FBSGraph: Accelerating Asynchronous Graph Processing via Forward and Backward Sweeping

Yu Zhang, Xiaofei Liao, *Member, IEEE*, Hai Jin, *Senior Member, IEEE*, Lin Gu, *Member, IEEE*, and Bing Bing Zhou

Abstract—Graph algorithm is pervasive in many applications ranging from targeted advertising to natural language processing. Recently, *Asynchronous Graph Processing* (AGP) is becoming a promising model to support graph algorithm on large-scale distributed computing platforms because it enables faster convergence speed and lower synchronization cost than the synchronous model for no barrier between iterations. However, existing AGP methods still suffer from poor performance for inefficient vertex state propagation. In this paper, we propose an effective and low-cost forward and backward sweeping execution method to accelerate state propagation for AGP, based on a key observation that states in AGP can be propagated between vertices much faster when the vertices are processed sequentially along the graph path within each round. Through dividing graph into paths and asynchronously processing vertices on each path in an alternative forward and backward way according to their order on this path, vertex states in our approach can be quickly propagated to other vertices and converge in a faster way with only little additional overhead. In order to efficiently support it over distributed platforms, we also propose a scheme to reduce the communication overhead along with a static priority ordering scheme to further improve the convergence speed. Experimental results on a cluster with 1,024 cores show that our approach achieves excellent scalability for large-scale graph algorithms and the overall execution time is reduced by at least 39.8 percent, in comparison with the most cutting-edge methods.

Index Terms—Graph algorithm, synchronous/asynchronous, convergence, cascade effect

1 Introduction

The advances in machine learning and data mining have led to a flurry of graph analytic techniques that typically require an iterative refinement process on large-scale graphs. Typical examples are SimRank [1], connected components [2], PageRank [3], adsorption [4], and expected hitting time [5]. These graph algorithms usually require numerous iterations to process a massive volume of data round by round until convergence. Under such circumstance, ensuring the quality-of-service (QoS, e.g., convergence speed) for these algorithms is regarded as a challenging task. It has been widely proved that general-purpose parallel frameworks [6] are not efficient to support this type of computation [7], [8]. Thus, several graph processing systems [9], [10] have been developed to encode graph algorithm as vertex/edge programs that can run on distributed computing platforms.

These graph processing systems fall into two categories, i.e., the ones using synchronous graph processing model (e.g., Pregel [11], GraphX [12], Blogel [13], Gemini [14]) or asynchronous graph processing model (e.g., GraphLab [15], PowerGraph [16], Maiter [17], [18], GiraphUC [19]). Although the synchronous processing model is easy to be implemented and reasoned about, recent work [20] shows that it suffers from poor performance over distributed platforms due to frequent global synchronization between iterations [21]. Different from the synchronous model, *Asynchronous Graph Processing* (AGP) does not use barrier between iterations and asynchronously updates each vertex's state according to the most recent states of its neighbours until the whole process converges. Thus, it is widely-used and recognized effective method on large-scale distributed platforms because it provides low synchronization cost, balanced load and so on.

However, the current AGP solutions [15], [16], [17], [18], [19], [20] all suffer from poor performance for inefficient vertex state propagation from two aspects. First, existing graph partitioning methods [22] are not suitable to AGP because they may divide vertices of a graph path into many partitions and also store them in a partition in an arbitrary order, although they have been extensively studied for balanced load and low communication cost. Then, in AGP, the vertices on a graph path may not be handled along their order on this path within each round. The whole graph has to be handled for many rounds to propagate each vertex's state to the others along graph path, because the state of an already-processed neighbor can only be updated in the next round. Second, with the current AGP execution modes, i.e., round-robin mode [15], [16], [17], [18] and prioritized mode [15], [16], [17], [18], vertex states either face slow

- *Y. Zhang, X. Liao, H. Jin, and L. Gu are with the Service 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: zhang_yu9068@163.com, {hjin, xfliao}@hust.edu.cn, anheeno@gmail.com.*
- *B.B. Zhou is with the School of Information Technologies, The University of Sydney, NSW 2006, Australia. E-mail: bbz@it.usyd.edu.au.*

Manuscript received 14 Nov. 2016; revised 10 Oct. 2017; accepted 24 Nov. 2017. Date of publication 8 Dec. 2017; date of current version 30 Mar. 2018. (Corresponding author: Xiaofei Liao.)
Recommended for acceptance by A. Singh.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TKDE.2017.2781241

backward propagation along graph path because each vertex needs a new round to affect the neighbors processed before it, or need highly additional overhead to be propagated. As a result, such inefficient state propagation of nonconvergent vertices (caused by the above two reasons) induces a long time to reprocess the whole graph for several rounds so as to update other vertex states.

We discover that, in AGP, if the vertices are handled sequentially along their order on a path within one graph processing round, the vertex state is able to work on the vertices to be processed after it on this path within the round. We name this feature as *cascade effect*. It is because that the most recent state of each vertex in AGP is allowed to be immediately used by its neighbors to update their states, while the state of an already-processed neighbor can only be updated in the next round. Such a cascade effect, in practice, enables us to quickly propagate vertex state along graph path with low overhead, motivating us to propose a more efficient AGP method.

Based on the observation, we propose a *Forward and Backward Sweeping* (FBS) execution method for AGP so as to accelerate its convergence speed via fully exploiting cascade effect. It first divides the graph as a set of paths without intersecting edges and ensures that the vertices of each path are indexed in a partition along their order on this path, aiming to help state propagation along graph path. After that, it takes the path as the basic unit for processing and the vertices of each path are asynchronously handled along their order on the path in a forward and backward sweeping way. In this way, for each path, its any vertex state can be quickly propagated to all its other vertices along the path within one forward and backward sweeping round, no matter how long the path is, getting a faster convergence speed. Besides, the vertices of the path swapped into cache in the current round have a high probability to be immediately accessed in the next round, helping to reduce the cache miss rate for short cache reuse distance.

Meanwhile, two optimization methods are also proposed to further improve the performance of our FBS over distributed platforms. In detail, a scheme is proposed to balance the communication cost and the convergence speed via setting suitable communication time interval. A static priority ordering scheme is employed to further improve the convergence speed via first processing the path with the largest number of remote neighbours residing on the other workers. To demonstrate the efficiency of our proposal, a runtime system, namely *FBSGraph*, is also designed and built. Experiments conducted on a cluster with 1024 cores show that our FBSGraph can achieve scalable performance for large-scale graph algorithms and significantly cut the execution time of state-of-the-art approaches by at least 39.8 percent.

In summary, we mainly make three contributions:

(1) We invent a FBS execution method for AGP to accelerate its convergence speed via exploiting our observed cascade effect of AGP.
(2) We propose a scheme to reduce the communication overhead and a static priority ordering scheme to further improve the convergence speed of our method on distributed platforms.
(3) Extensive experimental results validate the correctness and efficiency of our method by the fact that it much outperforms the current solutions.

The remainder of this paper is organized as follows. Section 2 discusses the related work. Section 3 illustrates the background and the inefficiency of current solutions. The details of our execution mode, optimizations for distributed platforms and its implementation are described in Section 4. Experimental results are presented and discussed in Section 5. Finally, Section 6 concludes this work.

2 Related Work

The importance of graph processing has popularized it to a number of usage scenarios [23] and generated a series of graph processing frameworks on multicore [24], [25], [26], [27] and distributed systems [28], [29], [30]. This paper focuses on distributed graph processing frameworks for its high scalability. Existing distributed graph processing systems fall into two categories, i.e., the ones based on synchronous processing model or asynchronous processing model.

Synchronous Graph Processing Systems. Pregel [11] is one of the most popular ones based on synchronous processing model and expresses the graph algorithm as a sequence of super-steps. Based on Pregel, several synchronous systems [31] have also been developed latter to optimize it for better performance. Pregelix [32] models Pregel's semantics as a logical query plan and implements those semantics as an iterative dataflow of relational operators. By taking this approach, Pregelix is able to offer a set of alternative strategies that can fit various workloads for better performance.

GraphX [12] leverages advances in distributed dataflow frameworks so as to bring low-cost fault tolerance to graph processing. Blogel [13] proposes to divide a graph into blocks and then run graph algorithms over these blocks in a parallel way for better locality. Although these synchronous systems are easy to be implemented and reasoned about, they needs strict synchronization between iterations. As a result, their performance suffers from high synchronization cost [33], e.g., load imbalance and network jitter.

Asynchronous Graph Processing Systems. Recently, some systems are proposed to support graph processing in an asynchronous manner. Because of no barrier between iterations, they have lower synchronization cost and enable faster convergence speed than the synchronous model by incorporating the most recent updates [20], [34]. GraphLab [15], [35] is a typical one and supports AGP via pipelined locking and data versioning for less network congestion and low network latency. It also provides round-robin mode and prioritized mode for user to choose the processing way of vertices for better performance.

