HGraph: I/O-efficient Distributed and Iterative Graph Computing by Hybrid Pushing/Pulling

Zhigang Wang, Yu Gu, Yubin Bao, Ge Yu, *Senior Member, IEEE,* Jeffrey Xu Yu, *Senior Member, IEEE,* and Zhiqiang Wei, *Member, IEEE*

Abstract—In the big data era, distributed computation is becoming a preferred solution for iterative graph analysis. However, graphs are rapidly growing in size and more importantly, there exist a lot of messages across iterations. For better scalability, many distributed systems keep graph data and message data on disk. Now these systems solely employ either pushing or pulling mode to manage data, but neither can always work well during the entire computation. This is mainly because I/O access patterns are dynamic and complex. This paper proposes a hybrid solution. It achieves the optimal performance in different scenarios by dynamically and adaptively switching modes between pushing and pulling. Specifically, we first devise a new block-centric pulling technique. It pulls messages much more I/O-efficiently than the existing vertex-centric pulling mode. We then combine pushing and pulling. For general-purpose, we categorize graph algorithms and accordingly present two seamless switching frameworks. We also design performance prediction components specialized to the two frameworks, to decide how and when we can switch modes. Some optimization strategies are also given to further enhance performance, such as priority scheduling and lightweight fault-tolerance. Extensive experiments against state-of-the-art solutions confirm the effectiveness of our proposals.

Index Terms—I/O-efficient, distributed iterative graph computing, pushing, pulling.

1 Introduction

NOWADAYS, most graph analysis jobs are iterative in nature over very large input data. Many distributed systems are then developed to cope with the performance challenge. *Pregel* [1] by Google as one of the early representative pioneers employs a Master-Slave framework (Fig. 1), where Master divides a job into several tasks running on computational nodes (Slaves) in a cluster. Typically, tasks first load and partition the graph in parallel. Then computations proceed through iterations (or "supersteps") to repeatedly update vertices and exchange intermediate results (messages). Supersteps are separated by global barriers to coordinate the progress. Besides, a node is referred to as the sender/receiver side when the task on it is sending/receiving messages.

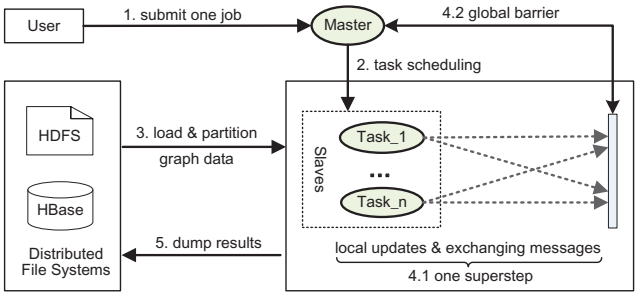


Figure 1. Illustration of distributed and iterative graph processing

Motivation: *Pregel* has been driving much of the research on enhancing performance, like partitioning [2], [3], communication [4], [5], and convergence [6]. This paper focuses on I/O-efficiency, as the memory resource can be easily exhausted with the drastically growing rate of graph data and the proportional increase of message data. Adding new physical nodes of course can alleviate memory pressure, but is not always economically feasible. By contrast, external storage is significantly cheaper than memory storage. As a result, a lot of efforts have been devoted into disk-based systems [8], [9], [10], [11], [12], [13], [14], [15].

Problem analysis: Almost all distributed graph systems take a pushing mode where messages from one source vertex (svertex) as a whole are generated and then actively pushed to destination vertices (dvertices). The advantage is that every svertex is accessed one time at most per superstep. However, dvertices will not consume messages until the next superstep so that they can receive required messages from all in-neighbors. Then messages might end up being kept on disk due to the large size, which incurs a lot of random writes. Another preferred mode is to pull messages from svertices on demand for each scheduled dvertex. Messages can be consumed immediately without possible writes. However, one svertex might be read multiple times if it is the common in-neighbor of several dvertices. That leads to considerably large costs of random reads since graph data usually reside on disk due to the rapidly growing data volume. For example, we run PageRank over a benchmark graph Orkut (see ork in Table 2). We find that the runtime per superstep increases from 12 to 116 seconds for pushing when many messages are stored on disk; and from 5 to 1305 seconds for pulling if most vertices reside on disk.

Challenges: Pushing and pulling are respectively sensitive to disk-resident message data and graph data. In the big data era,

- ∗*This work was partially done at Northeastern University.*
- *Z. Wang and Z. Wei (✉) are with the College of Information Science and Engineering, Ocean University of China, Qingdao, Shandong, China. E-mail: {wangzhigang,weizhiqiang}@ouc.edu.cn.*
- *Y. Gu (✉), Y. Bao and G. Yu are with the School of Computer Science and Engineering, Northeastern University, Shenyang, Liaoning, China. E-mail: {guyu, baoyubin, yuge}@mail.neu.edu.cn.*
- *J. X. Yu is with the Department of Systems Engineering and Engineering Management, the Chinese University of Hong Kong, Hong Kong, China. E-mail: yu@se.cuhk.edu.hk.*

Manuscript received October 30, 2018; revised XX XX, XXXX.

both kinds of data are usually on disk. Together with the fact that the number of messages might change with iterations, the performance comparison between pushing and pulling is non-deterministic. We then pursue a hybrid solution where the two modes can be switched adaptively if necessary. However, this is a non-trivial task. The reasons are twofold. First, directly combining pushing and pulling is not cost effective since the latter suffers from costly random reads. Second, to gain optimal performance, some important issues like switching efficiency and switching time need to be explored, which is especially difficult in a general system because graph algorithms have very different behaviors.

Our contributions: This paper explores a path to such a target hybrid solution. We first challenge the conventional wisdom that messages are pulled in a vertex-centric manner. Instead, we design a new Block-centric Pulling mode called BPull to reduce the number of random reads. It partitions vertices into several blocks so that vertices as dvertices in the same block can share a single request to pull messages from svertices. More importantly, the outgoing edges of a svertex are further divided into fine-grained subsets based on the distribution of dvertices among blocks. By this design, edges in a subset will be read as a whole to generate messages, and hence, the corresponding svertex is read only once.

Second, the existing pushing mode and our BPull are decoupled into a series of functions. By sharing the common vertex update operation, we can re-organize the execution order of such functions to smoothly switch modes. Further, to seek a proper switching point, a naturally desirable solution is to predict runtimes of the two modes, and then select an optimal one. More specifically, we dynamically compute the performance difference from network communication and disk I/O perspectives, in order to infer the runtime comparison result. For better accuracy, we basically divide graph algorithms into two categories based on the variation degree of runtimes among iterations. Accordingly, two prediction components are designed and respectively used in a Basic Hybrid-Switching framework BHS and a Generalized variant GHS. GHS is more general than BHS at the expense of requiring additional offline knowledge and modifications on the pushing mode.

We finally give two optimization strategies to further boost the overall performance. (1) Priority scheduling. Different nodes in BPull can freely send requests and the computation of a node cannot proceed until all required requests have been responded to. However, the default responding order is FIFO (first-in, first-out), which usually generates considerable waiting costs for nodes. Different from FIFO, now we continuously monitor the responding status of each node. Whenever necessary, we will dynamically prioritize requests so that requests urgently required by a node can immediately preempt the responding threads. This design thereby maximally unleashes the computational power. (2) Lightweight fault-tolerance. It is most likely that some nodes fail in a cluster. Existing fault-tolerance techniques [1], [16] can accelerate failure recovery but at the expense of archiving a lot of messages in the *failure-free* (no failure) execution. Instead, we propose to log some lightweight yet important metadata. Upon failures, such data can be used by our BPull to quickly re-generate messages required in recovery. As a result, our solution can provide competitive recovery efficiency with nearly-zero *failure-free* performance penalty.

Overall, benefitting from all techniques mentioned above, the resulting prototype system *HGraph* achieves up to 19x speedup, in comparison with up-to-date push- and pull-based systems.

We state that this paper extends a preliminary work [17] in the following aspects. (1) We design a priority scheduling policy to further enhance the performance of BPull. (2) We investigate the data access patterns of different graph algorithms and then propose a generalized variant GHS of BHS. (3) We present an efficient lightweight fault-tolerant method by fully utilizing BPull. (4) To show the generality and advantage, more graph algorithms and the most recently published system [14] are tested.

The remainder of this paper is organized as follows. Section 2 provides necessary background on pushing and pulling modes. Section 3 presents our novel BPull and Section 4 further introduces the basic and generalized hybrid solutions. Section 5 elaborates the priority scheduling and fault-tolerant optimizations. Section 6 contains a thorough experimental study. Section 7 overviews related works and Section 8 concludes this paper.

2 Overview of Push and Pull

We model a graph as a directed graph $G = (V,E)$, where V is a set of vertices and E is a set of edges. For an edge (u, v), u is the source vertex denoted as svertex, and v is the destination vertex denoted as dvertex. 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 [13]. This paper thereby assumes graph data reside on disk. Without loss of generality, we assume each node runs only one task and then discuss communication among nodes. Now we introduce existing pushing (Push) and pulling (Pull) modes.

Push: We use *Giraph* [9] as an example Pregel-like system. In one superstep, every vertex u in V is selectively scheduled to compute a user-defined function compute() (see Eq. (1)) in parallel, if u receives messages. For simplicity, let $M_I(u)$ and $M_O(u)$ denote the messages received and sent to/from u.

