		✓
GraphX [9]	✓		✓
MOCgraph [32]	✓		✓
Hama [3]	✓		✓
Pregelix [4]	✓		✓
Surfer [6]	✓		✓
Chronos [10]	✓	✓	
Kineograph [7]	✓	✓	
Kylin [11]		✓	
Seraph [30]		✓	
GraphLab PowerGraph [16, 8]		✓	

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 should 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. *Faunus* scans the whole graph at every superstep, which is efficient for PageRank (requiring all vertices and edges), but loads a large number of unnecessary data for algorithms where only some of vertices are required. *Giraph* exchanges data between memory and disk using the LRU replacement strategy, but the poor locality of data accesses limits the effectiveness. *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 re-partitioning 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 when the partition structure on each computational node is opened up to users.

Pull-based Systems: As listed in Table 1, 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, which obviously consumes much traffic. *GraphLab PowerGraph* employs a vertex-cut mechanism to reduce the network cost of sending requests and transferring messages at the expense of incurring the space cost of vertex replications. 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. 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* and *Kineograph* support push and pull meanwhile, but either push or pull is used for a given algorithm. However, the hybrid solution presented in this paper can switch between push and our block-centric pull (b-pull) adaptively during iterations to obtain optimal performance. Our b-pull 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 most algorithms, like PageRank and SSSP. *Blogel* and b-pull are technically orthogonal and have totally different implementation mechanisms.

3. OVERVIEW OF push, pull, AND hybrid

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-degree/out-degree of u is the number of its in-neighbors/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 [32]. This paper thereby assumes graph data (vertices and edges) reside on disk. In the following, we introduce the logic of push-based and pull-based approaches, and then give an overview performance analysis for push, pull, and our hybrid.

Push: *Giraph* [2], an open-source implementation of *Pregel*, takes a vertex-centric mechanism for iterative graph analysis in a sequence of supersteps. In one superstep, every vertex u in G computes a user-defined function in parallel. We denote this function as compute(). For simplicity, let $M_I(u)$ and $M_O(u)$ denote the mes-

sages received and sent to/from u, then the compute is as follows and an implementation of PageRank is given in Fig. 3(a).

$$\text{compute}(u^i, M_I^i(u)) \rightarrow (u^{i+1}, M_O^{i+1}(u)) \quad (1)$$

Specifically, in the i-th superstep, first, u gets a message iterator msgs of $M_I^i(u)$ which keeps messages received from in-neighbors in the (i-1)-th superstep, and then initializes a sum to zero (Lines 2-3). Second, after computing the sum of message values (Lines 4-5), its value is updated from u^i to u^{i+1} (Lines 6-8). Third, u sends the new updated value divided by its out-degree as messages $M_O^{i+1}(u)$ to all its out-neighbors (Lines 9-11). M_O^{i+1} for all vertices are the basis to form M_I^{i+1} in superstep (i+1). Finally, u will be inactive by *voting to halt* if the number of supersteps has reached to the maximal value maxNum. Note that an inactive vertex will become active again when it receives messages from its in-neighbors. We call this a push-based approach for the following two reasons. First, svertices send messages voluntarily. Second, messages are already 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 (getNumSupersteps() > maxNum)
13     voteToHalt();
14 }
```

(a) PageRank in PUSH

```
1  Vertex.pullRes (dvertex v) {
2    if (getResFlag() is True)
3      get the edge (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 (getNumSupersteps() > maxNum)
16     voteToHalt();
17   else
18     setResFlag();
19 }
```

(b) PageRank in PULL

Figure 3: push **and** pull

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

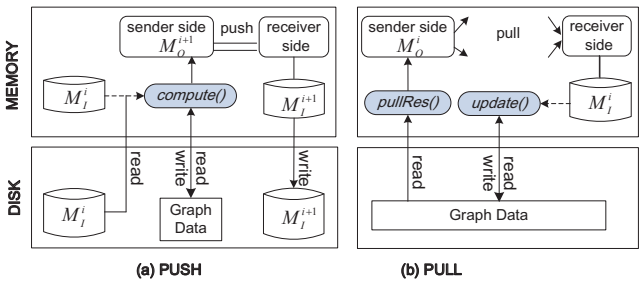


Figure 4: Data Flow: push **and** pull

Pull: The key idea of pull-based approaches is to decouple compute() into two functions, namely, pullRes() and update(), as shown in Eq. (2) and Eq. (3). We explain decoupled functions using PageRank in Fig. 3(b). The pullRes function is used by a svertex u to respond the pull request from a dvertex v for generating a desired message in $M_O^i(u)$ (Lines 1-6) in the similar way as done in pushing. The update function is similar like the compute function in push-based systems without pushing messages. It updates v's value using the pulled messages $M_I^i(v)$. In particular, a vertex u indicates that it should send messages to its out-neighbors by calling a setResFlag function (Line 18). This signal is kept at u. On demand, in the next superstep, u will respond a pull request from any its out-neighbors dvertices 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 when handling large graphs.

$$\text{pullRes}(u^i) \rightarrow M_O^i(u) \quad (2)$$

$$\text{update}(v^i, M_I^i(v)) \rightarrow v^{i+1} \quad (3)$$

Hybrid: In hybrid, we first decouple the computing logic for both push and pull, and then re-organize decoupled functions to support the switching operation. Thus, the performance of hybrid is determined by push and pull.

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.

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

Recall that the computing workload of pull is the same as push's but executed in two functions. Consequently, the difference between push and pull in terms of C_{cpu} can be considered to be negligible. Without loss of generality, suppose push and pull work using the synchronous computing model, even though both of them support the asynchronous iteration. push and pull thereby have the same $\mathcal{N}$ for a given algorithm. Obviously, we only need to discuss C_{net} and C_{io} below. For simplicity, we directly use the number of I/O bytes and network bytes as the values of C_{net} and C_{io}.

First, compared with push, the extra cost of sending pull requests in pull, can be offset by the large communication gains due to concatenating/combining messages (described in Sec. 4.2), when the number of messages is large. Second, in push, messages in M_O^{i+1} are carried across two consecutive supersteps and may end up to be kept on disk if the message volume exceeds the allocated memory resource, which incurs high I/O costs. This problem is avoided in pull because messages produced in pullRes() are consumed in update() immediately 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 messages on disk as reported in Sec 6.1.

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 pull (b-pull) in Section 4. In addition, b-pull also optimizes C_{net} compared with pull. Second, the decreased number of messages inevitably narrows the gap between push and b-pull in terms of C_{net} and C_{io}, even making push beat b-pull. Thus, we design a hybrid solution to chooses the profitable one to run iterative computations by switching between the two approaches adaptively (in Section 5).

4. BLOCK-CENTRIC PULLING METHOD

This section describes our block-centric pulling approach called b-pull by discussing three key issues. First, we present a data structure called VE-BLOCK for sending messages and storing a graph on disk. 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 in VE-BLOCK to enhance the efficiency. Before that, we list important symbols in Table 2.

4.1 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.

Table 2: Definitions of symbols

Symbol	Definition
T_i	A computational node.
$\mathcal{T}$	Number of computational nodes.
$\mathcal{V}$	Number of vertex blocks (Vblocks) which store vertices.
b_i, X_i	The i-th vertex block b_i and its metadata X_i.
g_{ij}	An edge block (Eblock) which stores outgoing edges.
E^t/$\mathcal{E}^t$	Edges read from disk by push/b-pull at superstep t.
f	Number of fragments covering all outgoing edges in $\mathcal{E}^t$.
$\mathcal{M}$	Number of messages produced at one superstep.
M_{disk}	Messages resident on disk in push at one superstep.
C_{io}(push)/ C_{io}(b-pull)	Number of bytes of disk-resident data accessed by push/b-pull at one superstep.
BR_i/BS_i	Message receiving/sending buffer on T_i in b-pull.
B_i, B	Message receiving buffer on T_i in push, and B=$\sum B_i$.
$\mathsf{B}_\perp$	Lower bound of B making C_{io}(push)<C_{io}(b-pull).
Q^t	Performance metric used in hybrid 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 a meta information, X_j, for a Vblock b_j in the computational node where b_j is stored. 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). X_j is used during pulling messages, as described in Section 4.2.

4.2 Pull Requesting and Pull Responding

Two components are used to pull messages. The pull requesting operation (Pull-Request) is performed at the computational node that requests messages to be consumed, and the pull responding operation (Pull-Respond) is performed at the computational node 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. At the same time, an active-flag vector is updated, to indicate whether a vertex value should be updated or not. After receiving all messages, each active vertex in b_i will perform the vertex-centric computation by calling update(). Like the active-flag vector, a responding-flag vector is used to indicate whether a svertex is supposed to send messages to its out-neighbors. Obviously, pulling messages is done in Vblocks, namely, the block-centric pulling mechanism.

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. To this end, T_y uses the meta information X_j in three steps. 1) T_y checks whether the Vblock responding indicator res in X_j is true. If it is true, T_y needs to check further, i.e., checking the i-th bit in the bitmap x_j. 2) If the i-th bit is on, it indicates that there are edges directed from vertices in b_j to vertices in b_i. 3) When both are true, T_y calls pullRes() for any vertex u as svertex in b_j to generate messages, if u's responding-flag is true.

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 v's id. v's id is thereby transferred only once, which reduces communication costs. In addition, if the message value is commutative and associative [17], values are supposed to be combined into 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.

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 the messages received from T_y into a buffer BR_x;
5         concatenate or combine messages in BR_x;
6         update the active-flag vector;
7     foreach vertex u in Vblock b_i do
8         u.update() if u is active;
9         update the responding-flag vector by invoking setResFlag();
```