In order to spare the overhead to handle converged vertices, REX [36] uses a delta-oriented method to minimize the amount of data being iterated over. PowerGraph [16] employs delta caching to skip the processing of converged vertices and vertex-cut method to divide graph for balanced load. Maiter [17], [18] proposes to asynchronously update vertex state by accumulating the recent state changes with its old value. Then it not only can efficiently bypass synchronous cost but also can only handle the vertices with state changes to avoid negligible updates with low overhead. GiraphUC [19] proposes a barrierless asynchronous parallel processing method, which supports AGP without the use of distributed locking and also ensures low communication overheads. However, these systems still suffer from inefficient vertex state propagation along graph paths.

Graph Partitioning Algorithms. Graph partitioning algorithms [22] have been extensively studied for decades and can be divided into edge-cut ones and vertex-cut ones. All these methods mainly try to get balanced load and minimize the communication cost. The popular graph partitioners, e.g., METIS [37], Chaco [38] and PMRSB [39], fall in the first category and divide a graph by cutting cross-partition edges, so as to construct balanced partitions with evenly distributed vertices and a minimal number of edges spanning the partitions.

However, edge-cut schemes may perform poorly for real-world graphs, because the highly skewed distribution of vertex degrees makes them difficult to balance load and keep a low communication cost among partitions [16]. Thus, some methods are recently proposed to perform vertex-cut partitioning via evenly dividing the set of edges. The typical ones, i.e., Random, Grid, Oblivious, HDRF and PDS, are proposed in PowerGraph [16] for user to choose the suitable partitioning scheme. PowerLyra [40] provides two additional methods, i.e., Hybrid and Hybrid-Ginger, to perform differentiated partitioning for high- and low-degree vertices, aiming to get less communication cost. Gemini [14] proposes a chunk-based partitioning method to preserve the vertex access locality. However, these solutions are not suitable to AGP due to ignoring its cascade effect, which may incur slow convergence speed for AGP.

3 Background & Problem Statement

In this section, we briefly introduce AGP and explain the inefficiency of existing AGP methods.

3.1 Asynchronous Graph Processing

In asynchronous graph processing [16], [17], [18], [36], each vertex i in a given graph maintains a state s_i and its value can be asynchronously updated in an iterative way based on the most recent unprocessed state changes, e.g., Δs_j, of its neighbors. Specifically,

$$\begin{cases} s_i &= \quad s'_i \oplus \Delta s_i \\ \Delta s_i &= \quad \sum \oplus f_{(j,i)}(\Delta s_j), \end{cases} \tag{1}$$

where s_i, s'_i and Δs_i are the recent state, old state and recent state change of vertex i, respectively. The initial values of s'_i and Δs_i are given by user. $\oplus$ is a user-specified operator, and $f_{(j,i)}(\Delta s_j)$ is a user given function calculating the effect of neighbour j's state change Δs_j on the state change of vertex i. In this model, a vertex allows the recent states of its neighbors to immediately work on it, which enables faster convergence and lower synchronization cost than the synchronous processing approach.

In detail, AGP consists of two steps for the iterative state update of each vertex i in a given graph. First, in the receive step, the state change effect, e.g., $f_{(j,i)}(\Delta s_j)$, from its any neighbours, e.g., j, are accumulated via $\oplus$ in an asynchronous way. Next, in the update step, the recently accumulated results, i.e., Δs_i, are added to s'_i using $\oplus$ for the state update of vertex i. Then, its state change effect $f_{(i,k)}(\Delta s_i)$ is pushed to its neighbours, e.g., k. All the vertices in the graph perform the same procedure round by round in an asynchronous way until their states converge, e.g., $|\Delta s_i| < \varepsilon$ (a given parameter) is met for each vertex i. When updating vertex state within each round, two execution modes [15], [16], [17], [18], [35], i.e., round-robin mode and prioritized mode, can be employed. The round-robin mode is the default one, where the state updates of vertices are conducted following their storing order in a partition. The prioritized mode proposes to first handle the vertex with the largest state change within each round.

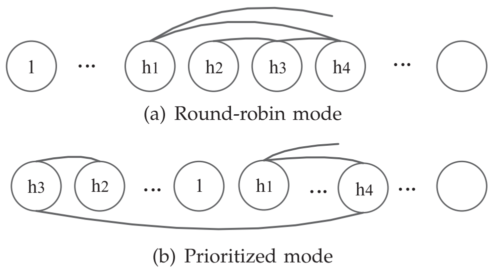


Fig. 1. Illustration of inefficient state propagation.

3.2 Inefficiency of Existing AGP Methods

Although existing AGP solutions perform better than the synchronous graph processing method, they still suffer from poor performance for inefficient vertex state propagation from two aspects.

First, using the current graph partitioning algorithms [22], the vertices in existing AGP methods may not be handled along graph path within each round, incurring slow vertex state propagation between vertices. Taken the toy graph in Fig. 1a as an example, the vertices on the path $\cdots - h_2 - h_3 - h_4 - h_1 - \cdots$ may be divided and stored in a partition in an arbitrary order, e.g., $\ldots$, h_1, h_2, h_3, h_4, $\ldots$. Then, in the round-robin mode, these vertices will be processed along the order of $\cdots$, h_1, h_2, h_3, h_4, $\cdots$ within each round. It means that h_1 will be handled before h_4 within each round. Because the states of already-processed vertices can only be updated in the next round, the large state change of h_3 (assume h_3 has the largest state change) can only work on h_4 after one round, and it needs a new round to affect h_1 as well as other vertices through h_1. For the priority mode, the vertices may be processed in the order of h_3, h_2, $\cdots$, h_1, $\cdots$, h_4, $\cdots$ within a round, according to their priorities (see Fig. 1b). Similarly, after working on h_4, state change of h_3 can only affect h_1 in a new round, because h_1 is processed before h_4. If there exists many unordered vertices like h_1, the convergence speed will be seriously affected.

Second, with the current AGP execution modes, important vertex states either face slow backward propagation along graph path or need highly additional overhead to be propagated. Take Fig. 1 as an example, when h_3 is processed, its state change Δs_{h_3} is propagated to both h_2 and h_4. In round-robin mode, as described in Fig. 1a, the state updates of vertices are conducted following their storing order in the partition and vertex h_2 is handled before h_3 in each round, although h_2, h_3 and h_4 are stored along the graph path (Note that h_2 can propagate its state to both h_3 and h_4 within a single round under such circumstances). Because the state of an already-processed neighbor can only be updated in the next round, the state update of h_2 according to h_3's state change will be delayed till the next round. Besides, vertex h_2 may also have neighbours placed before it. In such case, the large state change from vertex h_3, i.e., Δs_{h_3}, can only travel backward towards the beginning of the order one vertex each round. Although other small vertex state changes are processed, the vertices, e.g., h_2, ahead of h_3 may only converge when Δs_{h_3} is propagated to them for their state updates. As a result, it induces slow convergence speed.

The prioritized mode is proposed to accelerate the propagation of large vertex state change. For vertex processing, all state changes are sorted and the vertex with the largest state change will be processed first. Take Fig. 1b as an example, the largest state change of h_3 is processed first and its state change can be pushed to its neighbours, e.g., h_2 and h_4, and works on them within one round, rather than two rounds needed by the round-robin mode. However, in the prioritized mode, the state changes of vertices need to be sorted to determine their processing order, incurring high additional computational overhead.

These disadvantages of the current AGP methods raise an urgent and critical challenge to design a more efficient graph processing framework that can accelerate the convergence speed with low computation and communication overhead.

4 A Forward and Backward Sweeping Graph Processing Framework

In this section, we propose a graph processing framework, i.e., *FBSGraph*, to overcome the disadvantages of traditional AGP methods. It models a large graph using a collection of paths and asynchronously handles each path using a FBS execution mode for faster convergence speed with low computation overhead. Meanwhile, two optimizations are employed to reduce the communication cost and further accelerate the convergence speed on distributed platforms, respectively. Appendix C, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TKDE.2017.2781241 gives a real example to illustrate how our approach is applied.

4.1 FBS Execution for Accelerating Convergence

In practice, AGP has a feature called *cascade effect*, i.e., the state change of a vertex is able to affect the other vertices processed after it along a graph path if these vertices are handled according to their order on the path within a round of graph processing. Take the graph in Fig. 1 as an example, after processing the vertices along the path h_3–h_4–h_1–$\cdots$ for one round, the state change Δs_{h_3} of h_3 can be first propagated to h_4. Then Δs_{h_3} can be aggregated by h_4 and works on h_4 when h_4 is processed within the same round. Similarly, the newly generated state change calculated based on Δs_{h_3} is allowed to be propagated from h_4 to h_1 and immediately works on h_1 when h_1 is processed in the same round. We can observe that Δs_{h_3} in AGP can work on the other vertices, e.g., h_4 and h_1, in a single round, when these vertices are processed in the order of h_3, h_4, h_1, $\cdots$ within the round.

