HotGraph: Efficient Asynchronous Processing for Real-World Graphs

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

Abstract—For large-scale graph analysis on a single PC, asynchronous processing methods are known to converge more quickly than the synchronous approach, because of more efficient propagation of vertices state. However, current asynchronous methods are still very suboptimal in propagating state across different graph partitions. This presents a bottleneck for cross-partition state update and slows down the convergence of the processing task. To tackle this problem, we propose a new method, named the *HotGraph*, to faster graph processing by extracting a backbone structure, called *hot graph*, that spans all the partitions of the original graph. With this approach, most cross-partition state propagations in traditional solutions now take place within only a few hot graph partitions, thus removing the cross-partition bottleneck. We also develop a partition scheduling algorithm to maximize the hot graph's effectiveness by keeping it in memory and assigning it the highest priority for processing as much as possible. A forward and backward sweeping execution strategy is then proposed to further accelerate the convergence. Experimental results show that HotGraph can reduce the number of vertex state updates processed by 51.5 percent, compared with state-of-the-art schemes. Applying our optimizations further reduces this number by 72.6 percent and the execution time by 80.8 percent.

Index Terms—Graph processing, asynchronous, convergence, locality, I/O

1 Introduction

The analysis of real-world graphs is emerging as an important application area and has attracted significant attention from both academia and industry. For example, PageRank [1], [2] is employed to rank pages in the World Wide Web. Other examples [3], [4], [5] include the algorithms for analyzing social networks, protein regulation networks and so on. With the increase of memory size and computational ability, shared memory multi-core machines have become a promising platform to support large-scale graph algorithms. Many systems, such as X-Stream [6] and PathGraph [7] have been proposed to support graph processing on this platform. However, these systems face high load imbalance and slow convergence problems because of the synchronization between iterations.

Recently, a number of systems, such as GraphLab [8], [9], REX [10] and Maiter [11], are proposed to process graphs in an asynchronous way. They use no barrier between iterations and the new state of each vertex is allowed to be immediately used by other vertices. Therefore, the vertex state changes can be spread along paths in the graph without extra synchronization, enabling faster convergence than the synchronous approach. In practice, the graph is often divided into partitions for parallel processing using edge-cut partitions [12], [13] or vertex-cut partitions [7], [14]. The graph partitioning algorithms mainly focus on balancing load among partitions and keeping few connections between them for low communication overhead.

However, real-world graphs often have the power-law property regarding node degrees [14], meaning that a small portion of the vertices, which we call *hot vertices*, have much higher degrees than the others. These vertices naturally play a more important role than the others in graph processing, because most vertex state changes will pass through them. The existing graph partitioning algorithms, ignorant of such an interconnection characteristic, will likely divide the paths between those hot vertices into different partitions and then process them in a random order. As a result, most vertex state changes will undergo expensive cross-partition propagation along those paths, which seriously slows down the processing. Another problem of asynchronous graph processing is *slow backward propagation* by the traditional round-robin execution strategy [9], [10], [11]. Even with a single partition, the state change propagation along the reverse processing order (backward) can still be time-consuming.

We discover that state changes can be propagated between vertices much faster when 1) the vertices are divided into the same partition, and 2) intra-partition vertices are processed sequentially along the graph path, following what we call the *cascade effect*. We propose a novel method, *HotGraph*, for faster graph processing by virtue of this feature. The main idea is to build a backbone structure, called the *hot graph*, that consists of the hot vertices and

- *Y. Zhang, X. Liao, H. Jin, and L. Gu are with 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, {xfliao, hjin}@hust.edu.cn, anheeno@gmail.com.*
- *G. Tan is with the Shenzhen Institutes of Advanced Technology, Chinese Academy of Sciences, Shenzhen 518055, China. E-mail: guang.tan@siat.ac.cn.*
- *B.B. Zhou is with the School of Information Technologies, The University of Sydney, Sydney, NSW 2006, Australia. E-mail: bbz@it.usyd.edu.au.*

Manuscript received 22 Mar. 2016; revised 7 Oct. 2016; accepted 11 Oct. 2016. Date of publication 1 Nov. 2016; date of current version 13 Apr. 2017. Recommended for acceptance by J.D. Bruguera.
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/TC.2016.2624289

paths between them, which are extracted from the original graph and span all the partitions. In essence, this structure serves as a fast track for cross-partition state propagation. With this structure, most state propagation operations that may otherwise take place across partitions by traditional methods can now be done within the same partition, thus avoiding the cross-partition bottleneck.

As a further improvement, a partition scheduling algorithm is proposed to keep the hot graph in memory and to assign its partitions a high priority for fast memory access and privileged CPU usage. We develop a *forward and backward sweeping* (FBS) execution strategy that alternately processes the vertices in a forward and backward fashion along the graph paths, which speeds up the intra-partition convergence and also reduces the cache missing rate.

Experimental results show that, compared with a state-of-the-art asynchronous graph processing method, Maiter [11], HotGraph in its basic design reduces the number of state updates by 51.5 percent, and reduces the execution time by 55.9 percent. With our partition scheduling algorithm and FBS strategy, HotGraph can decrease the number of state updates by 72.6 percent. Compared with Maiter and Maiter-path [7], HotGraph enjoys performance improvements of 5.22 and 3.29 times, respectively. Compared with X-Stream [6] and GraphChi [15], two other latest graph processing systems on multi-core, HotGraph offers dramatic improvements of 19.1 and 28.3 times, respectively.

The remainder of the paper is organized as follows: Section 2 presents the the background and our motivation. Sections 3 and 4 describe the details of hot graph and optimization methods, followed by an experimental evaluation in Section 5. Section 6 gives a brief review of related work. Finally, we conclude the paper in Section 7.

2 Background and Motivation

2.1 Disadvantages of Existing Methods

In asynchronous graph processing [8], [9], [10], [11], [14], the vertex V_i's state in kth round, s_i^k, is updated by "$\oplus$" its previous state s_i^{k-1} and the current aggregated state changes of all its neighbors Δs_i^k, namely

$$s_i^k = s_i^{k-1} \oplus \Delta s_i^k. \tag{1}$$

The value of Δs_i^k is asynchronously calculated by processing and aggregating the state change Δs_j^{k-1}, or delta value, of its neighbors with user specified functions $f_{(j,i)}()$ and "$\oplus$". It allows the algorithm to obtain each vertex's state from the most recent state of its neighbor vertices, which enables faster convergence than the synchronous processing approach can achieve. However, its performance is still sub-optimal due to inefficient state change propagation.

