Maiter: An Asynchronous Graph Processing Framework for Delta-based Accumulative Iterative Computation

Yanfeng Zhang, Qixin Gao, Lixin Gao, *Fellow, IEEE*, Cuirong Wang

Abstract—Myriad of graph-based algorithms in machine learning and data mining require parsing relational data iteratively. These algorithms are implemented in a large-scale distributed environment in order to scale to massive data sets. To accelerate these large-scale graph-based iterative computations, we propose *delta-based accumulative iterative computation* (DAIC). Different from traditional iterative computations, which iteratively update the result based on the result from the previous iteration, DAIC updates the result by accumulating the “changes” between iterations. By DAIC, we can process only the “changes” to avoid the negligible updates. Furthermore, we can perform DAIC asynchronously to bypass the high-cost synchronous barriers in heterogeneous distributed environments. Based on the DAIC model, we design and implement an asynchronous graph processing framework, Maiter. We evaluate Maiter on local cluster as well as on Amazon EC2 Cloud. The results show that Maiter achieves as much as 60x speedup over Hadoop and outperforms other state-of-the-art frameworks.

Index Terms—delta-based accumulative iterative computation, asynchronous iteration, Maiter, distributed framework.

1 Introduction

The advances in data acquisition, storage, and networking technology have created huge collections of high-volume, high-dimensional relational data. Huge amounts of the relational data, such as Facebook user activities, Flickr photos, Web pages, and Amazon co-purchase records, have been collected. Making sense of these relational data is critical for companies and organizations to make better business decisions and even bring convenience to our daily life. Recent advances in data mining, machine learning, and data analytics have led to a flurry of graph analytic techniques that typically require an iterative refinement process [1], [2], [3], [4]. However, the massive amount of data involved and potentially numerous iterations required make performing data analytics in a timely manner challenging. To address this challenge, MapReduce [5], [6], Pregel [7], and a series of distributed frameworks [8], [9], [10], [7], [11] have been proposed to perform large-scale graph processing in a cloud environment.

Many of the proposed frameworks exploit vertex-centric programming model. Basically, the graph algorithm is described from a single vertex's perspective and then applied to each vertex for a loosely coupled execution. Given the input graph $G(V,E)$, each vertex $j \in V$ maintains a vertex state v_j, which is updated iteratively based on its in-neighbors' state, according to the update function f:

$$v_j^k = f(v_1^{k-1}, v_2^{k-1}, \ldots, v_n^{k-1}), \tag{1}$$

where v_j^k represents vertex j's state after the k iterations, and $v_1, v_2, \ldots, v_n$ are the states of vertex j's in-neighbors. The iterative process continues until the states of all vertices become stable, when the iterative algorithm converges.

Based on the vertex-centric model, most of the proposed frameworks leverage **synchronous iteration**. That is, the vertices perform the update in lock steps. At step k, vertex j first collects v_i^{k-1} from all its in-neighbors, followed by performing the update function f to obtain v_j^k based on these v_i^{k-1}. The synchronous iteration requires that all the update operations in the $(k-1)^{\text{th}}$ iteration have to complete before any of the update operations in the k^{th} iteration start. Clearly, this synchronization is required in each step. These synchronizations might degrade performance, especially in heterogeneous distributed environments.

To avoid the high-cost synchronization barriers, **asynchronous iteration** was proposed [12]. Performing updates asynchronously means that vertex j performs the update at any time based on the most recent states of its in-neighbors. Asynchronous iteration has been studied in [12], [13], [14]. Bypassing the synchronization barriers and exploiting the most recent state intuitively lead to more efficient iteration. However, asynchronous iteration might require more communications and perform useless computations. An activated vertex pulls all its in-neighbors' values, but not all of them have been updated, or even

- *Y. Zhang is with Northeastern University, Shenyang, P.R. China. E-mail: zhangyf@cc.neu.edu.cn*
- *Q. Gao is with Northeastern University, Qinhuangdao, P.R. China.*
- *L. Gao is with University of Massachusetts Amherst, U.S.A.*
- *C. Wang is with Northeastern University, Qinhuangdao, P.R. China.*

worse none of them is updated. In that case, asynchronous iteration performs a useless computation, which impacts efficiency. Furthermore, some asynchronous iteration cannot guarantee to converge to the same fixed point as synchronous iteration, which leads to uncertainty. More background introduction of the iterative graph processing can be found in Section 1 of the supplementary file.

In this paper, we propose **DAIC**, *delta-based accumulative iterative computation*. In traditional iterative computation, each vertex state is updated based on its in-neighbors' previous iteration states. While in DAIC, each vertex propagates only the "change" of the state, which can avoid useless updates. The key benefit of only propagating the "change" is that, the "changes" can be accumulated monotonically and the iterative computation can be performed asynchronously. In addition, since the amount of "change" implicates the importance of an update, we can utilize more efficient priority scheduling for the asynchronous updates. Therefore, DAIC can be executed efficiently and asynchronously. Moreover, DAIC can guarantee to converge to the **same** fixed point. Given a graph iterative algorithm, we provide the sufficient conditions of rewriting it as a DAIC algorithm and list the guidelines on writing DAIC algorithms. We also show that a large number of well-known algorithms satisfy these conditions and illustrate their DAIC forms.

Based on the DAIC model, we design a distributed framework, **Maiter**. Maiter relies on Message Passing Interface (MPI) for communication and provides intuitive API for users to implement a DAIC algorithm. We systematically evaluate Maiter on local cluster as well as on Amazon EC2 Cloud [15]. Our results are presented in the context of four popular applications. The results show that Maiter can accelerate the iterative computations significantly. For example, Maiter achieves as much as 60x speedup over Hadoop for the well-known PageRank algorithm.

The rest of the paper is organized as follows. Section 2 presents DAIC, followed by introducing how to write DAIC algorithms in Section 3. In Section 4, we describe Maiter. The experimental results are shown in Section 5. We outline the related work in Section 6 and conclude the paper in Section 7.

2 Delta-Based Accumulative Iterative Computation (DAIC)

In this section, we present delta-based accumulative iterative computation, DAIC. By DAIC, the graph iterative algorithms can be executed asynchronously and efficiently. We first introduce DAIC and point out the sufficient conditions of performing DAIC. Then, we propose DAIC's asynchronous execution model. We further prove its convergence and analyze its effectiveness. Under the asynchronous model, we also propose several scheduling policies to schedule the asynchronous updates.

2.1 DAIC Introduction

Based on the idea introduced in Section 1, we give the following 2-step update function of DAIC:

$$\begin{cases} v_j^k = v_j^{k-1} \oplus \Delta v_j^k, \\ \Delta v_j^{k+1} = \sum_{i=1}^{n} \oplus g_{\{i,j\}}(\Delta v_i^k). \end{cases} \tag{2}$$

$k = 1, 2, \dots$ is the iteration number. v_j^k is the state of vertex j after k iterations. Δv_j^k denotes the change from v_j^{k-1} to v_j^k in the '$\oplus$' operation manner, where '$\oplus$' is an abstract operator. $\sum_{i=1}^{n} \oplus x_i = x_1 \oplus x_2 \oplus \ldots \oplus x_n$ represents the accumulation of the "changes", where the accumulation is in the '$\oplus$' operation manner.