Algorithm 2: Pull-Respond (Vblock b_i that T_x holds)

```
1 foreach each Vblock's meta 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.getResFlag() 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     send messages in BS_y to T_x that requests messages for b_i;
```

4.3 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 including X_i, and the active-flag and responding-flag vectors, 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.

[1] VE-BLOCK can also be applied to any partitioning method by reordering vertices.

[2] B_i indicates the message receiving buffer size. We ignore the

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, for a given B_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).

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

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

I/O Cost: 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 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 b-pull are proportional to $\mathcal{V}$.

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

space of the sending buffer because *Giraph* immediately flushes out messages at the sender side once a sending threshold is satisfied.

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

5. THE HYBRID SOLUTION

This section first gives a performance analysis of push, pull and b-pull, and then describes a hybrid solution to combine push and b-pull to obtain optimal performance for different scenarios.

5.1 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. 5. As discussed in Section 3, we analyze C_{net} and C_{io}, the main factors affecting the performance.

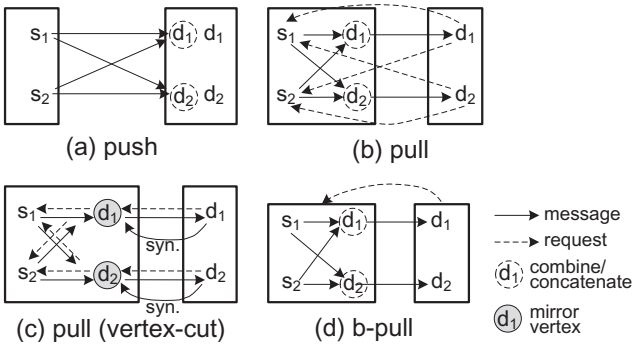


Figure 5: Comparisons of push**,** pull**,** pull **(vertex-cut), and** b-pull **(block-centric)**

Analyzing Communication Costs: Most push-based systems such as *Giraph* [2] and *GPS* [20] 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 the cost of concatenating/combining. By contrast, in pull-based approaches, messages are generated for requested svertices on demand. For example, in Fig. 5 (b), all messages for d_1 are generated based on the requested d_1 and then can be concatencated/combined. This mechanism makes concatenating/combining cost effective, but incurs extra cost of sending pull requests. In the worst case, pull requests will be sent up to $(|V|\mathcal{T})$ times. *GraphLab PowerGraph* employs a vertex-cut mechanism for efficiency and handles the computation by a Gather-Apply-Scatter model. As shown in Fig. 5(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 [8]. Finally, for b-pull, the upper bound of the number of requests is $\mathcal{V}\mathcal{T} \ll |V|$. As shown in Fig. 5(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 for pull with vertex-cut because *GraphLab PowerGraph* designs many sophisticated yet special policies to optimize this problem. As a result, for C_{net}, b-pull beats push and pull, and can offer a comparable performance to pull with vertex-cut.

Analyzing I/O Costs: The cost of reading svertices in existing pull-based approaches is considerable, because sending pull requests for each dvertex individually leads to frequent and random disk accesses (as shown in Figs. 8 and 10). However, b-pull 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 just analyze C_{io} for push and b-pull.

Let $G^t = (V^t, E^t)$ be the subgraph used in the t-th superstep, where $V^t \subseteq V$ and $E^t \subseteq E$. Also, $\mathcal{M}$ is the number of messages produced. $\mathcal{E}^t$ and E^t stand for the set of edges loaded from disk in b-pull and push, respectively. Now, we show the I/O cost of push in Eq. (7) and b-pull 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, $IO(F^t)$/$IO(V_{rr}^t)$ denotes the I/O cost of fragments' auxiliary data/values of svertices in Vblocks (i.e., V_{rr}^t) read by PullRespond. $IO(V^t)$ indicates the cost of updating vertex values, which is the same for both b-pull and push. Note that $IO(E^t)$ depends on the specific implementation of push [2, 32]. Without loss of generality, we only consider the implementation in *Giraph* [2], and other implementations can also be used in our hybrid solution.

$$C_{io}(\text{push}) = IO(V^t) + IO(E^t) + 2IO(M_{disk}) \quad (7)$$

$$C_{io}(\text{b-pull}) = IO(V^t) + IO(\mathcal{E}^t) + IO(F^t) + IO(V_{rr}^t) \quad (8)$$

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 size of F^t. Theorem 2 describes the sufficient condition of using b-pull. That is, $(\frac{|E|}{2} - f)$ is B's lower bound, namely $\mathsf{B}_\perp$, which makes push outperform b-pull in terms of I/O bytes. $|E|$ and f are available after building VE-BLOCK. Accordingly, we can decide whether using b-pull or not 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 $B \leq (\frac{|E|}{2} - f)$, then $C_{io}(\text{push}) \geq C_{io}(\text{b-pull})$.*

On the other hand, Theorem 2 only guarantees the effectiveness if each vertex should send messages to all its out-neighbors at every superstep, such as PageRank. For other algorithms, like SSSP, the number of vertices sending messages (called responding vertices) varies with iterations, i.e., $\mathcal{M}$ is not constant. As a result, for SSSP-like algorithms, the overall I/O cost comparison of push and b-pull 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 b-pull avoids disk I/O costs incurred by messages and greatly reduces the cost of accessing svertices, while simultaneously offering a comparable communication efficiency to pull with vertex-cut. That makes b-pull outperform pull with vertex-cut in disk scenarios. b-pull also beats push in most cases, but the decrease of $\mathcal{M}$ narrows the gap, even making push beat b-pull. This is because b-pull 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 b-pull have different favorite scenarios, and using a single one can hardly achieve optimal performance.

5.2 Switching between push and b-pull

In order to support the switching operation between push and b-pull, hybrid should accommodate them simultaneously from two perspectives: the computing functions for expressing their processing logics, and the data storage for supporting consistent and efficient data accesses.

Computing Functions: Like pull, b-pull also decouples compute() in push into pullRes() and update(). For push, compute() is divided into three functions: load(), update(), and pushRes(). Here, load() loads messages received from the previous superstep into a local buffer (Eq. (9)), to prepare to be consumed in update(). Following update(), pushRes() is immediately invoked to broadcast new messages $M_O^{i+1}(u)$ to u's out-neighbors (Eq. (10)).

$$\text{load}(M_O^{t-1}(u)) \rightarrow M_I^t(u) \quad (9)$$

$$\text{pushRes}(u^{t+1}) \rightarrow M_O^{t+1}(u) \quad (10)$$

Data Storage: The decoupling of compute() supports a seamless switching between push and b-pull. As shown in Fig. 6, when switching from b-pull to push, we first invoke b-pull's pullRes() and update() to update vertex values, and then immediately invoke push's pushRes() based on new vertex values. Obviously, although b-pull and push are completed in a single superstep, the seamless switching guarantees that update() is invoked only once for each vertex, which avoids reading/writing conflicts. Conversely, when switching from push to b-pull, push's load() and update() are invoked to update vertex values which will be used by b-pull's pullRes() at the next superstep. In addition, the shared update() makes push and b-pull can share vertex values efficiently, i.e., Vblocks in VE-BLOCK. However, for push, the access of edges, i.e., Eblocks in VE-BLOCK, is inefficient. The reason is that pushRes() requires all outgoing edges of vertex u, but they are organized in fragments and distributed among different Eblocks. We thereby store edges twice. One is organized by Eblocks and used in b-pull. The other one is organized in an adjacency list, like *Giraph*, and used in push.