Slow Cross-Partition Propagation. Consider an example in Fig. 1, assume V_1 has aggregated many state changes of vertices in the partition A because of its high degree, and needs to propagate them along a path to V_9 in the partition B. However, the vertices on this path are divided into six partitions, and therefore processed in a random order, for example, $V_9 \rightarrow V_1$. That is to say, the vertices $V_9, V_6, \ldots, V_3, V_2$ are processed before V_1. Note that the state of an already-processed neighbor can only be updated in the next round, according to the current execution models [9], [10], [11]. The

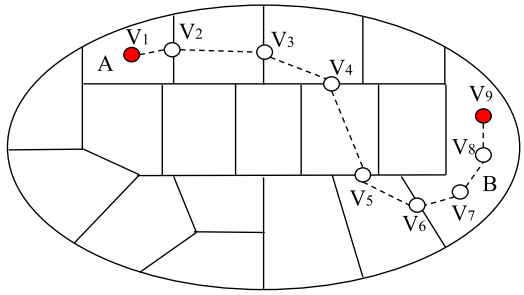


Fig. 1. Slow propagation of state changes.

state change Δs_1^k of V_1 in round k can only be propagated to V_2 in round $k+1$, to V_3 in round $k+2$ and so on. As a result, it needs at least six rounds until V_1's state change reaches V_9, contributing to the slow convergence of the processing task.

To confirm this, we evaluate the number of state changes processed for the convergence of SimRank [4] using the current graph partitioning method [7] on a real-world graph Friendster [16]. The results show that the number of state changes is increased by 2.1 times when the number of partitions is increased from 1 to 1,024 on a 32-core CPU. It is because the locally convergent vertices may need to be processed again for slowly propagated state changes from other partitions. In other words, graph partitioning for Friendster introduces a bottleneck and makes the asynchronous graph processing even slower for cross-partition state propagation.

Slow Intra-Partition Propagation. In the traditional asynchronous graph processing approach, vertices in a partition are always processed from the first to last according to their order in the partition in a round-robin way. As a result, the state changes of vertices can only be propagated in the reverse processing order, one vertex per round. For example, in Fig. 1, after processing the vertices of partition B in the order V_9, V_8, V_7 and V_6 for one round, the state change Δs_6^k of V_6 is propagated to V_7 after V_7 is processed. Δs_6^k can only take effect on V_7 in the next round, before its further propagation to V_8. Consequently, Δs_6^k can only reach V_9 after three rounds for three hops between V_6 and V_9. We call this phenomenon *slow backward propagation*.

2.2 Cascade Effect

Take the same example in Fig. 1, if all the vertices on the path from V_1 to V_9 are organized into the same partition and processed in the order $V_1, V_2, \ldots, V_8, V_9$. After processing the path for one round, the state change Δs_1^k of V_1 can be propagated to V_2. Then Δs_1^k can be aggregated by V_2 and works on V_2 when V_2 is processed within the current round. Similarly, the newly generated state change Δs_2^{k+1} of V_2, calculated based on $f_{(1,2)}$ (Δs_1^k), is allowed to immediately work on V_3 when V_3 is processed in the current round. That is to say, the state changes from the partition A can reach the partition B using a single round instead of eight rounds, thanks to the cascade effect. This motivates us to take advantage of the cascade effect and develop a new approach to accelerating the convergence of asynchronous graph processing.

3 HotGraph Approach

To fully exploit the cascade effect, we first construct a small-sized backbone structure called the *hot graph* to include the

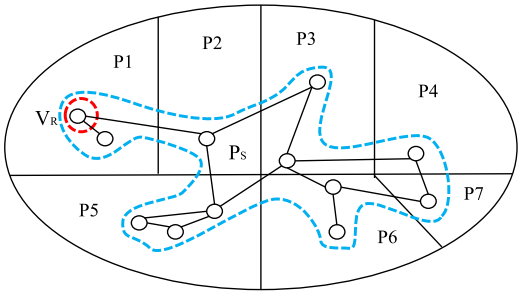


Fig. 2. An example of extracting a backbone structure P_s from a graph containing partitions P_1 to P_7. V_R is a hot vertex of P_1.

hot vertices and related paths, as shown in Fig. 2. In this way, most vertices state changes within different partitions can be quickly propagated to each other through the hot graph using much less cross-partition communication, accelerating the graph's global convergence. Its efficiency is discussed in Appendix A, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TC.2016.2624289.

3.1 Basic Concepts

Let $G = (V, E)$ denote a graph, where $V = \{V_l |\ 0 \leq l < n\}$ is a finite set of vertices, n is the number of vertices, and $E=\{(V_l, V_m) |\ V_l \in V, V_m \in V\}$ is a set of edges. The graph G is assumed to consist of several partitions, i.e., $G = \cup_{i=1}^{|P|} P_i$, where P_i is a graph partition and $|P|$ is the total number of partitions. To construct a hot graph, a set of hot vertices need to be extracted from each partition. Similar to the PageRank algorithm [17], whether a vertex is hot or cold depends on its degree $D(V_i)$, its neighbors' degrees $D(V_k)$, and the maximum degree of all vertices, D_{Max}. Without loss of generality, we define a concept, *structural hot degree (SHD)*, to evaluate the "hotness" of a vertex, calculated as

$$SHD(V_i) = D(V_i) + \alpha \cdot \frac{1}{D(V_i)} \cdot \sum_{V_k \in S(i)} D(V_k), \qquad (2)$$

where $S(i) = \{V_k |\ V_k$ is a vertex connected to $V_i\}$ and α $(0 \leq \alpha < 1)$ is the scaling factor set to $D_{Max}^{-\frac{1}{2}}$. This way, a high-degree vertex has less impact on the identification of its neighbors. A vertex V_i is considered hot if SHD(V_i) is higher than a threshold T. For example, in Fig. 3, when $\alpha = 0.3$ and $T = 4$, the vertex 11 is hot, while the vertex 9 is cold.

Here, the challenge is how to set an appropriate T to select a proper number of hot vertices. Too many hot vertices make too large a hot graph, which itself produces many partitions and undermines its efficiency. On the other hand, too few hot vertices may miss important vertices that are helpful for reducing cross-partition state propagation. In general, a suitable hot vertex ratio R should be small for real-world graphs because of the small proportions of high-degree nodes. However, to sort all vertices of the whole graph is still time-consuming considering the large graph sizes. To generate T quickly according R, we take $|V'|$ vertex samples that approximately reflect the status of the original graph, and sort them according to their SHD in descending order as a new set V'. T then shall be set to $SHD(V_{|V'|\cdot R})$, where $V_{|V'|\cdot R}$ is the $(|V'| \cdot R)$th sample vertex in set V'.

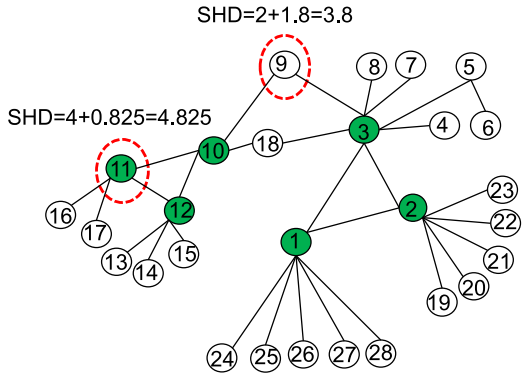


Fig. 3. An example illustrating how to identify hot vertices; $\alpha = 0.3$ and $T = 4$.