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

An implementation of a graph algorithm, PageRank, is given in Fig. 2(a). In the t-th superstep, u first gets an iterator msgs of $M_I^t(u)$ that keeps messages from in-neighbors in the (t-1)-th superstep (Line 2). After summing all messages, the vertex value is updated from u^t to u^{t+1} (Line 3). Then u sends u^{t+1} divided by its out-degree as messages $M_O^{t+1}(u)$ to all out-neighbors (Lines 5-6). $M_O^{t+1}(u)$ for every u forms M_I^{t+1} in superstep-(t+1). u *votes to halt* to terminate computations if the maximum number of supersteps, maxNum, has been reached. Fig. 3(a) gives an introduction to the data flow within a typical superstep t on one computational node. In *Giraph*, M_I^t and M_I^{t+1} are possibly spilled onto disk if a buffer overflow is detected.

```
1  Vertex.compute () {
2    Messages msgs = getRecMsg();
3    double newVal = 0.85*sum(msgs)
                + 0.15/getNumVertices();
4    setVal(newVal);
5    for (Edge e: getOutEdges().iterator())
6      sendMsg(e.Dst(), getVal()/outDeg());
7    if (getSuperStep() > maxNum)
8      voteToHalt(); }
```

(a) PageRank in Push

```
1  Vertex.pullRes (dvertex v) {
2    if (isRespond() is True)
3      get the edge e=(u, v); //u is svertex
4      sendMsg(e.Dst(),getVal()/outDeg());}
5  Vertex.update () {
6    Messages msgs = getRecMsg();
7    double newVal = 0.85*sum(msgs)
                + 0.15/getNumVertices();
8  setVal(newVal);
9    if (getSuperStep() > maxNum)
10     voteToHalt();
11   else
12     voteToRespond(); }
```

(b) PageRank in Pull

Figure 2. Algorithm implementation in Push and Pull

Pull: Unlike Push, Pull shifts the generation of messages in M_O^t or M_I^t from superstep (t-1) to t. That is, at superstep-t, the

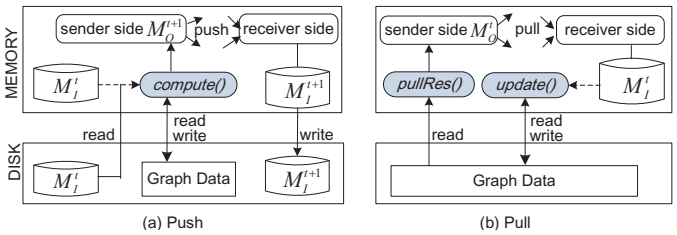


Figure 3. Data flow in Push and Pull at superstep-t

required messages are not available at the receiver side. Instead, they are pulled from svertices. Thus, Pull decouples compute() into two functions pullRes() and update(), as shown in Eq. (2) and Eq. (3). Fig. 2(b) shows the implementation of PageRank using Pull. Specifically, pullRes is run at the sender side to reactively respond to the pull request from v as dvertex, i.e., reading vertex u^t as svertex to generate required messages in $M_O^t(u)$ or $M_I^t(v)$ (Lines 1-4). While, update takes pulled messages from u's in-neighbors to update u^t (now as dvertex) at the receiver side. In particular, at superstep-(t-1), u indicates that it will respond to requests by invoking *voteToRespond* (Line 12). We outline the data flow of Pull in Fig. 3(b). Because messages in M_O^t or M_I^t are consumed immediately, there is no disk I/O for messages.

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

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

In particular, u is called as an **active vertex** if it is processed by compute() or update(). Similarly, it is a **responding vertex** if it generates messages in compute() or pullRes(). Let V_{act} and V_{res} be, respectively, the sets of active and responding vertices. Then before invoking compute(), Push loads outgoing edges in advance based on V_{act}. While this is done based on V_{res} for Pull.

Performance study: We then analyze the total runtime cost C for Push and Pull. Shown in Eq. (4), $\mathcal{N}$ is the number of supersteps. In one superstep, we use C_{cpu}, C_{net}, and C_{io}, to represent the CPU cost, communication cost, and disk I/O cost, respectively. Because both modes have the same compute workload and $\mathcal{N}$, the performance difference is dominated by C_{net} and C_{io}. For simplicity, we directly study network bytes and disk I/O bytes.

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

We first discuss C_{net}. Pull takes additional time to send pull requests. However, all messages regarding on a request are for the same dvertex. The good locality enables effective and efficient message combination/concatenation (see Section 3.2). In contrast, most push-based systems disable this optimization because of poor locality [4], [9]. Thus, the great communication gain can easily offset costs caused by sending pull requests. We then consider C_{io}. In Push, messages are carried across two consecutive supersteps and may end up being kept on disk, introducing excessive I/O costs. Pull naturally solves this problem but generates many random accesses to svertices when responding requests.

In conclusion, Push and Pull have different strengths. Our goal is to design hybrid solutions (Section 4) to adaptively choose a profitable mode. But before that, we introduce a block-centric technique in Section 3 to improve the I/O-efficiency of Pull.

3 BPULL: BLOCK-CENTRIC PULLING

This section describes BPull. We first present a data structure called VE-BLOCK, and then give the details on how to pull messages using VE-BLOCK. Finally, we discuss some key parameters in VE-BLOCK to enhance the efficiency of BPull.

3.1 Efficient graph storage VE-BLOCK

VE-BLOCK consists of two components: Vblocks for vertices and Eblocks for edges. Consider adjacency lists for representing a graph where every vertex keeps a quadruple $(id, val, |V_o|, V_o)$. Here 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, we simply range-partition [9] all vertices into $\mathcal{V}$ Vblocks, b_1-$b_\mathcal{V}$, in total[1]. A Vblock keeps a list of triples $(id, val, |V_o|)$. Given b_i, we have $\mathcal{V}$ Eblocks, g_{i1}-$g_{i\mathcal{V}}$, to maintain outgoing edges. In particular an edge (u, v) with u in b_i and v in b_j falls into g_{ij}. Further, 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.

For better performance, VE-BLOCK has built-in **Metadata**—X_j for b_j and Y_{ji} for g_{ji}. X_j keeps six items: the number of svertices in b_j (#), the total in/out-degree (*ind/outd*) of svertices, a bitmap x_j, a responding indicator (*res*), and the size of this block. In particular, the i-th bit in x_j is set if there exist edges directed from b_j to b_i. *res* is "true" if at least one svertex needs to respond to pull requests. Y_{ji} records the disk file size of g_{ji} and the numbers of edges, fragments, svertices, and dvertices.

Fig. 4 shows VE-BLOCK for an example graph. Vertices are partitioned into three Vblocks kept by two nodes T_1 and T_2. Edges are distributed into Eblocks accordingly. For example, (v_3, v_2) is assigned into g_{21} because v_3 belongs to b_2 and v_2 belongs to b_1. The bitmap in X_2 (111) indicates that vertices in b_2 have out-neighbors in the total three Eblocks g_{21}-g_{23}. Further, take g_{21} as an example. There exist two edges with two svertices $\{v_3, v_4\}$ and one dvertex $\{v_2\}$, as shown in Y_{21}.

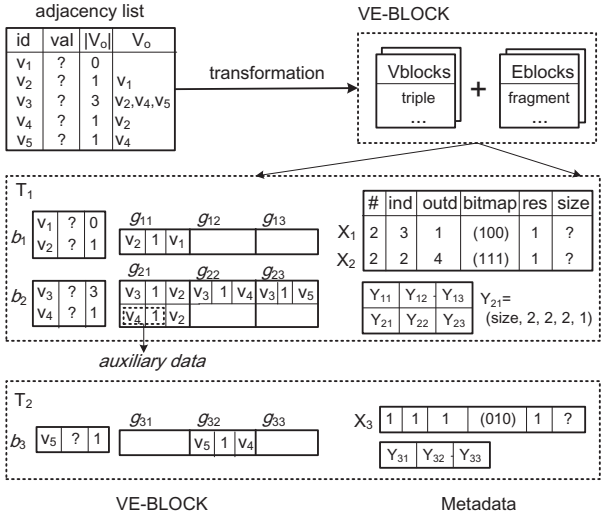


Figure 4. The VE-BLOCK structure

3.2 Pull requesting and pull responding

In order to pull messages, the requesting operation (Pull-Request) is performed at the receiver side that requests messages to be consumed; while, the responding operation (Pull-Respond) is performed at the sender side that generates messages on demand.

1. Any other partitioning method can also be used.

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

Algorithm 1: Pull-Request

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

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

Algorithm 2: Pull-Respond

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

Note that several svertices can generate messages to the same dvertex v. In Alg. 2, these message values can be concatenated to share the same id of v, to reduce communication costs. Further, if they are commutative and associative [1], they can be combined into a single one (Combiner). Similarly, in Alg. 1, messages can also be concatenated or combined to save memory space.

3.3 Determining the number of Vblocks

Clearly, the number of Vblocks, $\mathcal{V}$, determines the granularity of sending pull requests, and hence becomes critical for performance. We then discuss $\mathcal{V}$ in terms of memory usage and I/O cost.

Memory usage: The memory of a node T_i is mainly used by two buffers: BR_i for receiving messages in Pull-Request and BS_i for sending messages in Pull-Respond. Suppose T_i keeps a set V_i of vertices. *Giraph* (Push) uses B_i as the maximum number of messages in memory on T_i[2], i.e., available memory. Given B_i, we then analyze how to compute a proper Vblock granularity $\mathcal{V}_i$.

Suppose there exist $\mathcal{T}$ nodes. BS_i is divided into $\mathcal{T}$ sub-buffers, as these nodes may send pull requests to T_i at the same time. In fact, we can easily infer that both BS_i and BR_i are inversely proportional to the total number of Vblocks $\mathcal{V} = \sum \mathcal{V}_i$. This is because a large $\mathcal{V}$ can decrease the number of vertices in a Vblock. Then the number of messages temporarily kept in the sub-buffer on T_i and further received at the receiver side decreases.

For algorithms supporting Combiner, when dealing with a pull request initiated by b_x from T_j, T_i will not flush out messages until all of them have been produced. The goal is to thoroughly combine messages before sending to achieve high communication gains. Clearly, combination bounds the number of messages to $\frac{|V_j|}{\mathcal{V}_j}$. Thus, $\mathsf{BS}_i = \sum_{j=1}^{\mathcal{T}} \frac{|V_j|}{\mathcal{V}_j}$. On the other hand, messages for b_{x+1} are pre-pulled in an asynchronous fashion when vertices in b_x are being updated. We employ this design to reduce the waiting time. Then $\mathsf{BR}_i = 2\frac{|V_i|}{\mathcal{V}_i}$. Together, we compute every $\mathcal{V}_i$ by solving equations shown in Eq. (5) for $1 \le i \le \mathcal{T}$.

$$\mathsf{B}_i = \sum_{j=1}^{\mathcal{T}} \frac{|V_j|}{\mathcal{V}_j} + \frac{2|V_i|}{\mathcal{V}_i} \quad (5)$$

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

For concatenating algorithms, buffering all messages largely increases the memory usage. Hence, we immediately flush out messages whenever necessary, resembling *Giraph*. Then the size of BS_i can be ignored. Now B_i equals BR_i whose size is affected by the total in-degree of one Vblock. Pre-pulling is also disabled due to the memory usage bottleneck. We then set $\mathcal{V}_i$ using Eq. (6).

I/O costs: In BPull, Pull-Respond needs I/O costs to read the svertex value and auxiliary data for each possible fragment. Theorem 1 reveals that the expected number of all fragments is proportional to $\mathcal{V}$. Further, because disk I/Os in Pull-Request are independent of $\mathcal{V}$, the total I/O cost is proportional to $\mathcal{V}$.