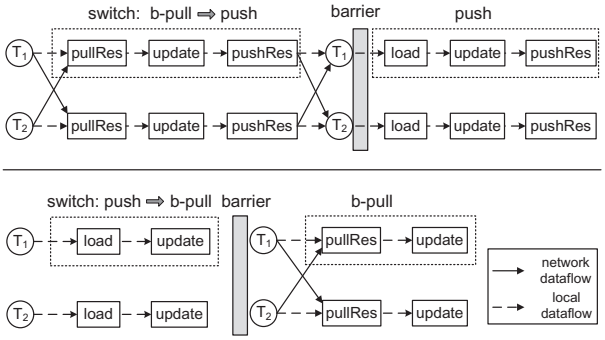


Figure 6: Switching between push and b-pull

5.3 Switching Time

The key to gain optimal performance in hybrid 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 [21] 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 should choose a metric which can characterize the performance of push and our b-pull.

Performance Metric and Switching Condition: The overall performance of push and b-pull is mainly dominated by the communication cost C_{net} and I/O cost C_{io}. Thus, at the t-th superstep, we use $\mathcal{M}_{co}$ (the number of concatenated or combined messages

across network in b-pull) 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 b-pull. Recall that $IO(M_{disk})$ is the number of written/read bytes incurred by writing/reading messages in push. Thus, $\big(IO(E^t)+IO(M_{disk})-IO(\mathcal{E}^t)-IO(F^t)\big)$ is the difference of push and b-pull 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 b-pull 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 (MB/s), and the network throughput (MB/s), respectively. Apparently, b-pull has superior performance if $Q^t \geq 0$.

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

Switching Interval: If we use the dynamic information collected at the t-th superstep to predict the metric at the $(t+\Delta t)$-th superstep, the switching interval is Δt, $\Delta t \geq 1$. As reported by Shang et al. [21], the accuracy of prediction is proportional to $\frac{1}{\Delta t}$. The dynamic information required by $Q^{t+\Delta t}$ includes C_{io}(push), C_{io}(b-pull) and $\mathcal{M}_{co}$. When running push, we can figure out the set of required Eblocks if b-pull is run, based on the distribution of edges used in pushRes(). After that, C_{io}(b-pull) is estimated using the metadata of Eblocks. $\mathcal{M}_{co}$ is estimated by $\mathcal{M}R_{co}$, where R_{co} is the concatenating/combining ratio in the last superstep using b-pull. In contrast, we can easily calculate C_{io}(push) and other dynamic information when running b-pull. Note that $\mathcal{M}$ is not available if hybrid is switching from push to b-pull, because no new message is produced (as shown in Fig. 6). Besides, although no extra work is incurred during switching from b-pull to push, there still exists a slight performance loss caused by resource contention, since pullRes() and pushRes() are run in a single superstep. Obviously, frequent switching is not cost effective. We thereby set Δt as 2.

The Boundary of hybrid: Shang et al. [21] divide well-known graph algorithms into three categories based on the change of active vertices, namely, *Always-Active-Style*, *Traversal-Style*, and *Multi-Phase-Style*. We use the three categories to discuss the boundary of hybrid, as the number of active vertices dominates that of responding vertices and then decides $\mathcal{M}$. First, for *Always-Active-Style*, an accurate prediction of Q^{t+2} is always available and can be used by hybrid to make a smart choice between b-pull and push, as the behavior of vertices stays the same during iterations, like PageRank. Second, the number of active vertices for *Traversal-Style* algorithms, such as SSSP, usually varies with iterations. Thus, the prediction accuracy may be poor. However, hybrid still works well, because the variation usually keeps monotonic for a long time and then hybrid benefits from switching during several subsequent supersteps. Finally, the current hybrid is not suitable for *Multi-Phase-Style* algorithms. This is because they exhibit a periodical change in terms of the active vertex volume, which prevents hybrid from accumulating the switching gain.

5.4 Execution of hybrid

Now, we present the execution of hybrid in Algorithm 3. Here, maxNum is the maximum number of supersteps. The graph loading operation includes two tasks (Line 1): 1) storing graph data, including organizing vertices in Vblocks and storing edges twice, and 2) calculating $B_{\perp}$ used in Theorem 2. After that, we decide the initial execution mode based on Theorem 2, i.e., b-pull if $B \leq B_{\perp}$, and push, otherwise (Lines 2-3). During iterations, if $preMode$ is not equal to $curMode$, we start the switching operation as shown in Fig. 6 (Lines 13-14). Otherwise, runPull() or runPush() is invoked based on $curMode$ and returns the dynamic information which is used to estimate Q^{t+2} to determine $curMode$ of the next superstep (Lines 6-11).

Algorithm 3: Hybrid-Execution (B, G, $maxNum$)

```
1  loading G, and calculating B⊥;
2  preMode ← initMode(B,B⊥);
3  curMode ← preMode;
4  for t = 1 to maxNum do
5      if preMode equals curMode then
6          if is_pull(curMode) then
7              [Mco,Cio(push),Cio(b-pull )] ← runPull();
8          else
9              [Mco,Cio(push),Cio(b-pull )] ← runPush();
10         preMode ← curMode;
11         curMode ← evaluate(Mco,Cio(push),Cio(b-pull ));
12     else
13         runSwitch(preMode, curMode);
14         preMode ← curMode;
```

6. PERFORMANCE STUDIES

We have implemented our hybrid solution in a prototype system called *HybridGraph*[3]. We compare *HybridGraph* with the two push-based systems *Giraph* and *MOCgraph* and the well-known open-source pull-based system *GraphLab PowerGraph*[4]. *MOCgraph* is developed on top of *Giraph* to accelerate the iteration performance by its message online computing technique. It is the up-to-date push-based system in terms of I/O-efficiency. *GraphLab PowerGraph* is a memory-resident system and we modify it to support access to vertices and edges on disk, to confirm the I/O-inefficiency of existing pull-based techniques. *GraphLab PowerGraph* is implemented using C++, while other systems tested are based on Java.

In the following, we use b-pull to indicate our basic pull-based approach using the block-centric mechanism. Furthermore, we use hybrid to stand for our hybrid solution combining push and b-pull. *GraphLab PowerGraph*'s pull approach is represented by pull. In addition, we use push and pushM to indicate the push approach used in *Giraph* and *MOCgraph*, respectively.

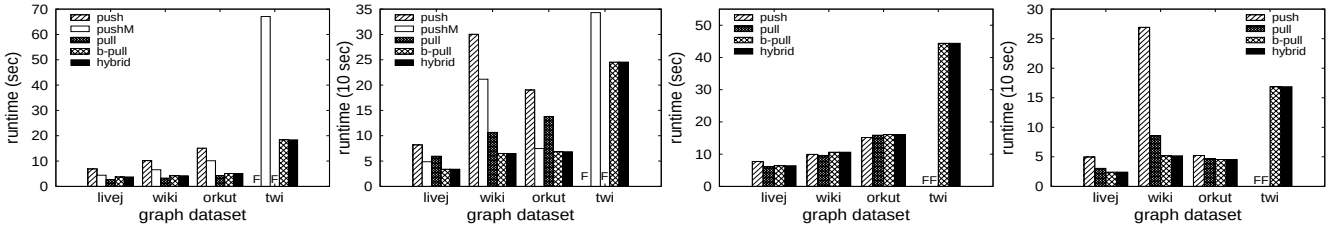
We conduct testing using two clusters, a *local cluster* with HDDs (7,200 RPM) and an *amazon cluster* with SSDs. Each of them consists of 30 computational nodes with one additional master node connected by a Gigabit Ethernet switch, where one node is equipped with 4 CPUs. Other configurations are listed in Table 3.

We test four graph algorithms, namely, PageRank [17], SSSP [17], LPA [19], and SA [15], using 6 real graphs listed in Table 4. By default, we use 5 computational nodes to handle the small graphs livej, wiki, and orkut, and 30 computational nodes for the large graphs twi, fri and uk. A graph to be tested is partitioned by the range method [2], since graph partitioning is time-consuming [23]. Fig. 3(b) in Section 3 describes the computing logic of PageRank. Other algorithms are described below.