Based on this observation, we propose a *Forward and Backward Sweeping* (FBS) execution method for AGP. It first divides the graph into paths via a path based partitioning

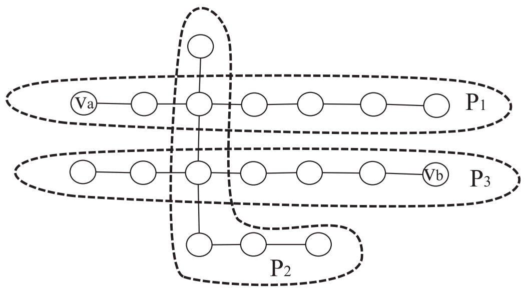


Fig. 2. Illustration of state change propagation along graph paths (e.g., state change of vertex v_a can be propagated to vertex v_b along the paths P_1, P_2, and P_3).

scheme, then uses a forward and backward sweeping mode to asynchronously handle vertices of each path according to their order on this path. In this way, as depicted in Fig. 2, large state changes of vertices can be quickly propagated to the other vertices along graph paths via using cascade effect, accelerating the convergence speed. The analysis to the efficiency, and worst- and best-case of our FBS approach in comparison with traditional techniques is discussed in Appendix A, available in the online supplemental material.

4.1.1 Path Based Graph Partition

Large-scale graphs are usually divided into partitions for parallel processing. However, as discussed in Section 3.2, with existing graph partition algorithms, the AGP may suffer from slow convergence speed because it cannot efficiently exploit the cascade effect to accelerate vertex state propagation along graph path. Therefore, we propose a new graph partition scheme to efficiently divide graph into paths in a parallel way and also ensure the vertices of each path still indexed in a partition along their original order on this path, aiming to help state propagation via cascade effect.

Specifically, let $G = (V, E)$ denote a graph, where V and E are the set of vertices and edges, respectively. We evenly divide the graph G into several partitions consisting of paths, i.e., $G = \cup P_i$, where $P_i = \cup_{(x\in P_i \wedge y\in P_i)} Path(x, y)$ is the ith graph partition, $Path(x, y)$ is a path between two vertices x and y, and any two paths have no intersecting edges. $Path(x, y) = \, < v_0, v_1, \ldots, v_k >$ is a sequence of connected edges via ordering the vertices, v_0, v_1, $\ldots$, v_k, such that $v_0 = x$, $v_k = y$, $\forall t \in [0, k-1] : (v_t, v_{t+1}) \in E$. When dividing the graph, as discussed in Appendix A, available in the online supplemental material, it should maximize the average length of all paths, i.e., $\overline{L} = \frac{1}{m}\sum_{j=1}^{m} L_j$ for faster convergence speed and also should minimize the value of $N_C = \sum_{v_l\in V} |v_l|$ so as to reduce the communication cost. There, m denotes the total number of paths, L_j denotes the length of the jth path, i.e., the number of edges in this path, and $|v_l|$ denotes the number of partitions assigned with the replicas of vertex v_l.

It is well known that the optimal solution to the graph partitioning problem is NP-complete. Thus we employ an approximate parallel method to evenly divide the graph and get the above goals. In detail, we first use a very lightweight chunk-based partition method [14] to divide the graph to several subgraphs with high locality. Then these subgraphs are evenly assigned to workers so as to divide the graph into paths in a parallel way. In order to further divide each subgraph into paths, as described in Algorithm 1, each worker takes a vertex as the root and travels

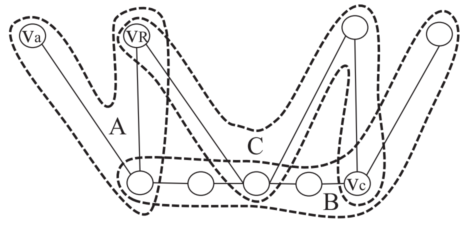


Fig. 3. An example to illustrate how to divide a subgraph into blocks A, B, and C via taking vertex v_R as the root.

the edges in a depth-first order. This algorithm is repeatedly triggered until all edges are visited. The visited edges and related vertices of each traversal can be gathered one by one as a path and stored in order in a *block* (see line 7). Note that the edges between any two vertices are visited only once as one edge and inserted into blocks, no matter they are directed/undirected edges.

In this way, as described in Fig. 3, all edges are assigned to blocks and no two blocks have intersecting edges. In addition, within each block, vertices are indexed following their original order on graph path to ensure efficient vertex state propagation along path. Note that, in Algorithm 1, each worker only deals with the edges of local subgraph (see line 3). Therefore, it does not introduce extra communication overhead among workers.

Algorithm 1. Graph Partition Algorithm

```
1:  procedure PARTITION (Vertex $v$)
2:     if vertex $v$ has unvisited local edges then
3:        /*Get successors of $v$ in local subgraph.*/  $S_{suc} \leftarrow$ GetLocalSuccessors($v$)
4:        for each successor $v_s$ of $v$ in set $S_{suc}$ do
5:           if edge $<v, v_s>$ is unvisited then
6:              Set edge $<v, v_s>$ as visited
7:              /*Inserted into its local block $b$.*/
                $b \leftarrow b \cup \{v, <v, v_s>\}$
8:              Partition($v_s$)
9:           end if
10:       end for
11:    else
12:       $b \leftarrow$ Create a new block
13:    end if
14: end procedure
```

During the above stage, it may produce some paths with very short lengths, e.g, the length of path A is two. The state changes on these paths can only be propagated to a small number of vertices within each processing round, potentially slowing down the convergence speed. Therefore, after that, we recheck all paths and merge the paths with short lengths in a head-to-tail way, aiming to increase the average length of the above generated paths. For example, as depicted in Fig. 3, path A (from vertex v_R to v_a) and path C (from vertex v_R to v_c) can be merged into one path. In order to reduce the merging overhead, our algorithm always try to merge the shortest path into a longer one. In the example shown in Fig. 3, vertices of path A should be inserted into path C and new path from vertex v_a to v_c is then generated.

Finally, the paths are selectively assigned to different partitions and all the partitions are tried to be of the same size in terms of the number of edges for balanced load among workers. The number of partitions is ensured to be equal to or more than the number of workers for parallel processing. Note that, a vertex may have multiple replicas distributed over several paths. Any state change on one replica will be notified and updated to the other ones, incurring communication cost. In order to reduce such overhead, i.e., minimizing the value of N_C, the paths with the highest correlative degree are tried to be assigned to the same partition. The correlative degree of paths p_i and p_j is evaluated via $D(p_i, p_j) = \frac{1}{max\{L_i, L_j\}} \cdot \sum_{v_l \in p_i \wedge v_l \in p_j} n_l$, where n_l is the number of replicas of vertex v_l on the longest one of p_i and p_j. L_i and L_j are the lengths of p_i and p_j, respectively. In this way, most or all replicas for the same vertex are tried to be assigned together, and it is expected to approximately get the minimum value of N_C.

4.1.2 FBS Execution

In AGP, as discussed in Section 3.2, vertex states suffer from slow backward propagation because the state of an already-processed neighbor can only be updated in the next round. Therefore, after dividing graph into paths, in our execution way, vertices on each path are alternately processed in a forward-by-backward way according to their order organized along the path.

Algorithm 2. Forward/Backward Sweeping

```
1:  procedure FBS(Path Table $T_p$) /*$T_p$ stores the vertices and related edges assigned to a path.*/
2:     Proc($T_p$, "$Forward$") /*Forward sweeping*/
3:     Proc($T_p$, "$Backward$") /*Backward sweeping*/
4:  end procedure
5:  procedure PROC(Path Table $T_p$, Order $FB$)
6:     if all vertices in $T_p$ are converged then
7:        Exit this procedure
8:     end if
9:     while $T_p$ has unprocessed vertices do
10:       if $FB$ is "$Forward$" then
11:          $j \leftarrow$ GetNext($T_p$)
12:       else
13:          $j \leftarrow$ GetPrevious($T_p$)
14:       end if
15:       $s_j \leftarrow s_j \oplus \Delta s_j$
16:       $S_{Neighbor} \leftarrow f_{(j,i)}(\Delta s_j)$
17:       for each $f_{(j,i)}(\Delta s_j) \in S_{Neighbor}$ do
18:          if local table $T$ has vertex $i$ then
19:             $\Delta s_i \leftarrow \Delta s_i \oplus f_{(j,i)}(\Delta s_j)$
20:          else
21:             OutPut($BufSend$, $i$, $f_{(j,i)}(\Delta s_j)$)
22:          end if
23:       end for
24:    end while
25: end procedure
```

The details of our FBS execution way are shown in Algorithm 2. The vertices on each path are handled in a forward and backward way (lines 10-14) as default until all states of vertices converge. In the forward sweeping, the present vertex state updating starts from the first one and moves forward until the path ends. In this way, state changes of vertices are able to be propagated to the vertices organized after them along the path. When the state of the

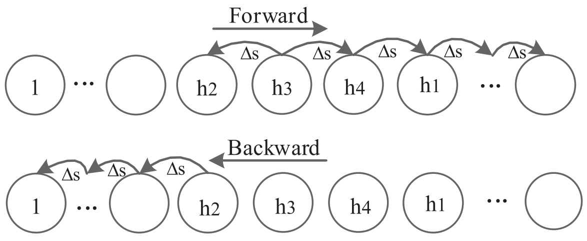


Fig. 4. Forward and backward sweeping mode.

last vertex is updated, the backward sweeping starts from the last vertex and moves backward until the first one. The state changes of these vertices can be propagated along path to the vertices organized before them. Therefore, vertex state changes are able to be propagated in cascade across the path after a FBS round, i.e., a forward round and a backward round. Note that the vertices on the path are ordered along the directed edges for directed graph. Thus, the vertices on a path may have converged after one forward sweeping and only needs a round (line 7).

