Efficient Graph Processing with Invalid Update Filtration

Long Zheng, *Member, IEEE*, Xianliang Li, Xi Ge, Xiaofei Liao, *Member, IEEE*, Zhiyuan Shao, *Member, IEEE*, Hai Jin, *Fellow, IEEE*, and Qiang-Sheng Hua, *Member, IEEE*

Abstract—Most of existing graph processing systems essentially follow pull-based computation model to handle compute-intensive parts of graph iteration for high parallelism. Considering all vertices and edges are processed in each iteration, pull model may suffers from a large number of invalid (vertex/edge) operations that do not contribute to graph convergence, leading to potential performance degradation. In this paper, we have the insight that these invalid operations can be filtered by leveraging a small fraction of critical information. However, most of critical information are often beyond the visibility of active vertices being processed. We present two novel filtration approaches to (cooperatively) identify out-of-visibility critical information with boundary-cut heuristics and speculative prediction for many graph algorithms.We have integrated both approaches and their hybrid solution into three state-of-art graph processing systems (including Ligra, Gemini, and Polymer). Experimental results using a wide variety of graph algorithms on both real-world and synthetic graph datasets show that neither of these approaches can have an absolute win for all graph algorithms. Boundary-cut, predictive, and hybrid approaches can improve the performance by 115.1, 38.1, and 136.6 percent on average.

Index Terms—Graph processing, pull computation model, invalid update, performance

1 Introduction

Graph has been widely used to represent the connections between different entities for many real-world applications, e.g., distributed optimization [1], web search [2], and social network analysis [3]. As graph scale is increasingly expanding with billions or even trillion edges, there becomes a high demand for graph processing in seeking top performance. Existing graph processing systems basically follow pull-based [4], [5], [6], [7] computation model (propagating the data via the in-coming edges) to parallelize the dense phase of graph processing where a very large number of active vertices need to be processed.

Not knowing which vertices would be activated, pull model in each iteration always has to schedule all vertices of input graph with their in-coming edges. Unfortunately, most of these scheduled vertices do not necessarily contribute to a *valid update* in the sense that the value of vertex can be modified with a new value. It is observed that these invalid operations for many graph algorithms consume a large number of compute and memory-bandwidth resources, leading to significant performance degradation [8], [9] (as will be also discussed in Section 2.2). In actual, invalid updates make no contribution to graph convergence, and they also do not impact the final result [9], [10]. It is therefore a natural consequence of skipping the invalid operations for enhancing the performance of graph processing.

Notify-pull model enforces that only vertices associated to activated vertices can be scheduled [11], but it still suffers from invalid updates since all converged vertices associated to the activated vertices will also be notified and scheduled. Break-early mechanism [8], [12] skips the processing of the remaining edges if a vertex finds a visited neighbor, but this optimization is only suitable for BFS, which has unique feature that all of its vertices are just traversed only once. $\triangle$-stepping SSSP [13] prioritizes to schedule the vertices with lower tentative distance to reduce the number of relaxation operations. Disjoint-set based CC [14] adopts disjoint-set to merge vertices in the same component into one set while scanning each edge only once. Unlike these algorithm-by-algorithm specialized designs that are often hard to understand and generalize [15], this work is devoted to a common, easy yet fast solution that can be applied for different algorithms with invalid operations filtered as well.

In this paper, we follow the convergence-guided ideology [8], [12] to filter the invalid operations. It has the basic principle that, if a given vertex has been already converged, all of its (remaining) associated edges can hence have no necessity of processing. It is notoriously difficult, if not impossible, to accurately judge the convergence of a vertex before it is finished due to several tremendous challenges.

First, vertex convergence is highly associated to the operation semantics of graph algorithm. For instance, vertex in CC can be converged by getting a minimal label while SSSP does

• *The authors are with the National Engineering Research Center for Big Data Technology and System, Services Computing Technology and System Lab, Cluster and Grid Computing Lab, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan 430074, China.*
E-mail: {longzh, xianliang, xge, xfliao, zyshao, hjin, qshua}@hust.edu.cn.

Manuscript received 29 Nov. 2018; revised 17 May 2019; accepted 23 May 2019. Date of publication 7 June 2019; date of current version 29 June 2021.
(Corresponding author: Xiaofei Liao.)
Recommended for acceptance by M. Piccardi.
Digital Object Identifier no. 10.1109/TBDATA.2019.2921358

2332-7790 © 2019 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See https://www.ieee.org/publications/rights/index.html for more information.

Authorized licensed use limited to: Huazhong University of Science and Technology. Downloaded on March 14,2023 at 12:58:49 UTC from IEEE Xplore. Restrictions apply.

so by finding a shortest path [16], [17]. Finding a shortest path for SSSP is often more complex than identifying a minimum label for CC [13]. In addition, even for the same graph algorithm, vertex convergence is notoriously difficult to identify due to the non-determinism of edge scheduling [18], [19]. This dynamic property results in non-trivial efforts to analyze the runtime status of vertex for convergence judgment. Second, vertex convergence is closely related to the inherent topology of graphs as well. A most intuitive situation for understanding this point is that, for a given graph, if we have gotten rid of a (few) particular edge(s) across vertices, the convergence of each vertex may have a significant difference before and after graph modification [20], [21]. This also makes it difficult to make an accurate judgment for vertex convergence with limited information of graph topology.

We have the insight that the vertex convergence for many graph algorithms (e.g., CC and SSSP) can be judged by leveraging a small fraction of critical information from a few (instead of all) vertices. For instance, in Dijkstra's single shortest path theory [17], a vertex can be considered converged if the shortest path among all unconverged vertices is found, even though not all vertices are converged. Unfortunately, most of critical information are often beyond the visibility of active vertices being processed. This is particularly true for existing graph systems using vertex-centric programming model [16], [22], [23] where each vertex can only capture the information from its neighbors.

One interesting and important observation, we have exploited in this work, is that these critical information during graph iteration for a wide variety of real-world graph algorithms often behaves two significant features.

First, vertex value for many graph algorithms (such as CC and SSSP) is monotonically-changing. For instance, all vertices in CC attempt to find a smaller label. This enables to find a strict boundary line, which can clearly separate vertices into the absolutely converged collection and uncertain collection. Once a vertex value that falls into the boundary can be then judged as converged immediately. Second, a vertex that has not been activated in the past few iterations for many graph algorithms have a high possibility of convergence. For instance, vertices for PageRank-Delta on *twitter-2010* have more than 99.2 percent possibility to be converged if it has not been activated in the last two iterations (as witnessed in Section 4.1). This allows to predict the convergence of vertices with a speculative decision.

This paper makes the following contributions:

- We make an in-depth comprehensive study to understand the performance issues arising from invalid operations for pull-based computation model (Section 2).
- We present a suite of novel filtration approaches, which can break through the local visibility limitation to capture critical information for the accurate and efficient vertex convergence judgment (Sections 3, 4, and 5).
- We integrate our filtration approaches into three state-of-the-art graph processing frameworks (including Ligra [9], Gemini [23], and Polymer [10]). Experimental results show boundary-cut, predictive, and hybrid judgment can improve the performance of graph processing by 115.1, 38.1, and 136.6 percent on average, respectively. (Section 6).

The rest of this paper is organized as follows. Section 2 describes background and motivation. Sections 3 and 4 present our invalid update filtration approaches. Section 5 puts these two approaches together. Section 6 shows the results. We survey related work in Sections 7 and 8 concludes this work.

2 Background and Motivation

We first review basic terminologies for parallel graph processing, followed by a motivation study for understanding the invalid update problem and their potential impact, finally motivating our approach.

2.1 Parallel Graph Processing

Graph (i.e., G) often consists of vertex (i.e., V) and edge (i.e., E), denoted as $G = (V, E)$. Vertex represents the entity while associated edges between entities indicate their relationship. For a directed graph, its edge often points from a source vertex to a destination vertex.

Vertex-Centric Programming. This model processes the graphs via "Think like a vertex" philosophy [2]. For each vertex, it has three basic steps with data gather, vertex update, and updated value scatter. Thus, programmers only need to focus on programming these three meta-operations on vertex. Vertex-centric programming model is easy to use, parallelize, and scale [16], [22], [24], [25].

Push Versus Pull. For a given vertex, there are two ways for message propagation. *Push* model schedules on the source vertex, and delivers its updated value to the (neighboring) destination vertices. It maintains a set of active vertices, also known as *Frontier*, to indicate which vertices should be computed during the iteration. *Pull* model schedules on the destination vertex. The core operations for each vertex have two types: Gather and Update. Gather operation collects information from in-coming vertices and their associated edges. Update operation tries to compute a new value of destination vertex with collected value(s). Note that all (active and inactive) in-coming source vertices will be scanned to gather their values in each iteration, leading to useless edge processing that does not necessarily contribute to valid vertex update (as will be discussed in Section 2.2).

2.2 Invalid Updates of Pull Model and Their Performance Impact: A Motivating Study

Invalid Updates of Pull Model. Invalid update indicates that the gathered information from a neighboring vertex via in-coming edge does not make a successful update on destination vertex. In pull scheduling process, for each edge *vSrc* → *vDst*, *vSrc* attempts to update *vDst* with the collected information, but this operation might not always be successful unless the given update condition is satisfied.

Fig. 3 depicts an illustrative example of how invalid updates incur under pull model for *Single-source Shortest Path* (SSSP) where update condition is that the cumulative value is less than the current value of the destination vertex. For vertex holding distance 9, it has three associated edges. Pull model will gather the information from these edges, and further check whether they can update it. For instance,

Authorized licensed use limited to: Huazhong University of Science and Technology. Downloaded on March 14,2023 at 12:58:49 UTC from IEEE Xplore. Restrictions apply.