After that, these hot vertices and the related paths to connect them are extracted from each P_i as a subgraph P_i', i.e., $P_i' = \cup_{(w \in P_i \vee u \in P_i) \wedge w \in S_{hot} \wedge u \in S_{hot}} Path_i(w, u)$, $E(P_i') \cup E(P_i'') = E(P_i)$, $E(P_i') \cap E(P_i'') = \emptyset$, and $V(P_i') \cap V(P_i'') \subseteq S_{hot}$, where $S_{hot} = \{v_i | SHD(v_i) \geq T\}$ is the set containing all hot vertices in G, P_i'' is the remaining subgraph after extracting P_i' from P_i, and E(X) and V(X) are the edge set and vertex set of subgraph X, respectively. $Path_i(w, u)$ is a part of the path from vertex w to u, contained in P_i, i.e., $\forall\ (V_l, V_m) \in Path_i(w, u) : (V_l, V_m) \in P_i$. Finally, we can get the hot graph P_s from the original graph G via $P_s = \cup_{i=1}^{|P|} P_i'$. The partitions of the hot graph are called *hot partitions*, while the partitions of the remaining graph are called *cold partitions*. In addition, no two partitions have intersecting edges. The details to extract the hot graph are given in Section 3.2.

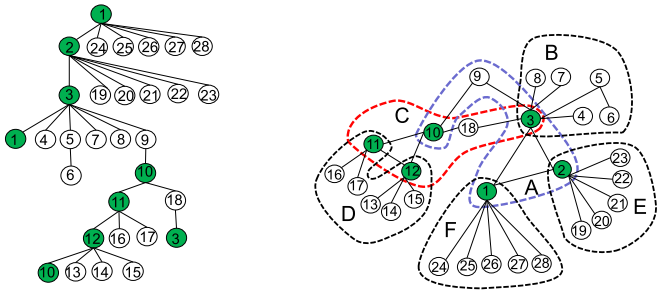
3.2 Hot Graph Extraction Algorithm

For a large graph, we first divide the graph into several subgraphs via the vertex-cut graph partition algorithm [14] and assign them to workers for hot graph extraction in parallel. In this way, the hot graph is expected to be extracted in a faster way and the vertices and edges on a path are tried to be divided into the same subgraph for quicker state propagation and better locality.

According to the threshold T, the set of hot vertices in each subgraph G^m can be determined as V^m*. To construct the hot and cold partitions from the assigned subgraph, each worker first randomly takes a hot vertex from V^m* as the root and traverses all the edges of its subgraph in a depth-first order, aiming to locally find the paths between hot vertices in the set V^m* (See Algorithm 1). The visited edges and related vertices of each traverse can be gathered one by one to form a path. The edges on the path between hot vertices are always divided into hot partitions (Lines 6 to 11), while the remaining edges are divided into cold partitions (Lines 16 to 19). This way, the graph can also be partitioned according to paths, improving the locality of vertex processing and also allowing efficient state propagation along the paths.

Fig. 4 gives an example to illustrate how to extract the hot graph from the subgraph in Fig. 3. The example subgraph is divided into six partitions, and the hot graph, constituted by A and C, has vertices spanning all other partitions.

After that, each worker begins to find the paths between hot vertices in different subgraphs in a cooperative way. For each hot vertex $V_i^m \in V^m*$, if it is connected to a cross subgraph vertex V_j^n in subgraph G^n, it takes vertex V_j^n as root



(a) Depth-first search (b) Partitions generated

Fig. 4. An example illustrating how to extract the hot graph.

and traverses subgraph G^n to find out the paths to local hot vertices in subgraph G^n. If a local hot vertex is found, this cross-subgraph path is inserted into hot partitions. Otherwise, the hot vertex V_i^m is only connected to cold vertices in subgraph G^n, path between V_i^m and V_j^n is inserted into the cold partitions.

Algorithm 1. Hot Graph Extraction Algorithm

```
1: procedure PARTITION(V_i, path, Partition P_hot, P_cold)
2:   if Vertex V_i has unvisited edges then
3:     V^suc ← findSuccessors(V_i)
4:     for each vertex V_j^suc in set V^suc and edge ⟨V_i, V_j^suc⟩ is
       unvisited do
5:       Set edge ⟨V_i, V_j^suc⟩ as visited.
6:       if V_j^suc is hot vertex then
7:         if path is not empty then /*Insert path into P_hot.*/
8:           P_hot ← P_hot ∪ path
9:           path ← ∅
10:        end if
11:        P_hot ← P_hot ∪ {V_i, ⟨V_i, V_j^suc⟩}
12:        Partition(V_j^suc, path, P_hot, P_cold)
13:      else /*V_j^suc is cold vertex.*/
14:        path ← path ∪ {V_i, ⟨V_i, V_j^suc⟩}
15:        Partition(V_j^suc, path, P_hot, P_cold)
16:        if path is not empty then
17:          path ← path−{V_i, ⟨V_i, V_j^suc⟩}
18:          P_cold ← P_cold ∪ {V_i, ⟨V_i, V_j^suc⟩}
19:        end if
20:      end if
21:    end for
22:  end if
23: end procedure
```

Note that all the partitions of the hot graph are of the same size in terms of number of edges for the sake of (coarse) load balancing, because the load of each partition is mainly dominated by the number of its edges. Although most vertex state changes need to be propagated through vertices in hot partition, they do not impose a heavy burden on the worker assigned with such a hot partition. This is because vertex state changes can be aggregated on vertices before their propagation as described in Equation (1) and several state change propagations in hot partition are combined into one. For good scalability, the balanced load among workers is achieved via the partition scheduling algorithm described in Section 4.1.

Using the above algorithm, each edge appears only once in all partitions. However, some vertices, such as V_3 in Fig. 4, may have multiple replicas distributed over different partitions. In this case, one of the replicas is nominated as the *master* which maintains the master version of the vertex data. It is usually included in the hot partition in order to keep it in memory to reduce synchronization cost. All the remaining replicas are the *mirrors* and their values are initialized according to the initial delta value of the master.

It should be noted that the partition may be much larger than the cache size, leading to poor locality for processing. Therefore, each partition is managed as a container of several equal-sized chunks and each worker is ensured to be able to load a chunk into cache for processing by adjusting the chunk size. This way, the chunks in each partition can be repeatedly processed in a forward and backward sweeping manner within several rounds, with a lower cache miss rate.

To effectively store the vertices of each chunk, a key-value table is established. Each table entry represents a vertex indexed by its key and with four other fields to describe corresponding information. The first two fields indicate vertex value and delta value, respectively. The third field stores the information of edges assigned to current chunk, e.g., priority. The forth field is a flag indicating whether current vertex is a mirror or a master version.

4 Optimization Techniques

With the hot graph, most state change propagations can now take place within a few hot graph partitions, handled in a faster way. To exploit the full power of this new structure, we propose the following two optimizations for maximum performance:

- First, we propose an algorithm that assigns higher priorities to the hot graph partitions for CPU usage and memory allocation, so that the hot graph's effectiveness is maximized;
- Second, we propose an efficient vertex execution method to replace the traditional round-robin execution model, in order to speed up the slow backward propagation discussed in Section 2.1.