The first update function says that a vertex state v_j^k is updated from v_j^{k-1} by accumulating the change Δv_j^k. The second update function says that the change Δv_j^{k+1}, which will be used in the next iteration, is the accumulation of the received values $g_{\{i,j\}}(\Delta v_i^k)$ from j's various in-neighbors i. The propagated value from i to j, $g_{\{i,j\}}(\Delta v_i^k))$, is generated in terms of vertex i's state change Δv_i^k. Note that, all the accumulative operation is in the '$\oplus$' operation manner.

However, not all iterative computation can be converted to the DAIC form. To write a DAIC, the update function should satisfy the following sufficient conditions.

The **first condition** is that,

- update function $v_j^k = f(v_1^{k-1}, v_2^{k-1}, \dots, v_n^{k-1})$ can be written in the form:

$$v_j^k = g_{\{1,j\}}(v_1^{k-1}) \oplus g_{\{2,j\}}(v_2^{k-1}) \oplus \ldots \oplus g_{\{n,j\}}(v_n^{k-1}) \oplus c_j \tag{3}$$

where $g_{\{i,j\}}(v_i)$ is a function applied on vertex j's in-neighbor i, which denotes the value passed from vertex i to vertex j. In other words, vertex i passes value $g_{\{i,j\}}(v_i)$ (instead of v_i) to vertex j. On vertex j, these $g_{\{i,j\}}(v_i)$ from various vertices i and c_j are aggregated (by '$\oplus$' operation) to update v_j.

For example, the well-known PageRank algorithm satisfies this condition. It iteratively updates the PageRank scores of all pages. In each iteration, the ranking score of page j, R_j, is updated as follows:

$$R_j^k = d \cdot \sum_{\{i|(i \to j) \in E\}} \frac{R_i^{k-1}}{|N(i)|} + (1-d),$$

where d is a damping factor, $|N(i)|$ is the number of outbound links of page i, $(i \to j)$ is a link from page i to page j, and E is the set of directed links. The update function of PageRank is in the form of Equation (3), where $c_j = 1 - d$, '$\oplus$' is '$+$', and for any page i that has a link to page j, $g_{\{i,j\}}(v_i^{k-1}) = d \cdot \frac{v_i^{k-1}}{|N(i)|}$.

Next, since Δv_j^k is defined to denote the "change" from v_j^{k-1} to v_j^k in the '$\oplus$' operation manner. That is,

$$v_j^k = v_j^{k-1} \oplus \Delta v_j^k, \tag{4}$$

In order to derive Δv_j^k we pose the **second condition**:

- function $g_{\{i,j\}}(x)$ should have the *distributive property* over '$\oplus$', *i.e.*, $g_{\{i,j\}}(x \oplus y) = g_{\{i,j\}}(x) \oplus g_{\{i,j\}}(y)$.

By replacing v_i^{k-1} in Equation (3) with $v_i^{k-2} \oplus \Delta v_i^{k-1}$, we have

$$\begin{aligned} v_j^k = & g_{\{1,j\}}(v_1^{k-2}) \oplus g_{\{1,j\}}(\Delta v_1^{k-1}) \oplus \ldots \oplus \\ & g_{\{n,j\}}(v_n^{k-2}) \oplus g_{\{n,j\}}(\Delta v_n^{k-1}) \oplus c_j. \end{aligned} \tag{5}$$

Further, let us pose the **third condition**:

- operator '$\oplus$' should have the *commutative property*, *i.e.*, $x \oplus y = y \oplus x$;
- operator '$\oplus$' should have the *associative property*, *i.e.*, $(x \oplus y) \oplus z = x \oplus (y \oplus z)$;

Then we can combine these $g_{\{i,j\}}(v_i^{k-2})$, $i = 1, 2, \ldots, n$, and c_j in Equation (5) to obtain v_j^{k-1}. Considering Equation (4), the combination of the remaining $g_{\{i,j\}}(\Delta v_i^{k-1})$, $i = 1, 2, \ldots, n$ in Equation (5), which is $\sum_{i=1}^{n} \oplus g_{\{i,j\}}(\Delta v_i^{k-1})$, will result in Δv_i^k. Then, we have the 2-step DAIC as shown in (2).

To initialize a DAIC, we should set the start values of v_j^0 and Δv_j^1. v_j^0 and Δv_j^1 can be initialized to be any value, but the initialization should satisfy $v_j^0 \oplus \Delta v_j^1 = v_j^1 = g_{\{1,j\}}(v_1^0) \oplus g_{\{2,j\}}(v_2^0) \oplus \ldots \oplus g_{\{n,j\}}(v_n^0) \oplus c_j$, which is the **fourth condition**.

The PageRank's update function as shown in Equation (4) satisfies all the conditions. $g_{\{i,j\}}(v_i^{k-1}) = d \cdot \frac{v_i^{k-1}}{|N(i)|}$ satisfies the second condition. '$\oplus$' is '$+$', which satisfies the third condition. In order to satisfy the fourth condition, v_j^0 can be initialized to 0, and Δv_j^1 can be initialized to $1-d$. Besides PageRank, we have found a broad class of DAIC algorithms, which are described in Section 3 of the supplementary file.

To sum up, DAIC can be described as follows. Vertex j first updates v_j^k by accumulating Δv_j^k (by '$\oplus$' operation) and then updates Δv_j^{k+1} with $\sum_{i=1}^{n} \oplus g_{\{i,j\}}(\Delta v_i^k)$. We refer to Δv_j as the *delta value* of vertex j and $g_{\{i,j\}}(\Delta v_i^k)$ as the *delta message* sent from i to j. $\sum_{i=1}^{n} \oplus g_{\{i,j\}}(\Delta v_i^k)$ is the accumulation of the received delta messages on vertex j since the k^{th} update. Then, the delta value Δv_j^{k+1} will be used for the $(k+1)^{\text{th}}$ update. Apparently, this still requires all vertices to start the update synchronously. That is, Δv_j^{k+1} has to accumulate all the delta messages $g_{\{i,j\}}(\Delta v_i^k)$ sent from j's in-neighbors, at which time it is ready to be used in the $(k+1)^{\text{th}}$ iteration. Therefore, we refer to the 2-step iterative computation in (2) as **synchronous DAIC**.