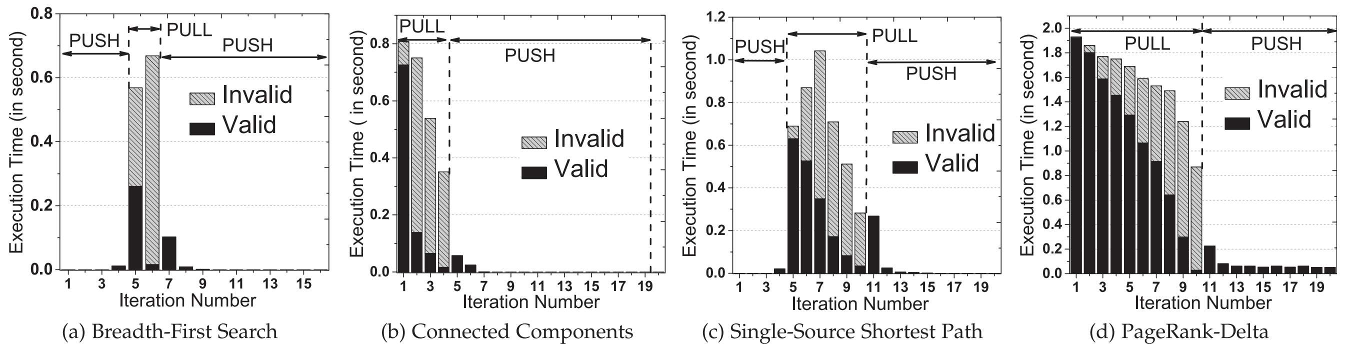


Fig. 1. Execution time breakdown over iteration. *Invalid* indicates the wasted time arising from invalid updates. Results for all tested graph algorithms are conducted based on twitter-2010.

the total path via in-coming edge $\overset{6}{\rightarrow}$ is 10, which is greater than the original value (9) of destination vertex. In-coming edge $\overset{4}{\rightarrow}$ has the similar result. Thus, no valid updates will be made until the edge $\overset{3}{\rightarrow}$ is processed.

More serious is that pull model is oblivious to the convergence of a vertex. It has to check its update procedure repeatedly, even though this vertex has already been converged. These invalid updates have extremely large number, which can be easily millions or even billions for large graphs [8], [12]. Also, they may consume a large amount of computational and bandwidth resources, resulting in the significant performance degradation [26].

Experimental Demonstration. We conduct a set of experiments to witness the performance impact of invalid updates on *twitter-2010*. Fig. 1 illustrates the results. We capture an execution snapshot of the first 20-time iteration using a hybrid computation model. It shows that pull phase basically occupies most of execution time. Particularly, pull phase for all graph algorithms involve a wealth of performance loss due to invalid updates. Invalid updates cause significant performance loss for BFS, CC, SSSP, and PageRank-Delta by 71.8, 61.5, 46.3, and 35.7 percent, respectively.

2.3 Overview of Our Approach

The key contribution of this work stems from the following observation that can greatly contribute to the easy, fast, and accurate convergence judgment of a vertex.

Observation. For a vertex in vertex-centric program, a small fraction of critical information out of its visibility facilitates the convergence judgment of this vertex.

Fig. 2 elaborates this observation via an example. For CC, all vertices try to replace their labels with a minimal label from neighbors. Suppose all vertex labels are initialized to vertex numbered identifier. For `vertex 2`, only associated `vertex 1` and `vertex 4` are visible to it. That is, the rest of graph structure is out of its visibility.

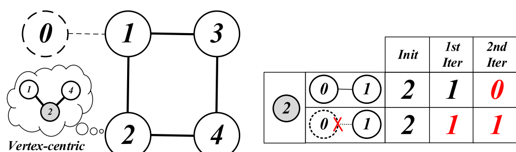


Fig. 2. An illustrative example of CC for elaborating our observation that a few out-of-visibility critical information is pretty helpful in facilitating the judgment of vertex convergence.

Let us consider the convergence of `vertex 2`. Assume `vertex 0` is not associated to `vertex 1`, `vertex 2` will be clearly converged with label 1 at the first iteration. Suppose `vertex 0` is associated to `vertex 1`, `vertex 2` will be converged with label 0 at the second iteration. In this graph, the association between `vertex 0` and `vertex 1` is the key to determine the convergence of `vertex 2`. However, this critical information is beyond the visible scope of `vertex 2` when it is active during iteration. We therefore have the insight of *whether and how we can capture these unavailable critical information in advance to help judge the convergence of a vertex for filtering its invalid updates.*

Unfortunately, it is a well-known hard problem to capture the critical information of vertex in a complete and accurate manner due to the complex data dependencies and intertwined topology of graphs. In this work, we recognize two major forms of critical information that can be identified easily for a wide variety of real-world graph algorithms.

Boundary-Cut Judgment. (Section 3) Many graph algorithms involve a common processing flow that, all vertices try to search an optimal value until no better results can be found for program convergence. For example, vertices in CC search for a minimal label. These algorithms aim at finding a boundary that is used for distinguishing converged vertices from the rest. A simple example is that in CC all vertices have to be attached to a vertex with minimal label in that component. Once a vertex finds the minimum label, it is definitely converged. This motivates us to develop a boundary-cut approach to judge the vertex convergence by finding such a bounded threshold for each vertex.

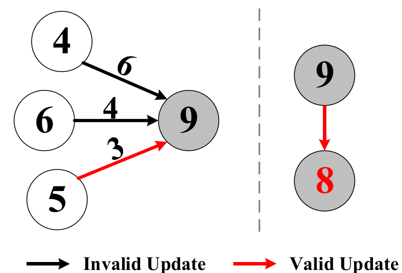


Fig. 3. Invalid updates for SSSP algorithm. The number inside a given vertex represents the shortest distance from source vertex to this vertex. The number on each edge indicates the distance from given source vertex to the point-to destination vertex. A valid update means that the source vertex has successfully updated the destination vertex with a new value. Otherwise it makes an invalid update.

Authorized licensed use limited to: Huazhong University of Science and Technology. Downloaded on March 14,2023 at 12:58:49 UTC from IEEE Xplore. Restrictions apply.

TABLE 1
Update Policies for Several Common Graph Algorithms

	update(u, v)
BFS	value[v] = MIN{value[u]+1, value[v]}
CC	value[v] = MIN{value[u], value[v]}
SSSP	value[v] = MIN{value[u]+weight(u, v), value[v]}

Predictive Judgment. (Section 4) For many graph algorithms, it is observed that their inactive vertices can be already converged with a high possibility if they are not re-activated in the last few rounds of iterations. For instance, our experiment in Section 4.1 shows that vertex for PageRank-Delta has over 99.2 percent possibility to be converged once it has not been activated during last two iterations. According to the vertex activation history, this also motivates us to develop a predictive approach to judge the vertex convergence with speculative decision.

We note that both vertex convergence judgment approaches have their respective benefits. No one can have an absolute win for all graph algorithms. Therefore, a hybrid solution between boundary-cut and predictive judgment is potentially beneficial (Section 5).

3 Boundary-Cut Judgment

This section elaborates the boundary-cut judgment, the key idea behind which lies on finding an appropriate boundary threshold for every graph algorithm. The vertices that satisfy the threshold condition can be considered converged. This approach is simple yet effective in quickly judging the vertex convergence for many graph algorithms.

3.1 Problem Statement

Table 1 depicts the core policies of vertex update for BFS, CC, and SSSP. All of them follow a relaxation-based fashion in the sense that vertices hold an upper bound value and will replace it once it gets a better one, with the monotonically-changing feature for vertex update. For this type of graph algorithms, when a vertex can be updated, at least two cases have to be considered:

Case 1. Suppose two vertices are connected directly. We define the condition when vertex u can update v as C_d

$$C_d(u,v) = ((u,v) \in E) \wedge (\text{update}(u,v) = \text{valid}). \tag{1}$$

C_d describes the requirement of u updating v by following two conditions: 1) u and v are associated, and 2) v can gather a better value from u (as depicted in Table 1).

Case 2. Suppose vertex u and v are not connected directly in graph. We define the condition when u can update v as C_c. Assume there are a finite set of vertices $c_i \in V$ where $i = \{0, 1, \ldots, k\}$

$$C_c(u,v) = C_d(u,c_0) \wedge C_d(c_0,c_1) \wedge \cdots \wedge C_d(c_k,v). \tag{2}$$

C_c implies u can update v via a chain of valid C_d conditions. u first updates its neighbors that then update their neighbors, and repeat this procedure until v can be reached.

Note that the sufficient condition for a vertex v being converged is that no vertices u can be found to meet the condition C_c. Our goal is to find those converged vertices as will be discussed in Section 3.2.

TABLE 2
Threshold and Convergence Condition for Different Graph Algorithms

	Threshold	Condition
BFS	MIN{value $[u] \| u \in Frontier$}	value[v] $<= thres + 1$
CC	MIN{value $[u] \| u \in Frontier$}	value[v] $<= thres$
SSSP	MIN{value $[u] \| u \in Frontier$}	value[v] $<= thres + Weight_{min}$

3.2 Calculating Boundary Threshold

By above Equation (2), all vertices that are not converged can be defined as follow:

$$S = \{v | v \in V, \exists u \in Frontier, C_c(u,v) = \text{TRUE}\}. \tag{3}$$

Equation (3) indicates all vertices that should be computed in the next iterations. All these vertices must be updated through a path from current set of vertices in *Frontier*. However, in practice it is hard to check whether a vertex is converged since it is non-trivial to check the existence of these paths, which is closely related to the graph topology.