4.1 Partition Scheduling Algorithm

We introduce a chunk based partition scheduling algorithm to give the hot partition chunks a higher priority for processing and keep them in memory for fast access. The priority of chunk C, donated as $Pri(C)$, is set with the joint consideration of its vertices' average structural hot degree $\overline{SHD}(C)$, the number of times it has failed to be processed $Num(C)$, and the number of its connections with the chunks that have just been processed, $Con(C)$. The basic scheduling rules are as follows:

- First, a chunk C should be set with the highest priority if its $\overline{SHD}(C)$ is the highest. This is because the chunk with a higher $\overline{SHD}(C)$ means that more state changes are accumulated on its vertices and need to be propagated to others.
- Second, the priority of a chunk C with a low $\overline{SHD}(C)$ should be raised if it has been ruled out for processing many times, i.e., a large $Num(C)$, in order to propagate state changes of vertices in a cold chunk to others.

- Third, a chunk C which has the most connections with the chunks having just been processed should be given a higher priority. This is due to the fact that the vertices state of each chunk can be propagated to others only through its neighboring chunks. Thus the chunks with more such connections are also expected to be first processed.

The above rules are captured by the following equation:

$$Pri(C) = \overline{SHD}(C) + \theta \cdot Num(C) + \beta \cdot Con(C), \quad (3)$$

where $0 < \theta < \overline{SHD}_{max}$ and $0 \leq \beta < \frac{\overline{SHD}_{max}}{Con_{max}}$ are scaling factors set at the preprocessing stage, $\overline{SHD}_{max}$ and Con_{max} are the maximum values of any chunk's $\overline{SHD}(C)$ and $Con(C)$, respectively. This way, a chunk with the highest $\overline{SHD}(C)$ can be guaranteed to be processed in the first place. The detailed calculation of Num(C) and Con(C) is described in Algorithm 2.

Algorithm 2. Partition Scheduling Algorithm

```
1: procedure SCHEDULING(Worker W_i, Chunk C_i)
2:   Set C_i as converged.
3:   IncreaseCon(C_i) /*Increase Con of chunks connected
     with C_i.*/
4:   for each unconverged chunk C_o do
5:     if Pri(C_o) > Pri(C_i) then
6:       Con(C_o)← 0
7:       Num(C_o)← 0
8:       if C_o is out-of-core then
9:         SwapIn(C_o)
10:      end if
11:      Process(W_i, C_o) /*Process chunk C_o on
         worker W_i.*/
12:    else
13:      Num(C_o)← Num(C_o)+ 1
14:    end if
15:  end for
16: end procedure
```

To store each chunk's state information, an in-memory auxiliary table is built. Each table entry corresponds to a chunk and contains seven fields. The first field stores chunk key value *chunkID* while the following fields store $Pri(C)$, $\overline{SHD}(C)$, $Num(C)$, $Con(C)$, neighbor chunks and a flag of three bits to indicate the memory management and chunk states. The *swap bit* in the flag indicates whether it can be swapped out of the memory. The *I/O bit* marks whether it is in the memory. If a chunk is being processed, it cannot be swapped out. Finally, the *convergence bit* is used to represent whether the vertices in this chunk have converged according to user specified convergence condition, e.g., a given number of rounds or local converge rate.

With this auxiliary table, Algorithm 2 is triggered to schedule the processing order of chunks when the processing of a chunk C_i is finished on worker W_i. It first increases Con of its neighbor chunks (Line 3). After that, it sequentially finds the unconverged chunk with a higher Pri than that of the finished chunk (Line 5). When a chunk is selected to be first processed, this chunk sets $Con = 0$ and $Num = 0$ (Lines 6 and 7). Meanwhile, it is loaded into memory for processing if it is out-of-core (Line 9). Then this chunk is assigned to worker W_i for processing (Line 11). This way, it can also get load balancing among workers. If an unconverged chunk cannot be processed at this moment it increases Num (Line 13).

4.2 Forward and Backward Sweeping Execution

To accelerate the backward propagation within one partition, a forward and backward sweeping execution approach is proposed. As we mentioned in Section 2, the traditional round-robin vertex processing approach needs more processing rounds for the reverse state propagation. Sometimes the number of rounds required is equal to the graph's diameter. In our approach, the vertices in a chunk are alternately processed in both forward and backward directions, according to their order along the path, until convergence. Specifically, assume the chunk C_o consists of several paths, i.e., $C_o = \cup_{(w \in C_o \wedge u \in C_o)} Path(w, u)$, where different paths have no shared edge, $Path(w, u) = <v_0, v_1, \ldots, v_k>$, $v_0 = w$ and $v_k = u$. Then, the FBS mode takes the path as the unit to processes the vertices until all vertices in this chunk have converged, and the vertices on each path $Path(w, u)$ are processed along the order of $v_0, v_1, \ldots, v_k$ in the forward sweeping round as well as $v_k, \ldots, v_1, v_0$ in the backward sweeping round.

In this way, the state change of a vertex can quickly reach other vertices in front of and behind it on the path in one FBS processing round via cascade effect, involving much fewer rounds than the traditional approach and with no additional computation overhead. For example, the Friendster [16] has a diameter of 32, but needs only 14 rounds to propagate the state change of a vertex to all others with our FBS execution model. Besides, using our FBS approach, the vertices of the chunk swapped into cache in the current round have a high probability to be immediately accessed in the next round for state propagation. It also helps to reduce the cache miss rate against the round robin strategy. The example to explain its benefits is given in Appendix B, available online.

Algorithm 3. Procedure of Executor

```
1: procedure PROCESS(Chunk C_o)
2:   while !MeetCondition(C_o) do
3:     while V_j ← C_o.GetNext(flag) do /*If flag is "1", it is the
       forward way and GetNext() gets the next data item. If
       flag is "−1", it is the backward way and GetNext()
       gets the previous data item.*/
4:       V_j.v ← V_j.v ⊕ V_j.Δv
5:       for each edge ⟨V_j, V_i⟩ of vertex V_j do
6:         Δv_i ← f_(j,i)(V_j.Δv)
7:         V_i.Δv ← V_i.Δv ⊕ Δv_i
8:       end for
9:       V_j.Δv ← Initial Value
10:    end while
11:    flag ← flag × (−1) /*Reverse the order.*/
12:  end while
13:  for each vertex V_j in Chunk C_o do
14:    if V_j is mirror vertex then
15:      Push its value as delta value to master.
16:      V_j.v ← Initial Value
17:    end if
18:  end for
19: end procedure
```

TABLE 1
Data Sets Properties

Data sets	Vertices	Edges	Sizes
Twitter [18]	41.7 million	1.4 billion	17.5 GB
Friendster [16]	65 million	1.8 billion	22.7 GB
uk2007 [19]	105.9 million	3.7 billion	46.2 GB
uk-union [19]	133.6 million	5.5 billion	68.3 GB
yahoo-web [20]	1.4 billion	6.6 billion	103 GB

The details of the FBS execution strategy are described in Algorithm 3. The vertices in each chunk are repeatedly processed in the FBS way. The inputs of our Algorithm 3 are user specified state calculation functions "$\oplus$", $f_{(j,i)}()$ and convergence condition $MeetCondition()$. If "delta value" field of any vertex receives with new updates, the corresponding chunk C_o is considered to be unconverged. In this case, a "$\oplus$" action will be first applied to this vertex, to accumulate the state changes into the vertex state ("value" field in vertex table) in local vertex table (Line 4). After that, it processes the "delta value" with user defined function $f_{(j,i)}()$ according to its edges in this chunk (Line 6) and then propagates the new "delta value" to connected vertices to update their "delta value" (Line 7). Finally, "delta value" will be initialized for future accumulation (Line 9).