Theorem 1. *Let $F[\mathcal{V}]$ denote the number of* fragments *related to u as* svertex. *$\mathbb{E}(F[\mathcal{V}])$ is the expected $F[\mathcal{V}]$. Then $\mathbb{E}(F[\mathcal{V}]) \propto \mathcal{V}$.*

Proof. Please see Section 1 in the supplementary file. □

In conclusion, we compute $\mathcal{V}_i$ using Eq. (5) or Eq. (6), which can satisfy the memory usage constraint and minimize I/O costs.

4 Adaptive Hybrid Solution

This section first gives the performance analysis of Push and BPull, and then describes two hybrid solutions to combine them for graph algorithms with different I/O patterns.

4.1 Performance analysis

We analyze C_{net} and C_{io}, the main factors affecting the performance (see Section 2). Note that dvertices in a Vblock share the

2. In fact, B_i in *Giraph* only indicates the receiving buffer size. We ignore the sending buffer size because *Giraph* immediately flushes out messages once the buffer overflows.

same pull request, which reduces the cost of sending requests. Together with the optimization of concatenation/combination, BPull outperforms Push in term of C_{net}. We then focus on C_{io}.

C_{io} of a complete superstep is the I/O cost caused by producing messages and updating vertices. Let $\mathcal{M}^t$ be the number of messages produced at the (t-1)-th superstep in Push or at the t-th superstep in BPull. Then E^{t-1} and $\mathcal{E}^t$ respectively stand for the set of edges read in Push and BPull for message generation. After completing the t-th superstep, we show the I/O costs of Push in Eq. (7) and BPull in Eq. (8). Here, M^t_{disk} is the set of messages resident on disk, and F^t is the set of fragments covering all edges in $\mathcal{E}^t$. We denote by $IO(.)$ the number of bytes of the given data. Specifically, $2IO(M^t_{disk})$ is the total number of write and read bytes regarding messages. $IO(F^t)$/$IO(V^t_{rr})$ denotes the I/O cost of fragments' auxiliary data (F^t)/svertices values in Vblocks (i.e., V^t_{rr}) read by Pull-Respond. $IO(V^t)$ and $IO(V^{t-1})$ respectively indicate the costs of updating vertices under BPull and Push, which are equal to each other.

$$C^t_{io}(\text{Push}) = IO(V^{t-1}) + IO(E^{t-1}) + 2IO(M^t_{disk}) \quad (7)$$

$$C^t_{io}(\text{BPull}) = IO(V^t) + IO(\mathcal{E}^t) + IO(F^t) + IO(V^t_{rr}) \quad (8)$$

We can compute the number of disk-resident messages by $|M^t_{disk}| = \mathcal{M}^t - \sum_{i=1}^{\mathcal{T}} B_i$ if $\mathcal{M}^t > \sum_{i=1}^{\mathcal{T}} B_i$; and zero, otherwise. However, the value of $\mathcal{M}^t$ (and hence $|M^t_{disk}|$) might dynamically change with t, leading to different I/O-efficiency comparison results. On the other hand, before exchanging messages, we cannot estimate the communication gain of BPull. The two factors render the overall performance comparison result of Push and BPull to be particularly non-deterministic, which motivates us to design a hybrid solution to smartly switch Push and BPull if necessary.

4.2 BHS: a basic hybrid-switching framework

A basic hybrid-switching framework (called BHS here) must accommodate Push and BPull from two perspectives: computing functions for algorithm implementation and data storage for consistent and efficient data accesses. The issue of switching time is also discussed for better performance.

Computing functions: Like existing Pull, BPull also decouples compute() in Push into pullRes() and update(). For Push, compute() is divided into three functions: loadM(), update(), and pushRes(). Here, loadM() loads messages received at the previous superstep into a local buffer (Eq. (9)), to prepare to be consumed in update(). $\Gamma^{in}(u)$ is the set of u's in-neighbors. Following update(), pushRes() is immediately invoked to broadcast new messages $M^{i+1}_O(u)$ to u's out-neighbors (Eq. (10)).

The decoupling of compute() supports a seamless switching by sharing update(). When switching from BPull to Push, we first invoke pullRes() and update() to update vertex values, and then immediately call pushRes() based on new values. Conversely, when switching from Push to BPull, loadM() and update() are invoked to update vertex values which will be used by pullRes() at the next superstep.

$$\text{loadM}()(\sum_{x \in \Gamma^{in}(u)} M^{t-1}_O(x)) \rightarrow M^t_I(u) \quad (9)$$

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

Data storage: The shared update() makes Push and BPull share common vertex values, i.e., Vblocks in VE-BLOCK. Particularly, although pulling and pushing messages are done in a single superstep when switching from BPull to Push, update() is invoked only once for each vertex, which avoids reading/writing conflicts. On the other hand, Eblocks cannot support efficient edge accesses for Push, because all edges of u are required in pushRes(), but they are maintained in different Eblock files. We thereby replicate edges in adjacency lists for Push.

Switching time: The key to gain optimal performance in BHS is deciding the right switching time. Recently Shang and Yu [18] report that for many graph algorithms, metrics collected by the current superstep can be used to predict those of the remaining supersteps. This paper follows the same principle but the difference is that we design a metric Q which can effectively characterize the performance of Push and our BPull.

Specifically, after superstep-t, we can compute Q^t based on C_{net} and C_{io}. For C_{net}, $Byte_m$ stands for the size of a destination vertex id if messages are concatenated, or the size of a whole message if combination is enable. $\mathcal{M}^t_{co}$ is the number of concatenated or combined messages across network in BPull. $\mathcal{M}^t_{co}Byte_m$ thereby denotes the extra communication cost of Push, compared with BPull. On the other hand, based on Eqs. (7) and (8), we can compute the difference in C_{io} by distinguishing random-read/write and sequential-read/write. Eq. (11) finally shows how to evaluate the performance difference between Push and BPull. Here, s_{rr}/s_{rw}/s_{sr} and s_{net} stand for the random-read/random-write/sequential-read throughput, and the network throughput, respectively. Apparently, BPull has superior performance if $Q^t \geq 0$.

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

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

Particularly, when actually running Push, we will know the set of *responding vertices* and its distribution among Vblocks. We then figure out the set of required Eblocks if BPull is run. Now C_{io}(BPull) can be estimated. In addition, $\mathcal{M}^t_{co}$ is estimated by the concatenating/combining ratio in the most recent BPull execution. Note, that if BPull has not yet been run, it is zero. In contrast, for estimation of Push, M^t_{disk} is inferred by comparing $\mathcal{M}^t$ against memory capacity. Further, after estimating how many edges will be read from disk based on the set of *active vertices*, we can compute C^t_{io}(Push). For both modes, collecting statistics and computing Q can be performed at the global barrier.

As reported by Shang et al. [18], the prediction accuracy is proportional to $\frac{1}{\Delta t}$, $\Delta t \geq 1$. However, $\Delta t = 1$ might cause frequent switching operations. In particular, when switching from BPull to Push, pullRes() and pushRes() are run in a single superstep. The potential resource contention results in a slight performance loss. Hence, our compromised solution is to set Δt as 2.

PUSH versus BPull in BHS: We do this analysis for better understanding the switching behaviors. By Eq. (11), the key point

is Q. Its sign decides that a specific superstep will run Push or BPull, i.e., which of the two modes has superior performance. Before analysis, we make an assumption that $V_{res} \equiv V$ among iterations. That means each vertex always broadcasts messages to all of its neighbors at each superstep. With this assumption, all edges will be read to generate messages, and then the numbers of messages and edges are equal to each other ($\mathcal{M}^t \equiv |E|$), which facilitates the I/O-cost estimation. Let f be the number of fragments in Eblocks. $B = \sum_{i=1}^{\mathcal{T}} B_i$ is the total memory capacity. Theorem 2 shows how to compare the I/O-efficiency based on easily collected factors B, f, and $|E|$. Then we can draw the following conclusions.

1) If $B \leq (|E| - f)$, BPull is more I/O-efficient than Push, because the latter needs to process a large amount of disk-resident messages. Together with the fact that BPull has consistent superior communication performance, we know Q is positive and hence BHS always runs BPull.
2) If B is so large that all data are kept in memory, I/O costs are zero. Q solely depends on the communication efficiency. BHS thereby consistently runs BPull.
3) If both conditions are not satisfied, with increase of B, many messages can be computed in memory, which decreases the message I/O cost of Push and even gradually offsets the communication gain of BPull. BHS might run Push or BPull, based on the online estimation of Q.

Theorem 2. $C_{\text{io}}(\text{Push}) \geq C_{\text{io}}(\text{BPull})$ *if* $B \leq (|E| - f)$.

Proof. Please see Section 2 in the supplementary file. □

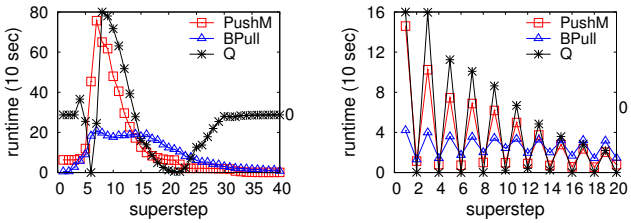
The assumption of $V_{res} \equiv V$ holds true for some algorithms like PageRank, but does not for others like Single-Source Shortest Path (SSSP). For the latter, $\mathcal{M}^t$ changes with iterations. It is difficult to make the theoretical analysis. However, BHS can collect enough information to compute Q and then smartly make the right choice. Besides, $|E|$, f and B are all available once VE-BLOCK is built. Thus, before iterations, BHS can select the initial mode for the first superstep by the discussion above and then dynamically switch modes if necessary.

4.3 GHS: a generalized hybrid-switching framework

Our further investigation reveals that BHS may not be a good choice for some other important algorithms. The main problem is the low prediction accuracy. We now provide a generalized variant.

Boundary of BHS: BHS works well for the category of algorithms where the runtime gradually changes in neighboring supersteps. The ***Gradual-Change*** property implies the mode that currently yields the best speedup is usually the optimal choice in the near future. Hence, it is reasonable to approximate $Q^{t+\Delta t}$ using Q^t. Many algorithms fall into this category, like SSSP, Random Walk, Connected Components, and PageRank. In contrast, for ***Sudden-Change*** algorithms, the runtime suddenly and frequently changes with iterations, leading to a large difference between Q^t and $Q^{t+\Delta t}$ even if $\Delta t = 1$. The prohibitively low prediction accuracy will guide BHS to select a suboptimal mode. Such algorithms include Maximal Independent Sets (MIS), Maximal Cliques, Vertex Coloring, Bipartite Matching, and so on.

For better understanding the weakness of BHS, we run two representative algorithms SSSP and MIS over a fri graph on Amazon EC2. Fig. 5 shows results of a push-based method (PushM[3] [13]) and our BPull. For more details about the experiment settings, please refer to Section 6.1.