We present a relaxed version that is insensitive to the graph topology. For facilitating the description, we define the relationship that the value of vertex u can induce the successful update of vertex v as $value(v) \preceq value(u)$. The main idea of relaxed solution is to simply use the monotonic features of vertex values without traversing the dependency relationship of graph topology. For example, in CC only those vertices with smaller labels have potential to update the ones with bigger labels, but not the converse. This inherent relationship $\preceq$ is insensitive to the graph topology, and deduces a relaxed definition of Equation (3) as follow:

$$S' = \{v | v \in V, \exists u \in Frontier, value(v) \preceq value(u)\}. \tag{4}$$

Note that $\preceq$ has a transitive relation. This property allows to check only once with optimal value in *Frontier* to update the concerned vertices, regardless of whether concerned vertices can be updated by the one(s) in *Frontier*. As a result, the next key question is to find an appropriate $thres$ that can isolate the potential vertices as follow:

$$thres = \{value(u) | u \in Frontier, \forall v \in Frontier, value(v) \preceq value(u)\}. \tag{5}$$

On the contrary, vertices that its value will never be successfully updated can be defined as below:

$$\overline{S'} = \{v | v \in V, value(v) \preceq thres\}. \tag{6}$$

Equation (6) can be used to separate all vertices into a definitely converged set and potentially unconverged set. Table 2 shows the threshold and convergence condition of BFS, CC, and SSSP, all of which basically have the similar threshold calculation. Note that the threshold computed from Table 2 is optimal (rather than set empirically) in each iteration, because a larger threshold may occur the incorrect

Authorized licensed use limited to: Huazhong University of Science and Technology. Downloaded on March 14,2023 at 12:58:49 UTC from IEEE Xplore. Restrictions apply.

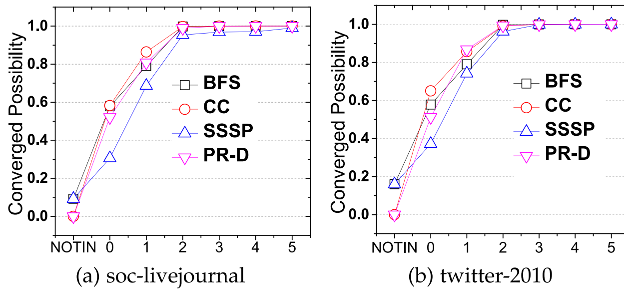


Fig. 4. Convergence possibility over M_g for different graph algorithms. NOTIN means vertex will not be processed during the whole processing.

results while a smaller threshold can not ensure the maximum degree of invalid updates being exploited. By contrast, our threshold used can promise both.

It is easy to understand that threshold-based method can offer the benefits for real-world graph applications. BFS has the unique feature of traversing vertices level by level, and only needs to find one in-coming edge associated with a vertex in frontier, which is easy. SSSP is relatively harder since value updated depends on the propagation path. However, our threshold can still work effectively and efficiently by finding a shortest path to satisfy the convergence condition as shown in Table 2.

Boundary-cut method can filter invalid processing for: 1) not only all edges of a vertex that has been already converged, and 2) but also the remaining edges of a vertex that is being processed to be converged. Case 1) is easy to understand. Case 2) is similar to break-early functionality in BFS where the vertex convergence can be easily judged by whether this vertex have been traversed [8], [12]. In the high-level design, the threshold used in our boundary-cut method also splits the vertices into the converged (i.e., visited) vertices and unconverged (i.e., unvisited) ones. The break-early functionality (useful for BFS only) can be understood as a special version of our boundary-cut method, which can be useful for not only BFS but also many other graph algorithms such as CC and SSSP.

3.3 Case Study: Connected Component

Algorithm 1 shows the modified CC by enabling boundary-cut judgment with the following differences compared to the plain CC. Before each iteration, a threshold label will be computed through FrontierScan (Lines 5-9), which aims to gather the minimal label from *Frontier*. We parallelize the gather procedure with parallel reduction. Once *Threslabel* is obtained, vertex program can judge the vertex convergence based on the comparison result of vertex label on *ThresLabel* via Converged (Line 2). Once a vertex has converged, its rest edges can be omitted as well.

Compared with the plain workflow of CC, the refined one only modifies to add a threshold label computation phase, which is easy to parallelize with sequential memory accesses. The threshold label for CC can be effectively used to filter converged vertices, because largest connected component often contains most vertices and edges [27]. All the vertices with the minimal label can be judged as converged, significantly reducing the total number of invalid updates.

Algorithm 1. CC with Boundary-Cut Judgment

1 **Procedure** Converged(*ID*, *ThresLabel*)
2 **return** *ID.Label* $\leq$ *ThresLabel*
3
4 **Procedure** FrontierScan(*Frontier*)
5 *ThresLabel* $\leftarrow +\infty$
6 /*Find minimal Label in Frontier*/
7 **for** $v \in$ *Frontier* **do**
8 *ThresLabel* $\leftarrow$ Min(v.*Label*, *ThresLabel*)
9 **return** *ThresLabel*
10
11 **Procedure** Compute(G)
12 *Frontier* $\leftarrow V$
13 *V.Label* $\leftarrow$ {0, 1, 2, 3, ..., n-1} /*Label initialization*/
14 **while** *Frontier* $\neq \emptyset$ **do**
15 **if** $|$ *Frontier.V*$|$ + $|$*Frontier.E*$|$ $>$ CriticalPoint **then**
16 *ThresLabel* $\leftarrow$ FrontierScan(*Frontier*)
17 Pull_Model(*G*, *Frontier*, *Threslabel*)
18 **else**
19 Push_Model(*G*, *Frontier*)

4 Predictive Judgment

In this section, we present an alternative approach to speculatively predict the convergence of vertex according to a history of update information.

4.1 Empirical Rule

We define M_g to represent the longest number of consecutive invalid vertex updates for a vertex during pull iteration. For example, $M_g = 2$ indicates that vertex has suffered two consecutive invalid updates before it is converged. Note that a number of vertices in SSSP may be never updated since they are isolated from the source vertex. We hence assign their M_g with NoTIN. For each vertex, we observe that its convergence obeys the following rule:

$$F(k) = P\{M_g <= k\}, \tag{7}$$

which indicates the convergence possibility of a vertex if it has consecutively suffered from invalid updates in the last k iterations. Fig. 4 shows the convergence possibility over M_g. All results are obtained from an offline trace analysis. We can find that vertices in a variety of graph algorithms have a very high ($> 95.3\%$) possibility of convergence once they have not been reactivated in the last two iterations.

4.2 Rule-Guided Speculation

Guided by the aforementioned rule, following branch prediction ideology [28], we propose a *Finite State Machine* (FSM) to schedule the vertex convergence by following $M_g = 2$. Fig. 5 shows the state transition diagram of FSM, which has six states below.

- *Initial*: This state indicates that the vertex is either initialized, or have suffered a valid update.
- *Pending*: This state indicates that the vertex has an invalid update from the Initial state. It will turn into Sleep state immediately if it has an invalid

Authorized licensed use limited to: Huazhong University of Science and Technology. Downloaded on March 14,2023 at 12:58:49 UTC from IEEE Xplore. Restrictions apply.

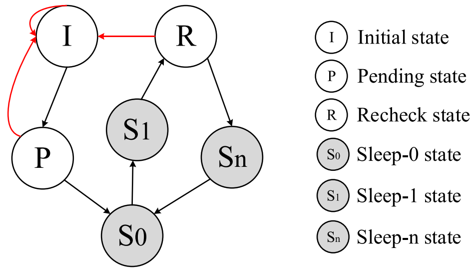


Fig. 5. Finite state machine with $M_g = 2$.

update once more. Otherwise, it will return back to `Initial`.

- *Sleep-0*: This state indicates that the vertex has at least two invalid updates. Vertices at the `Sleep-0` state will not be scheduled in the next iteration.
- *Sleep-1*: This state is automatically transferred from `Sleep-0` through one iteration. Vertices at the `Sleep-1` state will not be scheduled in the next iteration.
- *Recheck*: This state is automatically transferred from `Sleep-1` through one iteration. Vertices will be scheduled to recheck if they have been converged. `Recheck` state is designed for reducing negative effect of false prediction.
- *Sleep-n*: This state is transferred from `Recheck` with an invalid attempt. Vertices will not be scheduled, and turn into `Sleep-0` in the next iteration.

The FSM works as follows. All vertices are first labeled with `Initial` state, which will be changed based on the state transition diagram. All vertices with `Sleep`-prefixed states will not be scheduled. A vertex can trap into the `Sleep`-prefixed state with the premise that M_g consecutive invalid updates must suffer. `Pending` state is the very intermediate state to represent this. Therefore, if more consecutive invalid updates with larger M_g involved, extra pending state in FSM needs to be added between `Initial` state and `Sleep` state.

4.3 Handling False Speculation

For the correctness, we next discuss how we can justify the false speculation, which has two basic cases: 1) A few converged vertices are falsely judged as unconverged; 2) A few unconverged vertices are falsely judged as converged. Consider that the first case does not affect the final result. We focus on addressing the second case.

In practice, for these false unconverged vertices, we conservatively invoke one extra all-pull iteration before the normal model switching. Specifically, when the number of active vertices and edges is less than the critical point for model switch, we will restore an all-pull model to identify those mis-predicted vertices. All active vertices can be then held in *Frontier* to advance push execution. We note that the graph algorithms that involve no more than two pull iterations do not need to invoke one all-pull iteration.

Correctness Semantics. We informally show that our predictive judgment can behave as an asynchronous scheduling. In asynchronous scheduling model, active vertices will be held in a set and scheduled independently. As widely-studied in prior studies [22], [29], [30], it is proved that a wide variety of graph analytics applications can be ensured to be converged as long as all of its required vertices can been processed, no matter when and how they are scheduled during the whole graph processing. In our predictive judgment, all mis-predicted vertices will be definitely scheduled in the all-pull iteration in a way that the mis-prediction just defers the schedule order of these vertices. In this sense, our predictive judgment can be understood as a certain form of asynchronous model, which has been already proved to ensure the correctness of graph analytics applications with vertex-level consistency [22], [29], [30].