After a few FBS rounds, the chunk C_o converges and all mirror vertices in this chunk send their state values to related master vertices for overall state accumulation (Line 15) and the state value of all these mirror vertices will be initialized again for future accumulation (Line 16). The final state value of any vertex can be found in the "value" field in its master version. To enable the final delta value accumulation from workers with the mirror vertices to the worker with the master vertices, each worker maintains a communication queue with a lock to control the queue synchronization.

5 Experimental Results

In this section, we present experimental evaluation of our approach in comparison with state-of-the-art schemes.

Platform and Benchmarks. The hardware platform used in our experiments is a server containing four-way eight-core 2.60 GHz Intel Xeon CPU E5-2670 and each CPU has 20 M last-level cache, running a Linux operation system with kernel version 2.6.32. Its memory is 64 GB and the secondary storage for it is disk. It spawns a worker for each core to run benchmarks. The program is compiled with cmake version 2.6.4 and gcc version 4.7.2.

In experiments, four graph algorithms from web applications and data mining are employed as benchmarks: (1) PageRank [1], a popular algorithm for ranking web pages; (2) SSSP [5], which finds single source shortest paths in a graph; (3) SimRank [4], which is proposed to measure the similarity between two nodes in the network; (4) Adsorption [3], which is a graph-based algorithm that provides personalized recommendation for contents (e.g., video, music, document). The real world graph data sets used are specified in Table 1. For the adsorption algorithm, it generates weighted graph using log-normal parameters ($\mu = 0.4$, $\sigma = 0.8$) to generate the float weight of each edge following a log-normal distribution as Maiter [11].

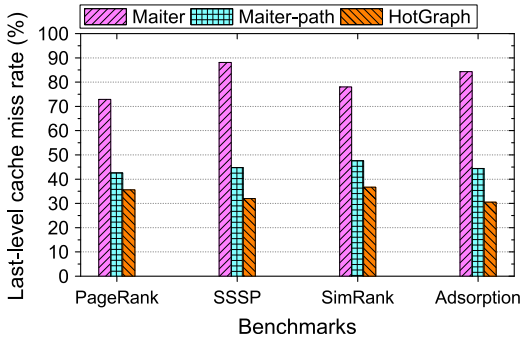


Fig. 5. Last-level cache miss rate.

Schemes for Comparison. Maiter [11] is a state-of-the-art system supporting asynchronous graph processing. It is shown to significantly outperform many alternative systems, such as Piccolo [21], Spark [22] and GraphLab [8]. In our experiments, Maiter is realized based on OpenMP 4.0 for better performance on shared memory multi-core machines. Our HotGraph system is implemented by modifying Maiter and incorporating our proposed design. The performance of HotGraph is mainly compared with Maiter [11] and Maiter-path. Instead of using a hash function to divide the graph, Maiter-path uses a more advanced graph partitioning algorithm PathGraph [7] on Maiter.

Two versions of HotGraph, namely HotGraph-RR-no and HotGraph-FBS-no, are also tested to show the performance contributions of the optimizations employed by HotGraph. HotGraph-RR-no uses the traditional round-robin execution strategy, and HotGraph-FBS-no uses our proposed FBS execution strategy, neither with our partition scheduling algorithm. In addition, HotGraph is also compared with X-Stream [6] and GraphChi [15], which are two other representative graph processing systems on multi-core. The size of each chunk is set to 2 MB and R is set to 0.5 percent for HotGraph.

5.1 Cache Miss Rate and Convergence Speed

We compare the last-level cache miss rates of Maiter, Maiter-path and HotGraph over yahoo-web using Cachegrind [23]. In Fig. 5, Maiter has a cache miss rate of up to 88.1 percent for SSSP, because irregular access in asynchronous graph processing makes Maiter frequently access the vertices out of cache. In comparison, Maiter-path partitions the graph into paths and has a lower cache miss rate 44.8 percent because the processing procedure often accesses the vertices along paths. However, Maiter-path still has a higher cache miss rate than HotGraph's 32.0 percent, because our FBS execution strategy can further improve the hit rate by changing the access order of vertices.

Fig. 6 depicts the memory miss rates of the three schemes over yahoo-web when the memory is restricted to particular sizes. As expected, the memory miss rate decreases with growing memory size. For instance, in SSSP, an increase of memory from 8 to 64 GB reduces Maiter's miss rate from 0.45 to 0.03 percent. In comparison, HotGraph has the lowest miss rate. Under 64 GB memory, for example, its miss rate is only 34.6 and 67.7 percent of Maiter and Maiter-path's results, respectively. It is because of two reasons. First, HotGraph has better locality due to the path-based partitioning and FBS execution strategy. Second, using the

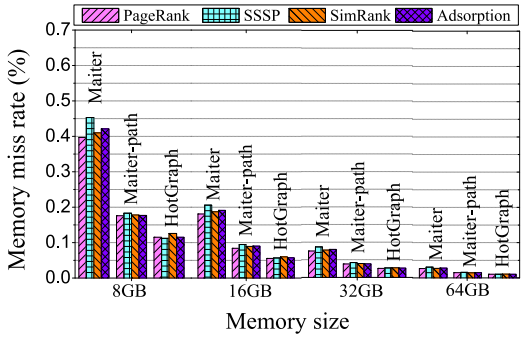


Fig. 6. Memory miss rate.

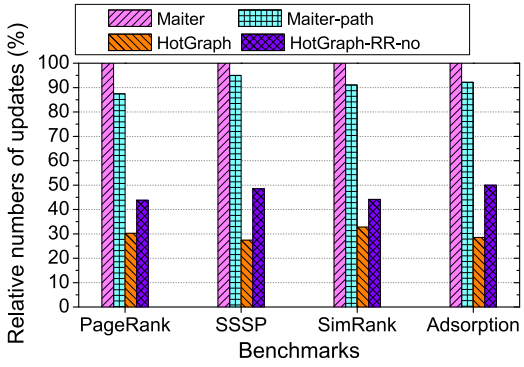


Fig. 7. Updates needed for convergence.

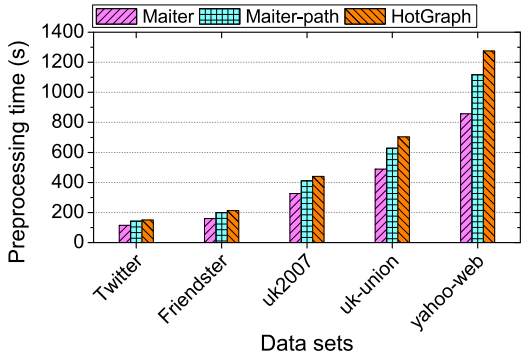


Fig. 8. Preprocessing times for different data sets.

