Front.Comput.Sci.
DOI

RESEARCH ARTICLE

An Effective Framework for Asynchronous Incremental Graph Processing

Xinqiao Lv[1], Wei Xiao[1], Yu Zhang[1], Xiaofei Liao[1], Hai Jin[1], Qiangsheng Hua[1]

1 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

Abstract Although many graph processing systems have been proposed, graphs in the real-world are often dynamic. It is important to keep the results of graph computation up-to-date. Incremental computation is demonstrated to be an efficient solution to update calculated results. Recently, many incremental graph processing systems have been proposed to handle dynamic graphs in an asynchronous way and are able to achieve better performance than those processed in a synchronous way. However, these solutions still suffer from suboptimal convergence speed due to their slow propagation of important vertex state (important to convergence speed) and poor locality. In order to solve these problems, we propose a novel graph processing framework. It introduces a dynamic partition method to gather the important vertices for high locality, and then uses a priority-based scheduling algorithm to assign them with a higher priority for an effective processing order. By such means, it is able to reduce the number of updates and increase the locality, thereby reducing the convergence time. Experimental results show that our method reduces the number of updates by 30%, and reduces the total execution time by 35%, compared with state-of-the-art systems.

Keywords incremental computation, graph processing, iterative computation, asynchronous, convergence

Received month dd, yyyy; accepted month dd, yyyy

E-mail: *zhyu@hust.edu.cn*

1 Introduction

Graphs have played an important role in real-world applications. Recently, algorithms involved in dynamic graph analysis have been widely studied such as analyzing complex relationship networks [1], ranking pages in web graphs for analyzing the evolution of social networks [2], and identifying essential proteins [3]. These algorithms need to update the computation result on the evolving graph in order to keep it up-to-date. Incremental computation [4–6] is an efficient technique for solving this problem. It reuses the result of the prior computation to accelerate convergence of a graph instead of rerunning the computation over the entire graph. Recently, asynchronous systems, such as GraphIn [7], which are based on the delta-based accumulative iterative computation model [8, 9], have been proposed to support incremental computation. Compared with synchronous systems [10–12], asynchronous systems discard the barriers, so that any processing unit vertex can immediately send the data to the successor vertex after completing the calculation step. Although current asynchronous incremental graph processing has a better state transfer method than that in the synchronous processing mode, its convergence rate is still suboptimal due to the lack of efficient partitioning method and its ineffective scheduling order.

The partitioning method involves dividing the graph into multiple parts for achieving high performance. Each partition holding a subgraph as a separate computing task can run parallel in a process or thread. Real-world graphs are always power-law graphs [13], and this means that a lower propor-

tion of vertices, which we call hot vertices, have much higher degree than the average vertex degree. These hot vertices not only easily lead to imbalanced subdivision, but also have a significant influence on the state change propagation. Some existing graph partitioning methods [14], such as hash-based vertex partition or random edge partition, tend to break the original connectivity of the graph structure, which results in low locality and high communication overhead. In addition, static partition methods are unsuitable for dynamic graphs, since changes in evolving graphs can skew these original partitions, resulting in load imbalance. In this paper, we take the efficiently performing vertex/edge additions situation into account, and the situation is a common phenomenon across most real-world applications.

The scheduling order [8] of the vertices will directly affect the convergence time of the iterative computation, and a proper scheduling algorithm will accelerate convergence of the graph. During the propagation of the state change, the predecessor vertex, called the pre-vertex, always affects the successor vertex, called the suc-vertex. Scheduling the vertex with larger state change in priority will accelerate all vertices reaching convergences. However, the traditional round-robin scheduling method does not take the state change propagation efficiency into account. For example, it is possible to schedule a low-priority vertex first, and then schedule a high-priority vertex, where low-priority means low state change. In this way the propagation effect of the latter would cause the former to be ineffective, resulting in poor convergence speed.

In this paper, we propose a new framework to partition graphs and schedule vertices, which leads to a higher access locality and faster convergence time. Concretely, our framework consists of two schemes: First, the partition strategy classifies the vertices into hot and cold vertices dependent on their degrees, where hot vertices have higher priority to influence others than cold vertices. Then the partition algorithm gathers hot vertices and their adjacent edges to build hot blocks and assigns them to the hot partitions dynamically while vertex/edge additions occur. Second, to further utilize the partition algorithm, a hot-priority scheduling algorithm is proposed. It manages each partition as several equal-sized chunks, and regards each chunk as scheduling units. For fast convergence, it assigns hot chunks a higher priority. Since the state change propagation from that chunk always has greater influence comparing with lower one, the chunk with a higher priority is always scheduled first during the iterative computation.

The experimental results show that our approach is effective with respect to execution time and the number of state updates. In addition, our experiment also shows that the algorithm optimizes the locality by comparing the cache hit rates. Compared with GraphIn, a state-of-the-art asynchronous incremental computation system, our approach is more efficient since it reduces the total execution time to lower than 35% and it reduces the number of state updates of vertices by 30%.

Our contributions are as follows:

- We first propose a partition approach, which improves access locality in asynchronous incremental computation and provides the basic for scheduling algorithm;
- We next propose a hot-priority scheduling algorithm for fast convergence by focusing special attention on the hot vertex;
- We implement our framework by modifying Maiter [8], which is an asynchronous iterative computation system for static graphs. The experimental results demonstrate that our approach is effective on evolving power-law graphs.

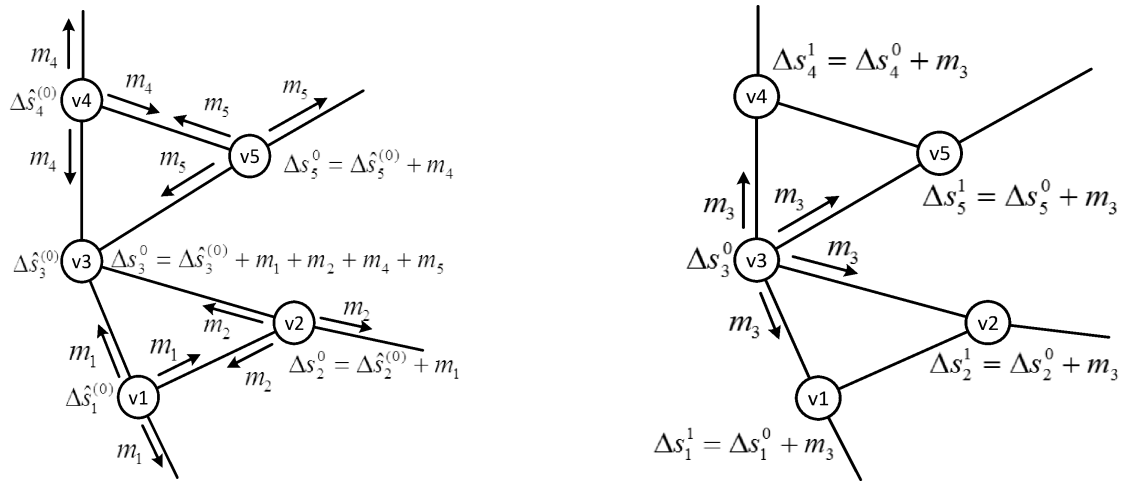
The rest of this paper is organized as follows. Section 2 introduces the background statement and challenges. Section 3 and Section 4 describe the details of the framework, followed by experiments described in Section 5. Section 6 briefly surveys related works. Finally, we conclude the paper in Section 7.

2 Background and Problem Statement

In this section, we first present basic concepts in graph. We next introduce Maiter and some key contents in asynchronous incremental graph processing. We finally dicuss some problems in partitioning and scheduling.