Algorithm 2. SSSP with Predictive Judgment

```
1  Procedure Converged(ID, Std)
2    flag ←Whether vertex ID is scheduled based on Std
3    return flag
4
5  Procedure FrontierScan(G, Frontier, Std)
6    /* shift state according to the update result */
7    for v ∈ V do
8      if v ∈ Frontier then
9        v.State ← Std.Initial
10     else
11       v.State ← Next status in Std
15
13 Procedure Compute(G)
14   Std ← Input state transition diagram
15   V. Distance ← {+∞, +∞, . . . , +∞}
16   V. State ← {I, I, ..., I} /* State initialization */
17   vSrc.Distance ← 0
18   Frontier += vSrc
19   while Frontier ≠ ∅ do
20     if |Frontier.V| + |Frontier.E| > CriticalPoint then
21       Pull_Model(G, Std, Frontier)
22       FrontierScan(G, Frontier)
23     else
24        if Last more than two iterations are in predictive
          mode then
25         /* All pull to re-identify active vertices */
26         AllPull_Model(G, Frontier)
27       else
28         Push_Model(G,Frontier)
```

4.4 Case Study: Single-Source Shortest Path

Algorithm 2 describes the modified SSSP by enabling predictive judgment. We define a structure *Std* for state transition diagram (Line 14). In the beginning, all vertices will be set to `Initial` state (Line 16). In each iteration, FrontierScan will be invoked to evaluate whether and how to shift the state according to the state transition diagram (Lines 7-11). Vertices with `Sleep`-prefixed states will not be scheduled (Lines 2-3). For the correctness, an all-pull iteration will be restored to find active vertices (Line 24).

Compared with the plain SSSP, the refined one modifies to add a status structure for each vertex and perform the scheduling according to a history of information in the last few iterations. All memory accesses in FrontierScan are sequential. For each vertex, we only need 3 bits (to represent six states), which are pretty lightweight in contrast to the whole graph storage. We particularly note that our FSM with $M_g = 2$ can be also extended to represent different graph algorithms with different M_g.

Authorized licensed use limited to: Huazhong University of Science and Technology. Downloaded on March 14,2023 at 12:58:49 UTC from IEEE Xplore. Restrictions apply.

TABLE 3
Characteristics of Different Filtration Approaches

Approach	Algorithm Features	Typical Algorithms
Boundary-cut	monotonic	BFS, CC, SSSP, Reachability
Predictive	predictable	BFS, CC, SSSP, PR-D, MIS, GC
Hybrid	monotonic & predictable	BFS, CC, SSSP
N/A	always-active, non-recursive	PageRank, TC

5 Put it All Together

Table 3 shows the details of our invalid filtration approaches and their applicability scope. It can be seen that boundary-cut approach works for graph algorithms where iterative values of vertices are monotonically changing such that a threshold can be obtained to make a clear distinction. In contrast, predictive approach has the requirement that vertex convergence is predictable in accordance with their update histories.

Hybrid Judgment. Note that many graph algorithms are not only monotonic but also predictable. We present a hybrid solution to combine both boundary-cut and predictive approaches for taking their respective advantages. Towards a hybrid design, we have two conditions of judging the convergence towards a vertex. Both conditions have their appropriate timing for convergence judgment on an applicable vertex. No one can be always better for all vertices.

Actually, both conditions can co-judge the vertex convergence without applicability conflicts in practice. We propose a cost-efficient hybrid method to apply two conditions independently. In this case, a vertex can be aggressively considered converged once it satisfies either of these two conditions. Specifically, we simply use an OR operation in the judgment condition in Converged function. This condition is relatively relaxed, and hence, more converged vertices will benefit from it. In hybrid method, predictive judgment gets involved from the 3rd iteration (since $M_g = 2$ in this paper). In other word, no misjudgment occurs in the first two pull iterations, enabling that those vertices judged as converged by boundary-cut method can be no longer processed in the all-pull iteration, further reducing a wealth of vertex rescheduling overhead in the all-pull iteration.

Discussion. We note that there still involve a very few graph algorithms that fit into neither of our approaches. For instance, PageRank needs to update all vertices in all iterations without involving invalid updates. Triangle Counting (TC) can be finished within only one iteration [9]. Despite these, the practicability of our work is still not limited since little runtime overhead is involved in our designs, as will be discussed in Section 6.5.

6 Evaluation

In this section, we evaluate the efficiency and effectiveness of our work for a wide variety of widely-used graph algorithms on both real-world and synthetic graph datasets.

6.1 Experimental Setup

We have integrated our approaches into three state-of-art hybrid graph processing frameworks, including Ligra [9], Gemini [23], and Polymer [10]. We change nothing for graph preprocessing, and use the build-in setting of each graph system for their pull-push mode switching by default. The statistical processing time in our tests only embraces the execution time. Note that we use the state transition diagram (as described in Fig. 5) for our predictive judgment.

TABLE 4
Graph Datasets

Dataset	$\lvert V\rvert$	$\lvert E\rvert$	Avg. Degree
enwiki-2013	4.2M	101.4M	24.1
soc-Livejournal	4.8M	69.0M	14.2
road	23.9M	58.3M	2.4
webbase	118.1M	1019.9M	8.6
twitter	61.6M	1468.4M	23.8
friendster	124.8M	1806.1M	14.5
rmat27	134.2M	2147.4M	16
rmat28	268.4M	4294.9M	16
rmat24-E(k)*	16.8M	$2^{24} \times k$	k

*We make $k = 8 \times i$ (where $0 < i < 9$) to generate graphs with different average degrees.

Graph Algorithms. We benchmark a wide variety of well-known graph algorithms as shown in Table 3, covering different categories and complexities:

- *Connected Components (CC)*, aiming at finding a maximal number of subgraphs where any two vertices can be connected via a chain of paths.
- *Single-source Shortest Path (SSSP)*, aiming at finding a path of a given vertex to every vertex such that the sum of the weights of their constituent edges is minimized.
- *PageRank-Delta (PR-D)*, aiming at getting the relative importance of factors named PR value in the given graph.
- *Maximal Independent Set (MIS)*, aiming at finding a maximal set of independent vertices from a given graph.
- *Graph Coloring (GC)*, aiming at coloring the graph with a least number of colors while ensuring that no two adjacent vertices are of the same color.
- *Breath-First Search (BFS)*, aiming at traversing the graph hop by hop to search the depth to all vertices from a root.

Note that Gemini does not provide PR-D, MIS, and GC. Ligra and Polymer do not provide GC. We implement the GC with the greedy algorithm [16]. We also complement to implement other absent graph algorithms for each system.

Graph Datasets. We benchmark all graph algorithms with a variety of graph collections, including: 1) six real-world graphs (coming from Stanford Large Network Dataset Collection[1] and Laboratory for Web Algorithmics[2]), and 2) two synthetic graphs (generated by the RMAT tool [31]). Table 4 depicts the details of graph datasets.

Platform. All experiments for are performed on a machine equipped with 2×Intel 14-core Xeon E5-2680 v4@2.40 GHz, 256 GB main memory, and 1 TB hard disk. We also deploy a

1. http://snap.stanford.edu/data
2. http://law.di.unimi.it/datasets.php

Authorized licensed use limited to: Huazhong University of Science and Technology. Downloaded on March 14,2023 at 12:58:49 UTC from IEEE Xplore. Restrictions apply.

TABLE 5
Execution Time (in Seconds) of Ligra, Gemini, Polymer with or without Using Boundary-Cut (-**B**), Predictive (-**P**), and Hybrid (-**H**) Judgment