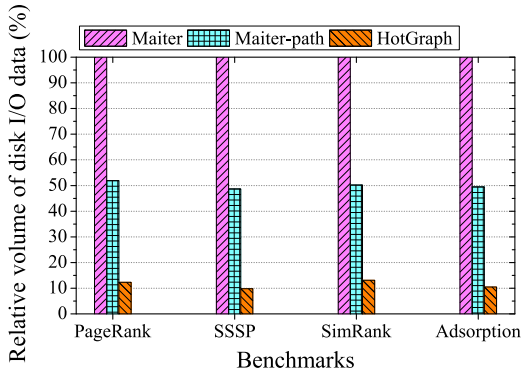


Fig. 9. Relative volume of data read/written through disk I/O.

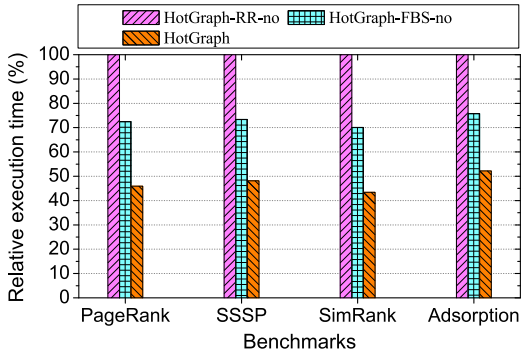


Fig. 10. Relative execution times of different schemes compared with HotGraph-RR-no.

partition scheduling algorithm, the vertices and edges in memory can be ensured to be the most needed ones by HotGraph, especially when the memory is limited.

The convergence speed is evaluated by the number of updates needed for the computation to converge, where a update is an operation to process a state change. From Fig. 7 we find that HotGraph-RR-no needs fewer updates to converge than Maiter and Maiter-path do over yahoo-web. It is because more locally convergent vertices of Maiter and Maiter-path need reprocessing for slower cross-partition state propagation than HotGraph-RR-no. For SSSP, HotGraph-RR-no involves only 48.5 and 51.1 percent as many updates as those by Maiter and Maiter-path, respectively. Meanwhile, we observe that the FBS strategy and the partition scheduling algorithm further improve the performance as expected. The number of updates needed by HotGraph is only 56.5 percent of that by HotGraph-RR-no for SSSP. It also means that HotGraph reduces the updates of Maiter by 72.6 percent.

5.2 Runtime Overhead

We evaluate the preprocessing times of Maiter, Maiter-path and HotGraph. This metric consists of the time to load data into the memory, and build and partition the graph in a parallel way. Fig. 8 shows that HotGraph takes more preprocessing time than other schemes. Although HotGraph needs 1.49 times as much time as Maiter's cost for yahoo-web, it brings significant benefits such as improved locality and higher convergence speed. Moreover, the hot graph can be reused after preprocessing.

Fig. 9 depicts the relative volume of data read or written through disk I/O over yahoo-web at runtime. For SSSP, the volume of data read or written through disk I/O by HotGraph are only 9.8 and 20.1 percent of Maiter and Maiter-path's costs, respectively. This is because HotGraph has the best locality and needs the fewest updates to converge.

5.3 Performance and Scalability

We evaluate the execution times of HotGraph, HotGraph-RR-no and HotGraph-FBS-no with yahoo-web. Fig. 10 gives the results relative to that of HotGraph-RR-no. We observe that HotGraph-FBS-no reduces the execution time by 26.6 percent for SSSP compared with HotGraph-RR-no, although they partition the graph in the same way. It is because that both the convergence and locality of HotGraph-RR-no are improved with the FBS strategy. Meanwhile, HotGraph reduces the execution time of SSSP by 34.4 percent compared with HotGraph-FBS-no, because HotGraph exploits the partition scheduling algorithm to converge more quickly and get a higher hit rate.

We also evaluate the execution times of the considered schemes over yahoo-web when the memory is restricted to

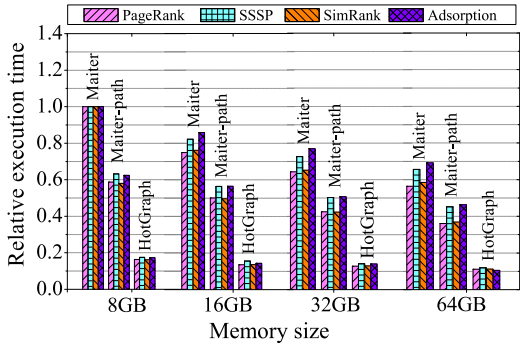


Fig. 11. Relative execution times compared with Maiter under different memory sizes.

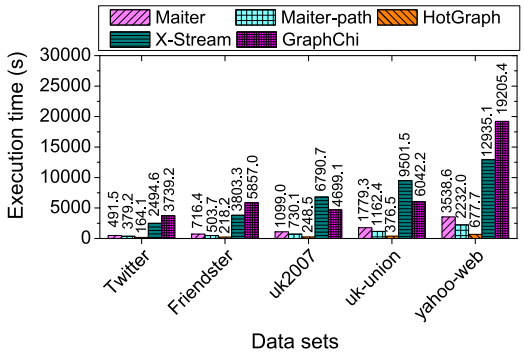


Fig. 12. Execution times of SimRank for different data sets.

particular sizes. Fig. 11 provides the relative results under 8 GB memory, using Maiter's execution time as a baseline. We see that Maiter-path obtains a speedup of 1.59 compared with that of Maiter on SimRank when the memory is 64 GB, suggesting the benefit of partitioning graph into paths. In comparison, HotGraph performs much better, with 5.22 and 3.29 times improvements over the two schemes, respectively, because of its higher hit rate and convergence speed. From Figs. 10 and 11, we can also find that the execution time of SimRank with Maiter is reduced by 55.9 percent by HotGraph in its basic version, and by 80.8 percent through our optimization techniques. Fig. 11 also shows that the execution times of different schemes decrease with growing memory size. Take SSSP as an example, the performance of HotGraph with 64 GB memory improves by a factor of 1.45 times over the case with 8 GB.

Fig. 12 depicts the execution time of SimRank using different schemes for different data sets. The superiority of HotGraph is confirmed again. It continues to perform the best over different data sets. In addition, the performance of X-Stream and GraphChi are also evaluated. Fig. 12 shows that X-Stream performs more poorly than both Maiter and Maiter-path, due to two reasons. First, X-Stream has to stream in all edges, even though many edges are useless for its convergence. Second, X-Stream suffers from a high synchronization cost. Similarly, GraphChi also performs poorly for high cost on graph edge sorting and slow convergence. For yahoo-web, Maiter yields a speedup 3.66 and 5.43 times as high as that of X-Stream and GraphChi, respectively, while HotGraph boosts the speedup by a factor of 19.1 and 28.3 times, respectively.

The scalability of these schemes is investigated as well. Fig. 13 describes the speedup of SimRank using the various schemes with different numbers of cores over yahoo-web.