- SSSP [17]: It finds the shortest distance between a given source vertex to any other vertex. Initially, the given source

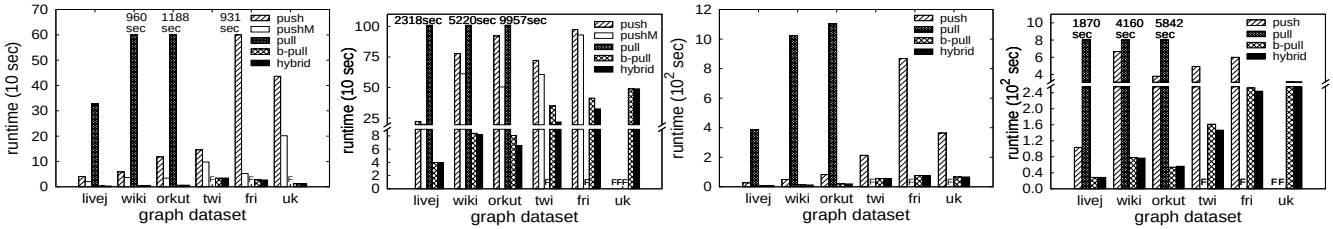
[3]The source code is available at https://github.com/HybridGraph.

[4]We use the synchronous version of *MOCgraph* and *GraphLab PowerGraph* in the testing, and *HybridGraph* can be extended to support the asynchronous iteration.



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

Figure 7: Testing runtime with *sufficient memory* on the *local cluster*



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

Figure 8: Testing runtime with *limited memory* on the *local cluster*

vertex is active and has the shortest distance 0 as its value. In every superstep, the value of a vertex is updated to be the minimum value received from its in-neighbors, if the minimum value is lower than the current one the vertex holds. The new minimum distance will be sent to its out-neighbors.

- LPA [19]: 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 neighbors have. Thus, messages, i.e., community labels, are not communicative. All vertices must be active, to collect the whole information of all neighbors.
- SA [15]: It is to simulate advertisements on social networks. Each vertex represents a person with a list of favorite advertisements as its value. Selected vertices are identified as sources and send values to their out-neighbors. For one vertex, any received advertisement is either further forwarded to out-neighbors or ignored, which is decided by his/her interests. Advertisements as messages are not communicative if we update a value by the advertisement that a maximum number of its in-neighbors have, like LPA.

Below, we first evaluate memory demands in particular when messages are far larger than the memory allowed to hold messages, to validate the effectiveness of b-pull and hybrid. Second, the details of switching between push and b-pull adaptively are given. Third, we study scalability. Finally, we report the performance of loading raw graph data, and then analyze the blocking time and network traffic of push and b-pull. For PageRank and LPA, the average metrics (e.g., runtime and I/O bytes) of one superstep are reported, by totally running 5 supersteps, as the workload of each superstep is constant. On the other hand, we run SSSP and SA until they converge and report the metrics of the whole iterations.

In particular, 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 same one given by Zhou et al. [32] for efficiency. That is, graph data reside on disk and message data may also reside on 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 orkut, 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 *GraphLab PowerGraph* (B_i=2.5 million) to guarantee that most vertices ($> 70\%$) reside in memory, because the runtime is unacceptable when B_i=0.5 million for small graphs. The LRU replacing strategy is used to manage vertices in *GraphLab PowerGraph*. For *MOCgraph*, other data (the remaining vertices, edges, and messages sent to disk-resident vertices) reside on disk. *GraphLab PowerGraph* manages edges and the remaining vertices on disk. Now, *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 addition, missing bars labelled with 'F' in the figures reported indicate unsuccessful runs.

Table 3: Configurations of two clusters

Cluster	RAM	Disk	$s_{rr}/s_{rw}/s_{sr}$ †	s_{net} ‡
local	6.0GB	500GB	1.177/1.182/2.358MB/s	112MB/s
amazon	7.5GB	30GB	18.177/18.194/18.270MB/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*.

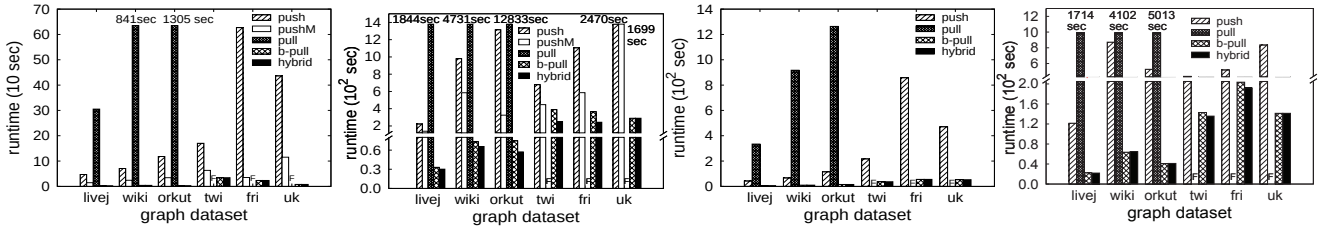
Table 4: Real Graph Datasets

Graph	Vertices	Edges	Degree	Type	Disk size
livej[5]	4.8M	68M	14.2	Social networks	0.50GB
wiki[6]	5.7M	130M	22.8	Web graphs	0.98GB
orkut[7]	3.1M	234M	75.5	Social networks	1.59GB
twi[8]	41.7M	1,470M	35.3	Social networks	12.90GB
fri[9]	65.6M	1,810M	27.5	Social networks	17.00GB
uk[10]	105.9M	3,740M	35.6	Web graphs	33.02GB

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

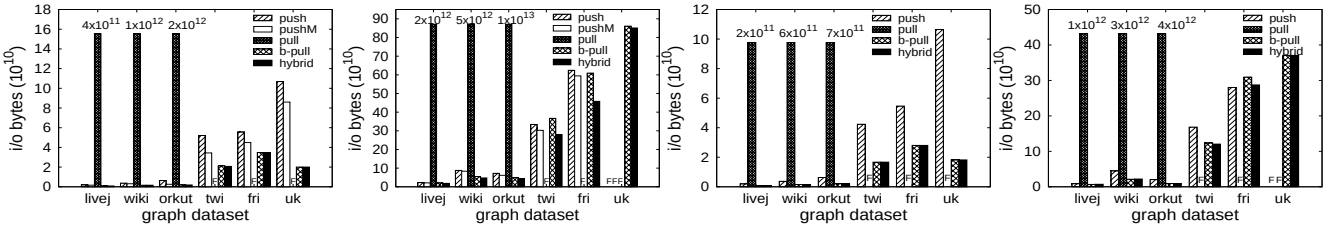
6.1 Overall Performance Evaluation

We test algorithms on all datasets in two scenarios: one is *sufficient memory* where push-based and pull-based approaches have different favorites (Fig. 7), and the other is *limited memory* where hybrid is supposed to be better (Fig. 8 and Fig. 9).



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

Figure 9: Testing runtime with *limited memory* on the *amazon cluster*



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

Figure 10: Testing I/O costs with *limited memory* on the *local cluster*

Runtime test using *sufficient memory* on the *local cluster*: In general, b-pull 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 b-pull reads edges based on the responding vertex set V_{res}, instead of the active vertex set V_{act} as push/pushM, where $V_{res} \subseteq V_{act}$. pushM beats push, because its message online computing can greatly alleviate the memory pressure to avoid starting Java garbage collection frequently. One observation we can make is that hybrid and b-pull even beat pull in some cases, as they send fewer pull requests than pull and can offer a comparable message transfer efficiency by combining messages. However, compared with pushM, the gains of hybrid and b-pull may be offset by extra costs of accessing svertices frequently (e.g., SSSP over orkut). Note that in *sufficient memory*, the communication cost dominates the final result of Q^t. hybrid thereby runs b-pull since it can efficiently concatenate/combine messages, leading to the same performance as b-pull's.

Runtime test using *limited memory* on the *local cluster*: Since pull-based approaches can avoid the expensive message disk I/Os, the speedup of b-pull/hybrid compared with push is even up to a factor of 35 (PageRank over uk). Compared with pushM, b-pull/hybrid can still offer roughly 7x speedup for SSSP over wiki and 16x speedup for PageRank over uk. Note that for SSSP over twi, b-pull does not work well as expected (only 1.7x faster than pushM). The reason is that the highly 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 b-pull during iterations. For SSSP, hybrid contributes to decreasing the runtime of b-pull by up to 37.6% over twi. For SA, the gain achieved by hybrid is up to 9% over twi and fri.

Runtime test using *limited memory* on the *amazon cluster*: This suite of experiments is run on the *amazon cluster* to show the impact of SSDs (Fig. 9). As expected, pull, pushM, b-pull and hybrid benefit from SSDs since SSDs exhibit faster random reads/writes than HDDs. Generally, the speedup is between 1.74 and 3.6. In particular, for SSSP over twi, b-pull is only 1.1 times faster than pushM, instead of 1.7 times on the *local cluster*, as the fast random read/write performance of SSDs narrows the runtime gap. However, our hybrid is still 1.7 times faster than pushM. In conclusion, b-pull and hybrid still perform best. On the other hand, an interesting observation is that the performance of push is not improved, and even worse in some 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 the node's in the *local cluster* (physical CPUs).