		soc	enwiki	road	webbase	twitter	friendster	rmat27	rmat28
CC	Ligra	0.08	0.12	20.91	2.85	2.53	5.22	7.99	11.89
	Ligra-B	0.04	0.05	18.83	1.43	0.44	2.93	2.78	4.17
	↑ perf.	132.3%	141.6%	11.2%	99.3%	475.3%	78.2%	187.4%	185.1%
	Ligra-P	0.07	0.11	13.50	2.44	2.47	4.53	7.46	11.19
	↑ perf.	12.1%	8.3%	54.8%	17.3%	2.4%	15.2%	7.1%	6.3%
	Ligra-H	0.04	0.04	11.63	1.21	0.41	2.88	2.55	3.75
	↑ perf.	137.6%	208.3%	80.3%	135.5%	517.1%	81.3%	213.3%	217.1%
	Gemini	0.04	0.13	12.43	2.03	2.06	6.35	5.13	9.74
	Gemini-B	0.03	0.06	10.62	1.37	0.47	3.12	1.88	3.47
	↑ perf.	37.1%	116.7%	17.1%	48.2%	337.6%	103.5%	173.9%	181.7%
	Gemini-P	0.04	0.11	8.75	1.74	1.89	5.98	4.98	8.76
	↑ perf.	3.3%	13.2%	42.1%	16.6%	9.3%	6.2%	3.4%	11.2%
	Gemini-H	0.03	0.06	8.52	1.28	0.45	2.76	1.62	2.91
	↑ perf.	38.2%	124.3%	45.9%	58.6%	357.8%	130.1%	216.7%	234.7%
	Polymer	0.11	0.18	18.63	2.31	2.37	5.97	6.77	9.13
	Polymer-B	0.05	0.07	17.62	1.39	0.62	3.45	2.04	3.58
	↑ perf.	123.4%	157.1%	5.7%	66.2%	282.3%	73.1%	231.9%	155.1%
	Polymer-P	0.11	0.17	11.75	1.97	2.17	5.63	6.13	8.87
	↑ perf.	4.7%	6.2%	59.4%	17.2%	9.7%	6.1%	10.4%	2.9%
	Polymer-H	0.05	0.06	9.76	1.23	0.58	3.13	1.92	3.27
	↑ perf.	132.2%	217.5%	90.8%	87.8%	308.6%	90.7%	252.6%	179.2%
SSSP	Ligra	0.26	0.28	12.13	3.26	4.17	14.57	25.13	63.72
	Ligra-B	0.21	0.23	12.17	2.87	3.36	10.33	19.21	47.26
	↑ perf.	23.4%	21.6%	-0.3%	13.6%	24.1%	41.0%	30.7%	34.8%
	Ligra-P	0.22	0.21	12.18	2.07	2.67	9.02	16.83	45.19
	↑ perf.	18.6%	34.7%	-0.4%	57.3%	56.1%	61.5%	49.1%	41.2%
	Ligra-H	0.17	0.19	12.13	1.88	2.29	8.29	14.67	38.76
	↑ perf.	52.9%	47.37%	-1.2%	73.4%	82.1%	75.8%	71.3%	64.4%
	Gemini	0.35	0.37	8.82	4.05	4.86	16.16	29.83	59.62
	Gemini-B	0.28	0.28	8.86	3.78	3.51	11.23	21.94	42.73
	↑ perf.	25.3%	32.1%	-0.5%	7.2%	38.4%	43.9%	36.7%	39.5%
	Gemini-P	0.31	0.24	8.78	2.63	3.23	9.73	19.37	37.82
	↑ perf.	13.2%	54.2%	0.1%	54.0%	50.4%	66.1%	54.2%	57.6%
	Gemini-H	0.26	0.22	8.93	2.21	2.93	8.89	16.31	31.04
	↑ perf.	34.6%	68.2%	-1.2%	83.3%	65.9%	81.8%	82.9%	92.1%
	Polymer	0.43	0.45	7.63	4.32	3.74	9.48	18.74	39.84
	Polymer-B	0.37	0.37	7.74	3.72	3.23	7.04	14.33	31.75
	↑ perf.	16.2%	23.6%	-1.4%	16.1%	15.8%	34.7%	30.8%	25.5%
	Polymer-P	0.31	0.33	7.68	3.25	3.01	7.17	11.92	29.63
	↑ perf.	38.7%	36.4%	-0.6%	32.9%	24.3%	32.2%	57.2%	34.5%
	Polymer-H	0.29	0.29	7.81	2.92	2.87	6.85	9.34	26.12
	↑ perf.	48.3%	55.2%	-2.3%	47.9%	30.3%	38.4%	100.6%	52.5%
PR-D	Ligra	0.69	0.67	2.03	10.35	19.24	45.27	30.13	74.15
	Ligra-P	0.52	0.55	1.72	7.23	12.47	31.53	22.36	49.86
	↑ perf.	33.7%	22.3%	18.4%	43.1%	54.3%	43.6%	35.7%	48.7%
	Gemini	0.42	0.48	1.39	6.21	13.12	31.75	19.86	51.47
	Gemini-P	0.33	0.37	1.02	4.67	8.79	22.36	13.37	33.93
	↑ perf.	27.2%	26.7%	36.3%	33.0 %	49.3%	42.1%	48.5%	51.7%
	Polymer	0.47	0.43	1.76	8.33	15.33	29.74	21.48	52.87
	Polymer-P	0.34	0.36	1.27	5.37	11.38	23.57	14.37	37.43
	↑ perf.	38.2%	19.4%	38.6%	55.1%	34.7%	26.2%	49.5%	41.3%
MIS	Ligra	0.58	0.63	1.23	1.76	5.83	11.17	25.4	42.17
	Ligra-P	0.42	0.44	1.07	1.39	4.71	7.15	16.9	28.83
	↑ perf.	37.6%	44.1%	15.6%	26.6%	23.8%	56.2%	50.3%	46.3%
	Gemini	0.83	0.77	1.37	2.32	5.37	14.26	29.76	35.37
	Gemini-P	0.62	0.51	1.19	1.84	4.23	8.67	19.43	23.53
	↑ perf.	33.7%	51.2%	15.1%	26.1%	26.9%	64.5%	53.1%	50.2%
	Polymer	0.47	0.73	0.87	1.53	7.54	17.87	23.16	38.62
	Polymer-P	0.36	0.49	0.73	1.21	5.87	12.34	15.77	28.93
	↑ perf.	30.1%	48.6%	19.2%	26.5%	28.4%	44.8%	46.9%	34.5%
GC	Ligra	3.72	4.62	1.17	8.52	47.38	79.32	64.73	113.26
	Ligra-P	2.61	3.42	1.04	6.57	34.56	55.08	47.26	83.28
	↑ perf.	42.5%	35.1%	12.6%	29.7%	37.1%	44.3%	37.2%	36.1%
	Gemini	6.78	6.43	1.78	6.51	43.55	82.78	74.47	125.88
	Gemini-P	4.33	4.93	1.44	4.55	33.49	52.63	53.46	87.37
	↑ perf.	57.6%	30.4%	23.6%	43.1%	30.1%	57.2%	39.3%	44.1%
	Polymer	4.23	5.37	0.79	9.72	43.62	69.42	58.35	87.63
	Polymer-P	3.17	3.87	0.72	7.14	31.52	43.52	42.12	61.94
	↑ perf.	33.4%	38.8%	9.8%	36.1%	38.4%	59.5%	38.5%	41.5%
BFS	Ligra	0.07	0.09	1.53	1.58	1.13	2.61	4.52	14.32
	Ligra-O	0.04	0.03	1.54	0.81	0.32	0.76	1.08	3.35
	↑ perf.	78.2%	188.6%	-0.9%	95.0%	252.3%	243.7%	319.6%	327.3%
	Ligra-B	0.04	0.04	1.54	0.78	0.33	0.74	1.05	3.42
	↑ perf.	73.7%	152.8%	-0.7%	102.1%	242.1%	234.7%	314.7%	318.3%
	Ligra-P	0.07	0.09	1.56	1.61	1.11	2.61	4.58	13.87
	↑ perf.	3.5%	-2.3%	-1.9%	-1.8%	1.8%	2.9%	-1.3%	3.2%
	Ligra-H	0.04	0.04	1.59	0.73	0.31	0.69	0.97	3.13
	↑ perf.	71.3%	168.3%	-3.2%	116.4%	232.4%	230.4%	313.2%	330.3%
	Gemini	0.15	0.09	3.17	2.17	1.32	3.57	6.32	21.38
	Gemini-O	0.06	0.05	3.24	1.29	0.54	0.89	1.56	4.73
	↑ perf.	139.1%	72.4%	-2.2%	68.2%	144.4%	301.2%	305.1%	352.1%
	Gemini-B	0.06	0.05	3.14	1.33	0.57	0.83	1.52	4.88
	↑ perf.	128.7%	76.2%	0.9%	63.2%	131.6%	330.1%	315.7%	338.1%
	Gemini-P	0.15	0.09	3.23	2.15	1.37	3.63	6.16	20.93
	↑ perf.	3.1%	-2.7%	-1.9%	0.9%	-3.7%	-1.6%	2.6%	2.2%
	Gemini-H	0.06	0.05	3.25	1.34	0.58	0.85	1.55	4.77
	↑ perf.	134.3%	79.3%	-2.5%	61.9%	127.6%	320.7%	307.7%	348.2%
	Polymer	0.18	0.14	1.25	1.25	1.52	4.21	3.17	8.93
	Polymer-O	0.05	0.05	1.32	0.64	0.47	1.32	0.87	2.56
	↑ perf.	260.3%	183.2%	-5.3%	95.3%	223.4%	218.9%	264.4%	248.8%
	Polymer-B	0.05	0.05	1.29	0.66	0.49	1.38	0.85	2.49
	↑ perf.	254.3%	182.7%	-3.1%	89.3%	210.2%	205.1%	272.9%	258.6%
	Polymer-P	0.17	0.15	1.23	1.21	1.57	4.26	3.01	8.53
	↑ perf.	5.9%	-3.6%	1.6%	3.3%	-3.2%	-1.2%	5.3%	4.7%
	Polymer-H	0.05	0.05	1.29	0.61	0.46	1.36	0.88	2.51
	↑ perf.	258.4%	183.2%	-3.2%	104.9%	230.4%	209.6%	260.2%	255.8%

distributed setting on a 4-node cluster. Each node is connected via a 1 Gb Ethernet. All graph algorithms are running with 28 threads on each machine.

6.2 Overall Performance

We evaluate the overall performance for three state-of-the-art graph systems with or without **B**oundary-cut (**-B**), **P**redictive (**-P**) judgment, and **H**ybrid approach **-H**, respectively. For BFS, we consider our approaches against a standard implementation and break-early-enabled implementation (i.e., **-O**). Table 5 shows the results.

Boundary-Cut Judgment. (*abbr*. **-B**) For CC, Ligra with boundary-cut judgment (i.e., Ligra-B) can provide 475.3 percent performance improvement at most over Ligra. Gemini-B is 337.6 percent faster than Gemini, and Polymer-B is 282.3 percent faster than Polymer. On average, boundary-cut judgment can provide BFS, CC, and SSSP for Ligra by 179.8, 163.8, and 23.6 percent performance improvement, Gemini by 173.1, 127.0, and 27.8 percent, and Ploymer by 183.8, 136.9, and 20.2 percent, respectively. Note that SSSP has relatively lower benefits than BFS and CC because more invalid updates are involved in BFS and CC over SSSP (as has been witnessed in Fig. 1). On average, Boundary-cut judgment can provide 115.1 percent performance improvement on all datasets. Results also show that sparse graphs such as webbase and road graphs (with small average degree) have relatively-small benefits from boundary-cut on BFS and CC due to small number of invalid updates. This fact is also witnessed in Section 6.7. Note that boundary-cut judgment can not be integrated with PageRank-Delta, MIS, and GC because they do not meet the monotonically-changing property.

Predictive Judgment. (*abbr*. **-P**) In comparison to the boundary-cut approach, predictive approach can handle more graph algorithms such as PageRank-Delta, MIS, and GC. Overall, Ligra-P outperforms Ligra for SSSP, PR-D, MIS, and GC by 39.8, 37.5, 37.6, and 36.2 percent on overage, respectively. Gemini-P has 43.7, 39.6, 40.1, and 40.7 percent performance improvement on average over Gemini. Polymer-P is 32.0, 37.9, 34.9, and 37.0 percent improvement over Polymer, respectively. On average, predictive judgment can provide 38.1 percent improvement on all datasets. Note that the benefits of BFS and CC using predictive judgment can be limited, since BFS and CC are often converged in a very few rounds of iterations without sufficient history for prediction.