Fig. 4 gives an example to illustrate how the FBS execution mode propagate vertex state for the toy graph described in Fig. 1. The vertices are stored in a block according to their sequence on the path, i.e., $\ldots$, h_2, h_3, h_4, h_1, $\ldots$. In forward sweeping, the state change of each vertex, e.g., h_3, is able to be pushed all the way towards the end. While in backward sweeping, the state change, e.g., Δs_{h_3}, is able to be pushed backward to the other vertices before h_3 on the path. Therefore, the state of each vertex is able to affect the other vertices on the same path within a FBS round. In this example, we can also find that our FBS approach also helps to reduce the cache miss rate. This is because that the vertices of a path swapped into cache in the current round have a high probability to be immediately accessed in the next round for state propagation. For example, the last vertex may be immediately processed in backward sweeping, after its processing in forward sweeping.

During a FBS round, each vertex will be processed according to user customized functions (lines 15-16). After that, the effects of its recent state change on its neighbours are sent to its neighbours (lines 17-23). As mentioned in Section 4.1.1, a vertex and its neighbours may be distributed over several workers. That is to say, the state change of one vertex needs to be sent to all its neighbours located on both local and remote workers. For local neighbours, the state change is directly accumulated with their states via the user defined "$\oplus$" operation (lines 18-19). Otherwise, the state change is buffered in $BufSend$ temperately and accumulated with other state changes targeting for the same remote neighbour, i.e., i, via "$\oplus$" operation (line 21). The accumulated changes then will be sent to targeting remote neighbours after a fixed time interval. Obviously, this potentially reduces the communication cost.

4.2 Optimizations for Distributed Platforms

Remember that partitions shall be distributed to multiple workers for parallel processing. As a result, the state changes of some vertices must be sent to their neighbours located on remote workers. This brings in significantly high communication overhead, imposing a negative impact on performance. Besides, the local state changes may become small after several rounds and are still treated the same as large state changes from other workers, slowing down the convergence speed. In the following part, we introduce a scheme to reduce the communication overhead and a static priority ordering strategy to arrange processing order of paths to accelerate the global convergence speed.

4.2.1 Communication Cost Optimization

As described in Section 4.1.2, all state changes sending to the same remote neighbour shall be buffered and accumulated in $BufSend$ within a time interval. How to set an appropriate time interval is a key issue to the performance. If a too short interval is set, state changes of present vertex will be continuously sent to its remote neighbours without an effective accumulation, causing a high communication overhead. If it is too long, state changes of present vertex will be buffered for a long time. In this case, the states of remote neighbours cannot be quickly updated, leading to a slow convergence speed. What is worse, each worker will keep conducting unnecessary computations without new updates, resulting in low resource utilization.

In order to balance the communication cost and the convergence speed, we propose two ways to set the suitable time interval. The theoretical justification on the proposed techniques for better performance is discussed in Appendix B, available in the online supplemental material.

Automatic Time Interval. We notice that the time interval can be set in an automatic way, according to the realtime execution conditions. The runtime system can periodically decrease or increase the interval until its value is suitable, according to the repeatedly profiled average value of processed state changes within a period of I_p.

In this scheme, users shall define an initial interval I_d, an adjustment parameter ΔI_d and an adjustment period I_p. The value of I_d is always increased by ΔI_d after a period of I_p to see if the average value of processed state changes increases within I_p, i.e.,

$$N(I_d + \Delta I_d) > N(I_d). \tag{2}$$

If so, we continue to increase I_d by ΔI_d. Otherwise, the value of ΔI_d is set to $-\Delta I_d$, i.e.,

$$\Delta I_d = -\Delta I_d. \tag{3}$$

Then it continues (2) until we get a suitable buffer interval with the largest value of N(I_d). The automatic way can dynamically change the time interval to suit the present execution conditions, but may face high runtime overhead caused by iteratively profiling N($I_d + \Delta I_d$) and N(I_d). Therefore, we further propose a heuristic time interval setting method with much lower computation overhead.

FBS-Round Based Time Interval. We observe that each vertex's state can be quickly propagated to the other connected vertices on the same worker within a FBS round. Hence, the local vertex states may converge after one or multiple FBS rounds. Meanwhile, $BufSend$ has accumulated large enough state changes of all local vertices to their remote neighbours. Therefore, it is natural to propagate the vertex state changes buffered in $BufSend$ to all remote neighbours on the other workers after certain rounds. Another benefit of setting FBS round as the interval is that the performance will

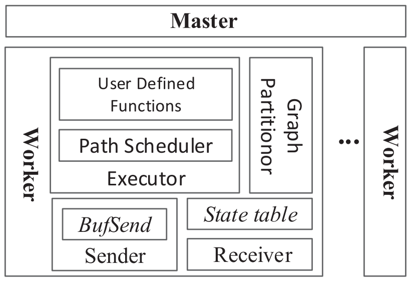


Fig. 5. Architecture overview of FBSGraph.

be less affected by the problem size. In default, the time interval is set to be one FBS round based on the fact that setting a FBS round as the interval often gets better performance. We will see that this approach indeed performs significantly better than the interval with more FBS rounds as well as the one using an automatic timer under most conditions.

4.2.2 Convergence Optimization

In order to reduce communication overhead, vertex state changes are often accumulated before being sent to the other workers. However, only after a small number of rounds, vertices on each worker quickly begin to locally converge, left with the remote vertex state changes which are much larger than the local ones and have a greater effect on the global convergence speed. The locally convergent vertices may need to be processed again for global convergence because of the remote state changes from other workers, slowing down the graph processing.

In our approach, the paths are able to be handled in an arbitrary order, without negative effect on its benefits. Note that the paths are handled in a random order by default. To accelerate the global convergence speed, we argue that the paths with large state changes from remote workers can be processed in first priority. Note that when graph partitions are evenly distributed over workers, the number of remote neighbours for each vertex is fixed and known. We introduce a static priority ordering scheme (SPO) to assign the processing order of paths on each worker. The basic concept is to assign the highest priority to the path with the largest number of remote neighbours residing on other workers. All paths then shall be processed according to their priories.

4.3 FBSGraph Framework Implementation

In this section, we build and implement our graph processing framework, i.e., *FBSGraph*, on a distributed platform, based on the FBS execution and optimizations. Fig. 5 gives the architecture of our FBSGraph framework. The whole framework consists of a *master* and multiple *workers*. The master coordinates the workers and monitors their statuses. It also judges whether the computation results have satisfied the convergence condition. When the condition is met, it immediately stops all workers and returns the results. Each worker is assigned with a graph partition and asynchronously updates the states of its vertices.

Within each worker, there is a *graph partitioner* in charge of path-based graph partitioning, an *executor* to schedule the processing order of paths and perform FBS execution on each path according to user defined functions, a *sender* dealing with state updates sending to remote vertex on other workers, a *receiver* to receive remote state updates, and a

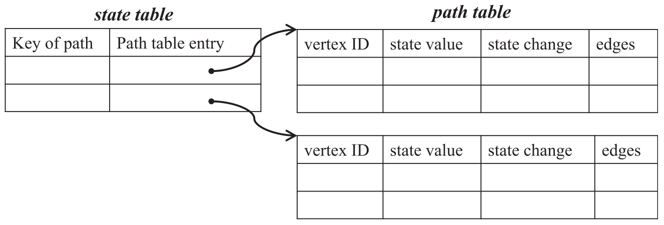


Fig. 6. Descriptions of state table and path table.

state table to store the states of all local vertices. Each data item maintained in the local in-memory state table has two fields as shown in Fig. 6. The first field stores a global unique key of each path. The second field is a pointer to a four dimensional table, i.e., *path table*, to describe all vertices and their related edges on this path. One data item in the path table indicates a vertex and contains four fields: vertex ID, state value, state change, and edges assigned in this path including associated edge information, e.g., priority.