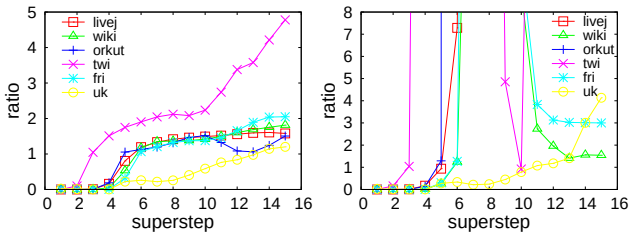
Testing disk I/O costs using *limited memory* on the *local cluster*: In the same setting used in Fig. 8, Fig. 10 reports I/O costs, in terms of the total number of read and written bytes. Obviously, pull exhibits an extremely expensive I/O cost due to random and frequent access to svertices, even though we use LRU to manage vertices. pushM beats push, as its message online computing technique consumes messages sent for memory-resident dvertices immediately. Finally, when running SSSP over twi and fri, the I/O cost of b-pull is usually more than that of push and pushM, since the gain incurred by avoiding message I/Os cannot offset the cost of accessing svertices and auxiliary data of fragments. However, hybrid can optimize it by switching adaptively.

6.2 Analyzing hybrid during Iterations

We validate the effectiveness of hybrid using the same setting in Figs. 8 and 9. We first explore the features of the performance metric Q^t, including the prediction accuracy and the impact of hardware characteristics. After that, a detailed analysis is given to show the resource requirements of hybrid. Except for the impact of hardware characteristics, other experiments are run on the *local cluster*.

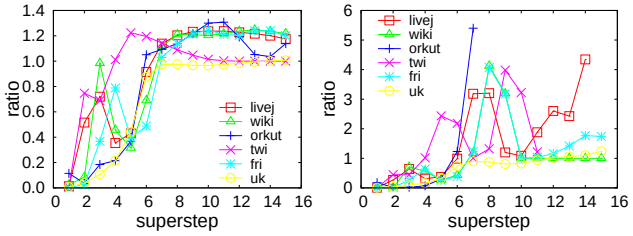
Testing the prediction accuracy of Q^t. The accuracy of predicting the performance metric is determined by C_{io}(push), C_{io}(b-pull) and $\mathcal{M}_{co}$. Figs. 11-13 report the prediction accuracy of them for SSSP and SA. The y-axis shows the ratio of the predicted value (collected at the t-th superstep) to the actual value (collected at the (t+2)-th superstep), since the switching interval is 2. The distance between the ratio and 1 is inversely proportional to the accuracy. We observe that the accuracy of SA is low, especially during supersteps 6-10. The reason is that the number of active vertices suddenly varies with iterations. For C_{io}(push), considering that edges are organized in blocks, even though one vertex is active, all edges

in the corresponding edge block should be accessed. The I/O cost is thereby not sensitive to the variation of active vertices, which partially offsets the impact of the sudden change of message I/Os, based on Eq. (7). As a result, C_{io}(push) has an extremely high accuracy. The accuracy of C_{io}(b-pull) is further boosted, because C_{io}(b-pull) does not include message disk I/Os (Eq. (8)).



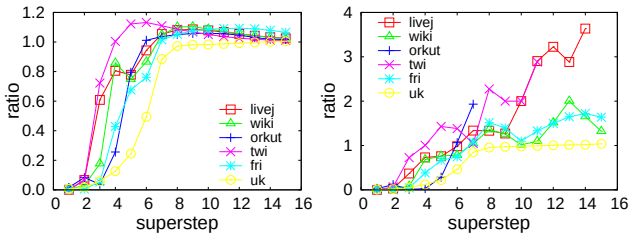
(a) SSSP (b) SA

Figure 11: The prediction accuracy of $\mathcal{M}_{co}$



(a) SSSP (b) SA

Figure 12: The prediction accuracy of C_{io}(push)



(a) SSSP (b) SA

Figure 13: The prediction accuracy of C_{io}(b-pull)

Evaluating the impact of hardware characteristics on Q^t. We explore the impact by running SSSP over twi, since hybrid achieves the most gain in this case, and other cases exhibit the similar phenomenon. Fig. 14 (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 11th superstep, and the other happens at the 26th superstep. For the two clusters, we observe that the switching points do not change. A straightforward explanation is that both b-pull and push can benefit from the fast random read/write performance of SSDs. From the perspective of Q^t, when the sign of Q^t changes, the number of messages is usually small. As listed in Table 3, s_{net} is significantly more than $s_{sr}/s_{rd}/s_{rw}$. Thus, the impact of the communication volume can be ignored. Furthermore, 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 $\big(C_{io}(\text{push}) - C_{io}(\text{b-pull})\big)$ which only relies on the graph topology and the special graph algorithm, and is orthogonal to the hardware characteristics. However, the absolute value of Q^t, $|Q^t|$, indicates the expected gain, which is affected by the hardware. For example, when $Q^t < 0$, $|Q^t_{HDD}| > |Q^t_{SSD}|$. That means, for the *local cluster*, switching from b-pull to push can achieve more gains, which is validated in Fig. 8 (b) and Fig. 9 (b). Taking SSSP over orkut and twi as examples, for HDDs, the gains of hybrid compared with b-pull are 19% and 38%, respectively. However, for SSDs, the gains are only 10% and 35%, respectively.

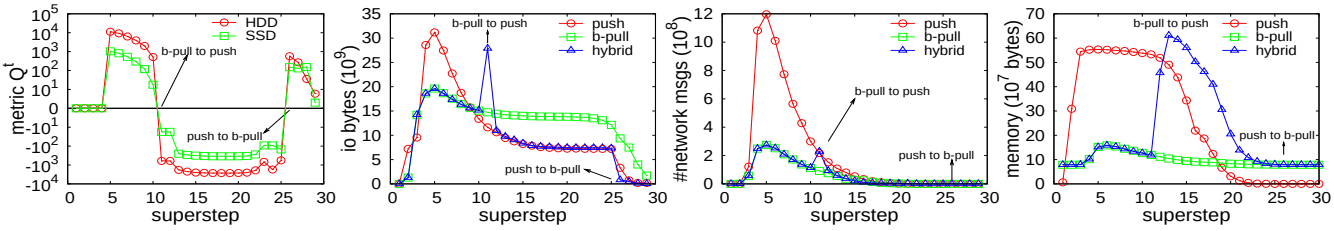
Analyzing resource requirements of hybrid. In this suite of experiments, we analyze resource requirements by running SSSP over twi. Here we ignore the report of SA since it has a similar performance with SSSP. We use push and b-pull as compared solutions, as hybrid always chooses one of them to execute iterative computations. Figs 14 (b)-(d) report the change of I/O-pressure, network communication costs, and memory usage. Generally, compared with push and b-pull, hybrid does not incur extra resource requirements if the switching operation does not happen, even though it maintains two replicas of edges. This is because when executing push/b-pull, hybrid only accesses edges in the adjacency list/Eblocks. However, in the case of switching from b-pull to push (at the 11th superstep), the resource requirements increase, because hybrid must pull messages from svertices, while simultaneously pushing new messages to dvertices. That means, at the 11th superstep, hybrid also processes messages which should have been handled at the 12th superstep in push. Although shifting the message processing does not incur extra 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, hybrid takes 22.944s to accomplish the computation at superstep 11. Compared with the sum of 17.512s (b-pull at superstep 11) and 5.01s (push at superstep 12), the performance degradation does not exceed 2%. This can be easily offset by the switching gains (70%/38%, compared with push/b-pull). By contrast, when switching from push to b-pull at superstep 26, messages that should have been generated and pushed at this superstep will be pulled at superstep 27. The resource requirements will not increase. Note that the memory usage of hybrid is more than that of push, even though switching from b-pull to push has been done at superstep 11. The is because hybrid always needs to maintain the metadata information of VE-BLOCK (used by b-pull).

6.3 Scalability

We test the scalability of PageRank in Fig. 15 using the-state-of-the-art push-based and pull-based approaches: pushM and hybrid. 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 b-pull integrated into hybrid. The former is much more expensive than the latter. Thus, we observe a super-linear performance degradation for pushM, when reducing the number of computational nodes. By contrast, the runtime increases sub-linearly for hybrid.