2.2 Asynchronous DAIC

DAIC can be performed asynchronously. That is, a vertex can start update at any time based on whatever it has already received. We can describe **asynchronous DAIC** as follows, each vertex j performs:

$$\begin{aligned} &\texttt{receive:} \begin{cases} \texttt{Whenever receiving } m_j, \\ \quad \Delta\check{v}_j \leftarrow \Delta\check{v}_j \oplus m_j. \end{cases} \\ &\texttt{update:} \begin{cases} \check{v}_j \leftarrow \check{v}_j \oplus \Delta\check{v}_j; \\ \texttt{For any } h\texttt{, if } g_{\{j,h\}}(\Delta\check{v}_j) \neq \mathbf{0}, \\ \quad \texttt{send value } g_{\{j,h\}}(\Delta\check{v}_j) \texttt{ to } h; \\ \Delta\check{v}_j \leftarrow \mathbf{0}, \end{cases} \end{aligned} \tag{6}$$

where m_j is the received delta message $g_{\{i,j\}}(\Delta\check{v}_i)$ sent from any in-neighbor i. The *receive* operation accumulates the received delta message m_j to $\Delta\check{v}_j$. $\Delta\check{v}_j$ accumulates the received delta messages between two consecutive update operations. The *update* operation updates $\check{v}_j$ by accumulating $\Delta\check{v}_j$, sends the delta message $g_{\{j,h\}}(\Delta\check{v}_j)$ to any of j's out-neighbors h, and resets $\Delta\check{v}_j$ to $\mathbf{0}$. Here, operator '$\oplus$' should have the *identity property* of abstract value $\mathbf{0}$, *i.e.*, $x \oplus \mathbf{0} = x$, so that resetting $\Delta\check{v}_j$ to $\mathbf{0}$ guarantees that the received value is cleared. Additionally, to avoid useless communication, it is necessary to check that the sent delta message $g_{\{j,h\}}(\Delta\check{v}_j) \neq \mathbf{0}$ before sending.

For example, in PageRank, each page j has a buffer ΔR_j to accumulate the received delta PageRank scores. When page j performs an update, R_j is updated by accumulating ΔR_j. Then, the delta message $d\frac{\Delta R_j}{|N(j)|}$ is sent to j's linked pages, and ΔR_j is reset to 0.

In asynchronous DAIC, the two operations on a vertex, receive and update, are completely independent from those on other vertices. Any vertex is allowed to perform the operations at any time. There is no lock step to synchronize any operation between vertices.

2.3 Convergence

To study the convergence property, we first give the following definition of the convergence of asynchronous DAIC.

Definition 1: Asynchronous DAIC as shown in (6) converges as long as that after each vertex has performed the receive and update operations an infinite number of times, $\check{v}_j^\infty$ converges to a fixed value $\check{v}_j^*$. Then, we have the following theorem to guarantee that asynchronous DAIC will converge to the same fixed point as synchronous DAIC. Further, since synchronous DAIC is derived from the traditional form of iterative computation, *i.e.*, Equation (1), the asynchronous DAIC will converge to the same fixed point as traditional iterative computation.

Theorem 1: If v_j in (1) converges, $\check{v}_j$ in (6) converges. Further, they converge to the same value, *i.e.*, $v_j^\infty = \check{v}_j^\infty = \check{v}_j^*$.

The formal proof of Theorem 1 is provided in Section 2 of the supplementary file. We explain the intuition behind Theorem 1 as follows. Consider the process of DAIC as information propagation in a graph. Vertex i with an initial value Δv_i^1 propagates delta message $g_{\{i,j\}}(\Delta v_i^1)$ to its out-neighbor j, where

$g_{\{i,j\}}(\Delta v_i^1)$ is accumulated to v_j and a new delta message $g_{\{j,h\}}(g_{\{i,j\}}(\Delta v_i^1))$ is produced and propagated to any of j's out-neighbors h. By synchronous DAIC, the delta messages propagated from all vertices should be received by all their neighbors before starting the next round propagation. That is, the delta messages originated from a vertex are propagated strictly hop by hop. In contrast, by asynchronous DAIC, whenever some delta messages arrive, a vertex accumulates them to $\check{v}_j$ and propagates the newly produced delta messages to its neighbors. No matter synchronously or asynchronously, the spread delta messages are never lost, and the delta messages originated from each vertex will be eventually spread along all paths. For a destination node, it will eventually collect the delta messages originated from all vertices along various propagating paths. All these delta messages are eventually received and contributed to any v_j. Therefore, synchronous DAIC and asynchronous DAIC will converge to the same result.

2.4 Effectiveness

As illustrated above, v_j and $\check{v}_j$ both converge to the same fixed point. By accumulating Δv_j (or $\Delta \check{v}_j$), v_j (or $\check{v}_j$) either monotonically increases or monotonically decreases to a fixed value $v_j^* = v_j^\infty = \check{v}_j^\infty$. In this section, we show that $\check{v}_j$ converges faster than v_j.

To simplify the analysis, we first assume that 1) only one update occurs at any time point; 2) the transmission delay can be ignored, *i.e.*, the delta message sent from vertex i, $g_{\{i,j\}}(\Delta v_i)$ (or $g_{\{i,j\}}(\Delta \check{v}_i)$), is directly accumulated to Δv_j (or $\Delta \check{v}_j$).

The workload can be seen as the number of performed updates. Let *update sequence* represent the update order of the vertices. By synchronous DAIC, all the vertices have to perform the update once and only once before starting the next round of updates. Hence, the update sequence is composed of a series of *subsequences*. The length of each subsequence is $|V|$, *i.e.*, the number of vertices. Each vertex occurs in a subsequence once and only once. We call this particular update sequence as *synchronous update sequence*. While in asynchronous DAIC, the update sequence can follow any update order. For comparison, we will use the same synchronous update sequence for asynchronous DAIC.

By DAIC, no matter synchronously and asynchronously, the propagated delta messages of an update on vertex i in subsequence k, *i.e.*, $g_{\{i,j\}}(\Delta v_i^k)$ (or $g_{\{i,j\}}(\Delta \check{v}_i)$), are directly accumulated to Δv_j^{k+1} (or $\Delta \check{v}_j$), $j = 1, 2, \ldots, n$. By synchronous DAIC, Δv_j^{k+1} cannot be accumulated to v_j until the update of vertex j in subsequence $k+1$. In contrast, by asynchronous DAIC, $\Delta \check{v}_j$ is accumulated to $\check{v}_j$ immediately whenever vertex j is updated after the update of vertex i in subsequence k. The update of vertex j might occur in subsequence k or in subsequence $k+1$. If the update of vertex j occurs in subsequence k, $\check{v}_j$ will accumulate more delta messages than v_j after k subsequences, which means that $\check{v}_j$ is closer to v_j^* than v_j. Otherwise, $\check{v}_j = v_j$. Therefore, we have Theorem 2. The formal proof of Theorem 2 is provided in Section 2 of the supplementary file.

Theorem 2: Based on the same update sequence, after k subsequences, we have $\check{v}_j$ by asynchronous DAIC and v_j by synchronous DAIC. $\check{v}_j$ is closer to the fixed point v_j^* than v_j is, *i.e.*, $|v_j^* - \check{v}_j| \leq |v_j^* - v_j|$.

2.5 Scheduling Policies

By asynchronous DAIC, we should control the update order of the vertices, *i.e.*, specifying the scheduling policies. In reality, a subset of vertices are assigned to a processor, and multiple processors are running in parallel. The processor can perform the update for the assigned vertices in a round-robin manner, which is referred to as *round-robin scheduling*. Moreover, it is possible to schedule the update of these local vertices dynamically by identifying their importance, which is referred to as *priority scheduling*. In [16], we have found that selectively processing a subset of the vertices has the potential of accelerating iterative computation. Some of the vertices can play an important decisive role in determining the final converged outcome. Giving an update execution priority to these vertices can accelerate the convergence.