Algorithm 3. Details of Executor

```
1:  procedure EXECUTOR(State Table $T$)
2:      /*Arrange processing order for paths in $T$.*/
        SPO($T$)
3:      while IsNotConverged($T$) do
4:          /*Asynchronously receive messages from other workers and accumulate it into $T$.*/
            NonBlockRecv($T$, $BufRecv$).
5:          while IntervalIsNotMeet() do
6:              /*Get the next path from the table $T$.*/
                $T_p \leftarrow$ Getpath($T$)
7:              /*Process the path $T_p$ in a FBS way.*/
                FBS($T_p$)
8:          end while
9:          /*Asynchronously send messages buffered in $BufSend$ to the other workers.*/
            NonBlockSend($BufSend$).
10:     end while
11: end procedure
```

With full information of paths on each worker, one typical execution procedure of our FBSGraph is described in Algorithm 3. To accelerate the convergence speed, all paths in a partition T are assigned with priorities via our SPO scheme to arrange their processing order (see line 2 and line 6). For each path, its vertices are processed in a FBS way according their storing order on the path table (line 7). State of one vertex is updated according to its related local neighbour state changes and remote neighbour state changes received by the receiver (line 4). Its new state changes sent to the other workers are first accumulated and buffered in the sender (line 21 of Algorithm 2). To efficiently reduce the communication cost, the sender sends state changes to remote workers after a time interval, e.g., after an automatic interval set by runtime or a few FBS rounds (line 5 and line 9). The above process continues until a stopping condition is satisfied (line 3).

5 PERFORMANCE EVALUATION

In this section, we compare the performance of our proposed approach with the other traditional AGP solutions. The discussion and evaluation of our approach for machine

TABLE 1
Data Sets Summary

Data sets	Vertices	Edges
Twitter [41]	41.7 million	1.4 billion
Com-Friendster [42]	65 million	1.8 billion
uk2007 [43]	105.9 million	3.7 billion
uk-union [43]	133.6 million	5.5 billion
yahoo-web [44]	1.4 billion	6.6 billion

learning algorithms is provided in Appendix E, available in the online supplemental material.

Platform and Benchmarks. The hardware platform used in our experiments is a cluster with 1024 cores residing on 64 nodes, which are interconnected by a 2-Gigabit Ethernet. Each node is a 2-way octuple-core with Intel Xeon E5-2670 2.60 GHz CPUs and 64 GB memory, running a Linux operation system with kernel version 2.6.32. A maximum of 16 workers are spawned for each node to run the benchmarks. Data communication is performed using openmpi version 1.6.3. The program is compiled with cmake version 2.6.4, gcc version 4.7.2, python version 2.7.6 and protobuf version 2.4.1.

In order to evaluate our approach, the following four typical graph algorithms from web applications and data mining are employed as benchmarks:

(1) *SimRank* [1], which is proposed to measure the similarity between two nodes in the network.
(2) *Connected Components (or CC)* [2], which is employed to find connected components in a graph.
(3) *PageRank* [3], which is a popular algorithm proposed for ranking web pages.
(4) *Adsorption* [4], which is a graph-based label propagation algorithm that provides personalized recommendation for contents (e.g., video, music, document, product, etc.).

The datasets used for these algorithms are real-world graphs downloaded from websites [41], [42], [43], [44] as described in Table 1. For the adsorption algorithm, it leverages the log-normal parameters ($\mu = 0.4$, $\sigma = 0.8$) to generate the float weight of each edge as Maiter [17], [18]. We run benchmark until its results converge, i.e., the state change Δs_i of each vertex i satisfies $|\Delta s_i| < 10^{-8}$. For each group of experiments, we run 50 times and calculate the average results.

Competitors. The performance of our proposed framework is compared with the following two execution

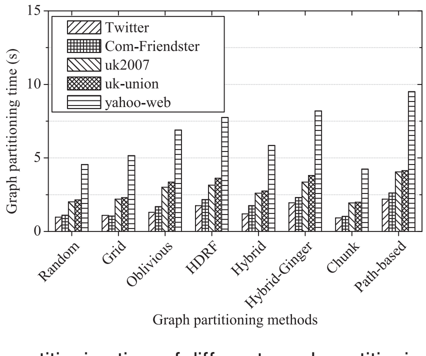


Fig. 7. Graph partitioning time of different graph partitioning methods for different data sets.

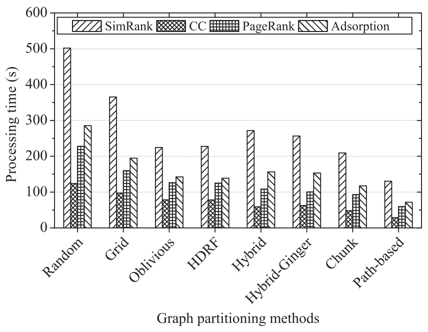


Fig. 8. Processing time of asynchronous algorithm via existing partitioning methods over yahoo-web.

methods built in Maiter [17], [18]: 1) Maiter-RR, which supports AGP using round-robin method; 2) Maiter-Pri, which uses a prioritized method with a sample-based coarse sorting procedure. Note that Maiter is implemented with the chunk-based method [14] to divide the graph for better locality with low overhead. Our FBSGraph is also implemented with several versions, i.e., FBSGraph-o, with fixed communication time interval of 100 ms, FBSGraph-n with a communication time interval of n FBS rounds, FBSGraph-d with an automatic time interval for communication and FBSGraph-SPO-1 associated with our SPO scheme based on FBSGraph-1. Their performance are also evaluated.

Besides, FBSGraph-SPO-1 is also compared with four other state-of-the-art systems, i.e., GraphLab [15], PowerGraph [16], GiraphUC [19] and Blogel [13], where the first three systems use asynchronous graph processing model and the fourth framework employs the synchronous model. For each system, we make our best effort to optimize the performance on every graph by carefully tuning the parameters, such as the partitioning method and the number of partitions.

5.1 Convergence Speed

We first evaluate the performance of FBSGraph to verify how our FBS approach affects the convergence speed compared with Maiter-RR and Maiter-pri for different benchmarks.

The performance of our path-based partitioning method is first compared with existing graph partitioning solutions [22], i.e., Random, Grid, Oblivious, HDRF, Hybrid, Hybrid-Ginger, and Chunk [14], which have been implemented on Maiter. Our partitioning approach needs additional preprocessing overhead to divide several subgraphs into paths in comparison with the chunk-based method. However, as described in Fig. 7, it is conducted in a parallel way and only needs 1.28, 1.59, 2.11, 2.15 and 5.27 seconds more than the chunk-based method to divide Twitter, Com-Friendster, uk2007, uk-union and yahoo-web, respectively. It only occupies a small proportion of the total execution time, because it only needs to traverse the graph twice and real-world graph is usually needed to be processed by an iterative graph algorithm for several rounds, even hundreds of rounds for convergence.

While our approach needs more preprocessing time than these methods, as shown in Fig. 8, the vertex processing time is significantly reduced, because it brings significantly more benefits (e.g., faster convergence speed and better locality) to AGP. For example, from Fig. 9, we can find that the AGP with our path-based partitioning method needs to

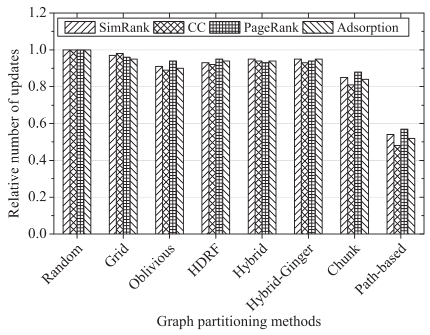


Fig. 9. Number of updates for convergence with existing partitioning methods over yahoo-web.

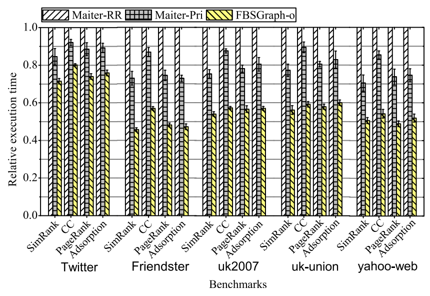


Fig. 10. Relative execution time of Maiter-Pri and FBSGraph-o against Maiter-RR.

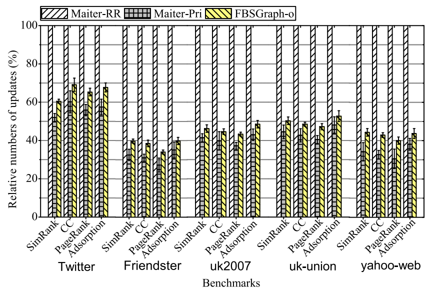


Fig. 11. Updates' efficiency of Maiter-Pri and FBSGraph-o against Maiter-RR for different benchmarks.

handle less updates to converge. Besides, such preprocessing only poses a one-time cost, while the partitioned results can be reused repeatedly by different algorithms.