Hybrid Approach. Results show that boundary-cut and predictive approaches have their respective benefits. No one can have an absolute win. We also find that boundary-cut judgment prefers fast-converged application such as CC while predictive approach prefers relative long-iteration such as SSSP for capturing sufficient history information. Hybrid approach is more efficient than both boundary-cut judgment and predictive judgment for most cases.

For CC and SSSP, it can be seen that the hybrid approach is more efficient than both boundary-cut and predictive approaches. To be specific, for CC, hybrid approach outperforms boundary-cut and predictive approaches by up to 80.5 and 502.6 percent performance improvement. On average, hybrid judgment can provide 136.6 percent performance improvement compared with the baseline version. For BFS, hybrid method has the almost same result with boundary-cut method. This is because that BFS has only two pull-based iterations with no need of predictive method.

Note that our boundary-cut approach and hybrid approach can provide the almost same benefit that break-

Authorized licensed use limited to: Huazhong University of Science and Technology. Downloaded on March 14,2023 at 12:58:49 UTC from IEEE Xplore. Restrictions apply.

TABLE 6
Execution Time for Hybrid Approach versus Specialized Optimizations

Algorithms	CC (seconds)			SSSP (seconds)		
	plain	disjoint-set	Hybrid	plain	△-stepping	Hybrid
soc	0.08	0.06	0.04	0.26	0.29	0.17
enwiki	0.12	0.13	0.04	0.28	0.27	0.19
road	20.91	0.57	11.6	12.13	0.60	12.13
webbase	2.85	1.73	1.21	3.26	2.54	1.88
twitter	2.53	1.60	0.41	4.17	3.12	2.29
friendster	5.22	3.61	2.88	14.57	13.57	8.29
rmat-27	7.99	5.15	2.55	25.13	22.38	14.67
rmat-28	11.89	8.32	3.75	63.72	54.62	38.76

early-enabled optimization (-O) can offer for BFS. As we discussed in Section 3.2, the traditional break-early functionality (useful for BFS only) can be understood as a special version of the boundary-cut judgment approach. Note that the vertices in BFS are traversed only once, which tends to cause a very few dense iterations (which are often no more than two). Therefore, we can see that the predictive judgment is of no use (since $M_g = 2$), further leading that hybrid approach is degraded into a boundary-cut method.

6.3 Compared with Specialized Optimizations

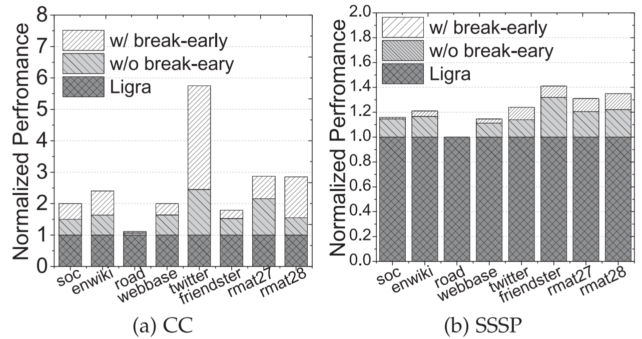
Compared to the plain design, many graph algorithms have also specialized optimizations for performance enhancement. For example, disjoint-set CC [14] presents efficient data structure called disjoint-set to efficiently merge vertices in the same components within only one iteration. △-stepping SSSP [32] prioritizes to schedule the activated vertices that hold shorter path for fast convergence. We also compare our design to these specialized designs.

Table 6 depicts the results. It can be seen that our hybrid judgment can provide better results over the specialized designs in most cases. To be specific, compared to the standard version of CC, disjoint-set CC is faster by -7.3%∼64.7% performance improvement while our hybrid solution can provide 81.2%∼517.3% performance improvement. In contrast to the plain SSSP, △-stepping SSSP have the performance benefits by -10.3%∼33.8% while our hybrid approach can have 47.4%∼82.1% performance improvement.

Note that *road* with the mesh topology has an exception, for which both CC and SSSP involve over hundreds of iterations with considerable synchronization overhead that the specialized optimizations can handle [15].

6.4 Benefit Details

We investigate the benefit details of boundary-cut and predictive approaches by investigating Ligra.



(a) CC (b) SSSP

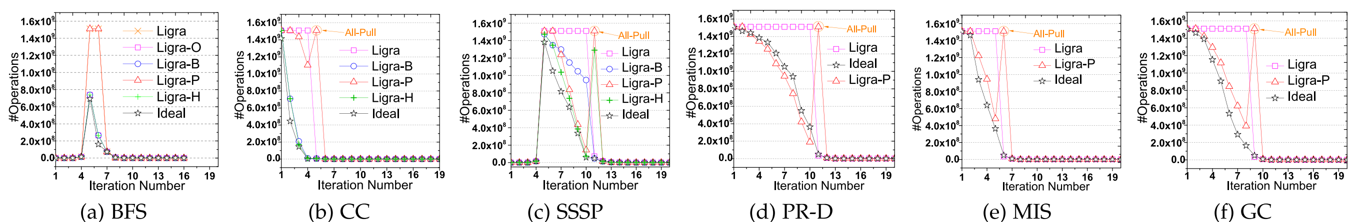
Fig. 7. Performance benefits from enabled break early optimization for CC and SSSP. Results are normalized to Ligra with its plain version.

Boundary-Cut Judgment. Fig. 6 shows the number of operations on the active vertices and their associated edges over iteration. The ideal number is obtained by an offline trace analysis. Compared to the plain Ligra, Ligra-B can significantly reduce the total number of operations by 57.3 percent (for BFS), 55.8 percent (for CC), and 21.9 percent (for SSSP). In particular, for CC, Ligra-B can offer the almost same results as the ideal case. SSSP has the deviation in a few iterations in contrast to ideal case for Ligra-B, because many converged vertices with longer distance than the threshold are ignored.

Fig. 7 breaks down the benefits from enabled break-early for other graph algorithms. We can see that enabled break-early of boundary-cut approach can further improve the performance of CC and SSSP by 2.34X and 1.13X at most, respectively. Few performance benefits are obtained for *road* since it involves no pull iteration for optimizations in SSSP.

Predictive Judgment. Fig. 6 depicts the number of operations for Ligra-P, which has significant difference from idea case for CC since it involves a very few (no more four) iterations for the history collection. In this case, vertices are hard to turn into the `sleep` status. In contrast, SSSP, PageRank-Delta, MIS, and GC with relatively-long iteration have better effects after their 3rd iteration with 36.3, 38.4, 27.7, and 31.5 percent operation reduction, respectively.

Fig. 8 shows that the mis-prediction occurs at a very low frequency by 1.2, 3.8, 0.9, 2.7, and 3.3 percent for CC, SSSP, PR-D, MIS, and GC, respectively. All-pull iteration does three-category jobs: 1) valid updates on active vertices that should be originally processed in the same iteration; 2) valid updates on the previously mis-predicted vertices; and 3) invalid updates. The first two jobs are normal working flows while only the last is the extra wasted work. Fig. 8 further shows that the runtime overhead of this wasted work is reasonably small, with 8.7, 5.2, 3.2, 7.3, and 4.7 percent for CC, SSSP, PR-D, MIS, and GC. In particular, BFS involves only two pull iterations, and hence, it has no mis-prediction rate and mis-prediction overhead.



(a) BFS (b) CC (c) SSSP (d) PR-D (e) MIS (f) GC

Fig. 6. Total number of operations on vertices and its associated in-coming edges over iteration for our approaches against Ligra and ideal situation.

Authorized licensed use limited to: Huazhong University of Science and Technology. Downloaded on March 14,2023 at 12:58:49 UTC from IEEE Xplore. Restrictions apply.

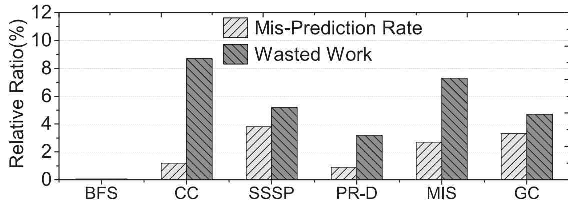


Fig. 8. Mis-prediction ratio (among all vertex convergence checkings) and runtime overhead ratio (relative to the overall execution time) in the all-pull iteration on twitter graph.

Hybrid Judgment. Fig. 6 also shows the number of operations for Ligra-H, which can reduce more invalid operations against Ligra-B and Ligra-P with 57.4 percent, 41.7 percent operation reduction for CC and SSSP, respectively. Hybrid solution can use alternative (boundary-cut) approach to filter many vertices in its first few iterations without processing all vertices before it formally enters into sparse stage. This is also the very reason why hybrid solution has the best fitting results against ideal case for both CC and SSSP.

6.5 Runtime Overhead

We also investigate the runtime overhead of our approaches. Table 7 illustrates the execution time results for CC and SSSP. Overall, both Ligra-B, Ligra-P and Ligra-H has incurred a negligible part of runtime overhead against execution time. To be specific, Ligra-B, Ligra-P, Ligra-H take 6.5, 4.7, and 9.2 percent overhead at most respectively. On average, they takes only 2.1, 2.5, and 3.9 percent overhead. In particular, for the large graph such as *twitter-2010*, SSSP has the total execution time by 4.17 seconds while the extra overhead for vertex convergence judgment only take 0.073 seconds, which can be considered negligible in practice.

Reasons are easy to understand. The extra overhead for boundary-cut judgment lies in reducing the minimal label in the *Frontier* vertices, while that for predictive judgment is to scan the vertices and shift the states of FSM. Operations responsible for these part of overhead has: 1) not only few numbers compared to all operations in each iteration, and 2) but also sequential memory accesses compared to the random accesses of original graph iterations.

6.6 Distributed Deployment

We also characterizee the performance of our approaches in a distributed four-node cluster. In the distributed settings, Gemini adopts a signal-slot abstraction to extend the push-pull mode. However, both signal and slot operations may be unnecessary for the invalid updates, which lead to both useless computation and prohibitive network communication. Interestingly, our approaches are efficient in improving the distributed communication overhead significantly by using the local vertex information only without remotely accessing the neighbors information. Thus, as shown in Fig. 9, we can see that up to 126.3 percent (CC on twitter), 53 percent (SSSP on rmat28), and 132.4 percent (CC on twitter) speedup can be achieved for Gemini-B, Gemini-P, and Gemini-H, respectively, with 51.2, 24.7, and 72.4 percent benefit on average.