2.1 Preliminary

A graph $G^t = (V^t, E^t)$ consists of a finite set V^t of vertices, $\{v_i \mid v_i \in V^t\}(0 \leqslant j < n)$, and a set E^t of directed edges (we view undirected edge as two directed edges), where $E^t = \left\{e = \left(v_i, v_j\right) \in E^t \mid v_i \in V^t, v_j \in V^t\right\}$. We call v_i and v_j as the resource vertex and the destination vertex of e, respectively. V^t represents the element of the graph and E^t represents the relationship between elements at the time t. Let $\left|V^t\right|$ denote the number of vertices in the set V^t, and $\left|E^t\right|$ represent the number of edges in set E^t. A parameter k is used to denote the number of partitions, and graph partitioner divides the graph into a collection $P_0^t, P_1^t, ...P_{k-1}^t$ that yields k sub-



(a) (b)

Fig. 1 A possible round-robin scheduling order is $v_4 \to v_5 \to v_1 \to v_2 \to v_3$. (a) illustrates the scheduling of v_4 to v_2. When scheduling v_4, it first updates its state value $s_4^1 = s_4^0 \bigoplus \Delta s_4^0$, then it pushes m_4 to its suc-vertex and sets Δs_4^0 to initial value zero. When scheduling v_5, it receives the message m_4 and accumulates m_4 to Δs_5^0, then it does the same work as v_4 does, updating its state value s_5^1, sending message m_5 to its suc-vertex and setting Δs_5^0 to 0. Similarly, v_1 and v_2 have the same job as v_4 and v_5 do. Finally, (b) illustrates the scheduling of v_3. v_3 also performs its update operation, and sends back m_3 to all its nerghours.

graphs. Let $\left|P_i^t\right| (0 \leqslant i < k)$ represent the number of elements in partition P_i^t.

In asynchronous delta-based accumulative iterative computation, according to Maiter [8], each vertex v_i updates its s_i^k and Δs_i^k starting from $\widehat{s^{(0)}}$ and $\Delta\widehat{s^{(0)}}$ independently and asynchronously, where s_i^k presents the state of v_i after k iterations, Δs_i^k presents the change from s_i^{k-1} to s_i^k, $\widehat{s^{(0)}}$ denotes the original state, and $\Delta\widehat{s^{(0)}}$ denotes the inital delta value of the new graph. For each iteration, there are two separate operations for vertex v_i:

- Accumulate operation: whenever receiving a message m_j (e.g., value is $g\{j,i\}\left(\Delta s_j^{k-1}\right)$) sent from any pre-vertex $v_j \in V$, it accumulates the m_j to Δs_i^k, according to this function:

$$\Delta s_i^k = \sum_{j=1}^{n} \bigoplus g\{j,i\}\left(\Delta s_j^{k-1}\right), \tag{1}$$

  where $g\{j,i\}()$ is a user-specified function used to calculate the value passed from vertex v_j to vertex v_i, and "$\bigoplus$" is an abstract operator.
- Update operation: it first updates its state s_i^{k-1} to s_i^k by accumulating Δs_i^k, according to the following function:

$$s_i^k = s_i^{k-1} \bigoplus \Delta s_i^k, \tag{2}$$

  then sends the message $g\{i,h\}\left(\Delta s_i^{k-1}\right)$ to any of v_i's out-neighbours v_h, and resets Δv_i to 0.

After several iterations of the above two operations, the vertex state should remain unchanged in the end. We determine graph convergence when all vertices in the graph are unchanged. Let $s^{(*)}$ denote the convergence point on G^t.

In asynchronous incremental graph processing, according to [7], in order to reuse the result of the prior computation instead of recomputing from scratch, the computation result on the previous graph is necessary to construct $\widehat{s^{(0)}}$ and $\Delta\widehat{s^{(0)}}$ when the new data arrives (G^t changes to G^{t+1}). For a graph data mining algorithm with the operator '$\bigoplus$' as '+', $\widehat{s^{(0)}}$ and $\Delta\widehat{s^{(0)}}$ are constructed in following way: for a kept vertex v_i, let $\widehat{s_i^{(0)}} = s_i^{(*)}$; for a newly added vertex v_j, let $\widehat{s_j^{(0)}} = 0$. To ensure that $s_j^1 = \widehat{s_j^{(0)}} \bigoplus \Delta\widehat{s_j^{(0)}}$, $\widehat{s_j^{(0)}}$ is calculated by $\Delta\widehat{s_j^{(0)}} = s_j^1 - \widehat{s_j^{(0)}}$, since '$\bigoplus$' is '+'. Typically, s_j^1 can be derived from the following form: $s_j^k = \sum_{r=1}^{n} \bigoplus g\{r,j\}\left(s_r^{k-1}\right)$.

2.2 Challenges of Existing Solutions

The placement of vertices and edges affects the communication and computation of graph processing and therefore plays a key role in performance. Since real-world graphs are often dynamic, it is necessary to provide efficient graph partitioning when a graph changes. Suppose that there is an initial partitioning of the original graph. If the new graph is placed following random partitioning, it will result in a high communication cost among partitions since random partitioning does not utilize any information about the structure of the graph. Moreover, a power-law distributed graph will always have poor performance if some partition methods ignore the important role of hot vertices. In addition, applying the of-

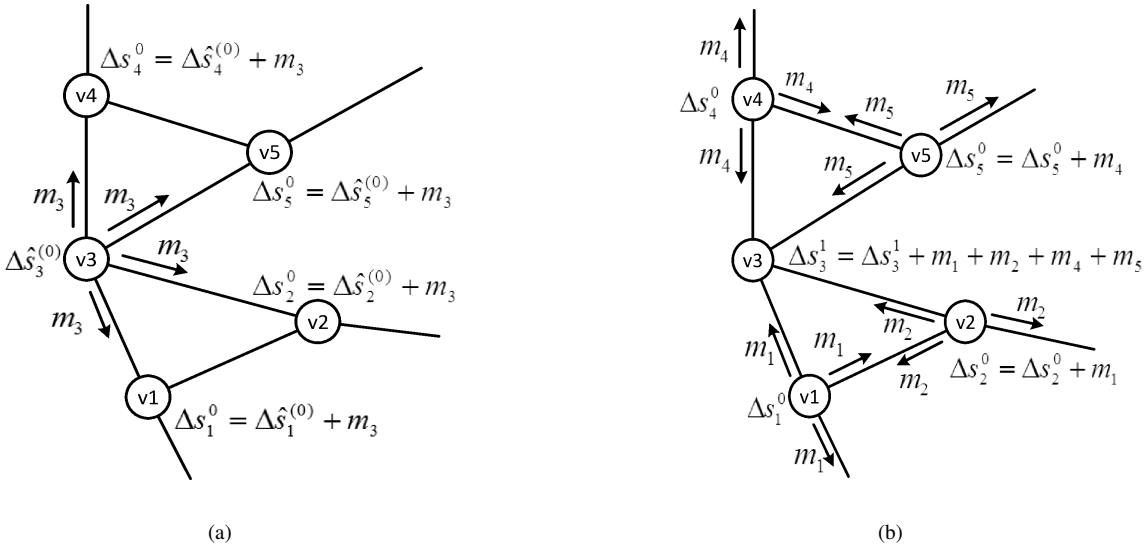


Fig. 2 A priority scheduling order is $v_3 \rightarrow v_4 \rightarrow v_5 \rightarrow v_1 \rightarrow v_2$. (a) illustrates the scheduling of v_3. v_3 updates the state value and sends m_3 to its suc-vertex. The rest of the vertices v_1, v_2, v_4, v_5 all have received m_3 before scheduling. (b) illustrates the scheduling of v_4 to v_2. When scheduling v_4, it should receive the message m_3 and accumulate m_3 to Δs_4^0, then it does the same work as v_3 does, updating its state value s_4^1, sending message m_4 to its suc-vertex and setting Δs_4^0 to 0. Similarly, v_1, v_2 and v_5 have the same job as v_4 does.

fline graph partition method [15] on a dynamic graph every time its structure changes is obviously inefficent due to the high overhead of repartitioning.