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

Figure 5. Illustration of runtime change (fri). Left y-axis indicates runtime per superstep; Right y-axis indicates the value of Q

Clearly, SSSP has a gradual runtime change after reaching the peak at superstep-6. When detecting $Q < 0$ at superstep-14, we can assume that the comparison result holds true in the near future, and so it does. Differently, MIS shows the sudden and frequent runtime change. BPull outperforms Push ($Q > 0$) at supersteps 1, 3, 5, ...; but underperforms ($Q < 0$) at 2, 4, 6, ... Switching modes at the right time can significantly drop the overall runtime, but it is very difficult. For example, following BHS, the predicted Q^2 is positive because $Q^1 > 0$, but actually it is not. Thus, we need a more generalized switching framework to accurately predict the I/O variation of sudden-change algorithms.

GHS for *sudden-change* algorithms: At superstep-t, since the accuracy of approximating Q^{t+1} with Q^t is low, a straightforward alternative is to directly compute Q^{t+1}. However, the computation must be accomplished before messages are pushed/pulled—the majority of workloads in superstep-(t+1). Then we can duly switch to the optimal mode. Below, we first analyze what statistics are required by computing Q^{t+1}, and then show how to modify the iterative framework for collecting statistics.

By Eq. (11), we know Q^{t+1} is dominated by the communication gain of BPull, the message I/O cost of Push, and the I/O difference of reading graph data under BPull and Push. The former two depend on how many messages are produced for superstep-(t+1), i.e., $\mathcal{M}^{t+1}$; while, the latter can be inferred by the sets of *active vertices* and *responding vertices* respectively at supersteps t and (t+1), i.e., V_{act}^t and V_{res}^{t+1}. More specifically, V_{act}^t and V_{res}^{t+1} are respectively formed by *Active Bitmap* and *Responding Bitmap* which are updated online (see Section 3.2). For $\mathcal{M}^{t+1}$, we can compute it by summing the number of messages $h^{t+1}(u)$ produced by each vertex u in V_{res}^{t+1}. Here, $h^{t+1}(u)$ is specific to applications and can be given as offline knowledge.

In BPull, the two required bitmaps are naturally provided at the end of superstep-t. We can duly compute Q^{t+1} before superstep-(t+1). In contrast, Push invokes compute() for each *active vertex* to update state and push messages. Clearly, the *Responding Bitmap* is not available until all *active vertices* have been updated. However, at that moment, all messages have been pushed, i.e., the majority of workloads have been done. We solve this problem by separating state update and message pushing, which are respectively performed by the decoupled functions update() and pushRes() in BHS. The two functions are run only when switching modes in BHS, but now, they are used in a regular push-based superstep. We first run update() and then insert a mini-barrier to save updated vertices onto disk. Following that, pushRes() is invoked to re-load vertices to generate messages. At the mini-barrier, the *Responding Bitmap* can be correctly collected.

3. PushM is an advanced pushing method tested in our experiments.

Compared with Push using compute(), the two-phase variant takes additional costs $IO(V_{res}^{t+1})/s_{sr}$ to re-load responding vertices. Further, the number of edges read from disk depends on V_{res}^{t+1} instead of V_{act}^{t}. Then the I/O cost of reading edges changes from $IO(E^t)$ to $IO(\mathcal{E}^{t+1})$. Together, we give the new performance metric $\mathbb{Q}^{t+1}$ in Eq. (12).

$$\mathbb{Q}^{t+1} = \frac{\mathcal{M}_{co}^{t+1} Byte_m}{s_{net}} + \frac{IO(M_{disk}^{t+1})}{s_{rw}} - \frac{IO(V_{rr}^{t+1})}{s_{rr}} + \frac{IO(V_{res}^{t+1}) + IO(M_{disk}^{t+1}) - IO(F^{t+1})}{s_{sr}} \quad (12)$$

The generalized hybrid-switching (GHS) framework combines BPull and the two-phase Push. At superstep-t, it first invokes pullRes() for BPull or loadM() for Push. Then update() is run to update vertices and collect required online statistics. Accordingly, we compute $\mathbb{Q}^{t+1}$ to guide how to produce messages. That is, vertex values are reloaded and then pushRes() outputs messages, if Push is a preferred mode at superstep-(t+1). Otherwise, nothing is done. The required messages will be pulled at superstep-(t+1).

BHS versus GHS: The switching decision in GHS is given before the most workloads of superstep-(t+1) are processed. Instead, for BHS, the decision is available at the end of superstep-(t+1). In this case, messages used in superstep-(t+1) are still pulled/pushed in the old mode. The delayed-response is acceptable for *gradual-change* algorithm, but cannot be tolerated for *sudden-change* algorithms because of fleeting optimization opportunities. Overall, as a general framework, GHS removes the gradual-runtime-changing assumption, but requires additionally *mini*-barriers and offline knowledge for performance predication. That means if we cannot specify the runtime changing pattern of an algorithm, we can conservatively run it under GHS.

Currently, we characterize the runtime pattern by the ways of compute() or update() &pullRes() being designed. In general, if the behaviors of vertex update and message generation is independent of the specific superstep counter, the workload will not change or just gradually change with iterations. The algorithm then belongs to the *gradual_change* category. Otherwise, it falls into the *sudden_change* category.

5 Optimizations

We finally introduce two optimizations for hybrid frameworks to further enhance the performance. They are priority scheduling in Section 5.1 and lightweight fault-tolerance in Section 5.2.

5.1 Priority scheduling in BPULL

In BPull, a computing node as a message receiver will broadcast pull requests among nodes for every local Vblock. Since all receivers work individually, a node as a message sender can receive more than one request at the same time. By default, it responds to such requests in parallel to maximize the throughput. However, the goal is difficult to be achieved because processing requests without coordination among nodes can make much of the node capacity run at low average utilization. Our solution is to dynamically prioritize the responding scheduling order based on global statistics. Below, we first describe the low resource utilization problem and its impact on overall runtime, and then present the details of priority scheduling.

Low resource utilization: We first give the explanation. Given $\mathcal{T}$ receivers, they can send requests at the same time. Thus, fully parallel responding requires that any sender has at least $\mathcal{T}$ CPU threads, which usually goes beyond the capacity of a single node. In this case, a sender responds to requests by the default OS scheduling policy—FIFO (first-in, first-out). On the other hand, real graphs generally have heavily skewed degree distribution, yielding unbalanced distribution of edges among Eblocks, and hence responding workloads among pull requests. Also, by Alg. 1, a receiver cannot initiate new requests for a Vblock until all of the old ones for another Vblock have been handled. Clearly, by FIFO, processing the old, heavyweight requests will certainly delay initiating new requests. Senders waiting for such new requests thereby block themselves, which makes resource under-utilization.

For better understanding, we demonstrate the phenomenon in Fig. 6. Here R_{ijj_k} is a request with responding workload $|R_{ijj_k}|$ or $|.|$ for short. It is sent by receiver T_j for b_{j_k} and processed by sender T_i. Suppose that a sender has one thread for responding and T_3 sequentially receives R_{321}, R_{331}, and R_{311}. By FIFO, T_3 firstly responds to R_{321} with the max workload, which takes so long that T_1 has consumed all buffered requests at some time x. Since Vblocks like b_1 on T_1 are now waiting for messages from T_3 (R_{311}), the update focus cannot be moved to the next batch of Vblocks like b_2. The responding thread on T_1 then keeps idle to wait for new requests like R_{112} for b_2 on T_1.

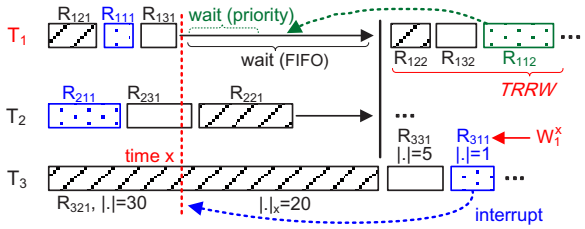


Figure 6. Responding to pull requests with FIFO (solid line) and Priority (dashed line) strategies. Each node has one thread for responding.

Increased overall runtime: At an iteration, a sender T_i will respond to all possible requests from any receiver. At time x, some requests have already been handled while others have not. The latter are called as the *total remaining responding workload* (*TRRW*). Now we assume all nodes have the same computing power only for simplicity. Then if T_i has the largest *TRRW* and its resource utilization is low, it becomes the slow, straggling sender, which increases overall runtime in distributed environments.

Priority Scheduling: Since the low resource utilization is caused by the individual FIFO responding design, now we propose a well-suited priority scheduling replacement strategy. The main idea is to keep the sender with the largest *TRRW* busy by repeatedly running the following three steps during a BPull iteration.

1) At any time x, seek the target sender $T_{\hat{i}}$ with the largest *TRRW* and then detect whether or not it is idle.
2) If yes, further seek a target receiver $T_{\hat{j}}$ that has the minimal workloads of pending requests before proceeding to a new local Vblock requiring messages from $T_{\hat{i}}$ (i.e., it will initiate a request to $T_{\hat{i}}$).
3) Prioritize the responding order of such pending requests so that $T_{\hat{j}}$ can quickly proceed to the new Vblock.

In step 1), computing *TRRW* of a sender T_i requires two kinds of information: a list L_i^x containing all requests that will be

processed by T_i after time x and the responding workload of each request within L_i^x. Initiating a request R_{ijj_k} means T_i contains responding vertices as svertices which will send messages along edges to dvertices on T_j. In another word, there is at least one Vblock on T_i, of which the responding indicator *res* and the j_k-th bit of the edge distribution *bitmap* are true (see Fig. 4). By detecting the two variables before an iteration (now $x = 0$), we can easily infer L_i^0 for every T_i. Then at any time x, we compute L_i^x just by removing handled requests from L_i^0. On the other hand, because requests in L_i^x have not been actually processed, we cannot exactly compute their responding workloads. Alg. 2 reveals that the responding work is to read vertices and edges, and then generate and transfer messages. Thus, the total size of Vblocks and Eblocks involved in responding is roughly proportional to the cost of disk I/O and network communication. We thereby use it to estimate the responding workload.