With the partitioned results, Maiter-RR, Maiter-Pri and our FBSGraph-o are able to handle the graph in a parallel way. Note that, the following experiments for Maiter-RR and Maiter-Pri use the chunk-based partitioning method because it performs better than the others as described in the above experiments. The execution time (including graph partitioning time) of Maiter-RR, Maiter-Pri and our FBSGraph-o are shown in Fig. 10. We can observe that Maiter-Pri converges more quickly than Maiter-RR. For example, Maiter-Pri reduces the execution time by up to 29.5 percent compared with Maiter-RR for SimRank over yahoo-web. This is because that Maiter-RR fails to distinguish the importance of each vertex state change and handles vertices in a round-robin mode. Nevertheless, our FBSGraph-o always outperforms Maiter-Pri and Maiter-RR for all four benchmarks. For example, FBSGraph-o reduces the execution time up to 36.5 percent against Maiter-Pri for CC over yahoo-web.

The reason is that our FBS mode needs much smaller number of updates to converge than Maiter-RR, meanwhile

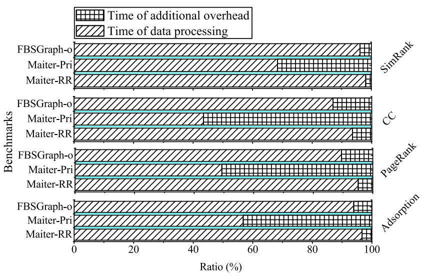


Fig. 12. Execution time breakdown of Maiter-RR, Maiter-Pri, and FBSGraph-o over yahoo-web.

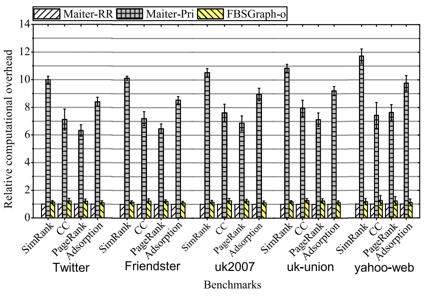


Fig. 13. Relative additional computational overhead of Maiter-Pri and FBSGraph-o against Maiter-RR.

requires much less runtime overhead against Maiter-Pri, by exploit the cascade effect, as shown in Figs. 11 and 12. Take PageRank algorithm over yahoo-web as example, it only needs about 40 percent of the updates required by Maiter-RR to converge to the same results. Note that the forward and backward sweeping way implemented in FBSGraph-o performs better than the way only using forward (or backward) sweeping, which is shown in Appendix D, available in the online supplemental material.

An interesting phenomenon observed from Fig. 11 is that our FBSGraph-o needs more updates to converge than Maiter-Pri. The reason is that Maiter-Pri is more aggressive in selecting and propagating large vertices state changes. However, the overall execution time of our FBSGraph-o is still much less than Maiter-Pri. That is because that Maiter-Pri requires much more runtime overhead to select large state changes as shown in Fig. 12. Taking CC as an example, the runtime overhead occupies up to 56.5 percent of the total execution time with Maiter-Pri, leading to an inefficient overall performance. While the ratio is only 6.5 and 13.0 percent for Maiter-RR and FBSGraph-o, respectively. For SimRank, the runtime overhead of FBSGraph-o is as less as 3.8 percent of the total execution time.

Fig. 13 describes the detailed additional computational overhead of Maiter-Pri and FBSGraph-o against Maiter-RR. We can observe that the runtime overhead of Maiter-Pri is much higher than that of either Maiter-RR or FBSGraph-o. Taking SimRank over yahoo-web as an example, the additional overhead of Maiter-Pri is even 11.7 times more than Maiter-RR, while the overhead of FBSGraph-o is only 1.18 times that of Maiter-RR.

The communication overhead of Maiter-Pri and FBSGraph-o against Maiter-RR is presented in Fig. 14. We can find that the communication overhead of Maiter-Pri is the lowest because it takes advantage of the priority-aware processing and therefore needs the smallest number of updates to converge. Fortunately, the communication

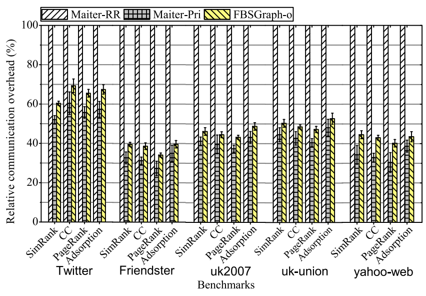


Fig. 14. Relative communication overhead of Maiter-Pri and FBSGraph-o against Maiter-RR.

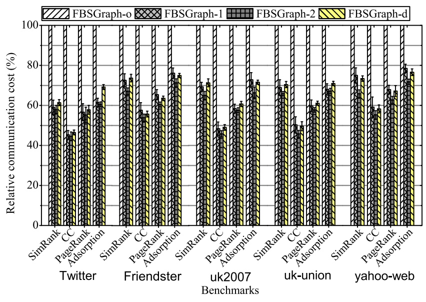


Fig. 15. Relative communication overhead of FBSGraph-o with different communication methods.

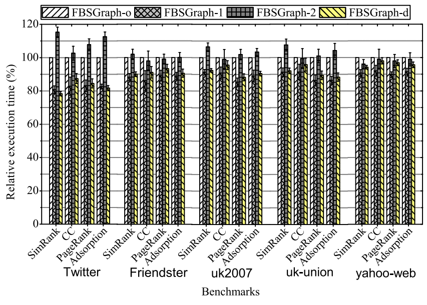


Fig. 16. Relative execution time of FBSGraph-o with different communication methods.

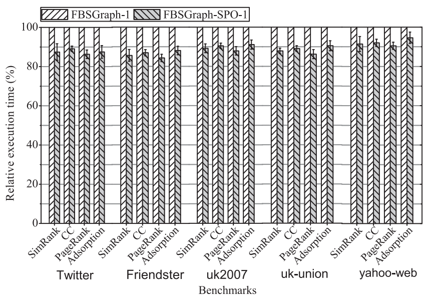


Fig. 17. Relative execution time of FBSGraph-1 with or without employing static priority ordering.

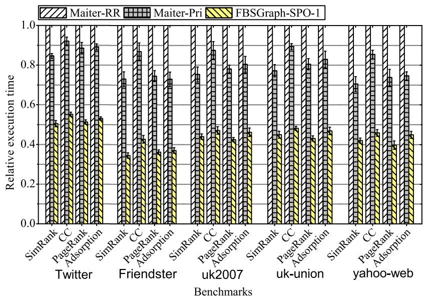


Fig. 18. Relative execution time of Maiter-Pri and FBSGraph-SPO-1 against Maiter-RR.

overhead of our FBS execution can be further reduced after integrated with a better communication interval setting and our proposed SPO optimization method.

5.2 Communication Efficiency

As mentioned in Section 4.2.1, in our approach, the communications among workers is triggered with different time intervals so as to improve the performance over distributed platform. In this section, we evaluate the efficiency of different time interval settings by comparing the performance of FBSGraph-o, FBSGraph-1, FBSGraph-2 and FBSGraph-d for different benchmarks.

The relative communication overhead of different time intervals against FBSGraph-o are presented in Fig. 15. We can observe that the communication overhead decreases with the increase of communication interval. However, a longer communication interval may induce more computational overhead to process nonsignificant vertex state change, lengthening the overall execution time.

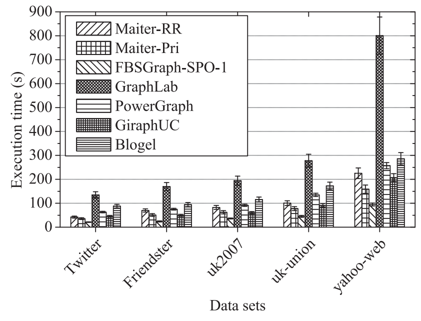
Observed from Fig. 16, FBSGraph-1 shows an advantage over the other three ones on execution time under most conditions. Taking PageRank over yahoo-web as an example, the performance of FBSGraph-1 is 9.1 percent better than that of FBSGraph-2. This is because, when the communication time interval is too long, the state changes cannot be quickly propagated to corresponding neighbours. Their states, therefore, cannot be updated with new information on time, potentially slowing down the convergence speed. It is also noticeable that FBSGraph-1 may outperform FBSGraph-d. For example, for PageRank over yahoo-web, FBSGraph-1 needs 7.9 percent less execution time than FBSGraph-d, because FBSGraph-d requires high profiling overhead for dynamical interval setting.