The round-robin scheduling algorithm still has room for improvement. It has suboptimal performance since it ignores the priority of scheduling vertex. We take PageRank as a trivial instance. Let us see what happens when considering the delta value as priority, which is proposed by Maiter [8]. Fig. 1 and Fig. 2 illustrate how the state change propagates in the round-robin way and priority scheduling, respectively.

Initially, the priority is determined by the delta value. There is an assumption that the current delta values of v_1, v_2, v_3, v_4, v_5 are 0.3, 0.2, 0.8, 0.3, 0.2, respectively. A possible round-robin scheduling is listed as follows: $v_4 \rightarrow v_5 \rightarrow v_1 \rightarrow v_2 \rightarrow v_3$. When scheduling v_i, it first receives the message m_j sent from its pre-vertex v_j and accumulates m_j to Δs_i^0. Then v_i updates its state value $s_i^1 = s_i^0 \bigoplus \Delta s_i^0$, where "$\bigoplus$" is "+". Finally, it pushes m_i to its suc-vertex and sets Δs_i^0 to initial value zero. At the time all vertices listed finish an iteration, the total delta value $T_d = 1.8$ drops to $T_d{}' = \Delta s_1 + \Delta s_2 + \Delta s_3 + \Delta s_4 + \Delta s_5 = 0.4 + 0.3 + 0 + 0.4 + 0.3 = 1.4$. With the same initialization condition, the scheduling order is changed to: $v_3 \rightarrow v_4 \rightarrow v_5 \rightarrow v_1 \rightarrow v_2$ in priority scheduling. The difference from the previous one is that the vertex v_3 with the highest delta value is scheduled first. The total delta value of all vertices is now $T_d{}' = \Delta s_1 + \Delta s_2 + \Delta s_3 + \Delta s_4 + \Delta s_5 = 0.189 + 0 + 0.712 + 0.189 + 0 = 1.09$. Comparing with the previous one, $T_d{}' = 1.4$, the priority scheduling method speeds up the delta value's approach to zero. However, the above priority scheduling method ignores the impacts of vertex degree.

We here compare the above priority scheduling with determining the priority by degrees. The initialization condition is that all vertices have the same initial delta value 0.2 except $\Delta\widehat{v_4^{(0)}} = 0.21$. The priority scheduling order is listed as: $v_4 \rightarrow v_5 \rightarrow v_1 \rightarrow v_2 \rightarrow v_3$ since the vertex v_4 has the highest priority decided by the delta value. In this case, the result is $T_d{}' = \Delta s_1 + \Delta s_2 + \Delta s_3 + \Delta s_4 + \Delta s_5 = 0.218 + 0.129 + 0 + 0.219 + 0.129 = 0.695$. With the same initialization condition, the priority scheduling order based on degrees is changed to: $v_3 \rightarrow v_4 \rightarrow v_5 \rightarrow v_1 \rightarrow v_2$ since the vertex v_3 has the highest degree. The total delta value of all vertices is now $T_d{}' = \Delta s_1 + \Delta s_2 + \Delta s_3 + \Delta s_4 + \Delta s_5 = 0.111 + 0 + 0.393 + 0.112 + 0 = 0.616$. The result shows that scheduling the vertex with high degree first has a better convergence speed.

In addition, the priority scheduling in Maiter is still a random scheduling, which always results in a large number of random access. Suppose that a possible priority scheduling order is $v_3 \rightarrow v_1 \rightarrow v_5 \rightarrow v_4$. However, these four vertices may not be in a contiguous storage. The result of processing may be ineffective due to poor locality.

3 Partition Scheme

In this section, we discuss the details of our partition scheme. Firstly, the original graph is decomposed into multiple sub-

graphs in a hash-based vertex partitioning. Subgraphs are assigned to each worker for spawning partitions in parallel. There are three steps to spawn partitions: (1) Distinguish a set of hot vertices from the original graph. (2) Build the hot vertex block consisting of vertex and edges incident to each hot vertex. (3) Group hot blocks to the same partition by a partition algorithm to acheive a better locality. Then a heuristic method is provided for assigning cold vertices and their incident edges. Secondly, when the graph is dynamic, we discuss how to maintain the characteristic of hot blocks dynamically when modifications occur. Our assumption is that at each time step, there are only additions of vertices and edges.

Algorithm 1 Partition Algorithm

Input: V, E, V_{hot}
Output: P_i

1: **for** each v_i in set V_{hot} **do**
2: **if** $D(v_i) \geq \tau$ **then**
3: $V_{suc} \leftarrow findsuc(v_i)$
4: **for** each edge $(v_r, v_j), v_r \in \{v_i \cup V_{suc}\}, v_j \in V_{suc}$ **do**
5: Set (v_r, v_j) as visited
6: $P_h \leftarrow P_h \cup \left\{v_r, (v_r, v_j), v_j\right\}$
7: **end for**
8: **end if**
9: **end for**
10: **for** each P_i in set P_c **do**
11: $P_i \leftarrow \cup rep(V_{hot})$
12: **end for**
13: **for** each v_i , $v_i \in V \wedge v_i \notin V_{hot}$ **do**
14: $V_{suc} \leftarrow findsuc(v_i)$
15: **for** each $(v_i, v_j), v_j \in V_{suc}$ and (v_i, v_j) is unvisited **do**
16: **if** $A(v_i) = \phi \wedge A\left(v_j\right) = \phi$ **then**
17: Assign $\left\{(v_i, v_j)\right\}$ to $minsize(P_c)$ /* The result of function $minsize(P_c)$ is a partition with the minimum size in set P_c.*/
18: **else if** $\left(A(v_i) \neq \phi \wedge A\left(v_j\right) = \phi\right) \vee (A(v_i) = \phi \wedge A\left(v_j\right) \neq \phi)$ **then**
19: Assign $\left\{(v_i, v_j)\right\}$ to $minsize(A(v_i))$
20: **else if** $A(v_i) \cap A\left(v_j\right) \neq \phi$ **then**
21: Assign $\left\{(v_i, v_j)\right\}$ to $minsize\left(A(v_i) \cap A\left(v_j\right)\right)$
22: **else if** $A(v_i) \cap A\left(v_j\right) = \phi$ **then**
23: Assign $\left\{(v_i, v_j)\right\}$ to $minsize\left(A(v_i) \cup A\left(v_j\right)\right)$
24: **end if**
25: **end for**
26: **end for**

A set of hot vertices needs to be identified firstly. Let $V_h = \{v_i \mid v_i \in V, D(v_i) \geqslant \tau\}$ to denote the set of hot vertices, where $D(v_i)$ is the degree of v_i, and τ is a system-supplied threshold. Note that if the threshold value decreases, a larger number of vertices would be selected as hot vertices. These hot vertices lead to a large number of hot blocks and hot partitions, reducing the system efficiency. From another point of view, a high threshold value would not be helpful. A suitable threshold value should be determined to match the hot vertices ratio of the real-world graph. As sorting all vertices of the whole graph is not efficient, a statistical sampling method can be considered. We randomly select a small subset of vertices $V' \subset V$ instead of the entire graph, and sort them to V'' ($V'' \cup V' = V', V'' \cap V' = V'$). Suppose that there is an R proportion of hot vertices in the graph; the threshold value shall be set to the degree of $(|V'| * R)^{th}$ sampled vertex in set V', which is $D\left(V''_{|V'|*R}\right)$.