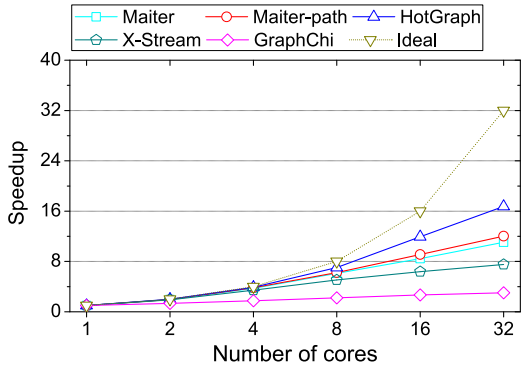


Fig. 13. Scalability of SimRank with Maiter, Maiter-path, HotGraph, X-Stream and GraphChi.

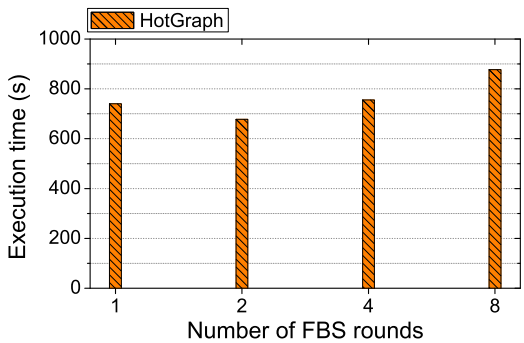


Fig. 14. Execution time of SimRank using HotGraph with different numbers of FBS rounds.

TABLE 2
Impact of α and R on SimRank

Time (s)	$R = 0.05\%$	$R = 0.5\%$	$R = 3\%$	$R = 10\%$
$\alpha = 0$	753.2	693.0	780.4	909.7
$\alpha = 0.0001$	746.0	677.7	774.5	905.1
$\alpha = 0.01$	773.8	724.3	806.1	948.5
$\alpha = 0.1$	813.1	783.7	850.9	983.3

We see that HotGraph has better scalability than other schemes. It is because HotGraph is able to efficiently propagate state changes across partitions when the number of partitions increases.

5.4 Impact of Parameters

The performance of HotGraph with different numbers of FBS rounds is evaluated for SimRank over yahoo-web, where a FBS round includes a forward round and a backward round. Fig. 14 shows that HotGraph achieves the best performance for two FBS rounds. More experimental results are shown in Appendix C, available online. It is because that it induces high data access to computation ratio when the number of sweeping is small. In contrast, it leads to much computation for the processing of state changes with low importance to convergence.

In addition, the execution time of SimRank using HotGraph is measured on yahoo-web for various values of α ($0 \leq \alpha \leq 0.1$) and R ($0 < R \leq 10$ percent), as listed in Table 2. From the table, we observe that a too small R may leave some vertices with high importance to convergence out of the hot graph, potentially leading to poor performance. On the contrary, the hot graph becomes too large and has to be divided

into several partitions, slowing down state propagation between partitions. Meanwhile, the performance also becomes worse when α is too large or too small. When the value of α is too large, the cold vertex with only a hot neighbor may be identified as hot vertex, lowering the performance. Conversely, too small α may induce the miss of much hotter vertex with numerous hot neighbors and reduce the performance as well. Therefore, a proper tradeoff should be pursued when choosing these parameters. More detailed evaluation is given in Appendix C, available online.

6 Related Work

Graph Processing Systems. Currently, several frameworks are proposed to support graph processing. Some of them are extended from general systems [21], [22], [24], [25], [26] and others are specific graph processing systems [13], [27], [28], [29], [30], [31].

GraphChi [15] proposes to reduce data access cost for graph processing on multi-core by avoiding random access to edges. However, it relies on a particular sorting of graph edges, and thus has high runtime costs. X-Stream [6] is proposed to exploit the sequential bandwidth in a low overhead way based on the edge-centric execution model. Xie et al. [32] try to handle each block of vertices repeatedly, reducing the ratio of data access cost to computation cost. It is inefficient when the vertices in each block are not highly connected with each other, for example in real-world graphs.

All these systems assume that synchronization between iterations is essential. Recently, some runtime systems, such as GraphLab [8], [9], are proposed to support graph processing in a fully asynchronous way, aiming to reduce the synchronization cost. In addition, some frameworks, such as REX [10] and Maiter [11] are proposed to avoid unnecessary computation and communication by exploiting the sparse computational dependencies. However, these schemes cannot efficiently propagate state changes in asynchronous graph processing, especially when the graph is divided into partitions.

Graph Partitioning Algorithms. Graph partitioning [33], [34] has been extensively studied in different fields for decades. However, all these schemes mainly try to minimize the communication cost and achieve balanced load.

The current algorithms can be divided into two categories. The first category is the edge-cut partition approach. It tries to divide a graph by cutting cross-partition edges, aiming to construct balanced partitions with evenly distributed vertices and a minimal number of edges spanning the partitions. The graph partitioners METIS [35], Chaco [36] and PMRSB [37] fall in this category. Some algorithms [38] based on edge-cut partition also use aggressive replication of the nodes in the graph to reduce the total network communication cost, supporting low latency querying for large-scale dynamically changing graphs. The VB-Partitioner [12] employs a vertex block-based partitioning approach, aiming at maximizing the amount of local graph processing and minimizing network I/O for inter-partition communication. Meanwhile, some efforts have been dedicated to partitioning a graph into similar sized partitions so that the workload of servers hosting these partitions will be more or less balanced. For example, Pregel [13] employs hash-based partitioning schemes to get balanced partitions.

For power-law graphs, the above algorithms may perform poorly, because the highly skewed distribution of node degrees makes it difficult to balance load among the partitions and keep a low communication cost. Recently, some methods, such as PowerGraph [14], proposed to perform vertex-cut partitioning. They equi-partition the set of edges to several partitions so that the amount of communication required to synchronize the state of vertex-copies is minimized. PathGraph [7] as a representative scheme tries to divide the graph into paths and processing each path as a unit, aiming to improve the locality of computation.

7 Conclusion and Future Work

Asynchronous graph processing has become a promising model to support large-scale real-world graph analysis because of better load balance and high convergence speed. However, state-of-the-art asynchronous solutions may still converge at suboptimal rates because of inefficient state propagation across different graph partitions. To address this problem, we propose the HotGraph approach to accelerate state propagation across partitions. Experimental results demonstrate dramatic performance improvements with our approach compared with state-of-the-art methods.

Our future work includes several aspects. First, HotGraph has been focused on static graphs so far. In the future, we will develop new algorithms to extract the hot graph in an incremental way and ensure load balance at the same time for evolving graph processing. Second, we will investigate how to dynamically adjust the number of hot vertices to get the best performance as well. Third, we will also explore how to extend HotGraph to distributed platforms. We may make each worker have a local hot partition to accelerate the convergence of its local subgraph and a global hot partition is used to accelerate convergence of the whole graph.

Acknowledgments

This paper is supported by National High-tech Research and Development Program of China (863 Program) under grant No. 2015AA015303, National Natural Science Foundation of China under grant No. 61322210, 61272408, 61433019, 61300040, Shenzhen Overseas High-level Talents Innovation and Entrepreneurship Funds No. KQCX20140520154115026 and Shenzhen Basic Research Program under Grant No. JCYJ20140610151856733. Xiaofei Liao is the corresponding author.