TABLE 7
Overhead of Our Approaches

	CC (seconds)				SSSP (seconds)			
	Ligra	OHD-B	OHD-P	OHD-H	Ligra	OHD-B	OHD-P	OHD-H
soc	0.08	0.002	0.002	0.004	0.26	0.003	0.004	0.006
enwiki	0.12	0.002	0.003	0.004	0.28	0.002	0.005	0.006
road	20.91	1.363	0.841	1.942	12.13	-	-	-
webbase	2.85	0.027	0.043	0.053	3.26	0.061	0.068	0.091
twitter	2.53	0.014	0.037	0.047	4.17	0.043	0.044	0.073
friendster	5.22	0.036	0.067	0.092	14.57	0.113	0.146	0.217
rmat-27	7.99	0.073	0.083	0.132	25.13	0.163	0.217	0.325
rmat-28	11.89	0.113	0.142	0.217	63.72	0.314	0.453	0.637

OHD Represents the Extra Overhead of Convergence Judgment. "-" Indicates That SSSP Does Not Trap Into Pull Model With No Invalid Operation Overhead Involved Thereby.

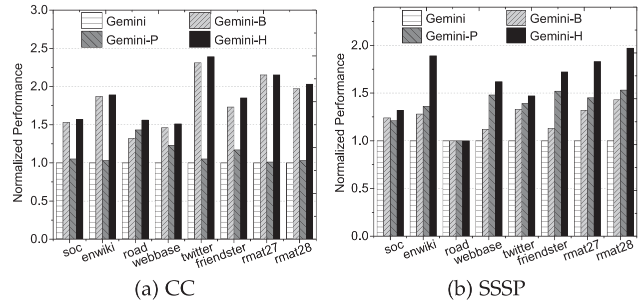


Fig. 9. Performance normalization on a 4-node cluster.

6.7 Sensitivity Study

We also characterize the performance variation of our approaches with different degrees and weights.

Average Degrees. Fig. 10 illustrates the comparative results. Overall, Ligra-H has the best effect. To be specific, for CC, Ligra-B and Ligra-H shows better efficiency against Ligra and Ligra-P as the average edges are changing. This is because more edges and vertices for CC enable that many edges can be skipped. For SSSP, all of them have the similar sensitivity. They have the increasing execution time nearly linear to graph size since the computation complexity is linearly related to the total number of edges.

Edge Weights. Table 8 depicts the results for SSSP. Overall, Ligra-B is efficient in handling the graphs with small edge weights. For example, it can achieve 448 percent performance improvement for maximum edge weight with 1. The underlying cause is that smaller edge weight means that vertices are easier to get converged and further skipped. Conversely, Ligra-P is efficient in handling the graphs with large edge weights. The reason is that vertices with larger edge weight need more iterations for the potential vertex convergence. Ligra-H outperforms both in most cases, but the benefits also

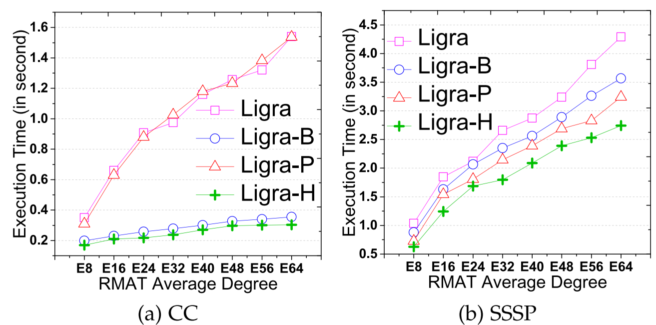


Fig. 10. Performance characterization of our approaches on RMAT24 with varying average degrees from 8 to 64.

Authorized licensed use limited to: Huazhong University of Science and Technology. Downloaded on March 14,2023 at 12:58:49 UTC from IEEE Xplore. Restrictions apply.

TABLE 8
Execution Time (in Seconds) of Our Filtration Approaches on *Twitter-2010* with the Maximum Weight Ranging from 1 to 127

	Ligra	Ligra-B	↑ perf.	Ligra-P	↑ perf.	Ligra-H	↑ perf.
twitter-1	2.14	0.39	448%	2.17	-1%	0.42	410%
twitter-3	2.72	0.59	361%	2.57	6%	0.56	386%
twitter-7	2.94	0.81	262%	2.72	9%	0.68	332%
twitter-15	3.65	1.80	103%	3.11	17%	1.21	201%
twitter-31	3.82	2.57	49%	2.89	32%	1.67	129%
twitter-63	4.17	3.36	24%	2.67	56%	2.29	82%
twitter-127	4.95	4.13	20%	3.09	63%	2.87	72%

degrade as the maximum weight is increasing because less edges can be filtered.

6.8 Discussion on Generalizability

Our predictive and boundary-cut methods have the generalizability in the real world. 1) *Predictive judgment*: As discussed, predictive judgment can be understood as a special form of asynchronous execution, which has been widely-used to drive the graph processing correctly. Thus, predictive judgment can extensively handle a wide variety of real-world graph algorithms that can be scheduled asynchronously. 2) *Boundary-cut judgment*: Although we only show the monotonicity of BFS, CC, and SSSP in this paper, all of them are the common graph applications that can be integrated as a core component for many complex graph applications. For example, as a basic traversal procedure, BFS can be used to represent many other graph traversal algorithms, such as Betweenness Centrality and Closeness Centrality [12]. Also, CC is a subroutine for many graph mining algorithms such as graph clustering [30]. Further, SSSP is widely used for the path navigation and social network analysis. We believe that these derivative algorithms can cover a broad category of important graph applications.

7 Related Work

Graph Processing with Hybrid Push-Pull Model. Hybrid model is often efficient to handle natural graphs [12], [33]. Ligra [9] presents a hybrid push-pull representation to improve the performance of general graph algorithms. Polymer [10] extends push-pull model with NUMA-aware optimization. Gemini [23] deploys the push-pull model in the distributed environment by using a signal-slot abstraction. These earlier studies deploy push-pull model in different situations, but their efficiency still suffers due to invalid operations, which are exactly addressed in this work.

Pull-Extended Optimizations. Pure pull conservatively scatters all the information from its neighboring vertices, leading to potentially unnecessary computations [9], [10], [23], [26]. Garaph [11] presents a notify-pull model to label all vertices connected to the frontier vertices. Only those labeled (rather than all) vertices can be scheduled. Notify-pull is efficient for the case where the scale of active vertices is small. Also, vertices with notify-pull is still needed to be scheduled even if it has been converged. Grazelle [34] parallelizes the edge scheduling of each vertex via a scheduler-aware interface to reduce the global synchronization overhead. Our work can be cooperatively-combined with these progressive efforts to make pull efficient in a new level.

Redundant Computation Reduction. There are also many attempts to reduce the redundant computation [35], [36]. Beamer et al. [8] propose an improved BFS with break-early optimization. A few studies prioritize to schedule certain vertices or edges to reduce computations, such as delta-stepping SSSP [32], [37]. Much effort has also been put into scheduling graph partitions based on graph components [35] or graph abstraction [38] to improve the efficiency of information propagation. Compared to these algorithm-by-algorithm specialized algorithms, we can not only reduce the redundant updates but also preserve the general applicability for more graph algorithms.

8 Conclusion

Existing graph-parallel processing systems essentially rely on pull computation model to expose high parallelism. This paper focuses on addressing the invalid updates of pull model for further performance enhancement. We have the insight that the invalid updates can be easily identified by additionally using a fraction of critical information. We therefore present two novel filtration approaches, boundary-cut and predictive judgments to exploit these critical information. We also combine these two judgments together to further reduce invalid updates. Our experimental results on a wide variety of graph algorithms show that boundary-cut, predictive, and hybrid judgment can improve the performance by 115.1, 38.1, and 136.6 percent on average.

Acknowledgments

We would like to thank the anonymous reviewers for their insightful comments and valuable feedback. This work is supported by the National Key Research and Development Program of China under Grant No. 2018YFB1003502, National Natural Science Foundation of China under Grant No.61825202, 61832006, and 61702201.

References

[1] Z. Li and Z. Ding, "Distributed optimization on unbalanced graphs via continuous-time methods," *Sci. China Inf. Sci.*, vol. 61, no. 12, pp. 129 204:1–129 204:3, 2018.
[2] 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. 36th ACM Int. Conf. Manage. Data*, 2010, pp. 135–146.
[3] J. Leskovec, K. J. Lang, A. Dasgupta, and M. W. Mahoney, "Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters," *Internet Math.*, vol. 6, no. 1, pp. 29–123, 2009.
[4] G. Dai, T. Huang, Y. Chi, N. Xu, Y. Wang, and H. Yang, "ForeGraph: Exploring large-scale graph processing on multi-FPGA architecture," in *Proc. 17th ACM/SIGDA Int. Symp. Field-Programmable Gate Arrays*, 2017, pp. 217–226.
[5] H. Jin, P. Yao, X. Liao, L. Zheng, and X. Li, "Towards dataflow-based graph accelerator," in *Proc. 37th Int. Conf. Distrib. Comput. Syst.*, 2017, pp. 1981–1992.
[6] H. Jin, P. Yao, and X. Liao, "Towards dataflow based graph processing," *Sci. China Inf. Sci.*, vol. 60, no. 12, pp. 126 102:1–126 102:3, 2017.
[7] P. Yao, L. Zheng, X. Liao, H. Jin, and B. He, "An efficient graph accelerator with parallel data conflict management," in *Proc. 27th Int. Conf. Parallel Archit. Compilation*, 2018, pp. 8:1–8:12.
[8] S. Beamer, K. Asanovic, and D. Patterson, "Direction-optimizing breadth-first search," *Sci. Program.*, vol. 21, no. 3–4, pp. 137–148, 2013.
[9] J. Shun and G. E. Blelloch, "Ligra: A lightweight graph processing framework for shared memory," in *Proc. 18th ACM SIGPLAN Symp. Principles Practice Parallel Program.*, 2013, pp. 135–146.