The hot block of a hot vertex $v_i \in V_h$ is denoted by $HVB_i = \left\{V_i^h \cup E_i^h\right\}$, where $V_i^h = \{v_i\} \cup \{v | v \in V, (v_i, v) \in E\}$, $E_i^h = \left\{(v', v'') | v' \in V_i^h, v'' \in V, (v', v'') \in E\right\}$. There are two types of partitions, P_h and P_c, where $P_h = \bigcup_{v_i \in V_h} HVB_i$ and P_c indicates the cold remaining subgraph. For each hot vertex in V_h, we concentrate the hot vertex and its incident edges to build hot blocks and assign them to the hot partition P_h. After finishing the hot vertices, we next handle the cold vertices with a heuristic partitioning. The details are shown in Algorithm 1.

The set of hot vertices is determined if vertex v_i satisfies $D(v_i) \geq \tau$. Each hot vertex $v_i \in V_{hot}$ and its incident edges are assigned to P_h (Lines 4-6). Note that the partition size should be kept in the average size in terms of similar number of edges in the interest of balancing load in a coarse way. It is necessary to take a new partition to hold edges instead of assigning the edge to the hot partition when the size of the partition approaches to $|E|/k$. To gather the message sent by the pre-vertex of the hot vertex placed in cold partition, some replicas of the hot vertex, $rep(V_{hot})$, are placed in cold partitions (Line 11). Subsequently, the heuristic rules are as follows: If neither the source nor the destination of edge (v_i, v_j) has been assigned to any partition, then the edge is placed in the partition with the minimum size in P (Lines 16-17), where $A(v_i)$ represents the set of partitions where vertex v_i has already been placed. If just one of v_i or v_j has been assigned, then the edge is placed in partition with the minimum size in $A(v_i)$ or $A(v_j)$ (Lines 18-19). If both of v_i and v_j have already been placed in some partition, and in addition, $A(v_i)$ and $A(v_j)$ have common elements, then the edge is placed in the partition with the minimum size in $A(v_i) \cap A\left(v_j\right)$ (Lines 20-21). The last case is that both v_i and v_j have already been placed in some partition, but $A(v_i)$ and $A(v_j)$ have no common elements; then the edge is placed in partition with the minimum

size in $A(v_i) \cup A\left(v_j\right)$ (Lines 22-23). In this case, a new replica of the vertex is created.

The modification to the structure of a graph over time can be exemplified as the addition of vertices and edges. When a vertex is added to the graph, the information about its first-level neighbors is available. To simplify processing, the added vertex and its neighbors are regarded as a group of newly added edges. For addition of vertices and edges, there can be two scenarios. Firstly, all existing vertices still maintain their characteristics whether they are hot or not. Secondly, some of the cold vertices will turn into hot since they acquire some new edges. For the first case, we assign new edges according to the heuristic rule in Algorithm 1 (Lines 13-26). For the second case, we must transfer those turned vertices from the cold partition to the hot partition, and keep the load balancing. To achieve this, we first take the edges of the turned vertices out of cold partitions, and then assign them again according to the rule of Algorithm 1. At this time, the input of Algorithm 1 is some new data ΔV, ΔE, ΔV_{hot}. $|E|/k$ also has grown.

The above partitioning splits vertices into multiple parts so that several replicas of one vertex may distribute across different partitions. The master should be selected to dominate the remaining replicas named mirrors. We always set the hot vertex in the hot partition as the master. It is also important to ensure that each cold partition has a similar number of masters due to the priority of chunk being confirmed by collecting the delta value of the master vertex, where the details will be provided in the next section.

In order to achieve a better access locality, we manage partitions in the form of chunks, where each partition consists of several chunks and the size of each chunk is set according to the server's CPU last-level cache size. In each chunk, a local key-value state table is created to manage the information of each vertex. The index of a vertex and four other fields make up a table entry. The *vid* is the index of a vertex, and next two fields store the *state value* and the *delta value*, respectively. The fourth field holds the edges assigned to the current chunk, and whether the vertex is a master or a mirror is marked in last field.

There are several motivations for giving special treatment to the hot vertex and for gathering them in same partition. First, hot vertices always collect greater delta values than others in terms of receiving more messages from their connections. Second, scheduling the vertex with higher degree would speed up iteration processing, where the example in Section 2 supports our argument. Third, to coordinate with the hot priority scheduling algorithm, we gather hot vertices to give hot partition a higher priority. Finally, when the system schedules a hot partition and finds that the hot vertex and its neighbours can be accessed locally, it will consume less execution time.

4 Hot Priority Scheduling Algorithm

In asynchronous incremental computation, since the communication between any two vertices is completely asynchronous, it means that the update operations can be executed in any order. In order to reach a fast convergence, the updates of a specific vertex whose priority is greater than others should receive preference to be performed. In this section, we show the details of the hot block priority scheduling algorithm.

4.1 Priority Definition

The traditional scheduling strategy selects vertices from the local table and performs the update operations in a cyclic paradigm. Though this strategy is simple and can avoid starvation, it ignores the current state of vertices and the connection relationship between vertices. However, two factors mentioned above can influence the convergence time of iterative computation to a great extent. We introduce a hot priority scheduling algorithm based on both current state and inherent topological information of vertices. It regards a chunk as the scheduling unit so that each worker can compute the scheduling unit repeatedly for lower cache miss rate. Each worker needs to maintain an auxiliary table to keep the scheduling information of a chunk. Table entries contain *ChunkID*, *Pri*, $Total\_delta$, the scheduling times T, where *ChunkID* marks the chunk, *Pri* is the priority of the chunk, $Total\_delta$ is the total delta value of all vertices contained in chunk, and T is the number of scheduling times.

We first define the priority of the chunk C:

$$Pri(C) = \alpha(T) * \overline{De(C)} + \beta * (1 - \alpha(T)) * Delta(C) \quad (3)$$

The priority of the chunk, denoted as $Pri(C)$, is determined by the interaction of three factors. The first factor, $\overline{De(C)}$ is the average degree of all vertices contained in the chunk. Obviously, this value will be higher in hot partitions. The second factor, $Delta(C)$ is the total delta value of all vertices contained in the chunk collected by the $Total\_delta$ field. This factor indicates the current state of the entire chunk and the influence on the other chunks. Last but not least, the third factor, scheduled times T, will affect the contribution to the priority of two factors above. The function $\alpha(T)$ $(0 < \alpha(T) < 1)$

is a monotonic decreasing function of T. This means that the value of $\alpha(T)$ will decrease as the scheduling times increase. In contrast, $(1-\alpha(T))$ and T maintain a positive correlation. $0 < \beta < \frac{\overline{De_{ave}}}{\overline{Delta_{ave}}}$ is the scaling factor set at the initial stage, where $\overline{De_{ave}}$ and $Delta_{ave}$ are the average values of all chunks' $\overline{De(C)}$ and $Delta(C)$, respectively.