In order to show the progress of the iterative computation, we quantify the iteration progress with L_1 norm of $\check{v}$, *i.e.*, $||\check{v}||_1 = \sum_i \check{v}_i$. Asynchronous DAIC either monotonically increases or monotonically decreases $||\check{v}||_1$ to a fixed point $||v^*||_1$. According to (6), an update of vertex j, *i.e.*, $\check{v}_j = \check{v}_j \oplus \Delta \check{v}_j$, either increases $||\check{v}||_1$ by $(\check{v}_j \oplus \Delta \check{v}_j - \check{v}_j)$ or decreases $||\check{v}||_1$ by $(\check{v}_j - \check{v}_j \oplus \Delta \check{v}_j)$. Therefore, by priority scheduling, vertex $j = arg \max_j |\check{v}_j \oplus \Delta \check{v}_j - \check{v}_j|$ is scheduled first. In other words, The bigger $|\check{v}_j \oplus \Delta \check{v}_j - \check{v}_j|$ is, the higher update priority vertex j has. For example, in PageRank, we set each page j's scheduling priority based on $|R_j + \Delta R_j - R_j| = \Delta R_j$. Then, we will schedule page j with the largest ΔR_j first. To sum up, by priority scheduling, the vertex $j = arg \max_j |\check{v}_j \oplus \Delta \check{v}_j - \check{v}_j|$ is scheduled for update first.

Theorem 3 guarantees the convergence of asynchronous DAIC under the priority scheduling. The proof of Theorem 3 can be found in Section 2 of the supplementary file. Furthermore, according to the analysis presented above, we have Theorem 4 to support the effectiveness of priority scheduling.

Theorem 3: By asynchronous priority scheduling, $\check{v}'_j$ converges to the same fixed point v_j^* as v_j by synchronous iteration converges to, *i.e.*, $\check{v}'^\infty_j = v_j^\infty = v_j^*$.

Theorem 4: Based on asynchronous DAIC, after the same number of updates, we have $\check{v}'_j$ by priority scheduling and $\check{v}_j$ by round-robin scheduling. $\check{v}'_j$ is closer to the fixed point v_j^* than $\check{v}_j$ is, *i.e.*, $|v_j^* - \check{v}'_j| \leq |v_j^* - \check{v}_j|$.

3 Writing DAIC algorithms

In this section, we provide the guidelines of writing DAIC algorithms. Given an iterative algorithm, the following steps are recommended for converting it to a DAIC algorithm.

- **Step1: Vertex-Centric Check**. Check whether the update function is applied on each vertex, and write the vertex-centric update function f. If not, try to rewrite the update function.
- **Step2: Formation Check**. Check whether f is in the form of Equation (3)? If yes, identify the sender-based function $g_{\{i,j\}}(v_i)$ applied on each sender vertex i, the abstract operator '$\oplus$' for accumulating the received delta messages on receiver vertex j.
- **Step3: Properties Check**. Check whether $g_{\{i,j\}}(v_i)$ has the distributive property and whether operator '$\oplus$' has the commutative property and the associative property?
- **Step4: Initialization**. According to (2), initialize v_j^0 and Δv_j^1 to satisfy $v_j^1 = v_j^0 \oplus \Delta v_j^1$, and write the iterative computation in the 2-step DAIC form.
- **Step5: Priority Assignment** (Optional). Specify the scheduling priority of each vertex j as $|\check{v}_j \oplus \Delta\check{v}_j - \check{v}_j|$ for scheduling the asynchronous updates.

Following the guidelines, we have found a broad class of DAIC algorithms, including single source shortest path (SSSP), PageRank, linear equation solvers, Adsorption, SimRank, etc. Table 1 shows a list of such algorithms. Each of their update functions is represented with a tuple ($g_{\{i,j\}}(x)$, $\oplus$, v_j^0, Δv_j^1). In Table 1, matrix A represents the graph adjacency information. If there is an edge from vertex i to vertex j, $A_{i,j}$ represents the edge weight from i to j, or else $A_{i,j} = 0$. More details of these DAIC algorithms are described in Section 3 of the supplementary file.

4 Maiter

To support implementing a DAIC algorithm in a large-scale distributed manner and in a highly efficient asynchronous manner, we propose an asynchronous distributed framework, Maiter. Users only need to follow the guidelines to specify the function $g_{\{i,j\}}(v_i)$, the abstract operator '$\oplus$', and the initial values v_j^0 and Δv_j^1 through Maiter API (Maiter API is described in Section 4 of the supplementary file). The framework will automatically deploy these DAIC algorithms in the distributed environment and perform asynchronous iteration efficiently.

Maiter is implemented by modifying Piccolo [11], and Maiter's source code is available online [17]. It relies on message passing for communication between vertices. In Maiter, there is a master and multiple workers. The master coordinates the workers and monitors the status of workers. The workers run in parallel and communicate with each other through MPI. Each worker performs the update for a subset of vertices. In the following, we introduce Maiter's key functionalities.

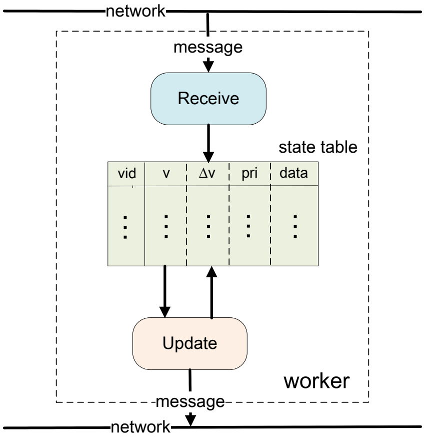


Fig. 1. Worker overview.

Data Partition. Each worker loads a subset of vertices in memory for processing. Each vertex is indexed by a global unique *vid*. The assignment of a vertex to a worker depends solely on the vid. A vertex with vid j is assigned to worker $h(j)$, where $h()$ is a hash function applied on the vid. Besides, preprocessing for smart graph partition can be useful. For example, one can use a lightweight clustering algorithm to preprocess the input graph, assigning the strongly connected vertices to the same worker, which can reduce communication.

Local State Table. The vertices in a worker are maintained in a local in-memory key-value store, *state table*. Each state table entry corresponds to a vertex indexed by its vid. As depicted in Fig. 1, each table entry contains five fields. The 1st field stores the vid j of a vertex; the 2rd field stores v_j; the 3rd field stores Δv_j; the 4th field stores the priority value of vertex j for priority scheduling; the 5th field stores the input data associated with vertex j, such as the adjacency list. Users are responsible for initializing the v fields and the Δv fields through the provided API. The priority fields are automatically initialized based on the values of the v fields and Δv fields. Users read an input partition and fills entry j's data field with vertex j's input data.