6.4 The performance of loading graph data

Fig. 16 reports the performance of loading graph data in *HybridGraph* using three data structures *adj* (i.e., adjacency list, used by push), VE-BLOCK (used by b-pull), and *adj*+VE-BLOCK (used by hybrid). We measure the performance in terms of runtime and I/O bytes written onto local disks. The y-axis indicates the performance ratio of *adj*/VE-BLOCK/*adj*+VE-BLOCK to *adj*. Building VE-BLOCK increases the runtime of loading data, as each adjacency list must be parsed into fragments (computation-intensive) and additional auxiliary data of fragments are written onto disk (I/O-intensive). Furthermore, compared with VE-BLOCK, although *adj*+VE-BLOCK keeps a replica of edges in an adjacency list, the overall loading runtime just slightly increases, due to the fast sequential write. Generally, for b-pull and hybrid, the extra cost in loading data can be easily offset by the performance gains during computation, as shown in Fig. 8 and Fig. 9.

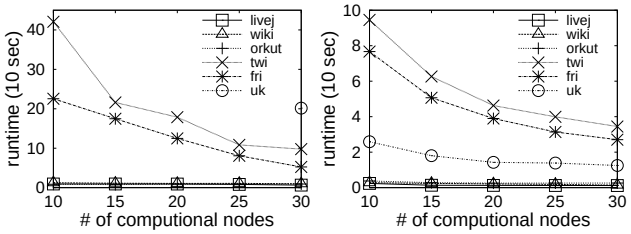


(a) Performance metric (b) Disk I/O costs (c) Network costs (d) Memory usage

Figure 14: Analyzing Q^t, I/O costs, network costs, and memory usage for hybrid (SSSP **over** twi, ***limited memory***)

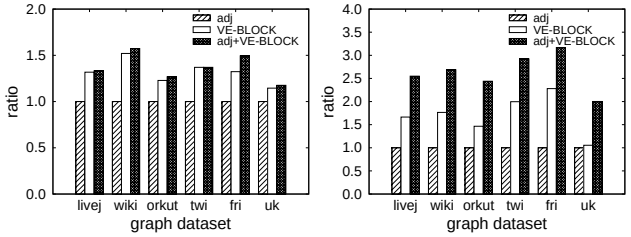
6.5 Blocking time and network traffic

This section analyzes the efficiency of push and b-pull 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. 7 (a). Fig. 17 shows the average value and fluctuant range (min-max) for blocking time over wiki and orkut datasets. Note that b-pull starts exchanging messages from the 2nd superstep.



(a) pushM (PageRank) (b) hybrid (PageRank)

Figure 15: Scalability of computations (*local cluster*)



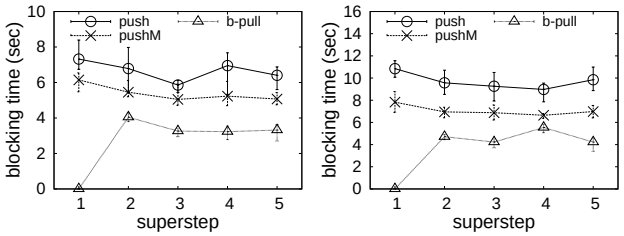
(a) Runtime (b) I/O costs

Figure 16: The performance of loading graph in *adj*, VE-BLOCK **, and *adj*+**VE-BLOCK **, on the *local cluster***

Fig. 18 shows the network traffic for push and b-pull. The network traffic includes all input and output on bytes, and is extracted by GANGLIA [11], a cluster monitoring tool, where the monitoring interval is for every 2 seconds. In particular, we disable the combining function of b-pull, 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.

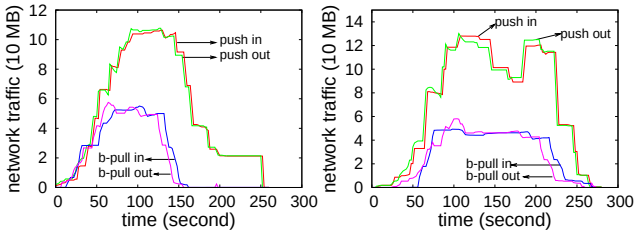
It is worth noting that in push and pushM, all distributed tasks will produce and send messages in parallel, whereas in b-pull, the message operation is triggered by pulling requests from destination vertex blocks. Our tests show b-pull offers a comparable parallelism and superior communication efficiency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

[11]Ganglia. http://ganglia.sourceforge.net/



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

Figure 17: Blocking time: push **vs.** b-pull



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

Figure 18: Network traffic: push **vs.** b-pull

7. CONCLUSION

This paper proposes a new adaptive and I/O-efficient message processing mechanism based on the system *HybridGraph* we have developed for vertex-centric computing on Cloud. A large number of messages generated in iterations have to be stored on disk when memory is not sufficient, and it is most likely to happen when graphs become larger and larger. 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 cost reduction shifts to the sender side, and becomes the I/O cost of reading a graph. Therefore, by effectively organizing a graph beforehand, we design a new pulling approach b-pull to reduce I/O costs. Furthermore, to obtain optimal performance, push and b-pull are seamlessly combined in our hybrid framework and adaptively switched according to the variation in message scale. In experiments, we show that our proposed b-pull and hybrid methods can obviously outperform the up-to-date push-based and pull-based methods for computations of disk-resident graph data.

Acknowledgements. This work is supported by the National Basic Research Program of China (973 Program) under Grant No. 2012CB316201, the National Natural Science Foundation of China (61433008, 61472071, and 61272179), Research Grants Council of the Hong Kong SAR, China No. 14209314 and 418512, and China Scholarship Council. Authors are also grateful to anonymous reviewers for their constructive comments. Yu Gu is the corresponding author of this work.

8. REFERENCES

[1] Faunus. http://thinkaurelius.github.io/faunus/.

[2] Giraph. http://giraph.apache.org/.

[3] Hama. https://hama.apache.org/.

[4] Y. Bu, V. Borkar, J. Jia, M. J. Carey, and T. Condie. Pregelix: Big (ger) graph analytics on a dataflow engine. *Proc. of the VLDB Endowment*, 8(2):161–172, 2014.

[5] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst. Haloop: efficient iterative data processing on large clusters. *Proc. of the VLDB Endowment*, 3(1-2):285–296, 2010.

[6] R. Chen, X. Weng, B. He, and M. Yang. Large graph processing in the cloud. In *Proc. of SIGMOD*, pages 1123–1126. ACM, 2010.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] U. Kang, C. E. Tsourakakis, and C. Faloutsos. Pegasus: A peta-scale graph mining system implementation and observations. In *ICDM*, pages 229–238. IEEE, 2009.

[15] 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.

[16] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein. Distributed graphlab: a framework for machine learning and data mining in the cloud. *Proc. of the VLDB Endowment*, 5(8):716–727, 2012.

[17] 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.

[18] 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.

[19] 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.

[20] S. Salihoglu and J. Widom. Gps: A graph processing system. In *Proc. of SSDBM*, page 22. ACM, 2013.

[21] Z. Shang and J. X. Yu. Catch the wind: Graph workload balancing on cloud. In *Proc. of ICDE*, pages 553–564. IEEE, 2013.

[22] 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.

[23] I. Stanton and G. Kliot. Streaming graph partitioning for large distributed graphs. In *Proc. of SIGKDD*, pages 1222–1230. ACM, 2012.

[24] S. Tasci and M. Demirbas. Giraphx: parallel yet serializable large-scale graph processing. In *Euro-Par 2013 Parallel Processing*, pages 458–469. Springer, 2013.

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

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

[27] 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.

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

[29] 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.

[30] Z. Yang, J. Xue, Z. Qu, S. Hou, and Y. Dai. Seraph: An efficient system for parallel processing on a shared graph, 2013.

[31] J. Yin and L. Gao. Scalable distributed belief propagation with prioritized block updates. In *Proc. of CIKM*, pages 1209–1218, 2014.

[32] 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. ARCHITECTURE OF *HybridGraph*

Fig. 19 gives the overview architecture of *HybridGraph* which is implemented on top of Apache Hama 0.2.0-incubating. Like *Giraph*, *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, mainly including: 1) dividing a graph job into several tasks and scheduling them to computational nodes to run in parallel; 2) coordinating the progress of tasks through a synchronous barrier; 3) detecting the fault computational node and re-scheduling tasks on it. The fault tolerance is managed by writing/reading checkpoint files, resembling *Pregel*.