References

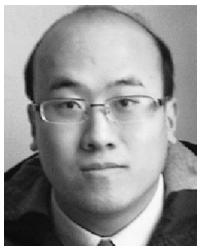
[1] D. Horowitz and S. D. Kamvar, "The anatomy of a large-scale social search engine," in *Proc. 19th Int. Conf. World Wide Web*, 2010, pp. 431–440.

[2] P. Wang, B. Xu, Y. Wu, and X. Zhou, "Link prediction in social networks: The state-of-the-art," *Sci. China Inf. Sci.*, vol. 58, no. 1, pp. 1–38, 2015.

[3] 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.

[4] 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.

[5] M. E. Newman and M. Girvan, "Finding and evaluating community structure in networks," *Phys. Rev. E*, vol. 69, no. 2, pp. 26–113, 2004.
[6] A. Roy, I. Mihailovic, and W. Zwaenepoel, "X-stream: Edge-centric graph processing using streaming partitions," in *Proc. 24th ACM Symp. Operating Syst. Principles*, 2013, pp. 472–488.
[7] P. Yuan, W. Zhang, C. Xie, H. Jin, L. Liu, and K. Lee, "Fast iterative graph computation: A path centric approach," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2014, pp. 401–412.
[8] 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.
[9] 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.
[10] 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.
[11] 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, Aug. 2014.
[12] K. Lee and L. Liu, "Efficient data partitioning model for heterogeneous graphs in the cloud," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2013, pp. 1–12.
[13] G. Malewicz, et al., "Pregel: A system for large-scale graph processing," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2010, pp. 135–146.
[14] 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.
[15] A. Kyrola, G. E. Blelloch, and C. Guestrin, "GraphChi: Large-scale graph computation on just a PC," in *Proc. 10th USENIX Symp. Operating Syst. Des. Implementation*, 2012, pp. 31–46.
[16] Stanford, "Stanford dataset," 2016. [Online]. Available: http://snap.stanford.edu/data/
[17] L. Page, S. Brin, R. Motwani, and T. Winograd, "The PageRank citation ranking: Bringing order to the web," Stanford Digit. Library Technol. Project, Stanford Univ., Stanford, CA, USA, Tech. Rep. SIDL-WP-1999–0120, 1998.
[18] H. Kwak, C. Lee, H. Park, and S. Moon, "What is Twitter, a social network or a news media?" in *Proc. 19th Int. Conf. World Wide Web*, 2010, pp. 591–600.
[19] L. for Web Algorithmics, "Datasets," 2016. [Online]. Available: http://law.di.unimi.it/datasets.php
[20] Y. Lab, "Datasets," 2016. [Online]. Available: http://webscope.sandbox.yahoo.com/catalog.php?datatype=g
[21] R. Power and J. Li, "Piccolo: Building fast, distributed programs with partitioned tables," in *Proc. 9th USENIX Symp. Operating Syst. Des. Implementation*, 2010, pp. 1–14.
[22] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," in *Proc. 2nd USENIX Conf. Hot Topics Cloud Comput.*, 2010, pp. 1–10.
[23] J. Fitzhardinge, "Cachegrind," 2016. [Online]. Available: http://www.valgrind.org/
[24] 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.
[25] J. Ekanayake, et al., "Twister: A runtime for iterative MapReduce," in *Proc. 19th ACM Int Symp. High Perform. Distrib. Comput.*, 2010, pp. 810–818.
[26] D. G. Murray, M. Schwarzkopf, C. Smowton, S. Smith, A. Madhavapeddy, and S. Hand, "CIEL: A universal execution engine for distributed data-flow computing," in *Proc. 8th USENIX Symp. Netw. Syst. Des. Implementation*, 2011, pp. 113–126.
[27] L. Chen, X. Huo, B. Ren, S. Jain, and G. Agrawal, "Efficient and simplified parallel graph processing over CPU and MIC," in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, 2015, pp. 819–828.
[28] 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.
[29] 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.
[30] A. Roy, L. Bindschaedler, J. Malicevic, and W. Zwaenepoel, "Chaos: Scale-out graph processing from secondary storage," in *Proc. 25th Symp. Operating Syst. Principles*, 2015, pp. 410–424.
[31] Y. Zhou, L. Liu, K. Lee, C. Pu, and Q. Zhang, "Fast iterative graph computation with resource aware graph parallel abstractions," in *Proc. 24th Int. Symp. High-Perform. Parallel Distrib. Comput.*, 2015, pp. 179–190.
[32] W. Xie, G. Wang, D. Bindel, A. Demers, and J. Gehrke, "Fast iterative graph computation with block updates," *Proc. VLDB Endowment*, vol. 6, no. 14, pp. 2014–2025, 2013.
[33] 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. 1–15.
[34] 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.
[35] G. Karypis and V. Kumar, "Multilevel graph partitioning schemes," in *Proc. Int. Conf. Parallel Process.*, 1995, pp. 113–122.
[36] B. Hendrickson and R. Leland, "A multi-level algorithm for partitioning graphs," in *Proc. ACM/IEEE Conf. Supercomputing*, 1995, pp. 1–28.
[37] S. T. Barnard, "PMRSB: Parallel multilevel recursive spectral bisection," in *Proc. ACM/IEEE Conf. Supercomputing*, 1995, pp. 1–27.
[38] J. Mondal and A. Deshpande, "Managing large dynamic graphs efficiently," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2012, pp. 145–156.



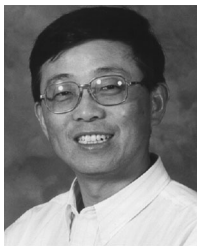
Yu Zhang received the PhD degree in computer science from Huazhong University of Science and Technology (HUST), in 2016. He is now a postdoctor in the School of Computer Science, HUST. His research interests include big data processing, cloud computing, and distributed systems. His current topic mainly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

Xiaofei Liao received the PhD degree in computer science and engineering from 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 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 Huazhong University of Science and Technology (HUST), in 1994. He is a Cheung Kung scholars chair professor of computer science and engineering with HUST, China. In 1996, he was awarded a German Academic Exchange Service fellowship to visit the Technical University of Chemnitz, 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 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. He is a fellow of the CCF, senior member of the IEEE, and a member of the ACM.

Lin Gu received the BS degree from the School of Computer Science and Technology, Huazhong University of Science and Technology, China, in 2009, and the MS and PhD degrees in computer science from the University of Aizu, Fukushima, Japan, in 2011 and 2015, respectively. She is now in the Services Computing Technology and System Lab, Cluster and Grid Computing Lab, 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.

Guang Tan received the PhD degree in computer science from the University of Warwick, United Kingdom, in 2007. He is currently a professor in Shenzhen Institutes of Advanced Technology, Chinese Academy of Sciences, China, where he works on the design of distributed systems and networks. From 2007 to 2010, he was a postdoctoral researcher with INRIA-Rennes, France. He has published more than 30 research articles in the areas of peer-to-peer computing, wireless sensor networks, and mobile computing. His research is sponsored by National Science Foundation of China and Chinese Academy of Sciences. He is a member of the IEEE, the ACM, and the CCF.



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