Further, selecting $T_{\hat{j}}$ in step 2) depends on two phases: finding all candidate receivers in Phase-I and then performing the selection in Phase-II. A receiver is regarded as a candidate if it will send requests to $T_{\hat{i}}$ at some future time, i.e., it has at least one element in $L_{\hat{i}}^x$ of $T_{\hat{i}}$. On the other hand, $T_{\hat{i}}$ only cares about who can firstly initiate a request to make it quickly busy again. The goal of Phase-II is then to identify such a candidate, i.e., $T_{\hat{j}}$. Note that requests are for updating vertices. A receiver initiate them in order of processing Vblocks. The problem is then simplified to compare the arriving times of the first request $R_{\hat{i}jj_k}$ specific to Vblock b_{j_k} on each candidate T_j. However, at time x, perhaps T_j is currently processing Vblock b_{j_s}, and then will proceed to $b_{j_{s+1}}, ..., b_{j_{k-1}}$ and b_{j_k}. Here such pending Vblocks from j_s to j_{k-1} do not require messages from $T_{\hat{i}}$, and hence, we call requests from them as *pending requests*. We then estimate how long T_j can proceed to b_{j_k} by quantifying the *responding workload of pending requests* (*PRRW*). More specifically, a Vblock might initiate many *pending requests* but only the one with the maximal workload dominates the runtime in distributed environments. We then compute *PRRW* denoted by W_j^x for T_j by summing up the maximal workload of every pending Vblock, mathematically shown in Eq. (13). The candidate with the minimal *PRRW* is then selected as $T_{\hat{j}}$.

$$W_j^x = \sum_{l=s}^{k-1} \max\{|R_{ijj_l}| \; | 1 \leq i \leq \mathcal{T}\} \tag{13}$$

Thirdly, it is most likely that requests from $T_{\hat{j}}$ arrives at a sender T_i later than those from other receivers. Nevertheless, by priority scheduling in step 3), the former are still firstly processed so that $T_{\hat{j}}$ can proceed to the target Vblock b_{j_k} without any delay. For example, T_1 in Fig. 6 is the target sender since it is idle and has the largest *TRRW*. It estimates itself as the target receiver because R_{112} for b_2 can make it busy again and $W_1^x = |R_{311}| = 1$ is the minimal among candidates. R_{311} then increases its priority so that T_3 can immediately respond to it.

Generally, let N_{thd} specify the number of threads used for responding per node. The responding parallelism α is equal to $\min\{\mathcal{T}, N_{thd}\}$. We then relax the condition of seeking $T_{\hat{i}}$ as "top-α largest *TRRW*" so that more senders can avoid idling if $\alpha \geq 2$.

Now we clearly know we can identify $T_{\hat{i}}$ and $T_{\hat{j}}$ by analyzing the request initiation relationship between senders and receivers. Such a global analysis requires online statistics. Some are available at the very beginning of an iteration, including *res*, *bitmap*, and the size of Vblocks and Eblocks required by step 1), and the order of processing Vblocks required by step 2). Also, we can easily collect other real-time statistics including the requests that have been handled in step 1) and the Vblocks that are currently processed on a candidate receiver in step 2).

Last but not least, priority scheduling can also optimize I/O costs for algorithms supporting Combiner. As analyzed in "Memory usage" in Sec. 3.3, when running these algorithms, sub-buffers for sending messages will compete for limited memory resource, which affects the number of Vblocks. However, now limited by the explicit responding parallelism, a sender handles at most α requests at the same time. The new design decreases the number of sub-buffers from $\mathcal{T}$ down to α. We then compute the number of Vblocks $\mathcal{V}_i^\alpha$ for T_i by Eq. (14), rather than Eq. (5). With the assumption that every node has the same memory capacity, Theorem 3 tells us this design decreases the total number of Vblocks, and hence the I/O costs (see "I/O costs" in Sec. 3.3).

$$\mathsf{B}_i = \sum_{j=1}^{\alpha} \frac{|V_j|}{\mathcal{V}_j^\alpha} + \frac{2|V_i|}{\mathcal{V}_i^\alpha} \tag{14}$$

Theorem 3. *Assume that $B_i = B_j$ for $\forall 1 \leq i, j \leq \mathcal{T}$. Let $\mathcal{V}$ and $\mathcal{V}^\alpha$ respectively denote the total numbers of* Vblocks *without and with the limitation of the responding parallelism. Then $\mathcal{V} \geq \mathcal{V}^\alpha$ for algorithms supporting Combiner.*

Proof. Please see Section 3 in the supplementary file. □

5.2 Lightweight fault-tolerance

Today's graph systems typically tolerate failures by checkpointing [1], [19] which periodically archives vertex values during the *failure-free* (no failure) execution and upon failures, rolls back computations on all tasks to the most recent checkpoint for recovery. In addition, we can also log outgoing messages per superstep [1] to confine rollback-recovery to failed/restarted tasks only. This optimization accelerates recovery but slows down the *failure-free* execution. This is because the volume of messages over supersteps can be large and then we need to keep them on external storage. However, hybrid frameworks allow us to design a new lightweight fault-tolerance mechanism to strike a good balance between recovery efficiency and *failure-free* performance.

Pull-confined failure recovery: The main idea is to re-generate messages required by recovery using BPull, instead of proactively logging them. Recall that BPull naturally allows messages to be efficiently pulled on demand of specific dvertices. Upon failures, vertices as dvertices on failed/restarted tasks can normally broadcast requests to collect messages for recovery. In contrast, vertices on surviving tasks do nothing but only respond to requests. More specifically, for correct re-generation, surviving tasks need to restore the responding context, including svertices in Vblocks, edges in Eblocks, and the *Responding Bitmap* (see Sec. 3.2). The former two are directly available on local disk, while the latter is dynamically updated in memory over iterations. We then log the latter per iteration for retrieving information. Since the bitmap has much smaller size than messages, the performance penalty for *failure-free* computations is significantly reduced.

Since pull requests are broadcasted only by failed vertices, we call the new mechanism as pull-confined failure recovery. It can be run at any superstep, because Vblocks, Eblocks and the bitmap are always available in BHS or GHS even a superstep originally runs Push. In particular, as shown in Fig. 7, if the failed superstep (t+3) originally runs Push, we should manually switch

from BPull to Push after recovery so that messages required by Push at superstep-(t+3) can be prepared in advance.

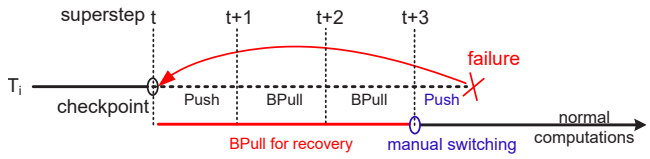


Figure 7. Failure recovery with BPull

Performance analysis: The difference between our pull-confined recovery (CpPull) and existing log-based recovery (CpLog) is how to get required messages. Theorem 4 gives the sufficient condition of CpPull outperforming CpLog—that is, the ratio of failed nodes to all nodes λ is less than $\frac{1}{2}$. In real distributed environments, the probability that several nodes fail at the same time is extremely low. Thus, CpPull is a lightweight yet practical fault-tolerant framework.

Theorem 4. *Suppose that $\mathcal{T}_F$ of $\mathcal{T}$ tasks/nodes fail at superstep-(t+1).* CpPull *outperforms* CpLog *if $\mathcal{T}_F \leq \lambda\mathcal{T}$, where $\lambda \geq \frac{1}{2}$.*

Proof. Please see Section 4 in the supplementary file. □

6 Performance Studies

We develop a prototype system *HGraph*[4] to implement the block-centric pulling mode (BPull), and the basic (BHS) and generalized (GHS) switching frameworks. We compare it against two pushing systems *Giraph* (Push) [9] and *MOCgraph* (PushM) [13]. They are all based on Java for a fair comparison. However, existing pulling systems are all written in C++ and memory-resident. We then select the well-known representative *GraphLab* (Pull) [2] as the competitor and modify it to support disk operations to confirm the I/O-inefficiency. Note that *MOCgraph* is built on top of *Giraph*, but it can directly consume messages if messages are commutative and their dvertices are in memory. It is the up-to-date Java-based pushing system in terms of I/O-efficiency. We also conduct experiments against the most recently published pushing system *GraphD* [14], which is based on C++ but we consider it for a complete performance study. In particular, our techniques with priority scheduling are marked with a suffix-marker "+".

6.1 Experimental setup

Cluster configurations: We conduct testing on Amazon EC2. The cluster consists of 30 nodes with one additional master connected by a Gigabit Ethernet switch. Every node is equipped with 4 virtual CPUs, 7.5GB RAM and 30GB SSD[5]. The random-read/random-write/sequential-read/network throughput $s_{rr}/s_{rw}/s_{sr}/s_{net}$ is 18.2/18.2/18.3/116MB/s[6].

Algorithms: We test six benchmark graph algorithms, including PageRank, SSSP, Label Propagation Algorithm (LPA) [20], Simulating Advertisement (SA) [21], Maximal Independent Sets (MIS) and Bipartite Matching (BM). They are widely used in Internet, Social Networks, and many other applications. Table 1 summarizes their features from two perspectives: 1) the message compression policy: *combination* or *concatenation*; 2) the runtime changing pattern with iterations: *gradual_* or *sudden_change*. *gradual_change* algorithms are further divided into two categories based on whether or not all vertices will respond to requests. In particular, PushM cannot run *concatenation* algorithms since messages are not commutative. GHS is very suitable for *sudden_change* algorithms because of its prominent prediction component. Overall, the selected algorithms can achieve full test coverage of suitability and efficiency for solutions involved in our experiments. For more details about the graph algorithms, please refer to Section 5 in the supplementary file.

Table 1
Features of benchmark graph algorithms

	gradual_change ($V_{res} \equiv V$)	*gradual_change* ($V_{res} \not\equiv V$)	*sudden_change*
combination	PageRank	SSSP	MIS
concatenation	LPA	SA	BM

Graph datasets: All tests are run over six real graphs listed in Table 2. We convert these graphs into undirected ones as inputs when running MIS and BM. Further, to construct a bipartite graph for BM, vertices are hashed into two distinct sets and only edges between the sets are preserved. Besides, each edge is assigned a weight randomized between 0 and 1 when generating messages in SSSP. By default, a graph is partitioned by the range method [9] for *Giraph*, *MOCgraph*, and *HGraph*. For *GraphLab*, many intelligent methods are provided, but only Oblivious is used since others exhibit the similar I/O-performance. Finally, *GraphD* only provides a built-in hash partitioning method.