Receive Thread and Update Thread. As described in Equation (6), DAIC is accomplished by two key operations, the receive operation and the update operation. In each worker, these two operations are implemented in two threads, the *receive thread* and the *update thread*. The receive thread performs the receive operation for all local vertices. Each worker receives the delta messages from other workers and updates the Δv fields by accumulating the received delta messages. The update thread performs the update operation for all local vertices. When operating on a vertex, it updates the corresponding entry's v field

TABLE 1
A list of DAIC algorithms

algorithm	$g_{\{i,j\}}(x)$	$\oplus$	v_j^0	Δv_j^1
SSSP	$x + A(i,j)$	min	∞	0 $(j = s)$ or ∞ $(j \neq s)$
Connected Components	$A(i,j) \cdot x$	max	-1	j
PageRank	$d \cdot A(i,j) \cdot \frac{x}{\lvert N(j)\rvert}$	$+$	0	$1 - d$
Adsorption	$p_i^{cont} \cdot A(i,j) \cdot x$	$+$	0	$p_j^{inj} \cdot I_j$
HITS $(authority)$	$d \cdot A(i,j) \cdot x$	$+$	0	1
Katz metric	$\beta \cdot A(i,j) \cdot x$	$+$	0	1 $(j = s)$ or 0 $(j \neq s)$
Jacobi method	$-\frac{A_{ji}}{A_{jj}} \cdot x$	$+$	0	$\frac{b_j}{A_{jj}}$
SimRank	$\frac{C \cdot A(i,j)}{\lvert I(a)\rvert \lvert I(b)\rvert} \cdot x$	$+$	$\lvert I(a) \cap I(b)\rvert$ $(a \neq b)$ or 1 $(a = b)$	$\frac{\lvert I(a)\rvert \lvert I(b)\rvert}{C}$ $(a \neq b)$ or 0 $(a = b)$
Rooted PageRank	$A(j,i) \cdot x$	$+$	0	1 $(j = s)$ or 0 $(j \neq s)$

and Δv field, and sends messages to other vertices.

Scheduling within Update Thread. The simplest scheduling policy is to schedule the local vertices for update operation in a round robin fashion. The update thread performs the update operation on the table entries in the order that they are listed in the local state table and round-by-round. The static scheduling is simple and can prevent starvation.

However, as discussed in Section 2.5, it is beneficial to provide priority scheduling. In addition to the static round-robin scheduling, Maiter supports dynamic priority scheduling. A *priority queue* in each worker contains a subset of local vids that have larger priority values. The update thread dequeues the vid from the priority queue, in terms of which it can position the entry in the local state table and performs an update operation on the entry. Once all the vertices in the priority queue have been processed, the update thread extracts a new subset of high-priority vids for next round update. The extraction of vids is based on the priority field. Each entry's priority field is initially calculated based on its initial v value and Δv value. During the iterative computation, the priority field is updated whenever the Δv field is changed (*i.e.*, whenever some delta messages are received).

The number of extracted vids in each round, *i.e.*, the priority queue size, balances the tradeoff between the gain from accurate priority scheduling and the cost of frequent queue extractions. The priority queue size is set as a portion of the state table size. For example, if the queue size is set as 1% of the state table size, we will extract the top 1% high priority entries for processing. In addition, we also use the sampling technique proposed in [16] for efficient queue extraction, which only needs $O(N)$ time, where N is the number of entries in local state table.

Message Passing. Maiter uses OpenMPI [18] to implement message passing between workers. A message contains a vid indicating the message's destination vertex and a value. Suppose that a message's destination vid is k. The message will be sent to worker $h(k)$, where $h()$ is the partition function for data partition, so the message will be received by the worker where the destination vertex resides.

A naive implementation of message passing is to send the output messages as soon as they are produced. This will reach the asynchronous iteration's full potential. However, initializing message passing leads to system overhead. To reduce this overhead, Maiter buffers the output messages and flushes them after a timeout. If a message's destination worker is the host worker, the output message is directly applied to the local state table. Otherwise, the output messages are buffered in multiple *msg table*s, each of which corresponds to a remote destination worker. We can leverage early aggregation on the msg table to reduce network communications. Each msg table entry consists of a destination vid field and a value field. As mentioned in Section 2.1, the associative property of operator '$\oplus$', *i.e.*, $(x \oplus y) \oplus z = x \oplus (y \oplus z)$, indicates that multiple messages with the same destination can be aggregated at the sender side or at the receiver side. Therefore, by using the msg table, Maiter worker combines the output messages with the same vid by '$\oplus$' operation before sending them.

Iteration Termination. To terminate iteration, Maiter exploits *progress estimator* in each worker and a global *terminator* in the master. The master periodically broadcasts a *progress request signal* to all workers. Upon receipt of the termination check signal, the progress estimator in each worker measures the iteration progress locally and reports it to the master. The users are responsible for specifying the progress estimator to retrieve the iteration progress by parsing the local state table. After the master receives the local iteration progress reports from all workers, the terminator makes a global termination decision in respect of the global iteration progress, which is calculated based on the received local progress reports. If the terminator determines to terminate the iteration, the master broadcasts a *terminate signal* to all workers. Upon receipt of the terminate signal, each worker stops updating the state table and dumps the local table entries to HDFS, which contain the converged results. Note that, even though we exploit a synchronous termination check periodically, it will not impact the asynchronous computation. The workers proceed after producing the local progress reports without waiting for the master's feedback.

Fault Tolerance. The fault tolerance support for synchronous computation models can be performed through checkpointing, where the state data is checkpointed on the reliable HDFS every several iterations. If some workers fail, the computation rolls back to the most recent iteration checkpoint and resumes from that iteration. Maiter exploits Chandy-Lamport [19] algorithm to design asynchronous iteration's fault tolerance mechanism. The checkpointing in Maiter is performed at regular time intervals rather than at iteration intervals. The state table in each worker is dumped to HDFS every period of time. However, during the asynchronous computation, the information in the state table might not be intact, in respect that the messages may be on their way to act on the state table. To avoid missing messages, not only the state table is dumped to HDFS, but also the msg tables in each worker are saved. Upon detecting any worker failure (through probing by the master), the master restores computation from the last checkpoint, migrates the failed worker's state table and msg tables to an available worker, and notifies all the workers to load the data from the most recent checkpoint to recover from worker failure. For detecting master failure, Maiter can rely on a secondary master, which restores the recent checkpointed state to recover from master failure.

5 Evaluation

This section evaluates Maiter with a series of experiments.

5.1 Preparation

Frameworks for Comparison. We pick a few popular frameworks for comparison, including **Hadoop** [6], memory-based iMapReduce (**iMR-mem**) [20], file-based iMapReduce (**iMR-file**) [20], **Spark** [21], file-based PrIter (**PrIter-mem**) [16], file-based PrIter (**PrIter-file**) [16], **Piccolo** [11], and GraphLab [8]. The GraphLab framework provides both synchronous execution engine (**GraphLab-Sync**) and asynchronous execution engine. Moreover, under the asynchronous execution engine, GraphLab supports both fifo scheduling (**GraphLab-AS-fifo**) and priority scheduling (**GraphLab-AS-pri**). To evaluate Maiter with different scheduling policies, we consider the round robin scheduling (**Maiter-RR**) as well as the priority scheduling (**Maiter-Pri**). In addition, we manually add a synchronization barrier controlled by the master to let these workers perform DAIC synchronously. We call this version of Maiter as **Maiter-Sync**. More details of these frameworks can be found in Section 5.1 of the supplementary file.

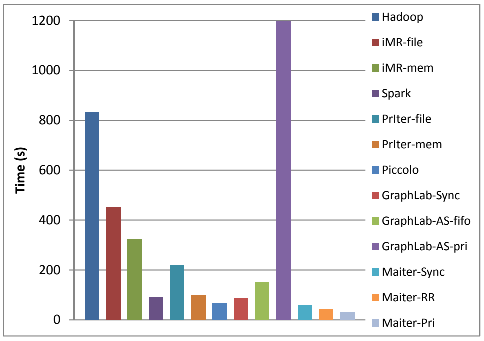


Fig. 2. Running time of PageRank on Google Webgraph on local cluster.