Algorithm 2 Hot Priority Scheduling Algorithm

Input: W_i, C_i
Output:
1: $Total\_delta(C_i) \leftarrow 0$
2: Accumulate the *delta value* of vertices in *PQueue* to $Total\_delta$
3: Write *PQueue* to local table
4: $PriMax \leftarrow Pri(C_i)$
5: $C_{sc} \leftarrow C_i$
6: **for** each chunk C_j held by W_i **do**
7: **if** $Pri(C_j) > PriMax$ **then**
8: $PriMax \leftarrow Pri(C_j)$
9: $C_{sc} \leftarrow C_j$
10: **end if**
11: **end for**
12: $process(W_i, C_{sc})$
13: $T \leftarrow T + 1$

There are three characteristics of this design: (1) The chunk with higher $\overline{De(C)}$, which can easily collect more messages due to its complex connectivity, always has priority to be scheduled. (2) As the final convergence condition of a chunk depends on its total delta value, the chunk with the larger $Total\_delta$ should have priority to be scheduled. (3) With the increase of scheduling times, the centre of priority is shifted to the delta value, which can reflect the actual convergence state of the graph.

4.2 Scheduling Scheme

The scheduling process is given in Algorithm 2, which determines the execution sequence of the chunk. There are two new variables. $PriMax$ represents the current maximum of the priority. Csc stands for the chunk to which $PriMax$ belongs, and it also represents the candidate for the next round of scheduling. In the beginning, suppose that a round of scheduling execution has been completed, and the C_i has finished the scheduling. The algorithm sets the $Total\_delta$ of C_i to 0 (Line 1), where C_i is temporarily convergent. Then it initializes $PriMax$ and Csc by $Pri(C_i)$ and C_i, respectively (Lines 4-5). By traversing the Pri of all chunks, the final Csc is found (Lines 6-11), then the algorithm assigns Csc to W_i to execute (Line 12). Of course, after each round of scheduling, the scheduling time T needs to be increased by 1 (Line 13). There are two reasons for us not to worry that some chunks will be starved for scheduling. First, the chunk that has just been scheduled is in a temporary state of convergence. That means that the subgraph converges locally, and it may still be activated by the other chunks. So the second factor of $Pri(C)$ is possibly 0 (Line 1), and the scheduled chunk will lose the competitive advantage of $Primax$. Furthermore, if an old chunk fails to be $PriMax$ many times, it would hold more $Total\_delta$, which will be the main contribution to be $PriMax$, as the T continues grow.

4.3 Processing of Chunk

To ensure thread safety for a message passing between multiple workers, each worker maintains a thread-safe queue to synchronize communication messages (Lines 2-3) in Algorithm 2. The process of a chunk consists of two stages: Updating operation and forwarding operation as shown in Algorithm 3. In the initialization state before a round of processing, each vertex has stored the received message containing the status change information in the *delta value* field. If a chunk is considered to be unconverged (Line 2), an updating operation is performed on each vertex in the chunk: (1) A user-defined state accumulate function updates the state value (Line 4); (2) the changes of the state value processed by user-specified functions $g\{i, j\}()$ are accumulated by each local out-neighbor (Lines 5-8); (3) the delta value is reset to the initial value for the next accumulation (Line 9). The above three steps are repeated until the chunk converges.

When the chunk is converged, each mirror vertex in the chunk should forward its *state values* as the message to the related master vertex for global updating. Therefore, those temporarily converged chunks may be activated again by receiving the message from these scheduled chunks. The forwarding operation obtains the *chunkID*, which holds the master of mirror vertex v_i (Line 15). If the target chunk C_m is processing at this point, to avoid breaking thread safety, the system temporarily stores data into a thread-safe data queue instead of writing it to the local table directly (Line 17). After the chunk C_m completes the processing, it will update the $Total\_delta$ field of the chunk by accumulating the *delta value* of the vertices in *PQueue*, and writing the data to the local table synchronously (Lines 2-3) in Algorithm 2. If the target chunk C_m is waiting for being scheduled, the system updates the state table of target chunk C_m directly with s_i (Lines 18-21). It is worth noting that we associate the updating of $Total\_delta$ with the updating of the local state ta-

ble. The reason is that in this way we can make $Total\_delta$ reflect the state of the current chunk in a timely manner; it helps the scheduler to select a proper chunk to process. Once the message has been sent, we must set the state value of the mirror vertex to the initialization value for future operations (Line 22).

Algorithm 3 Processing Chunk

Input: C_0
Output:
1: // Updating operation
2: **while** C_0 is not converge **do**
3: **for** each vertex v_i in chunk C_0 **do**
4: $s_i \leftarrow s_i + \Delta s_i$
5: **for** each edge (v_i, v_j) of vertex v_i **do**
6: $\Delta s_j{}' \leftarrow g\{i, j\}(\Delta s_i)$
7: $\Delta s_j \leftarrow \Delta s_j + \Delta s_j{}'$
8: **end for**
9: $\Delta s_i \leftarrow 0$
10: **end for**
11: **end while**
12: // Forwarding operation
13: **for** each vertex v_i in chunk C_0 **do**
14: **if** v_i is mirror **then**
15: $C_m \leftarrow getchunk(v_i)$
16: **if** C_m is running **then**
17: Write s_i as delta value to $PQueue$
18: **else**
19: $C_m.Total\_delta \leftarrow C_m.Total\_delta + s_i$
20: Write s_i as delta value to local table
21: **end if**
22: $v_j \leftarrow inital\ value$
23: **end if**
24: **end for**

5 Evaluation and Analysis

In this section, some experimental evaluations of our method are presented to prove the effectiveness of our framework.

5.1 Preparation

5.1.1 Experimental Environment

The experiments are performed on a server holding a 2-way 8-core 2.60GHz Intel Xeon CPU E5-2670 and 64GB memory, running a Linux operation system with kernel version 2.6.32. The last-level cache size of each CPU is 20480kb. Each core serves each worker as a thread for parallel processing. The size of each chunk is designed according to the server's last-level cache size. So here we set the chunk size to 2MB to ensure that the entire data of the chunk can be loaded into at least the last-level cache when a thread running in a core schedules the target chunk. The proportion R of hot vertices in a graph is 0.5%.

5.1.2 Datasets and Algorithm

The datasets including various sizes of real-world graphs are summarized in Table 1: Orkut, Soc-LiveJournal, Twitter2010 [16], and Friendster from Stanford large network dataset collection. In practice, we employ a popular graph data mining algorithm, PageRank, as benchmark; this is a base rank algorithm for several important algorithms.

Table 1 Graph Datasets Summary

Datasets	Vertices	Edges
Orkut(Ok)	3,072,441	117,185,083
Soc-LiveJournal(Lj)	4,847,571	68,993,773
Twitter2010(Tw)	41,652,230	1,468,365,182
Friendster(Fs)	65,608,366	1,806,067,135

5.1.3 Schemes for Comparison

In practice, our framework has been implemented in the delta-based accumulate iterative computation model by modifying Maiter, which is one of the state-of-the-art systems for supporting asynchronous graph processing. Compared with some alternative frameworks, like Spark [17], GraphLab [18], and Piccolo [19], Maiter [8] has been shown to acheive significantly superior outperform in the iterative computation. We evaluate and analyze the experiment from two aspects: First, we compare our framework with Maiter-RR as well as Maiter-Pri in two ways on the original graph, where Maiter-RR uses the robin-round scheduling and Maiter-pri uses the priority scheduling applied by Maiter. The two ways are the cache miss rate and the scheduling overhead. Next, for an evolving graph, we compare our framework with GraphIn-RR and GraphIn-Pri in three ways on a dynamic graph, where both of them are implemented on GraphIn [7], which is the state-of-the-art system for supporting asynchronous incremental computation and also based on Maiter. These three ways are the overhead of preprocessing, the number of updates, and the total execution time, respectively. All of the above four schemes are using a hash method to divide graphs. For convenience, we name our approach hot block priority scheduling (Hot-Priority).