B. AN EXAMPLE USING b-pull

We discuss computing SSSP by our b-pull approach on two computational nodes, T_1 and T_2. Fig. 20 shows the VE-BLOCK for an example graph, where v_3 is the source vertex to compute. Given a vertex id v_i, val is the shortest distance, $|V_o|$ is the out-degree, and V_o is the out-neighbors of v_i. The vertices of the example graph G are partitioned into three Vblocks: $b_1 = \{v_1, v_2\}$, $b_2 = \{v_3, v_4\}$, and $b_3 = \{v_5\}$, and the edges of G are distributed into Eblocks accordingly. Here, b_i is with Eblocks, g_{i1}, g_{i2}, and g_{i3}. Suppose b_1 and b_2 with their Eblocks are assigned to the computational node

T_1, and b_3 with its Eblocks is assigned to T_2. X_i is the meta information of Vblock b_i, and is only maintained by the computational node to which b_i belongs. In particular, for instance, the bitmap in X_1 (100) indicates that the vertices in b_1 only have out-neighbors in Eblock g_{11}. Here, the res of X_2 is set to be 1 since the value of source vertex v_3 is 0 and should be sent to its out-neighbors.

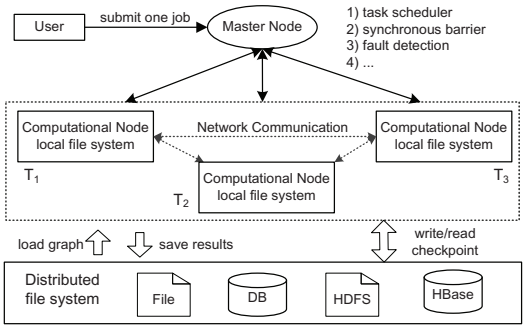


Figure 19: *HybridGraph* Architecture

We discuss the iterative process of push and our b-pull in Fig. 21. In the push-based approach, in the 1st superstep, the source-vertex v_3 updates its shortest distance to be zero and then pushes messages to its 3 out-neighbors v_2, v_4, and v_5. In the 2nd superstep, they update their shortest distances, and push messages to their out-neighbors. In every superstep, the messages will be stored at the receiver side to be used for the next superstep. The SSSP computation will terminate in 4 supersteps.

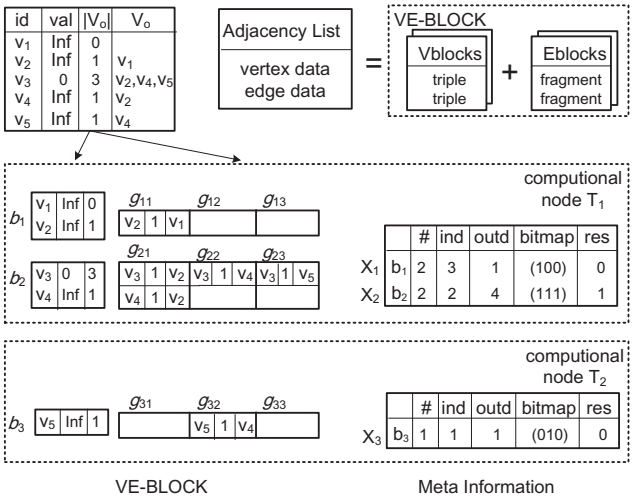


Figure 20: The VE-BLOCK structure

In our b-pull-based approach, in the 1st superstep, the source vertex v_3 only updates its value to be zero. There are no any messages sending. In the 2nd superstep, via pull requesting based on Vblock ids, v_2, v_4, and v_5 request messages to be sent from the vertex v_3, and then update their own values. Different from the push-based approach, the messages received in the same superstep will be consumed immediately. b-pull needs 5 supersteps, which is one more than push. But in the last one, and no data are transferred except block-centric pull requests. The cost is negligible, and we thereby state that b-pull and push have the same supersteps for a given algorithm in the synchronous computing model in Eq. (4).

Next, we further explain b-pull using Pull-Request (Algorithm 1) and Pull-Respond (Algorithm 2). Consider the 2nd superstep in our b-pull. The computational node T_1 sends pull requests for

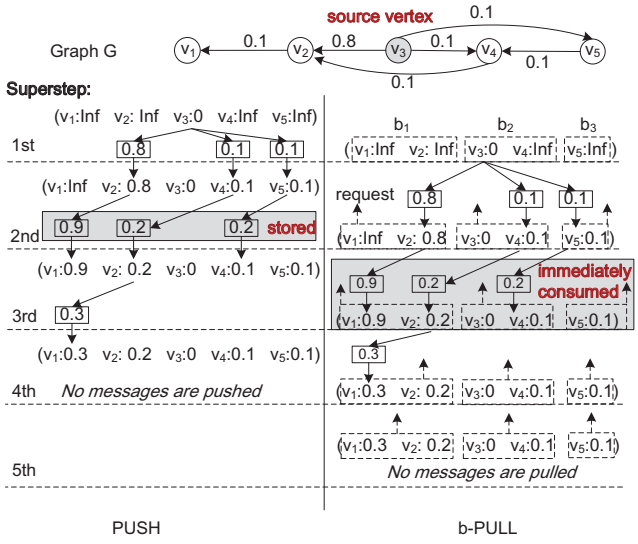


Figure 21: Illustration of SSSP in push and b-pull

Vblocks b_1 and b_2, and the computational node T_2 sends pull requests for Vblock b_3 simultaneously. We discuss the process for requesting b_1 in Fig. 22. The other are the same. As shown in Fig. 22, ① T_1 sends the pull request for b_1 to T_1 and T_2. ② T_1 receives the request from itself, and then checks the meta information for Vblocks T_1 holds. Vblock b_1 is skipped, as the res of X_1 is 0. However, the res of X_2 is 1 and the 1st bit in the bitmap in X_2 is 1, which indicates that some vertices in b_2 should broadcast the most recent values, and there exist edges from b_2 to b_1 in Eblock g_{21}. As a result, b_2 may produce messages to be sent to vertices in b_1. Then, v_3 and v_4 in b_2 are checked while scanning g_{21}. Since only v_3 is the responding vertex at the 1st superstep, we obtain the value of v_3 and the outgoing edge $(v_3, v_2, 0.8)$ with a weight 0.8. ③ T_1 invokes v_3.pullRes() to produce a message $(v_2, 0.8)$, and then puts it into the sub-buffer of BS_1. ④ Messages in sub-buffer are sent to T_1 and are kept in the receiving buffer BR_1. ⑤ When all messages have been pulled from T_1 and T_2, T_1 invokes update() for the active vertex v_2 in b_1 to update its value (v_2.update()). ⑥ The new value of v_2 is stored, and v_2 is marked as responding to respond pull requests at the next superstep.

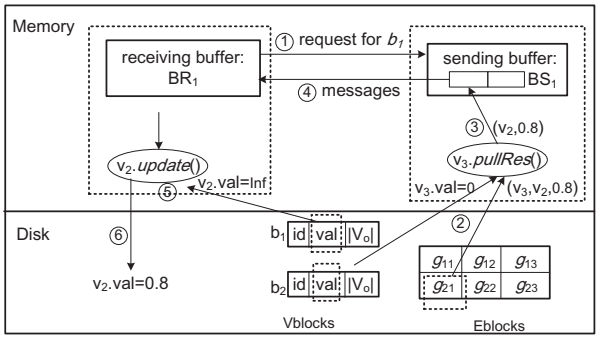
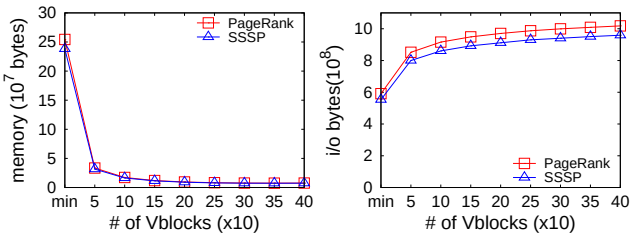


Figure 22: Message data-flow for b_1 on T_1 in the 2nd superstep

C. TESTING THE IMPACT OF THE VEBLOCK GRANULARITY

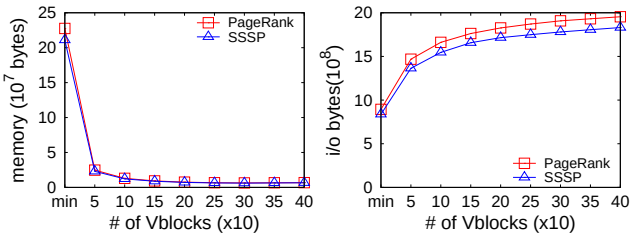
We conduct testing to explore the impact of $\mathcal{V}$ using PageRank

and SSSP over graphs livej and wiki described in Section 6. The testing is done using 5 computational nodes in our *local cluster*, where each node is with 6GB available memory. Fig. 23 and Fig. 24 show the curve of the memory requirement measured by summing up the size of messages in buffer and metadata in VE-BLOCK on each computational node (BR_i+BS_i+$Byte$(VE-BLOCK)), and I/O bytes, respectively, when varying the number of Vblocks. In these figures, 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 computational 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) and accessed on average. We also show the overall runtime when varying $\mathcal{V}$ in Fig. 25. We find that for SSSP there exists a turning point between $\mathcal{V}$=5 (min) and $\mathcal{V}$=100 ($x=10$), especially for wiki. 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 (in Figs. 23(b) and 24(b)), more redundant edges may be read from disk, as they are stored in the same Eblock together with the required edges.