Authorized licensed use limited to: Huazhong University of Science and Technology. Downloaded on March 14,2023 at 12:58:49 UTC from IEEE Xplore. Restrictions apply.

[10] K. Zhang, R. Chen, and H. Chen, "NUMA-aware graph-structured analytics," in *Proc. 20th ACM SIGPLAN Symp. Principles Practice Parallel Program.*, 2015, pp. 183–193.

[11] L. Ma, Z. Yang, H. Chen, J. Xue, and Y. Dai, "Garaph: Efficient GPU-accelerated graph processing on a single machine with balanced replication," in *Proc. USENIX Annu. Tech. Conf.*, 2017, pp. 195–207.

[12] H. Liu and H. H. Huang, "Enterprise: Breadth-first graph traversal on GPUs," in *Proc. 27th Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2015, pp. 1–12.

[13] V. T. Chakaravarthy, F. Checconi, P. Murali, F. Petrini, and Y. Sabharwal, "Scalable single source shortest path algorithms for massively parallel systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, no. 7, pp. 2031–2045, Jul. 2017.

[14] P. Erdds and A. Renyi, "On random graphs I," *Publicationes Mathematicae Debrecen*, vol. 6, pp. 290–297, 1959.

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

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

[17] J. C. Chen, "Dijkstra's shortest path algorithm," *J. Formalized Math.*, vol. 15, no. 9, pp. 237–247, 2003.

[18] G. Wang, W. Xie, A. J. Demers, and J. Gehrke, "Asynchronous large-scale graph processing made easy," in *Proc. 6th Biennial Conf. Innovative Data Syst. Res.*, 2013, pp. 3–6.

[19] R. R. McCune, T. Weninger, and G. Madey, "Thinking like a vertex: A survey of vertex-centric frameworks for large-scale distributed graph processing," *ACM Comput. Surveys*, vol. 48, no. 2, pp. 1–25, 2015.

[20] S. Dolev, Y. Elovici, and R. Puzis, "Routing betweenness centrality," *J. ACM*, vol. 57, no. 4, pp. 1–27, 2010.

[21] T. Opsahl, F. Agneessens, and J. Skvoretz, "Node centrality in weighted networks: Generalizing degree and shortest paths," *Social Netw.*, vol. 32, no. 3, pp. 245–251, 2010.

[22] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein, "Distributed GraphLab: A framework for machine learning and data mining in the cloud," *Proc. VLDB Endowment*, vol. 5, no. 8, pp. 716–727, 2012.

[23] X. Zhu, W. Chen, W. Zheng, and X. Ma, "Gemini: A computation-centric distributed graph processing system," in *Proc. 12th USENIX Symp. Operating Syst. Des. Implementation*, 2016, pp. 301–316.

[24] L. Zheng, X. Liao, and H. Jin, "Efficient and scalable graph parallel processing with symbolic execution," *ACM Trans. Archit. Code Optimization*, vol. 15, no. 1, pp. 188–199, 2018.

[25] X. Zhu, W. Han, and W. Chen, "GridGraph: Large-scale graph processing on a single machine using 2-level hierarchical partitioning," in *Proc. USENIX Annu. Tech. Conf.*, 2015, pp. 375–386.

[26] J. Malicevic, B. Lepers, and W. Zwaenepoel, "Everything you always wanted to know about multicore graph processing but were afraid to ask," in *Proc. USENIX Annu. Tech. Conf.*, 2017, pp. 631–643.

[27] A. Broder, R. Kumar, F. Maghoul, P. Raghavan, S. Rajagopalan, R. Stata, A. Tomkins, and J. Wiener, "Graph structure in the web," *Comput. Netw.*, vol. 33, no. 1–6, pp. 309–320, 2000.

[28] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*. Amsterdam, The Netherlands: Elsevier, 2011.

[29] C. Xie, R. Chen, H. Guan, B. Zang, and H. Chen, "SYNC or ASYNC: Time to fuse for distributed graph-parallel computation," in *Proc. 20th ACM SIGPLAN Symp. Principles Practice Parallel Program.*, 2015, pp. 194–204.

[30] Z. Ai, M. Zhang, and Y. Wu, "Squeezing out all the value of loaded data: An out-of-core graph processing system with reduced disk I/O," in *Proc. USENIX Annu. Tech. Conf.*, 2017, pp. 125–137.

[31] D. Chakrabarti, Y. Zhan, and C. Faloutsos, "R-MAT: A recursive model for graph mining," in *Proc. 4th SIAM Int. Conf. Data Mining*, 2004, pp. 442–446.

[32] U. Meyer and P. Sanders, "$\triangle$-stepping: A parallelizable shortest path algorithm," *J. Algorithms*, vol. 49, no. 1, pp. 114–152, 2003.

[33] M. Besta, M. Podstawski, L. Groner, E. Solomonik, and T. Hoefler, "To push or to pull: On reducing communication and synchronization in graph computations," in *Proc. 26th Int. Symp. High-Perform. Parallel Distrib. Comput.*, 2017, pp. 93–104.

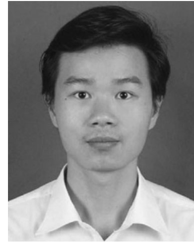
[34] S. Grossman, H. Litz, and C. Kozyrakis, "Making pull-based graph processing performant," in *Proc. 23rd ACM SIGPLAN Symp. Principles Practice Parallel Program.*, 2018, pp. 246–260.

[35] P. Yuan, W. Zhang, C. Xie, H. Jin, L. Liu, and K. Lee, "Fast iterative graph computation: A path centric approach," in *Proc. 26th Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2014, pp. 401–412.

[36] A. Kusum, K. Vora, R. Gupta, and I. Neamtiu, "Efficient processing of large graphs via input reduction," in *Proc. 25th ACM Int. Symp. High-Perform. Parallel Distrib. Comput.*, 2016, pp. 245–257.

[37] J. Zhang and J. Li, "Degree-aware hybrid graph traversal on FPGA-HMC platform," in *Proc. 18th ACM/SIGDA Int. Symp. Field-Programmable Gate Arrays*, 2018, pp. 229–238.

[38] M. Zhang, Y. Wu, Y. Zhuo, X. Qian, C. Huan, and K. Chen, "Wonderland: A novel abstraction-based out-of-core graph processing system," in *Proc. 23rd Int. Conf. Archit. Support Program. Languages Operating Syst.*, 2018, pp. 608–621.

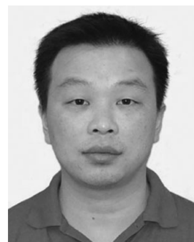


Long Zheng received the PhD degree from the Huazhong University of Science and Technology (HUST), in 2016. He is now a postdoc with the School of Computer Science and Technology, Huazhong University of Science and Technology, in China. His current research interests include program analysis, runtime systems, and configurable computer architecture with a particular focus on graph processing. He is a member of the IEEE.

Xianliang Li is currently working toward the master's degree in the School of Computer Science and Technology, Huazhong University of Science and Technology (HUST), China. His current research interests focus on data mining and graph processing.

Xi Ge is currently working toward the master's degree in the School of Computer Science and Technology, Huazhong University of Science and Technology (HUST), China. His current research interests focus on deep learning and graph processing.

Xiaofei Liao received the PhD degree in computer science and engineering from the Huazhong University of Science and Technology (HUST), China, in 2005. He is now the vice dean with the School of Computer Science and Technology, HUST. He has served as a reviewer for many conferences and journal papers. His research interests include the areas of system software, P2P system, cluster computing, and streaming services. He is a member of the IEEE.



Zhiyuan Shao received the PhD degree in computer science and engineering from the Huazhong University of Science and Technology (HUST), China, in 2005. He is now an associate professor with the School of Computer Science and Engineering, HUST. He has served as a reviewer for many conferences and journal papers. His research interests include the areas of operating systems, virtualization technology for computing system, and big-data processing. He is a member of the IEEE.

Authorized licensed use limited to: Huazhong University of Science and Technology. Downloaded on March 14,2023 at 12:58:49 UTC from IEEE Xplore. Restrictions apply.

Hai Jin received the PhD degree in computer engineering from the Huazhong University of Science and Technology (HUST), in 1994. He is a Cheung Kung Scholars chair professor of computer science and engineering at Huazhong University of Science and Technology (HUST), China. In 1996, he was awarded a German Academic Exchange Service fellowship to visit the Technical University of Chemnitz in Germany. He worked with the University of Hong Kong between 1998 and 2000, and as a visiting scholar with the University of Southern California between 1999 and 2000. He was awarded Excellent Youth Award from the National Science Foundation of China in 2001. He is the chief scientist of ChinaGrid, the largest grid computing project in China, and the chief scientists of National 973 Basic Research Program Project of Virtualization Technology of Computing System, and Cloud Security. He has co-authored 15 books and published more than 600 research papers. His research interests include computer architecture, virtualization technology, cluster computing and cloud computing, peer-to-peer computing, network storage, and network security. He is a fellow of the IEEE and a member of the ACM.

Qiang-Sheng Hua received the BEng and MEng degrees from the School of Information Science and Engineering, Central South University, China, in 2001 and 2004, respectively, and the PhD degree from the Department of Computer Science, University of Hong Kong, China, in 2009. He is currently an associate professor with the Huazhong University of Science and Technology, China. He is interested in parallel and distributed computing, including algorithms, and implementations in real systems. He is a member of the IEEE.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/csdl.

Authorized licensed use limited to: Huazhong University of Science and Technology. Downloaded on March 14,2023 at 12:58:49 UTC from IEEE Xplore. Restrictions apply.