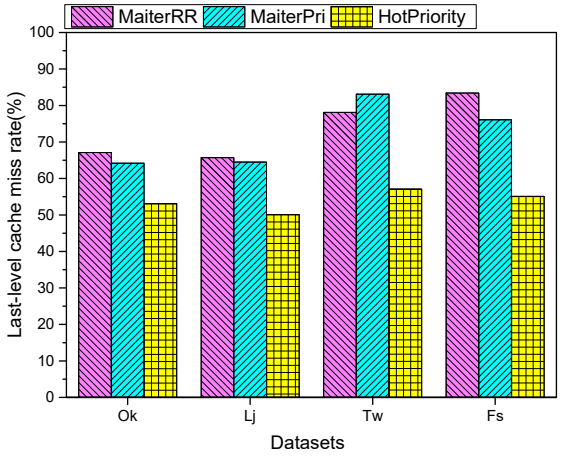


Fig. 3 Last-level cache miss rate

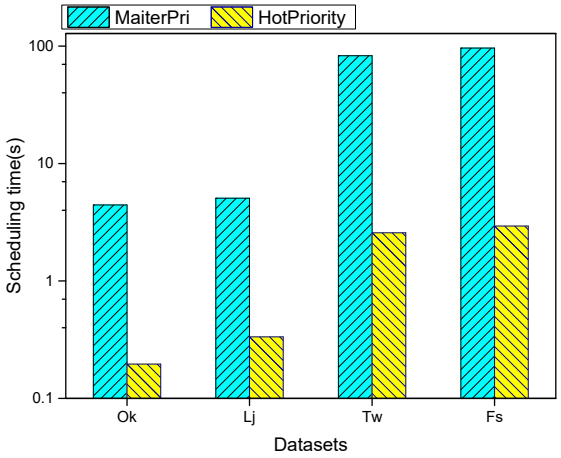


Fig. 4 Scheduling time on different datasets

5.2 Evaluation

In order to simulate that the graph is changing dynaimcally, we first load a large proportion of the total graph, i.e. 95%, as the original graph for processing, then load a small proportion of the remaining graph each time, i.e. 1%. We assume that the graph changes only five times.

5.2.1 Cache Miss Rate

We here compare our method with Maiter on the measure of last-level cache miss rate on different graphs. From Fig. 3, both Maiter-RR and Maiter-Pri run with a higher cache miss rate. Their average cache miss rate is as high as 65%. The reason is that the above two scheduling methods on the basis of hash partitioning can easily lead to irregular access. When a partition stores a large amount of data greater than the cache size, and random access occurs frequently, it results in frequent cache substitution. By comparison, after our preprocessing, hot vertices that are frequently accessed can quickly forward messages locally. Furthermore, vertices in a scheduling chunk can be updated repeatedly until the chunk converges. Therefore, the average cache miss rate of our method is always less than Maiter-RR.

5.2.2 Scheduling Overhead

We evaluate the scheduling overhead of Maiter-Pri and Hot-Priority. We think of a simple polling schedule for its local table, Maiter-RR, does not have scheduling overhead due to the order in which the vertex is executed is fixed. For Maiter-Pri, it needs to maintain a priority queue for each worker to support dynamic priority scheduling, which is the result of the next round of scheduling, the *vids* of vertices. This result is generated by two steps: (1) Randomly sampling and sorting to determine boundary; (2) selecting the vertices whose priority is larger than boundary and loading them into the priority queue to wait for upcoming scheduling. The time complexity of step (1) is fixed, and it can be considered as $O(MlogM)$, where M is the number of sample vertices. Step (2) requires $O(N)$ time to extract the priority queue, where N is the local state table size. However, Hot-Priority just needs $O(n)$ time to select a chunk from several chunks, where n is the number of chunks, so that from the experiment in Fig. 4, its scheduling overhead is far less than Maiter-Pri.

5.2.3 Preprossing Overhead

We evaluate the preprocessing time of Maiter, GraphIn, and Hot-Priority. Fig. 5 shows the overhead of the preprocessing on four datasets when loading the original graph. Fig. 6 shows the result when processing different incremental data on Friendster. The overhead of preprocessing of Maiter and GraphIn is only loading data, since they use the hash partition method, which always loads data directly into the corresponding partition according to the hash function. In contrast to them, Hot-Priority needs to repartition the graph data for generating hot partitions. Theoretically, Hot-Priority will inevitably consume more preprocessing time. Actually, the time consumption of our approach is approximately 1.4 times that of Maiter and GraphIn. However, Hot-Priority will compensate for this additional overhead in the subsequent phase with the obvious benefits such as high locality and short convergence times.

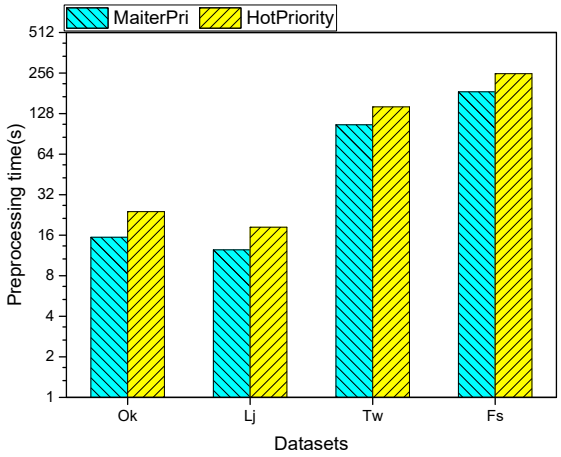


Fig. 5 Preprocessing time on different datasets

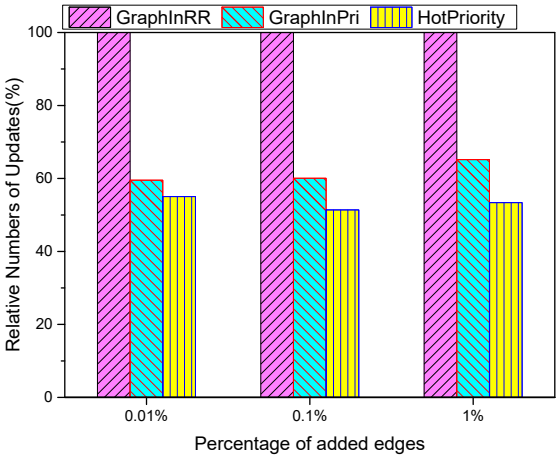


Fig. 7 Relative numbers of updates

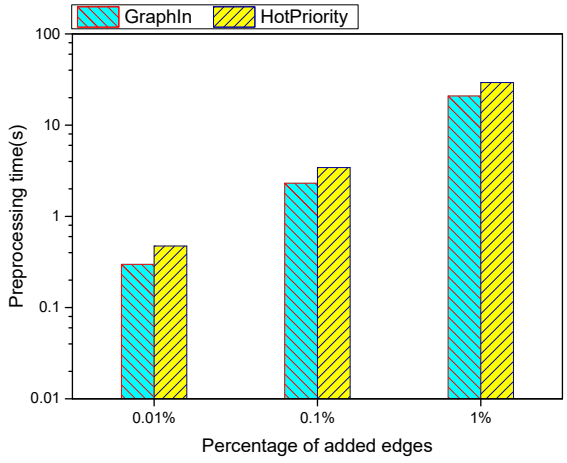


Fig. 6 Preprocessing time on dynamic graph

5.2.4 Number of Updates

We evaluate the number of updates of GraphIn-RR, GraphIn-Pri, and Hot-Priority on FriendSter with different percentages of change. Fig. 7 shows the results relative to that of GraphIn-RR. It shows that Hot-Priority is always better than GraphIn-Pri while the number of updates in GraphIn-RR is the highest and is much higher than Hot-Priority. The randomness of the GraphIn-RR scheduling will cause the local temporarily converged vertices to be reactivated frequently. These vertieces should perform the reprocessing since those slower state propagations from hot vertices arrive. However, Hot-Priority always selects vertices with higher delta value and higher degree to schedule, which will help spread the state change and accelerate convergence. There are two reasons causing the experimental results of Hot-Priority and GraphIn-Pri to be similar. On the one hand, Hot-Priority is based on the chunk-unit. Accordingly, when scheduling a chunk from some cold chunks, some of the vertices in the cold chunk may not be activated without receiving the message, but the entire chunk is scheduled. This will increase the number of updates for some vertices but happen very rarely, since the hot chunk always has a higher scheduling priority. On the other hand, although GraphIn-Pri always extracts the priority queue with the highest priority for scheduling, it ignores the impact of the degree of hot vertices.