Table 2
Real graph datasets (M: million)

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

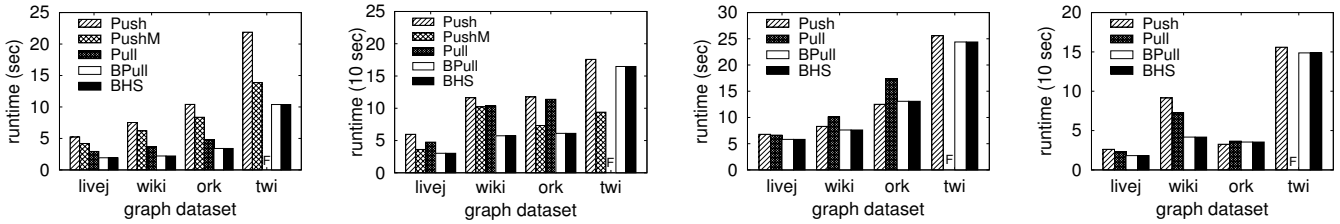
[7] http://snap.stanford.edu/data/soc-LiveJournal1.html
[8] http://haselgrove.id.au/wikipedia.htm
[9] http://socialnetworks.mpi-sws.org/data-imc2007.html
[10] http://an.kaist.ac.kr/traces/WWW2010.html
[11] http://snap.stanford.edu/data/com-Friendster.html
[12] http://law.di.unimi.it/webdata/uk-2007-05/

Experiments design: Section 6.2 studies the overall performance to validate the effectiveness of our techniques. Section 6.3 gives the detailed performance analysis of BHS. Section 6.4 further tests GHS using MIS and BM. Section 6.5 explores the performance of different fault-tolerant frameworks. Section 6.6 compares *HGraph* and *GraphD*. Section 6.7 reports the impact of graph partitioning.

In particular, we state two testing scenarios, *sufficient memory* where all systems manage data in memory and *limited memory* where some data will be spilled onto disk. For the latter, we give a detailed description in Section 6 in the supplementary file.

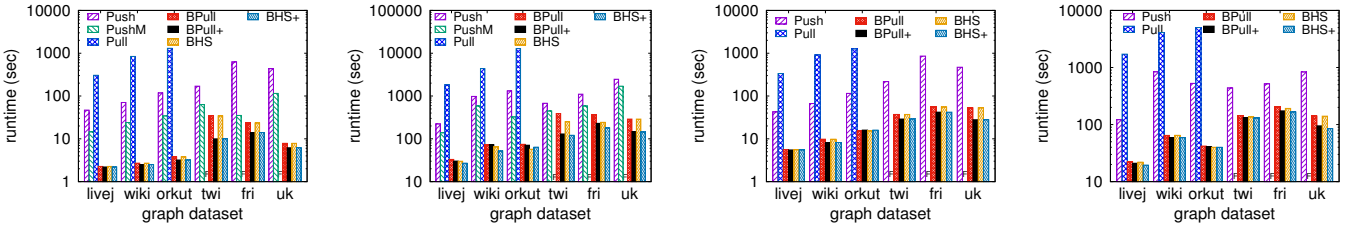
Without otherwise specified, for LPA and PageRank, the average runtime of one superstep is reported, by totally running 30 supersteps, as the workload per superstep is constant. Other algorithms are run until they converge and we report the runtime of the entire computation. In addition, the symbol 'F' indicates an unsuccessful run. All tests are run in the synchronous manner.

4. https://github.com/HybridGraph/HGraph
5. We also run all tests on a local cluster with HDD and achieve the similar performance improvement.
6. Reported by *fio-2.0.13* and *iperf-2.0.5*.



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

Figure 8. Runtime with *sufficient memory*. BPull/BHS is comparable to or even better than competitors



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

Figure 9. Runtime with *limited memory*

6.2 Overall performance evaluation

We test algorithms except MIS and BM in two scenarios: one is *sufficient memory* where push-based and pull-based modes have different favorites as shown in Fig. 8, and the other is *limited memory* where BHS+ is the best as shown in Fig. 9. In particular, the Push vs. BPull analysis in Section 4.2 reveals that BPull consistently outperforms Push among iterations, if $V_{res} \equiv V$ with low-budget memory allocation (for PageRank and LPA in Fig. 9) and/or the memory resource is sufficient (for all cases in Fig. 8). In these cases, BHS always selects BPull as the preferred mode and hence, they perform very similarly.

Runtime with *sufficient memory*: In general, BPull has superior performance to Push because of combining/concatenating messages. The performance gap is especially large for SSSP and SA over a large diameter graph wiki. In this case, there exists a long-tail convergent stage where many active vertices do not generate messages after updates, i.e., $V_{res}^t \subseteq V_{act}^t$. Then BPull might read fewer edges than Push and PushM (see Section 2). PushM outperforms Push because directly consuming messages can alleviate the memory pressure and then avoid frequently starting Java garbage collection. Another observation is that BPull even beats Pull in some cases, as the former sends fewer pull requests and can offer a comparable message transfer efficiency by combination. Finally, without I/O operations, priority scheduling brings marginal benefit. We then omit the report to reduce clutter.

Runtime with *limited memory*: The speedup of BPull compared with Push is up to a factor of 56 (PageRank over uk). Even compared with PushM, BPull still offers roughly 15x speedup in the same case. For SSSP and SA, the number of messages dynamically changes, which weakens the competitive advantage of pulling and hence BPull does not work well as expected. For example, BPull runs only 1.7x faster than PushM. However, by switching modes, BHS runs up to 36% and 7% faster than BPull, respectively for SSSP and SA.

Priority scheduling creates another performance gap between BPull and Push/PushM. Here we set $\alpha = 5$ because one node can totally run 8 CPU threads and three of them are respectively used by the file system, the daemon process, and the vertex update function. We find that the largest speedup is increased to 70x compared with Push and 19x against PushM. Among all cases, we gain the best runtime improvement on twi. The reason is that the highly skewed power-law degree distribution exacerbates the unbalanced responding workload distribution among requests. The default FIFO policy cannot cope with this challenge very well. Further, BHS+ also includes an engineering effort—recording the offset of edges per svertex to skip useless edges in Push. It brings great runtime reduction on wiki because many supersteps run Push in the long-tail convergent stage.

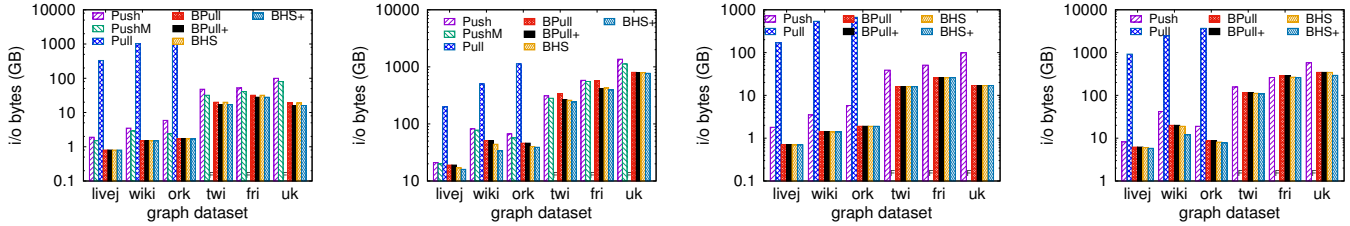
I/O costs with *limited memory*: Fig. 10 reports I/O costs denoted by the total number of read and write bytes. Pull entails substantial costs because of the random and frequent access to svertices. On the other hand, PushM beats Push since messages sent for in-memory dvertices are directly consumed. In particular, when running SSSP over twi, BPull usually costs more than Push and PushM. That means the gain achieved by eliminating message I/Os cannot defeat the cost incurred by accessing svertices and auxiliary data of fragments. However, BHS can optimize it by switching modes adaptively. Note that on large graphs twi, fri, and uk, $\alpha = 5 < \mathcal{T} = 30$. Priority scheduling thereby reduces I/O costs for PageRank and SSSP because of the decreased number of Vblocks. Further, by recording edge offsets, we can observe the significant I/O reduction in BHS+ for SSSP and SA over wiki.

6.3 Analysis of BHS

We now validate the effectiveness of BHS using the same setting in Fig. 9. We first explore the impact of hardware characteristics on the performance metric Q. A detailed analysis is then given to show the resource requirements when switching modes. Here, we test SSSP over twi since BHS achieves the most gain in this case.

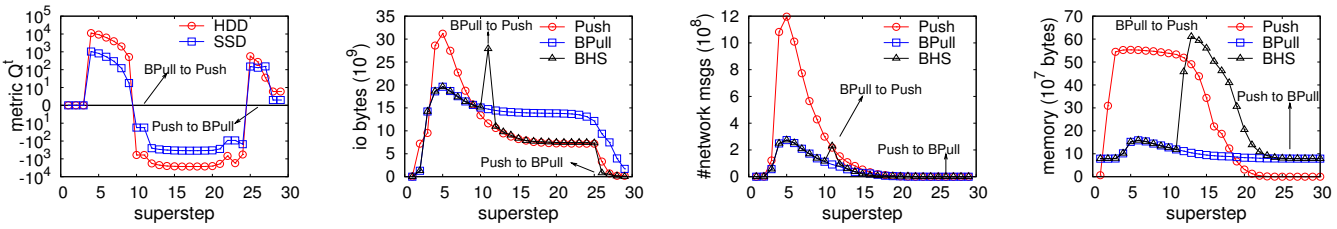
Impact of hardware characteristics: We plot values of Q^t under HDD[13] and SSD in Fig. 11 (a). There exist two switching points

13. Now each Amazon EC2 node is equipped with 30GB HDD (low disk throughput) while other configurations remain unchanged.



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

Figure 10. I/O costs with *limited memory*

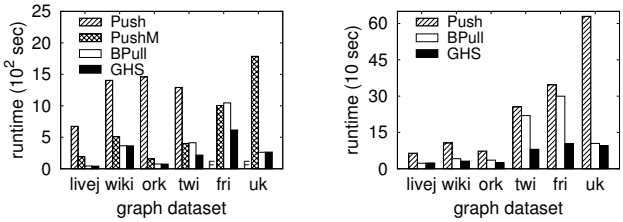


(a) Performance metric (b) Disk I/O costs (c) Network costs (d) Memory usage

Figure 11. Variation of Q^t, I/O costs, network costs, and memory usage for BHS (SSSP over twi, *limited memory*)

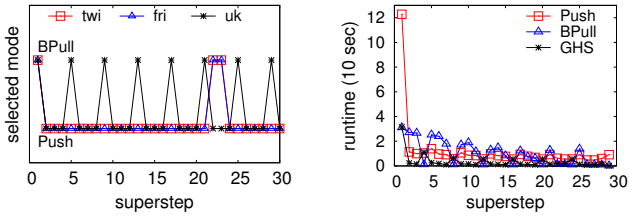
respectively at the 10th and the 25th supersteps, both of which do not change with different external storage mediums. We now give the explanation. When the sign of Q^t changes, the number of messages is small. Together with the fact that $s_{net} \gg s_{rr}/s_{rw}/s_{sr}$ and s_{rr}, s_{rw}, and s_{sr} are close in values, by Eq. (11), we know that the sign of Q^t is mainly dominated by C_{io}, which is orthogonal to hardware characteristics.