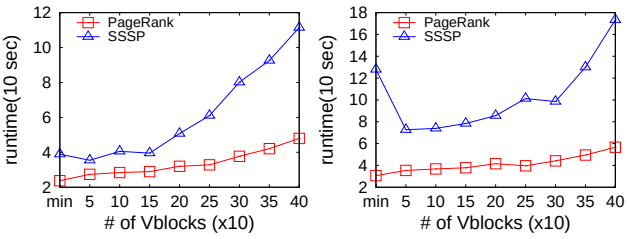
(a) Memory requirement (b) I/O bytes

Figure 23: Testing memory requirements and I/O bytes (PageRank **and** SSSP **over** livej)



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

Figure 24: Testing memory requirements and I/O bytes (PageRank **and** SSSP **over** wiki)



(a) livej (b) wiki

Figure 25: Testing runtime (PageRank **and** SSSP)

D. PROOF

The proof of Theorem 1.

PROOF. Let $F[\mathcal{V}]$ denote the number of fragments associated with u as svertex in Vblock, and $\mathbb{E}(F[\mathcal{V}])$ 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 $\mathbb{E}(F[\mathcal{V}])\propto\mathcal{V}$. □

The 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 b-pull, at each superstep, edges may be involved in update() to update vertex values, besides being used in pullRes() to broadcast messages. By contrast, they are accessed only once in push because compute() covers the logics of update() and pullRes(). When all vertices broadcast messages to their neighbors along outgoing edges, $|E^t| = |E| = \mathcal{M}$. Accordingly, in the worst case, $IO(\mathcal{E}^t) = 2IO(E^t) = 2S_e|E| = 2S_e\mathcal{M}$. Second, the I/O bytes of accessing disk-resident messages in push is: $2(\mathcal{M}-B)S_m$. For b-pull, considering that $S_m\geq S_v$ and $S_m\geq S_f$, we have:

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

Finally, suppose $B\leq(\frac{|E|}{2}-f)$, we can infer that:

$$C_{io}(\text{push})-C_{io}(\text{b}-\text{pull}) \geq 2S_m(|E|-B-f)-S_e|E| \geq 0$$

, because $S_m\geq S_e$. □

E. THE EFFECTIVENESS OF COMBINING

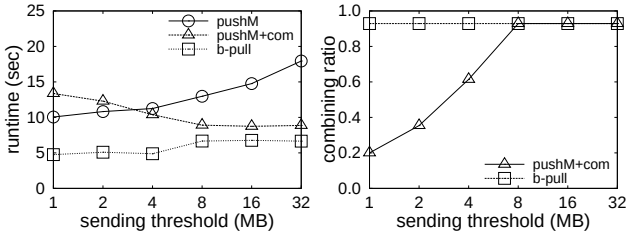
Distributed systems usually set a sending threshold to control the communication behavior, in order to make full use of the network idle time and reduce the overhead of building connections. Assume that the threshold is 2. As shown in Fig. 5 (a), for push, s_1 generates two messages $m(s_1,d_1)$ and $m(s_1,d_2)$ 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 combining. By contrast, for pull (including b-pull), 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 obviously guarantees that all messages for d_1 can be combined.

We modify *MOCgraph* to support combining messages at the sender side. The modified version is identified as pushM +com.

Table 5: Comparisons of runtime in five scenarios about *GraphLab PowerGraph* (seconds)

scenarios	PageRank			SSSP			LPA			SA		
	livej	wiki	orkut	livej	wiki	orkut	livej	wiki	orkut	livej	wiki	orkut
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

We define the combining ratio as $\frac{\#\_of\_combined\_messages}{\#\_of\_total\_messages}$. Not surprisingly, as shown in Fig. 26 (a) (in the same setting used in Fig. 7 (a)), when varying the sending threshold from 1MB to 32MB, the runtime of pushM increases because a large threshold cannot make full use of the network idle time. By contrast, pushM +com works well, as many messages can be combined (i.e., a large combining ratio shown in Fig. 26 (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 b-pull, the communication gain is orthogonal to the threshold. Consequently, the only challenge for b-pull is to set a reasonable threshold to make full use of network resources. Fortunately, this constraint is relatively loose. As shown in Fig. 26, b-pull works well from 1MB to 4MB. This paper thereby uses 4MB as the default sending threshold.



(a) Runtime (b) Combining Ratio

Figure 26: Testing the effectiveness of combining for pushM and b-pull (PageRank over orkut)

F. EVALUATING THE PERFORMANCE OF MODIFIED *GraphLab PowerGraph*

In this suite of experiments, we use five testing scenarios to confirm that our extension does not incur extra costs compared with the original *GraphLab PowerGraph*. 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*: edges reside on disk and each task caches 2.5 million vertices at most in memory.

Table 5 reports the performance in five scenarios using 5 computational nodes of the *local cluster*. Apparently, *ext-mem* achieves a comparable performance with the original *GraphLab PowerGraph*, which validates that our extension is reasonable. Furthermore, 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 overall performance seriously degrades, due to frequently reading/writing disk-resident vertices, even though we have employed the LRU replacing strategy.

G. DETAILED DISCUSSION ON THE BOUNDARY OF hybrid

The switching mechanism of our hybrid relies on the sign of the performance metric Q^t which is determined by sub-metrics C_{io}(push), C_{io}(b-pull) and $\mathcal{M}_{co}$. We predict the three sub-metrics using the mechanism proposed by Shang et al. [21]. That is, metrics collected in the t-th superstep can be used to predict those in the (t+2)-th superstep. Shang et al. [21] divide well-known graph algorithms into three categories based on the change of active vertices, namely, *Always-Active-Style*, *Traversal-Style*, and *Multi-Phase-Style*. We discuss the boundary of hybrid accordingly.

Always-Active-Style: Every vertex in each superstep sends messages to all its neighbors. The typical algorithm is PageRank. We can always achieve the predicted Q^{t+2} accurately, since the behavior of vertices does not change. hybrid can choose an optimal mechanism based on Q^{t+2}.

Traversal-Style: Some vertices are treated as starting points, and the other ones are involved in computing based on the user-defined processing logic. SSSP follows this style. Not surprisingly, when the number of active vertices is drastically changing, the prediction mechanism proposed by Shang et al. [21] works poorly. Specifically, when the active vertex scale is decreasing, we have $Q^t > Q^{t+2}$, because Q^t is proportional to the number of messages. If we use Q^t as the predicted value of Q^{t+2}, the switching timing from b-pull to push (i.e., the sign changes from positive to negative) will be put off. Nevertheless, hybrid still achieves an impressive gain in runtime. This is because the sign of Q^t keeps unchanged in the subsequent n supersteps (due to a continuous decrease of the number of active vertices), and then hybrid can always benefit from the switching operation. Here, n depends on the specified algorithm and the graph topology. Take SSSP over twi as an example. n=15 after switching from b-pull to push (as shown in Fig. 14 (a)). Similarly, switching from push to b-pull will be put off when the number of active vertices is increasing. In conclusion, the total gain in runtime is up to 37.6%.

Multi-Phase-Style: The entire computation is divided into a number of phases, and each phase needs several supersteps. The behavior of vertices changes repeatedly among phases. That means the increase and decrease of the active vertex scale will occur periodically. Like *Traversal-Style*, the prediction accuracy is poor. But the difference is that the sum of gains after executing the delayed switching is negligible, because the sign of Q^t changes frequently. Taking the minimum spanning tree algorithm as an example [21], n=1 after switching from b-pull to push.

In conclusion, our hybrid works well for *Always-Active-Style* algorithms. For *Traversal-Style* algorithms, the gain is still impressive, although the prediction accuracy is poor. Finally, the current hybrid is not suitable for *Multi-Phase-Style* algorithms. For the last one, a possible solution is to analyze the historical information of iterations to explore the change style, and then guide the switching operation. We will validate the effectiveness of this solution as future work.