Experimental Cluster. The experiments are performed on a cluster of local machines as well as on Amazon EC2 Cloud [15]. The *local cluster* consisting of 4 commodity machines is used to run small-scale experiments. Each machine has Intel E8200 dual-core 2.66GHz CPU, 3GB of RAM, and 160GB storage. The Amazon EC2 cluster involves 100 medium instances, each with 1.7GB memory and 5 EC2 compute units.

Applications and Data Sets. Four applications, including PageRank, SSSP, Adsorption, and Katz metric, are implemented. We use Google Webgraph [22] for PageRank computation. We also generate synthetic massive data sets for PageRank and other applications. The details of synthetic data sets can be found in Section 5.2 of the supplementary file.

5.2 Running Time to Convergence

Local Cluster Results. We compare different frameworks on running time in the context of PageRank computation. Due to the limited space, the termination approach of PageRank computation is presented in Section 5.3 of the supplementary file. Fig. 2 shows the PageRank running time on Google Webgraph on our local cluster. Note that, the data loading time for the memory-based systems (other than Hadoop, iMR-file, iMR-mem) is included in the total running time.

By using Hadoop, we need 27 iterations and more than 800 seconds to converge. By separating the iteration-variant state data from the static data, iMR-file reduces the running time of Hadoop by around 50%. iMR-mem further reduces it by providing faster memory access. Spark, with efficient data partition and memory caching techniques, can reduce Hadoop time to less than 100 seconds. PrIter identifies the more important nodes to perform the update and ignores the useless updates, by which the running time is reduced. As expected, PrIter-mem converges faster than PrIter-file. Piccolo utilizes MPI for message passing to realize fine-grained updates, which improves the performance.

GraphLab variants show their differences on the performance. GraphLab-Sync uses a synchronous engine and completes the iterative computation with-

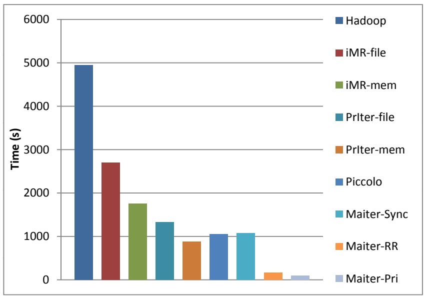


Fig. 3. Running time of PageRank on 100-million-node synthetic graph on EC2 cluster.

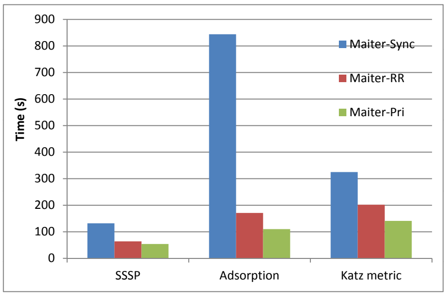


Fig. 4. Running time of other applications (SSSP, Adsorption, and Katz metric) on EC2 cluster.

in less than 100 seconds. GraphLab-AS-fifo uses an asynchronous engine and schedules the asynchronous updates in a FIFO queue, which consumes much more time. The reason is that the cost of managing the scheduler (through locks) tends to exceed the cost of the main PageRank computation itself. The cost of maintaining the priority queue under asynchronous engine seems even much larger, so that GraphLab-AS-pri converges with significant longer running time. More experimental results focusing on demonstrating the difference between GraphLab and Maiter can be found in Section 5.4 of the supplementary file.

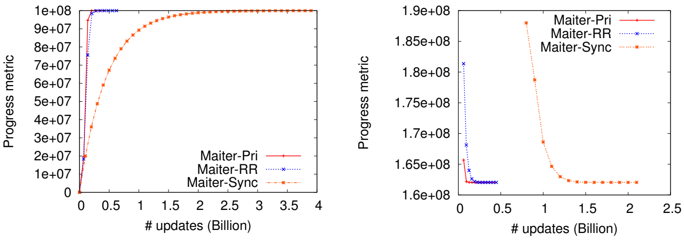
The framework that supports synchronous DAIC, Maiter-Sync, filters the zero updates ($\Delta R = 0$) and reduces the running time to about 60 seconds. Further, the asynchronous DAIC frameworks, Maiter-RR and Maiter-Pri, can even converge faster by avoiding the synchronous barriers. Note that, our priority scheduling mechanism does not result in high cost, since we do not need distributed lock for scheduling asynchronous DAIC. In addition, in priority scheduling, the approximate sampling technique [16] helps reduce the complexity, which avoids high scheduling cost.

EC2 Results. To show the performance under large-scale distributed environment, we run PageRank on a 100-million-node synthetic graph on EC2 cluster. Fig. 3 shows the running time with various frameworks. We can see the similar results. One thing that should be noticed is that Maiter-Sync has comparable performance with Piccolo and PrIter. Only DAIC is not enough to make a significant performance improvement. However, the asynchronous DAIC frameworks (Maiter-RR and Maiter-Pri) perform much better. The result is under expectation. As the cluster size increases and the heterogeneity in cloud environment becomes apparent, the problem of synchronous barriers is more serious. With the asynchronous execution engine, Maiter-RR and Maiter-Pri can bypass the high-cost synchronous barriers and perform more efficient computations. As a result, Maiter-RR and Maiter-Pri significantly reduce the running time. Moreover, Maiter-Pri exploits more efficient priority scheduling, which can achieve 60x speedup over Hadoop. This result demonstrates that only with asynchronous execution can DAIC reach its full potential.

To show that Maiter can support more applications, we also run other applications on EC2 cluster. We perform SSSP, Adsorption, and Katz metric computations with Maiter-Sync, Maiter-RR, and Maiter-Pri. We generate weighted/unweighted 100-million-node synthetic graphs for these applications respectively. Fig. 4 shows the running time of these applications. For SSSP, the asynchronous DAIC SSSP (Maiter-RR and Maiter-Pri) reduces the running time of synchronous DAIC SSSP (Maiter-Sync) by half. For Adsorption, the asynchronous DAIC Adsorption is 5x faster than the synchronous DAIC Adsorption. Further, by priority scheduling, Maiter-Pri further reduces the running time of Maiter-RR by around 1/3. For Katz metric, we can see that Maiter-RR and Maiter-Pri also outperform Maiter-Sync.

5.3 Efficiency of Asynchronous DAIC

As analyzed in Section 2.4, with the same number of updates, asynchronous DAIC results in more progress than synchronous DAIC. In this experiment, we measure the number of updates that PageRank and SSSP need to converge under Maiter-Sync, Maiter-RR, and Maiter-Pri. In order to measure the iteration process, we define a *progress metric*, which is $\sum_j R_j$ for PageRank and $\sum_j d_j$ for SSSP. Then, the *efficiency* of the update operations can be seen as the ratio of the progress metric to the number of updates.



(a) PageRank

(b) SSSP

Fig. 5. Number of updates vs. progress metric.