Resource requirements: Figs 11 (b)-(d) respectively report the change of I/O-pressure, network communication costs, and memory usage, among supersteps. In the case of "BPull→Push" at superstep-11, the resource requirements increase, because messages which will be handled at the next superstep in Push are now processed in advance. The sudden increase of requirements may slightly slow down the performance due to resource contention. However, this can be easily offset by the switching gains. By contrast, when switching from Push to BPull at superstep-26, messages that should have been generated and pushed at this superstep will be pulled at the next superstep. The resource requirements will not increase. Note that because of additionally maintained *Metadata* of VE-BLOCK, BHS consumes more memory than Push even though the pushing mode is selected.



(a) MIS (b) BM

Figure 12. Runtime of GHS (*limited memory*)



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

Figure 13. Mode selection on BM (*limited memory*)

6.4 Analysis of GHS

We then verify the effectiveness of GHS using MIS and BM with *limited memory*. Because the I/O-inefficiency of Pull has been validated in Section 6.2, we remove it from the tests. Besides, GHS cannot benefit from priority scheduling because of frequent switching operations. We then disable this optimization.

The runtime comparison results are presented in Fig. 12. Overall, GHS generates great performance improvement. Taking MIS as an example, it is approximately 48% faster than BPull over twi. For BM, the improvement is even up to 65% over fri.

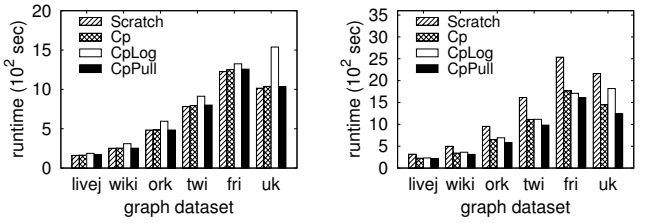
We then design experiments to show how likely GHS can seek the optimal mode. We show the mode decision per superstep on twi, fri, and uk (Fig. 13 (a)), since GHS works especially well on these large graphs. We further demonstrate the runtime change on the graph with the most significant performance improvement (Fig. 13 (b)). Clearly, GHS can exactly capture the runtime variation and then select the optimal for a specific superstep.

6.5 Fault tolerance

We next test fault-tolerance. Besides CpLog and our CpPull, we also test another two solutions to explore the tradeoff between the archiving data cost and the recovery efficiency. One is Scratch that recomputes from scratch upon failures since nothing is prepared during the *failure-free* execution. The other is Cp that periodically makes a checkpoint of vertex values [19]. All tests are run on top of BHS with *limited memory*. Besides, we simulate a failure by manually killing a task at the end of iterations. By offline analysis, the checkpointing interval is set as 10.

Figs. 14 shows the overall runtime of the entire computation w/ and w/o failures, respectively. Compared with Cp, CpLog greatly

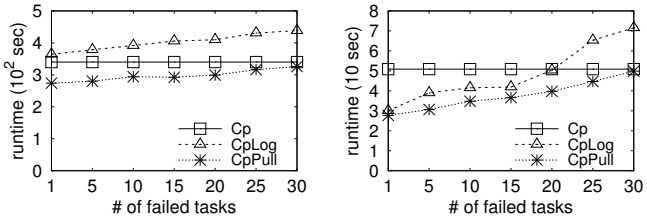
drops the runtime when encountering failures, at the expense of degrading the *failure-free* execution. The performance loss is up to 52% over uk. However, the degradation in CpPull is less significant because no message is logged. Even so, it can still quickly recover failures by re-generating messages on-demand. The lightweight logging policy and the fast recovery method then yield up to 32% improvement (over uk), compared with CpLog.



(a) The failure-free execution (b) The execution with one task failed

Figure 14. Overall runtime of LPA

We then explore the performance features by varying the number of failed tasks. Shown in Fig. 15 (a), both CpLog and CpPull scale with failed tasks due to increased recovery workloads, while Cp does not since it always rolls back the computation on all tasks. By Theorem 4, CpPull outperforms CpLog when $\mathcal{T}_F \leq \frac{1}{2}\mathcal{T} = 15$ and so it does. However, even when $\mathcal{T}_F > 15$, we still observe superior performance. This is because under CpLog, restarted tasks also need to log newly generated messages for possible failures in the future, which increases the runtime especially when the number of failed tasks is large (see Fig. 15 (b)).

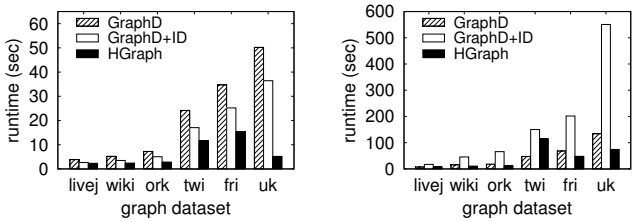


(a) Overall runtime (b) Runtime of recomputations

Figure 15. Performance analysis when varying the number of failed tasks (PageRank over uk)

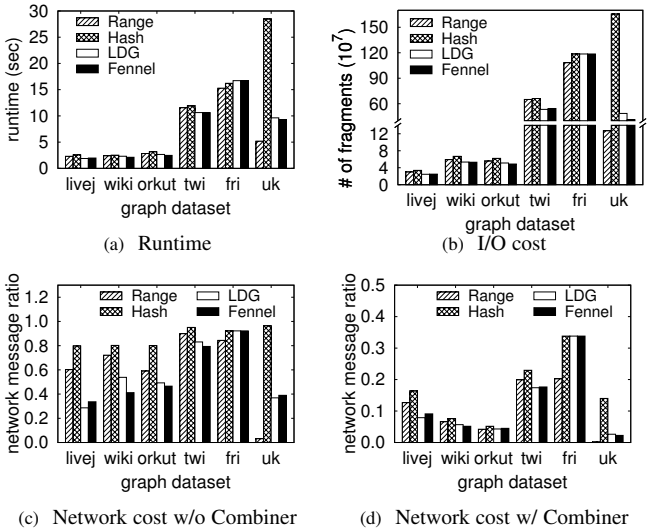
6.6 Comparison with GraphD

Next we compare our Java-based *HGraph* with the state-of-the-art C++-based *GraphD*, by running the widely used benchmark PageRank with *limited memory*. *HGraph* employs the optimal BPull with priority scheduling. In particular, *GraphD* can efficiently combine messages by recoding vertex ids ("ID"). As shown in Fig. 16(a), from the runtime perspective, HGraph is 9.7 and 7.0 times faster, at most, than GraphD and GraphD+ID, respectively. The outstanding performance comes from zero message I/Os and low random read costs. We then measure the elapsed time of loading graph and building VE-BLOCK in *HGraph*. We compare it to the preprocessing time of competitors (Fig. 16(b)). Because GraphD+ID incurs extra effort for recoding IDs, it takes an average of 4 times longer than the preprocessing time of GraphD. HGraph directly stores the loaded sub-graph in each task to minimize the partitioning cost. Thus, even with the time of building VE-BLOCK, it still provides a comparable cost to GraphD. Together, HGraph achieves overall success although C++ is usually more efficient than Java.



(a) Runtime (b) Preprocessing cost

Figure 16. Performance analysis: *GraphD* vs. *HGraph* (PageRank)



(a) Runtime (b) I/O cost

(c) Network cost w/o Combiner (d) Network cost w/ Combiner

Figure 17. Impact of different graph partitioning strategies (PageRank)

6.7 Impact of graph partitioning

Fig. 17 finally shows the impact of different graph partitioning strategies on *HGraph*, all of which try to improve the locality of partitioned sub-graphs to reduce network costs, and are scalable to very large graphs. Specifically, the competitors include: Range [9] preserving locality naturally provided by input graphs; Hash, a widely used method with good efficiency but poor locality; and LDG [22] and Fennel [23], two variants streaming graph data and improving locality by the distribution of already placed vertices.

By PageRank, we observe a slight variation in runtime in sub-figure(a) on all graphs except uk. We explain it from two perspectives. Firstly, sub-figure(b) shows that good locality facilitates clustering edges and then reduces the number of fragments, but the effect is limited. Thus, the costs of reading svertices and auxiliary data will slightly decrease. Secondly, sub-figure(c) validates that advanced partitioning methods indeed optimize the number of network messages across sub-graphs. However, *HGraph* can completely combine messages sent to the same dvertex at the sender side. This optimization significantly compresses network messages and hence weakens the effect of partitioning, as shown in sub-graph(d). The exception is uk with very good locality in input data. Range fully utilizes this advantage and then outperforms other competitors in terms of disk I/Os and network costs.

6.8 Extended discussion

Now we have known that *HGraph* is I/O-efficient. Besides, its built-in combination or concatenation design can significantly reduce the network I/O costs, which is also the main goal of graph

partitioning. As a result, the performance of *HGraph* is not very sensitive to different partitioning strategies (see Fig. 17). Note also that the new BPull component just manages pull requests in a block-centric manner, but messages are still generated by the traditional vertex-centric principle and then transmitted along edges one-hop per superstep. Thus, the message spreading speed, or the algorithm convergence speed, will not be affected.

Finally, our techniques can be implemented on top of any pushing- or pulling-based underlying systems. No matter which one is selected, we should implement another one and then the switching framework. In particular, VE-BLOCK required by BPull can be built by further partitioning the sub-graph on each node. We make our project publicly available so that users can quickly test our techniques and/or their own advanced ideas.

7 Related Work

We now summarize representative distributed graph systems and existing fault-tolerant techniques, to highlight our contributions.

Push-based systems: Pushing naturally expresses the logic of graph algorithms. Many systems are then push-based [1], [4], [5], [6], [24], [25], [26], [27], [28] and work well when all data are kept in memory. There exist efforts targeting the disk setting for better scalability. Early pioneers [7], [8], [9], [10], [11], [12], [29], [30], [31] manage data on HDFS or local disk in a naive way, rendering them I/O-inefficient. If messages are commutative and their dvertices are in memory, *MOCgraph* [13] directly consumes them to reduce I/O costs. *Chaos* [32] focuses on streaming edges and ignores communication costs with the assumption of high-speed network. *GraphD* [14], as well as our *HGraph*, removes this assumption since many clusters still work in the Gigabit Ethernet network. The closed-source *Turbograph++* [15] overlaps opereations of CPU, disk and network, for full resource utilization, which is nice complement to our design.