5.3 Comprehensive Performance Comparison

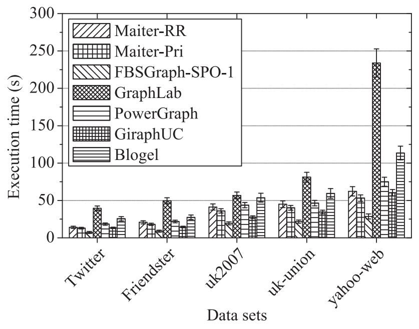
In this section, we first compare FBSGraph-SPO-1 with FBSGraph-1 over different data sets. Fig. 17 shows that our SPO scheme indeed improves the performance. For example, FBSGraph-SPO-1 reduces the execution time by 9.5 percent for PageRank over yahoo-web, compared with FBSGraph-1.

Then the relative execution time (including graph partitioning time) of FBSGraph-SPO-1, Maiter-Pri against Maiter-RR for different benchmarks are presented in Fig. 18. It can be clearly seen that FBSGraph-SPO-1 significantly outperforms the other two methods for all four benchmarks. Specially, for PageRank over yahoo-web, the execution time of Maiter-RR is reduced to 39.7 percent by FBSGraph-SPO-1. Compared with Maiter-Pri, the execution time of FBSGraph-SPO-1 is only 53.7 percent for the CC and 60.2 percent for Adsorption over yahoo-web.

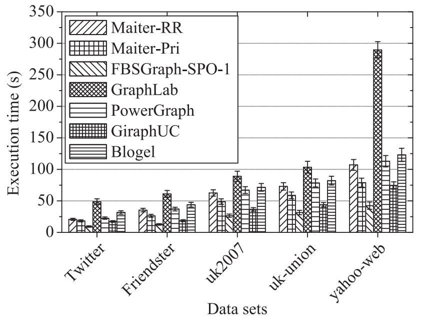
After that, we further compare the performance of FBSGraph-SPO-1 with four state-of-the-art systems, i.e., GraphLab, PowerGraph, GiraphUC and Blogel. Fig. 19 gives the execution time (including graph partitioning time) of different asynchronous graph algorithms with them over different data sets. It is obvious that FBSGraph-SPO-1 gets better performance than the others. For example, over yahoo-web, FBSGraph-SPO-1 only needs 94.6 seconds for SimRank to converge, while the execution time of PowerGraph is up to 257.1 seconds. This is because that PowerGraph not only



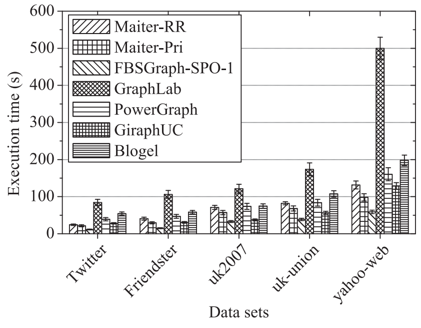
(a) SimRank



(b) CC



(c) PageRank



(d) Adsorption

Fig. 19. Execution time of different asynchronous graph algorithms with existing solutions.

faces slower convergence speed than FBSGraph-SPO-1, but also needs higher communication cost to synchronize the state between master replica and mirror replicas. Although GiraphUC ensures low communication overhead via batch communication, it also suffers from slow state propagation along graph path. For SimRank over yahoo-web, GiraphUC still needs 206.9 seconds to converge.

Besides, we can also find that Blogel has worse performance than PowerGraph. This is because that Blogel suffers from high synchronization cost, including load imbalance and network jitter, for its global barrier between iterations. Besides, such a synchronization cost also aggravates the inefficient vertex state propagation along graph path, inducing slower convergence speed. As a result, for SimRank over yahoo-web, Blogel needs 285.7 seconds. Although GraphLab uses asynchronous model and needs less vertex updates to converge, it needs expensive distributed locking to ensure consistency and also needs more communication cost to propagate vertex state for poorer locality. Its performance is even worse than Blogel. Note that we can also observe similar results for CC, PageRank and Adsorption.

Finally, the scalability of Maiter-RR, Maiter-Pri and FBSGraph-SPO-1 are evaluated by executing SimRank over yahoo-web. We increase the number of cores from 64 cores to 1024, as shown in Fig. 20. An ideal curve with a linear speedup is also drawn in the figure to give a reference. The advantage of our FBSGraph-SPO-1 can always be observed, especially when the number of cores is large. It is because that FBSGraph-SPO-1 takes the cascade effect into consideration and is also able to use a coarse-grained communication scheme. As a result, its communication overhead is significantly reduced. To prove this discussion, in Fig. 21, we also evaluate the communication overhead of Maiter-Pri, FBSGraph-SPO-1 and Maiter-RR upon different benchmarks. We can see that FBSGraph-SPO-1 requires the lowest communication overhead.

6 Conclusion

Present AGP solutions on large-scale distributed platforms either suffer from very suboptimal convergence speed or significant additional computation overhead. In this paper,

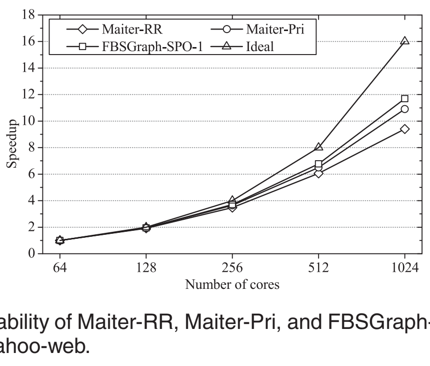


Fig. 20. Scalability of Maiter-RR, Maiter-Pri, and FBSGraph-SPO-1 evaluated over yahoo-web.

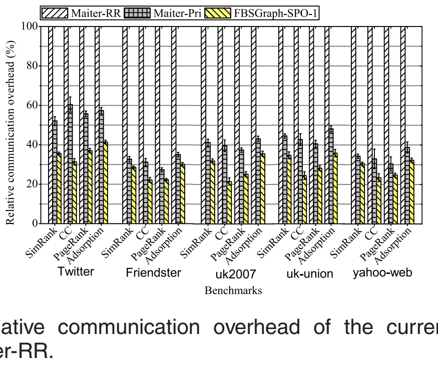


Fig. 21. Relative communication overhead of the current solutions against Maiter-RR.

we propose a forward and backward sweeping execution framework, i.e., FBSGraph, by exploiting the *cascade effect*, to accelerate the convergence speed of large-scale AGP with little additional overhead on distributed platforms. Extensive experiments are conducted using four benchmarks on a cluster of 1024 cores and the experimental results demonstrate that our new method greatly outperforms the state-of-the-art approaches.

The topologies of graph also affects the overall performance of AGP. For example, in real-world graph, a small portion of the vertices may have much higher degree than the others. They naturally play a more important role than the others on the convergence speed of AGP, because most vertex state changes will pass through them. Thus new graph partitioning strategies may need to be developed to better suit the AGP model under this condition. For example, we should consider how to partition graph to efficiently propagate state changes of the vertices with high degree using the cascade effect. This is one of our future research tasks. Besides, the current version of our approach focuses on static graph and its extension to the support of evolving graph processing is taken as a future work via incrementally dividing graph based on paths according to graph changes.

ACKNOWLEDGMENTS

This paper is supported by the National Natural Science Foundation of China under grant No. 61702202, 61628204, 61732010, and 61672251, China Postdoctoral Science Foundation Funded Project under grant No. 2017M610477 and 2017T100555, National Key Research and Development Program of China under grant No. 2017YFB1001603.

REFERENCES

[1] G. Jeh and J. Widom, “Simrank: A measure of structural-context similarity,” in *Proc. 8th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, 2002, pp. 538–543.

[2] S. Hong, N. C. Rodia, and K. Olukotun, “On fast parallel detection of strongly connected components (SCC) in small-world graphs,” in *Proc. Int. Conf. High Perform. Comput., Netw. Storage Anal.*, 2013, pp. 1–11.

[3] L. Page, S. Brin, R. Motwani, and T. Winograd, “The pagerank citation ranking: Bringing order to the web,” Technical report, Stanford Digital Library Technologies Project, 1998.

[4] S. Baluja, et al., “Video suggestion and discovery for youtube: Taking random walks through the view graph,” in *Proc. 17th Int. Conf. World Wide Web*, 2008, pp. 895–904.

[5] H. Chen, H. Jin, and X. Cui, “Hybrid followee recommendation in microblogging systems,” *Sci. China Inform. Sci.*, vol. 60, no. 012102, pp. 1–14, 2017.

[6] J. Dean and S. Ghemawat, “Mapreduce: simplified data processing on large clusters,” *Commun. ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[7] U. Kang, C. E. Tsourakakis, and C. Faloutsos, “Pegasus: A peta-scale graph mining system implementation and observations,” in *Proc. 9th IEEE Int. Conf. Data Mining*, 2009, pp. 229–238.