On the EC2 cluster, we run PageRank on a 100-million-node synthetic graph and SSSP on a 500-million-node synthetic graph. Fig. 5a shows the progress metric against the number of updates for PageRank. In PageRank, the progress metric $\sum_j R_j$

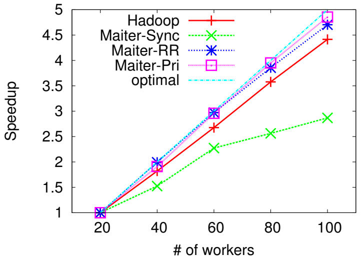


Fig. 6. Scaling performance.

should be increasing. Each R_j^0 is intialized to be 0 and each ΔR_j^1 is initialized to be $1-d=0.2$ (the damping factor $d=0.8$). The progress metric $\sum_j R_j$ is increasing from $\sum_j R_j^1=\sum_j(R_j^0+\Delta R_j^1)=0.2\cdot N$ to N, where $N=10^8$ (number of nodes). Fig. 5b shows the progress metric against the number of updates for SSSP. In SSSP, the progress metric $\sum_j d_j$ should be decreasing. Since d_j is initialized to be ∞ for any node $j\neq s$, which cannot be drawn in the figure, we start plotting when any $d_j<\infty$. From Fig. 5a and Fig. 5b, we can see that by asynchronous DAIC, Maiter-RR and Maiter-Pri require much less updates to converge than Maiter-Sync. That is, the update in asynchronous DAIC is more effective than that in synchronous DAIC. Further, Maiter-Pri selects more effective updates to perform, so the update in Maiter-Pri is even more effective.

5.4 Scaling Performance

Suppose that the running time on one worker is T. With optimal scaling performance, the running time on an n-worker cluster should be $\frac{T}{n}$. But in reality, distributed application usually cannot achieve the optimal scaling performance. In order to measure how asynchronous Maiter scales with increasing cluster size, we perform PageRank on a 100-million-node graph on EC2 as the number of workers increases from 20 to 100. We consider the running time on a 20-worker cluster as the baseline, based on which we determine the running time with optimal scaling performance on different size clusters. We consider Hadoop, Maiter-Sync, Maiter-RR, and Maiter-Pri for comparing their scaling performance.

Fig. 6 shows the scaling performance of Hadoop, Maiter-Sync, Maiter-RR, and Maiter-Pri. We can see that the asynchronous DAIC frameworks, Maiter-RR and Maiter-Pri, provide near-optimal scaling performance as cluster size scales from 20 to 100. The performance of the synchronous DAIC framework Maiter-Sync is degraded a lot as the cluster size scales. Hadoop splits a job into many fine-grained tasks (task with 64MB block size), which alleviates the impact of synchronization and improves scaling performance.

More experimental results, such as the communication cost and the scaling performance when increasing input size, are provided in Section 5.5 and Section 5.6 of the supplementary file.

6 Related Work

The original idea of asynchronous iteration, chaotic iteration, was introduced by Chazan and Miranker in 1969 [12]. Motivated by that, Baudet proposed an asynchronous iterative scheme for multicore systems [13], and Bertsekas presented a distributed asynchronous iteration model [14]. These early stage studies laid the foundation of asynchronous iteration and have proved its effectiveness and convergence. Asynchronous methods are being increasingly used and studied since then, particularly so in connection with the use of heterogeneous workstation clusters. A broad class of applications with asynchronous iterations have been correspondingly raised [23], [24], such as PageRank [25], [26] and pairwise clustering [27]. Our work differs from these previous works. We focus on a particular class of iterative algorithms and provide a new asynchronous iteration scheme, DAIC, which exploits the accumulative property.

On the other hand, to support iterative computation, a series of distributed frameworks have emerged. In addition to the frameworks we compared in Section 5, many other synchronous frameworks are proposed recently. HaLoop [28], a modified version of Hadoop, improves the efficiency of iterative computations by making the task scheduler loop-aware and employing caching mechanisms. CIEL [29] supports data-dependent iterative algorithms by building an abstract dynamic task graph. Pregel [7] aims at supporting graph-based iterative algorithms by proposing a graph-centric programming model. REX [30] optimizes DBMS recursive queries by using incremental updates. Twister [31] employs a lightweight iterative MapReduce runtime system by logically constructing a reduce-to-map loop. Naiad [9] is recently proposed to support incremental iterative computations.

All of the above described works build on the basic assumption that the synchronization between iterations is essential. A few proposed frameworks also support asynchronous iteration. The partial asynchronous approach proposed in [32] investigates the notion of partial synchronizations in iterative MapReduce applications to overcome global synchronization overheads.GraphLab [33] supports asynchronous iterative computation with sparse computational dependencies while ensuring data consistency and achieving a high degree of parallel performance. PowerGraph [34] forms the foundation of GraphLab, which characterizes the challenges of computation on natural graphs. The authors propose a new approach to distributed graph placement and representation that exploits the structure of power-law graphs. GRACE [35] executes iterative computation with asynchronous engine while letting users implement their algorithms with the synchronous BSP programming model. To the best of our knowledge, our work is the first that proposes to perform DAIC for iterative algorithms. We also identify a broad class of iterative algorithms that can perform DAIC.

7 Conclusions

In this paper, we propose DAIC, delta-based accumulative iterative computation. The DAIC algorithms can be performed asynchronously and converge with much less workload. To support DAIC model, we design and implement Maiter, which is running on top of hundreds of commodity machines and relies on message passing to communicate between distributed machines. We deploy Maiter on local cluster as well as on Amazon EC2 cloud to evaluate its performance in the context of four iterative algorithms. The results show that by asynchronous DAIC the iterative computation performance is significantly improved.

Acknowledgments

This work was partially supported by U.S. NSF grants (CCF-1018114, CNS-1217284), National Natural Science Foundation of China (61300023, 61272179), Fundamental Research Funds for the Central Universities (N120416001, N120816001, N100704001), China Mobil Labs Fund (MCM20122051), and MOE-Intel Special Fund of Information Technology (MOE-INTEL-2012-06).

References

[1] S. Baluja, R. Seth, D. Sivakumar, Y. Jing, J. Yagnik, S. Kumar, D. Ravichandran, and M. Aly, "Video suggestion and discovery for youtube: taking random walks through the view graph," in *Proc. Int'l Conf. World Wide Web (WWW '08)*, 2008, pp. 895–904.

[2] H. H. Song, T. W. Cho, V. Dave, Y. Zhang, and L. Qiu, "Scalable proximity estimation and link prediction in online social networks," in *Proc. Int'l Conf. Internet Measurement (IMC '09)*, 2009, pp. 322–335.

[3] D. Liben-Nowell and J. Kleinberg, "The link-prediction problem for social networks," *J. Am. Soc. Inf. Sci. Technol.*, vol. 58, pp. 1019–1031, May 2007.

[4] S. Brin and L. Page, "The anatomy of a large-scale hypertextual web search engine," *Comput. Netw. ISDN Syst.*, vol. 30, pp. 107–117, April 1998.

[5] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," in *Proc. USENIX Symp. Opearting Systems Design & Implementation (OSDI '04)*, 2004, pp. 10–10.