Giraph++ [6] and *Blogel* [24] propose a complete block-centric model where vertices in the same block can freely update themselves and directly communicate with each other. Differently, *HGraph* still follows the vertex-centric programming principle for easy use although BPull can pull messages in blocks. *Blogel* further supports block-level communication for very specific algorithms like connected components where vertices within one block share the same value. It is technically orthogonal to our combining or concatenating technique used in BPull.

Pull-based systems: To our knowledge, there exist several pull-based systems designed for memory-resident analysis [2], [19], [33], [34]. All of them follow the vertex-centric pulling principle and hence suffer from performance penalty caused by random reads in disk scenarios. In particular, *GraphLab* replicates vertices for efficient communication [2]. However, memory resources can be quickly exhausted [13], which severely degrades the performance. Some systems state that they support Push and Pull [35], [36]. While, only one is used for a given algorithm. In particular, *Pregel+* [37] can run the two modes at different supersteps but only for operating different kinds of messages.

Fault tolerance: Most fault-tolerance techniques are based on checkpointing [1], [38] and there exist many efforts for optimization [16], [19], [39]. Additionally logging messages can further boost the recovery efficiency [1], [40] but at the expense of *failure-free* performance loss. However, our solution achieves an overall success by BPull. Besides, reactive solutions can recover failures without checkpointing, but they either replicate vertices [41] in limited memory or work for specific applications [42].

8 Conclusion and Future Work

This paper proposes a new adaptive and I/O-efficient message processing mechanism for graph computing on cloud. For general purpose, we combine existing pushing and our improved pulling modes with different switching principles by investigating I/O access patterns. We also design some optimizations for efficiency, including priority scheduling and lightweight fault-tolerance. Experiments verify the great advantage of our proposals.

Note that some graph mining algorithms like pattern matching and dense subgraph retrieval, will broadcast subgraphs as messages, and the size of a subgraph changes with iterations. That challenges the fixed message buffer budget and the static VE-BLOCK design, and might result in the message buffer overflow. However, dynamically adjusting the granularity of VE-BLOCK can change the memory requirement of buffering messages. We plan to investigate this solution as future work.

Acknowledgments

This work was supported by the National Key R&D Program of China (2018YFB1003404), the National Natural Science Foundation of China (61902366, 61902365, and 61872070), China Postdoctoral Science Foundation Grant (2019M652474 and 2019M652473), and Research Grants Council of the Hong Kong SAR (China No. 14203618 and 14202919). The authors also would like to thank all anonymous reviewers.

References

[1] 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*. ACM, 2010, pp. 135–146.

[2] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "Powergraph: Distributed graph-parallel computation on natural graphs," in *Proc. of OSDI*, vol. 12, 2012, p. 2.

[3] M. Onizuka, T. Fujimori, and H. Shiokawa, "Graph partitioning for distributed graph processing," *Data Science and Engineering*, vol. 2, no. 1, pp. 94–105, 2017.

[4] S. Salihoglu and J. Widom, "Gps: A graph processing system," in *Proc. of SSDBM*. ACM, 2013, p. 22.

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

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

[7] U. Kang, C. E. Tsourakakis, and C. Faloutsos, "Pegasus: A peta-scale graph mining system implementation and observations," in *Proc. of ICDM*. IEEE, 2009, pp. 229–238.

[8] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst, "Haloop: efficient iterative data processing on large clusters," *In Proc. of the VLDB Endowment*, vol. 3, no. 1-2, pp. 285–296, 2010.

[9] "Apache giraph," http://giraph.apache.org/.

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

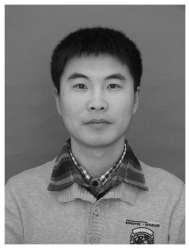
[11] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica, "Graphx: Graph processing in a distributed dataflow framework," in *Proc. of OSDI*, 2014, pp. 599–613.

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

[13] C. Zhou, J. Gao, B. Sun, and J. X. Yu, "Mocgraph: Scalable distributed graph processing using message online computing," *In Proc. of the VLDB Endowment*, vol. 8, no. 4, pp. 377–388, 2014.

[14] D. Yan, Y. Huang, M. Liu, H. Chen, J. Cheng, H. Wu, and C. Zhang, "Graphd: distributed vertex-centric graph processing beyond the memory limit," *TPDS*, vol. 29, no. 1, pp. 99–114, 2018.

[15] S. Ko and W.-S. Han, "Turbograph++: A scalable and fast graph analytics system," in *Proc. of SIGMOD*. ACM, 2018, pp. 395–410.
[16] C. Xu, M. Holzemer, M. Kaul, and V. Markl, "Efficient fault-tolerance for iterative graph processing on distributed dataflow systems," in *In Proc. of ICDE*. IEEE, 2016, pp. 613–624.
[17] Z. Wang, Y. Gu, Y. Bao, G. Yu, and J. X. Yu, "Hybrid pulling/pushing for i/o-efficient distributed and iterative graph computing," in *Proc. of SIGMOD*. ACM, 2016, pp. 479–494.
[18] Z. Shang and J. X. Yu, "Catch the wind: Graph workload balancing on cloud," in *Proc. of ICDE*. IEEE, 2013, pp. 553–564.
[19] J. Xue, Z. Yang, Z. Qu, S. Hou, and Y. Dai, "Seraph: an efficient, low-cost system for concurrent graph processing," in *Proc. of HPDC*. ACM, 2014, pp. 227–238.
[20] U. N. Raghavan, R. Albert, and S. Kumara, "Near linear time algorithm to detect community structures in large-scale networks," *Physical Review E*, vol. 76, no. 3, p. 036106, 2007.
[21] 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*. ACM, 2013, pp. 169–182.
[22] I. Stanton and G. Kliot, "Streaming graph partitioning for large distributed graphs," in *Proc. of SIGKDD*. ACM, 2012, pp. 1222–1230.
[23] C. Tsourakakis, C. Gkantsidis, B. Radunovic, and M. Vojnovic, "Fennel: Streaming graph partitioning for massive scale graphs," in *Proc. of WSDM*. ACM, 2014, pp. 333–342.
[24] 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*, vol. 7, no. 14, pp. 1981–1992, 2014.
[25] S. Tasci and M. Demirbas, "Giraphx: parallel yet serializable large-scale graph processing," in *Euro-Par 2013 Parallel Processing*. Springer, 2013, pp. 458–469.
[26] 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.
[27] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi, "Naiad: a timely dataflow system," in *Proc. of SOSP*. ACM, 2013, pp. 439–455.
[28] W. Fan, J. Xu, Y. Wu, W. Yu, J. Jiang, Z. Zheng, B. Zhang, Y. Cao, and C. Tian, "Parallelizing sequential graph computations," in *Proc. of SIGMOD*. ACM, 2017, pp. 495–510.
[29] U. Kang, H. Tong, J. Sun, C.-Y. Lin, and C. Faloutsos, "Gbase: a scalable and general graph management system," in *Proc. of SIGKDD*. ACM, 2011, pp. 1091–1099.
[30] S. Seo, E. J. Yoon, J. Kim, S. Jin, J.-S. Kim, and S. Maeng, "Hama: An efficient matrix computation with the mapreduce framework," in *CloudCom*. IEEE, 2010, pp. 721–726.
[31] "Titan," http://titan.thinkaurelius.com/.
[32] R. Amitabha, B. Laurent, M. Jasmina, and Z. Willy, "Chaos: Scale-out graph processing from secondary storage," in *Proc. of SOSP*. ACM, 2015, pp. 410–424.
[33] 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*. ACM, 2013, p. 9.
[34] 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*. IEEE, 2013, pp. 193–198.
[35] 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*. ACM, 2014, p. 1.
[36] 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*. ACM, 2012, pp. 85–98.
[37] D. Yan, J. Cheng, Y. Lu, and W. Ng, "Effective techniques for message reduction and load balancing in distributed graph computation," in *Proc. of WWW*, 2015, pp. 1307–1317.
[38] "Apache spark," http://spark.apache.org/.
[39] Z. Wang, Y. Gu, Y. Bao, G. Yu, and L. Gao, "An i/o-efficient and adaptive fault-tolerant framework for distributed graph computations," *Distributed and Parallel Databases*, vol. 35, no. 2, pp. 177–196, 2017.
[40] Y. Shen, G. Chen, H. Jagadish, W. Lu, B. C. Ooi, and B. M. Tudor, "Fast failure recovery in distributed graph processing systems," *In Proc. of the VLDB Endowment*, vol. 8, no. 4, pp. 437–448, 2014.
[41] M. Pundir, L. M. Leslie, I. Gupta, and R. H. Campbell, "Zorro: Zero-cost reactive failure recovery in distributed graph processing," in *Proc. of SoCC*. ACM, 2015, pp. 195–208.
[42] S. Schelter, S. Ewen, K. Tzoumas, and V. Markl, "All roads lead to rome: optimistic recovery for distributed iterative data processing," in *Proc. of CIKM*. ACM, 2013, pp. 1919–1928.



Zhigang Wang received the PhD degree in computer software and theory from Northeastern University, China, in 2018. He is currently a lecturer in the College of Information Science and Engineering, Ocean University of China. His research interests include cloud computing, distributed graph processing and machine learning. He is a member of the China Computer Federation (CCF). He received the CCF Outstanding Doctoral Dissertation Award in 2018.

Yu Gu received the PhD degree in computer software and theory from Northeastern University, China, in 2010. Currently, he is a professor and the PhD supervisor at Northeastern University, China. His current research interests include big data analysis, spatial data management and graph data management. He is a senior member of the China Computer Federation (CCF).

Yubin Bao received the PhD degree in computer software and theory from Northeastern University, China, in 2003. Currently, he is a professor at Northeastern University, China. His current research interests include data warehouse and OLAP, graph data management, and cloud computing. He is a senior member of the China Computer Federation (CCF).

Ge Yu received the PhD degree in computer science from Kyushu University of Japan, in 1996. He is currently a professor and the PhD supervisor at Northeastern University of China. His research interests include distributed and parallel database, OLAP and data warehousing, data integration, graph data management, etc. He is a member of ACM, a senior member of IEEE, and a Fellow of the China Computer Federation (CCF).

Jeffrey Xu Yu has held faculty positions with the University of Tsukuba and Australian National University. Currently, he is a professor in the Department of Systems Engineering and Engineering Management, Chinese University of Hong Kong, Hong Kong. His current research interests include graph processing and social network analysis. He is a member of ACM and a senior member of IEEE.

Zhiqiang Wei received the PhD degree from Tsinghua University, China, in 2001. He is currently a professor with the Ocean University of China. He is also the director of High Performance Computing Center at the Pilot National Laboratory for Marine Science and Technology (Qingdao). His current research interests are in the fields of intelligent information processing, social media and big data analytics. He is a member of IEEE and CCF.