5.2.5 Execution Time

Fig. 8 shows the execution time of the considered schemes on Friendster with the different percentages of change. We observe that the Hot-Priority approach reduces execution time by 35% compared with GraphIn-RR, and by about 15% compared with GraphIn-Pri for PageRank. In comparison, Hot-Priority performs better, that is, the updates in Hot-Priority are more effective than that in the robin-round way. Further, Hot-Priority selects more effective updates to execute, which can hit cache frequently and propagate state change effectively by selecting higher-degree or more influential vertices.

6 Related Work

To date, a great deal of work has been devoted to the development of graph processing. On the one hand, some graph processing research [4, 6, 17, 19–21] has been applied to general big data processing platforms. On the other hand, some

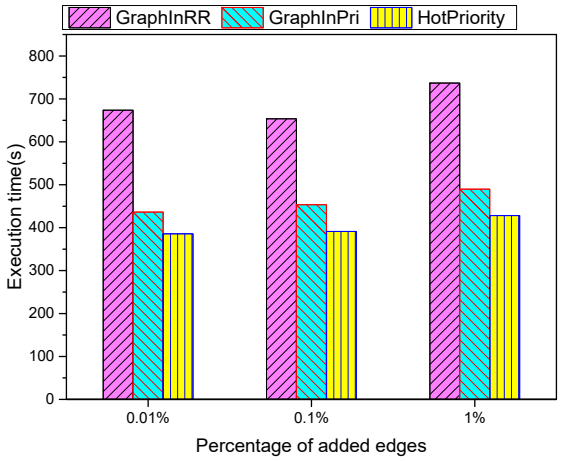


Fig. 8 Execution time on Friendster

specific graph processing systems [8,9,11,13,18,22,23] have also been developed. Some of them work in a bulk synchronous parallel way such as Pregel. In addition, a few proposed systems such as graphLab, PowerGraph and Maiter support asynchronous iteration.

Google's Pregel defines a series of successive supersteps to support graph-based iterative algorithms. Since Pregel has been proposed, many open-source Pregel-like frameworks including giraph++ [24], Pregel+ [25], and GPS [26] have been developed for supporting optimization, as GPS proposes the large adjacency list partitioning to deal with high-degree vertices, and Pregel+ exploits vertex mirroring and message combining to reduce message communications. Furthermore, in the vertex-centric model, GraphChi [27], a large-scale graph processing system in the multicore setting, uses a parallel sliding windows method to improve out-of-core access. In contrast to the vertex-centric model, there are also various programming paradigms. X-Stream [23] has a more sequential access by using an edge-centric graph computation model but still has a unpredictable access pattern. Xie et al. [28] propose a block-oriented iterative graph computation model that updates a block instead of updating a vertex, but it ignores the priority. PathGraph [29] implements the path-centric abstraction for fast iterative graph computation by dividing a graph into paths and reviewing each path as a processing unit, which allows fast loading and locality-optimized computing.

All of the above described systems organize iterative computation into global synchronous supersteps. In contrast to the synchronous model, GraphLab [18] proposes a data pulling programming paradigm and sparse computational dependencies to support asynchronous execution, which can eliminate the waiting time for synchronous barriers. Additionally, in Maiter [8], the graph iterative computation can be executed efficiently and asynchronously. Under the asynchronous delta-based accumulative iterative computation model, each vertex in Maiter propagates the change of the state instead of the state. Since the changes can be accumulated, Maiter allows all communications between vertices to be propagated in a completely asynchronous way for avoiding unnecessary communication and computation. It also proposes a priority scheduling algorithm to accelerate convergence in asynchronous iterative computation. PowerSwitch [30] uses two patterns, synchronous and asynchronous, for distributed parallel-graph computation by adaptively switching two patterns during graph computation.

There are several systems for supporting incremental parallel processing on massive datasets. Incoop [4] saves and reuses states at the granularity of tasks to support incremental processing for one-step applications. DryadInc [10] allows their applications to reuse prior computation results for supporting incremental processing. Rather than focusing on one-pass applications, several recent studies have addressed the incremental processing for graphs. Kineograph [11] reuses prior states by constructing incremental snapshots of the evolving graph. i2MapReduce [6] supports a fine-grained incremental computation. In contrast to synchronous updates, GraphIn [7] applys asynchronous updates to support incremental computation. However, the existing asynchronous updates are still suboptimal due to their suboptimal state propagation and ineffective scheduling order. Our work illustrates an effective improvement on asynchronous incremental computation.

Graph partitioning is an essential step prior to graph processing since subgraphs should be stored in distributed memory instead of entire graphs. PowerGraph [13] combines the foundation of Graphlab with the vertex-cut partition in order to maintain efficient communication and well-balanced load in power-law graphs. Several greedy heuristics [13] are also proposed to decrease communication cost and ensure well-balanced loads on skewed graphs. Fennel [31], a framework for streaming graph partitioning, achieves high-quality partitions by partitioning a graph in a single pass. For changes in graphs, [32] extends the streaming graph partitioning to reassign the vertices to partitions in a restreaming fashion. [33] presents a novel model for the problem of partitioning dynamic graphs.

7 Conclusion and Future Work

In this paper, we propose an effective framework to improve access locality and accelerate state change propagation during iterative computation in asynchronous incremental graph processing. Our framework consists of two parts, the first is a partition method, and the other is a priority scheduling algorithm. Our framework can achieve a high cache hit rate and accelerate convergence with an acceptable preprocessing overhead, and we report these experiments to evaluate our method. The results show that by employing our approach, the current system can efficiently improve the performance of graph processing. Our future work includes two aspects. First, we will work on extending the hot block priority approach to distributed platforms for supporting large-scale graph data. Second, we will explore new partition algorithms to support dynamically generating hot blocks in the deletion case when the graph evolves.

Acknowledgments

This paper is supported by National Natural Science Foundation of China under grant No. 61702202, China Postdoctoral Science Foundation Funded Project under grant No. 2017M610477 and 2017T100555.