[6] Hadoop. http://hadoop.apache.org/.

[7] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *Proc. ACM SIGMOD*, 2010, pp. 135–146.

[8] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein, "Distributed graphlab: A framework for machine learningand data mining in the cloud," *Proc. VLDB Endow.*, vol. 5, no. 8, 2012.

[9] F. McSherry, D. Murray, R. Isaacs, and M. Isard, "Differential dataflow," in *Proc. Biennial Conf. Innovative Data Systems Research (CIDR '13)*, 2013.

[10] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: cluster computing with working sets," in *Proc. USENIX Workshop Hot Topics in Cloud Computing (HotCloud '10)*, 2010.

[11] R. Power and J. Li, "Piccolo: Building fast, distributed programs with partitioned tables," in *Proc. USENIX Symp. Opearting Systems Design and Implementation (OSDI '10)*, 2010.

[12] D. Chazan and W. Miranker, "Chaotic relaxation," *Linear Algebra and its Applications*, vol. 2, no. 2, pp. 199 – 222, 1969.

[13] G. M. Baudet, "Asynchronous iterative methods for multiprocessors," *J. ACM*, vol. 25, pp. 226–244, April 1978.

[14] D. P. Bertsekas, "Distributed asynchronous computation of fixed points," *Math. Programming*, vol. 27, pp. 107–120, 1983.

[15] Amazon ec2. http://aws.amazon.com/ec2/.

[16] Y. Zhang, Q. Gao, L. Gao, and C. Wang, "Priter: a distributed framework for prioritized iterative computations," in *Proc. ACM Symp. Cloud Computing (SOCC '11)*, 2011.

[17] Maiter project. http://code.google.com/p/maiter/.

[18] Open mpi. http://www.open-mpi.org/.

[19] K. M. Chandy and L. Lamport, "Distributed snapshots: determining global states of distributed systems," *ACM Trans. Comput. Syst.*, vol. 3, no. 1, pp. 63–75, Feb. 1985. [Online]. Available: http://doi.acm.org/10.1145/214451.214456

[20] Y. Zhang, Q. Gao, L. Gao, and C. Wang, "imapreduce: A distributed computing framework for iterative computation," *J. Grid Comput.*, vol. 10, no. 1, pp. 47–68, 2012.

[21] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for. in-memory cluster computing," in *Proc. USEINX Symp. Networked Systems Design and Implementation (NSDI'12)*, 2012.

[22] Stanford dataset collection. http://snap.stanford.edu/data/.

[23] A. Frommer and D. B. Szyld, "On asynchronous iterations," *J. Comput. Appl. Math.*, vol. 123, pp. 201–216, November 2000.

[24] J. C. Miellou, D. El Baz, and P. Spiteri, "A new class of asynchronous iterative algorithms with order intervals," *Math. Comput.*, vol. 67, pp. 237–255, January 1998.

[25] F. McSherry, "A uniform approach to accelerated pagerank computation," in *Proc. Int'l Conf. World Wide Web (WWW '05)*, 2005, pp. 575–582.

[26] G. Kollias, E. Gallopoulos, and D. B. Szyld, "Asynchronous iterative computations with web information retrieval structures: The pagerank case." in *PARCO*, ser. John von Neumann Institute for Computing Series, vol. 33, 2005, pp. 309–316.

[27] E. Yom-Tov and N. Slonim, "Parallel pairwise clustering." in *Proc. SIAM Intl. Conf. Data Mining (SDM '09)*, 2009, pp. 745–755.

[28] Y. Bu, B. Howe, M. Balazinska, and D. M. Ernst, "Haloop: Efficient iterative data processing on large clusters," *Proc. VLDB Endow.*, vol. 3, no. 1, 2010.

[29] 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. USEINX Symp. Networked Systems Design and Implementation (NSDI '11)*, 2011.

[30] M. S. R., I. G. Ives, and G. Sudipto, "Rex: Recursive, deltabased datacentric computation," *Proc. VLDB Endow.*, vol. 5, no. 8, 2012.

[31] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S.-H. Bae, J. Qiu, and G. Fox, "Twister: a runtime for iterative mapreduce," in *Proc. IEEE Int'l Workshop MapReduce (MapReduce '10)*, 2010, pp. 810–818.

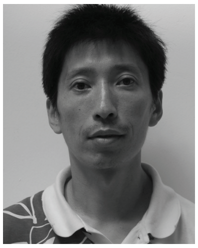
[32] K. Kambatla, N. Rapolu, S. Jagannathan, and A. Grama, "Asynchronous algorithms in mapreduce," in *Proc. IEEE Conf. Cluster (Cluster' 10)*, 2010, pp. 245 –254.

[33] L. Yucheng, G. Joseph, K. Aapo, B. Danny, G. Carlos, and M. H. Joseph, "Graphlab: A new framework for parallel machine learning," in *Proc. Int'l Conf. Uncertainty in Artificial Intelligence (UAI '10)*, 2010.

[34] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "Powergraph: distributed graph-parallel computation on natural graphs," in *Proc. USENIX Symp. Opearting Systems Design and Implementation (OSDI '12)*, 2012, pp. 17–30.

[35] G. Wang, W. Xie, A. Demers, and J. Gehrke, "Asynchronous large-scale graph processing made easy," in *Proc. Biennial Conf. Innovative Data Systems Research (CIDR '13)*, 2013.

Yanfeng Zhang received the BSc, MSc, and PhD degrees in computer science from Northeastern University, China, in 2005, 2008, and 2012 respectively. He is currently working in Computing Center at Northeastern University, China. He had been a visiting PhD student in University of Massachusetts Amherst during August 2009 to April 2012. His current research consists of large scale data mining, distributed systems, and cloud computing. He has published many technical papers in the above areas. His paper in ACM Cloud Computing 2011 was honored with “Paper of Distinction”.



Qixin Gao received the Ph.D. degree in Institute of Computer Science and Engineering, Northeastern University, Shenyang, China, in 2008. He is currently working in Northeastern University at Qinhuangdao, China. His current research interests include image processing, visual perception, and massive data processing.

Lixin Gao is a professor of Electrical and Computer Engineering at the University of Massachusetts at Amherst. She received her Ph.D. degree in computer science from the University of Massachusetts at Amherst in 1996. Her research interests include social networks, Internet routing, network virtualization and cloud computing. Between May 1999 and January 2000, she was a visiting researcher at AT&T Research Labs and DIMACS. She was an Alfred P. Sloan Fellow between 2003-2005 and received an NSF CAREER Award in 1999. She won the best paper award from IEEE INFOCOM 2010, and the test-of-time award in ACM SIGMETRICS 2010. Her paper in ACM Cloud Computing 2011 was honored with “Paper of Distinction”. She received the Chancellor’s Award for Outstanding Accomplishment in Research and Creative Activity in 2010, and is a fellow of IEEE.

Cuirong Wang received the Ph.D. degree from Northeastern University, Shenyang, China, in 2003. She is currently a Professor with the computer Science Departmen-t, Northeastern University at Qinhuangdao, China. Her current research interests include data center networks, cloud computing, and wireless sensor networks. She has been a main researcher in several National Nature Science Foundation research projects of China. Dr. Wang is an advanced member of China Computer Federation.