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

[9] N. Satish, et al., “Navigating the maze of graph analytics frameworks using massive graph datasets,” in *Proc. ACM SIGMOD Int. Conf. Manag. Data*, 2014, pp. 979–990.

[10] Y. Lu, J. Cheng, D. Yan, and H. Wu, “Large-scale distributed graph computing systems: An experimental evaluation,” *Proc. VLDB Endowment*, vol. 8, no. 3, pp. 281–292, 2014.

[11] G. Malewicz, et al., “Pregel: A system for large-scale graph processing,” in *Proc. ACM SIGMOD Int. Conf. Manag. Data*, 2010, pp. 135–146.

[12] 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. 11th USENIX Symp. Operating Syst. Des. Implementation*, 2014, pp. 599–613.

[13] D. Yan, J. Cheng, Y. Lu, and W. Ng, “Blogel: A block-centric framework for distributed computation on real-world graphs,” *Proc. VLDB Endowment*, vol. 7, no. 14, pp. 1981–1992, 2014.

[14] 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.

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

[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] Y. Zhang, Q. Gao, L. Gao, and C. Wang, “Maiter: An asynchronous graph processing framework for delta-based accumulative iterative computation,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 25, no. 8, pp. 2091–2100, 2014.

[18] Z. Wang, L. Gao, Y. Gu, Y. Bao, and G. Yu, “A fault-tolerant framework for asynchronous iterative computations in cloud environments,” in *Proc. 7th ACM Symp. Cloud Comput.*, 2016, pp. 71–83.

[19] M. Han and K. Daudjee, “Giraph unchained: Barrierless asynchronous parallel execution in pregel-like graph processing systems,” *Proc. VLDB Endowment*, vol. 8, no. 9, pp. 950–961, 2015.

[20] K. Vora, C. Tian, R. Gupta, and Z. Hu, “Coral: Confined recovery in distributed asynchronous graph processing,” in *Proc. 22nd Int. Conf. Archit. Support Program. Languages Operating Syst.*, 2017, pp. 223–236.

[21] Y. Zhang, X. Liao, H. Jin, G. Tan, and G. Min, “Inc-part: Incremental partitioning for load balancing in large-scale behavioral simulations,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 26, no. 7, pp. 1900–1909, Jul. 2015.

[22] S. Verma, L. M. Leslie, Y. Shin, and I. Gupta, “An experimental comparison of partitioning strategies in distributed graph processing,” *Proc. VLDB Endowment*, vol. 10, no. 5, pp. 493–504, 2017.

[23] Y. Zhang, L. Gu, X. Liao, H. Jin, D. Zeng, and B. B. Zhou, “Frank: A fast node ranking approach in large-scale networks,” *IEEE Netw.*, vol. 31, no. 1, pp. 36–43, 2017.

[24] Y. Chi, G. Dai, Y. Wang, G. Sun, G. Li, and H. Yang, “Nxgraph: An efficient graph processing system on a single machine,” in *Proc. IEEE Int. Conf. Data Eng.*, 2016, pp. 409–420.

[25] Z. Ai, M. Zhang, Y. Wu, X. Qian, K. Chen, and W. Zheng, “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.

[26] Y. Zhang, X. Liao, X. Shi, H. Jin, and B. He, “Efficient disk-based directed graph processing: A strongly connected component approach,” *IEEE Trans. Parallel Distrib. Syst.*, to be published, doi: 10.1109/TPDS. 2017.2776115.

[27] S. Maass, C. Min, S. Kashyap, W. Kang, M. Kumar, and T. Kim, “Mosaic: Processing a trillion-edge graph on a single machine,” in *Proc. 12th Eur. Conf. Comput. Syst.*, 2017, pp. 527–543.

[28] A. Ching, S. Edunov, M. Kabiljo, D. Logothetis, and S. Muthukrishnan, “One trillion edges: Graph processing at Facebook-scale,” *Proc. VLDB Endowment*, vol. 8, no. 12, pp. 1804–1815, 2015.

[29] B. Shao, H. Wang, and Y. Li, “Trinity: A distributed graph engine on a memory cloud,” in *Proc. ACM SIGMOD Int. Conf. Manag. Data*, 2013, pp. 505–516.

[30] Y. Zhang, X. Liao, H. Jin, and G. Tan, “Sae: Toward efficient cloud data analysis service for large-scale social networks,” *IEEE Trans. Cloud Comput.*, vol. 5, no. 3, pp. 563–575, Jul.-Sep. 2017.

[31] S. Salihoglu and J. Widom, “Optimizing graph algorithms on pregel-like systems,” *Proc. VLDB Endowment*, vol. 7, no. 7, pp. 577–588, 2014.

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

[33] Y. Zhang, X. Liao, H. Jin, and G. Min, “Resisting skew-accumulation for time-stepped applications in the cloud via exploiting parallelism,” *IEEE Trans. Cloud Comput.*, vol. 3, no. 1, pp. 54–65, Jan.-Mar. 2015.

[34] 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. 1–12.

[35] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein, "Graphlab: A new framework for parallel machine learning," in *Proc. 26th Conf. Uncertainty Artif. Intell.*, 2010, pp. 1–10.

[36] S. R. Mihaylov, Z. G. Ives, and S. Guha, "Rex: recursive, delta-based data-centric computation," *Proc. VLDB Endowment*, vol. 5, no. 11, pp. 1280–1291, 2012.

[37] D. LaSalle and G. Karypis, "Multi-threaded graph partitioning," in *Proc. 27th Int. Symp. Parallel Distrib. Process.*, 2013, pp. 225–236.

[38] B. Hendrickson and R. Leland, "A multi-level algorithm for partitioning graphs," in *Proc. ACM/IEEE Conf. Supercomputing*, 1995, pp. 1–28.

[39] S. T. Barnard, "Pmrsb: Parallel multilevel recursive spectral bisection," in *Proc. ACM/IEEE Conf. Supercomputing*, 1995, pp. 1–27.

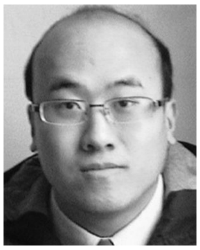
[40] R. Chen, J. Shi, Y. Chen, and H. Chen, "Powerlyra: Differentiated graph computation and partitioning on skewed graphs," in *Proc. 10th Eur. Conf. Comput. Syst.*, 2015, pp. 1:1–1:15.

[41] H. P. Haewoon Kwak, Changhyun Lee, and S. Moon, "What is twitter, a social network or a new media?" 2010. [Online]. Available: http://an.kaist.ac.kr/traces/WWW2010.html

[42] Stanford large network dataset collection, 2016. [Online]. Available: http://snap.stanford.edu/data/

[43] Laboratory for web algorithmics. datasets, 2016. [Online]. Available: http://law.di.unimi.it/datasets.php

[44] Yahoo! lab. datasets, 2016. [Online]. Available: http://webscope.sandbox.yahoo.com/catalog.php?datatype=g



Yu Zhang received the PhD degree in computer science from the Huazhong University of Science and Technology (HUST), in 2016. He is now a postdoctor in the School of Computer Science, HUST. His research interests include big data processing, cloud computing, and distributed systems. His current topic mainly focuses on application-driven big data processing and optimizations.

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 a professor in the School of Computer Science and Engineering, HUST. His research interests include in the areas of system virtualization, system software, and cloud computing. He is a member of the IEEE.

Hai Jin received the PhD degree in computer engineering from HUST, in 1994. He is a Cheung Kung Scholars chair professor of computer science and engineering with the Huazhong University of Science and Technology (HUST), in 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 the 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 scientist of the National 973 Basic Research Program Project of Virtualization Technology of Computing System, and Cloud Security. He is a fellow of the CCF, senior member of the IEEE, and a member of the ACM. He has coauthored 22 books and published more than 800 research papers. His research interests include computer architecture, virtualization technology, cluster computing and cloud computing, peer-to-peer computing, network storage, and network security.

Lin Gu received the BS degree from the School of Computer Science and Technology, Huazhong University of Science and Technology, China, in 2009. She received the MS and PhD degrees in computer science from the University of Aizu, Fukushima, Japan, in 2011 and 2015, respectively. She is now with the Services Computing Technology and System Lab, Cluster and Grid Computing Lab in the School of Computer Science and Technology, Huazhong University of Science and Technology. Her current research interests include cloud computing, big data, and software-defined networking. She is a member of the IEEE.

Bing Bing Zhou received the graduate degree in electronic engineering, in 1982 from the Nanjing Institute of Technology in China, and the PhD degree in computer science, in 1989 from Australian National University, Australia. He is an associate professor in the School of Information Technologies, University of Sydney, Australia (2003-present). Currently, he is the theme leader for distributed computing applications in the Centre for Distributed and High Performance Computing at the University of Sydney.