References

1. Baluja S, Seth R, Sivakumar D, Jing Y S, Yagnik J, Kumar S, Ravichandran D, Aly M. Video suggestion and discovery for youtube: taking random walks through the view graph. In: Proceedings of the 17th International Conference on World Wide Web, 2008, 895–904
2. Wang P, Xu B W, Wu Y R, Zhou X Y. Link prediction in social networks: the state-of-the-art. SCIENCE CHINA Information Sciences, 2015, 58(1):1–38
3. Shang X Q, Wang Y, Chen B L. Identifying essential proteins based on dynamic protein-protein interaction networks and rna-seq datasets. SCIENCE CHINA Information Sciences, 2016, 59(7):1–11
4. Bhatotia P, Wieder A, Rodrigues R, Acar U A, Pasquin R. Incoop: mapreduce for incremental computations. In: Proceedings of the 2nd ACM Symposium on Cloud Computing, 2011, 1–14
5. Zhang Y F, Gao Q X, Gao L X, Wang C R. Imapreduce: a distributed computing framework for iterative computation. In: Proceedings of the 25th IEEE International Symposium on Parallel and Distributed Processing, 2011, 1112–1121
6. Zhang Y F, Chen S M, Wang Q, Yu G. I2mapreduce: incremental mapreduce for mining evolving big data. In: Proceedings of the 32nd IEEE International Conference on Data Engineering, 2016, 1482–1483
7. Yin J T, Gao L X. Asynchronous distributed incremental computation on evolving graphs. In: Proceedings of the 2016 Machine Learning and Knowledge Discovery in Databases, 2016, 722–738.
8. Zhang Y F, Gao Q X, Gao L X, Wang C R. Maiter: an asynchronous graph processing framework for delta-based accumulative iterative computation. IEEE Transactions on Parallel & Distributed Systems, 2014, 25(8):2091–2100
9. Mihaylov S R, Ives Z G, Guha S. Rex: recursive, delta-based data-centric computation. Proceedings of the VLDB Endowment, 2012, 5(11):1280–1291
10. Popa L, Budiu M, Yu Y, Isard M. Dryadinc: reusing work in large-scale computations. In: Proceedings of the 2009 Conference on Hot Topics in Cloud Computing, 2009, 1–5
11. Cheng R, Hong J, Kyrola A, Miao Y S, Weng X T, Wu M, Yang F, Zhou L D, Zhao F, Chen E H. Kineograph: taking the pulse of a fast-changing and connected world. In: Proceedings of the 7th ACM European Conference on Computer Systems, 2012, 85–98
12. Murray D G, Mcsherry F, Isaacs R, Isard M, Barham P. Naiad: a timely dataflow system. In: Proceedings of the 24th ACM SIGOPS Symposium on Operating Systems Principles, 2013, 439–455
13. Gonzalez J E, Low Y C, Gu H J, Bickson D, Guestrin C. Powergraph: distributed graph-parallel computation on natural graphs. In Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation, 2012, 17–30
14. Verma S, Leslie L M, Shin Y, Gupta I. An experimental comparison of partitioning strategies in distributed graph processing. Proceedings of the Vldb Endowment, 2017, 10(5):493–504
15. Karypis G, Kumar V. Multilevel graph partitioning schemes. In: Proceedings of the 1995 International Conference on Parallel Processing, 1995, 113–122
16. Kwak H, Lee C, Park H, Moon S. What is twitter, a social network or a news media? In: Proceedings of the 19th International Conference on World wide web, 2010, 591–600
17. Zaharia M, Chowdhury M, Franklin M J, Shenker S, Stoica I. Spark: cluster computing with working sets. In: Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing, 2010, 1–10
18. Low Y C, Gonzalez J E, Kyrola A, Bickson D, Guestrin C E, Hellerstein J. Graphlab: a new framework for parallel machine learning. In: Proceedings of the 26th Conference on Uncertainty in Artificial Intelligence, 2010, 1–10
19. Power R, Li J Y. Piccolo: building fast, distributed programs with partitioned tables. In: Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation, 2010, 1–14
20. Bu Y Y, Howe B, Balazinska M, Ernst M D. Haloop: efficient iterative data processing on large clusters. Proceedings of the Vldb Endowment, 2010, 3(1):285–296
21. Ekanayake J, Li H, Zhang B J, Gunarathne T, Bae S H, Qiu J, Fox G. Twister: a runtime for iterative mapreduce. In: Proceedings of the

19th ACM International Symposium on High Performance Distributed Computing, 2010, 810–818
22. Malewicz G, Austern M H, Bik A J, Dehnert J C, Horn I, Leiser N, Czajkowski G. Pregel: a system for large-scale graph processing. In: Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data, 2010, 135–146
23. Roy A, Mihailovic I, Zwaenepoel W. X-stream: edge-centric graph processing using streaming partitions. In: Proceedings of the 24th ACM Symposium on Operating Systems Principles, 2013, 472–488
24. Tian Y Y, Balmin A, Corsten S A, Tatikonda S, McPherson J. From "think like a vertex" to "think like a graph". Proceedings of the Vldb Endowment, 2013, 7(3):193–204
25. Yan D, Cheng J, Lu Y, Ng W. Effective techniques for message reduction and load balancing in distributed graph computation. In: Proceedings of the 24th International Conference on World Wide Web, 2015, 1307–1317
26. Salihoglu S, Widom J. Gps: a graph processing system. In: Proceedings of the 2013 Conference on Scientific and Statistical Database Management, 2013, 1–12
27. Kyrola A, Blelloch G, Guestrin C. Graphchi: large-scale graph computation on just a pc. In: Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation, 2012, 31–46
28. Xie W L, Wang G Z, Bindel D, Demers A, Gehrke J. Fast iterative graph computation with block updates. Proceedings of the Vldb Endowment, 2013, 6(14):2014–2025
29. Yuan P P, Zhang W Y, Xie C F, Jin H, Liu L, Lee K. Fast iterative graph computation: a path centric approach. In: Proceedings of the 2014 International Conference for High Performance Computing, Networking, Storage and Analysis, 2015, 401–412
30. Xie C N, Chen R, Guan H B, Zang B Y, Chen H B. Sync or async: time to fuse for distributed graph-parallel computation. In: Proceedings of the 20th ACM Sigplan Symposium on Principles and Practice of Parallel Programming, 2015, 194–204
31. Tsourakakis C, Gkantsidis C, Radunovic B, Vojnovic M. Fennel: streaming graph partitioning for massive scale graphs. In: Proceedings of the 7th ACM International Conference on Web Search and Data Mining, 2014, 333–342
32. Nishimura J, Ugander J. Restreaming graph partitioning: simple versatile algorithms for advanced balancing. In: Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, 2013, 1106–1114
33. Abdolrashidi A, Ramaswamy L. Continual and cost-effective partitioning of dynamic graphs for optimizing big graph processing systems. In: Proceedings of the 2016 IEEE International Congress on Big Data, 2016, 18–25

Xinqiao Lv received his Ph.D. degree in computer science and engineering from Huazhong University of Science and Technology (HUST). He is now an associate professor in school of Computer Science and Engineering at HUST. His main research interests include big data processing, cloud computing and distributed systems.

Wei Xiao received a BE degree in computer science from Huazhong University of Science and Technology (HUST) in 2015. He is currently a master in school of computer science at HUST. His research interests include graph processing and cloud computing.

Yu Zhang received a Ph.D. degree in computer science from Huazhong University of Science and Technology (HUST) in 2016. He is now a postdoctor in school of computer science at HUST. His research interests include big data processing, system software and architecture. His current topic mainly focuses on application-driven big data processing and optimizations.

Xiaofei Liao received a Ph.D. degree in computer science and engineering from Huazhong University of Science and Technology (HUST), China, in 2005. He is now a Professor in school of Computer Science and Engineering at HUST. His research interests are in the areas of system virtualization, system software, and Cloud computing.

Hai Jin is a Cheung Kung Scholars Chair Professor of computer science and engineering at Huazhong University of Science and Technology (HUST) in China. Jin received his PhD in computer engineering from HUST in 1994. In 1996, he was awarded a German Academic Exchange Service fellowship

to visit the Technical University of Chemnitz in Germany. Jin worked at The University of Hong Kong between 1998 and 2000, and as a visiting scholar at the University of Southern California between 1999 and 2000. He was awarded Excellent Youth Award from the National Science Foundation of China in 2001. Jin 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. Jin is a Fellow of CCF, senior member of the IEEE and a member of the ACM. He has co-authored 22 books and published over 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.

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