Supplementary File of the TPDS manuscript

by Yanfeng Zhang, Qixin Gao, Lixin Gao, *Fellow, IEEE*, Cuirong Wang
zhangyf@cc.neu.edu.cn

Abstract—This supplementary file contains the supporting materials of the TPDS manuscript – "Maiter: An Asynchronous Graph Processing Framework for Delta-based Accumulative Iterative Computation." It improves the completeness of the TPDS manuscript.

1 Iterative Graph Processing

The graph algorithm can be abstracted as the operations on a graph $G(V, E)$. Peoples usually exploit a vertex-centric model to solve the graph algorithms. Basically, the graph algorithm is described from a single vertex's perspective and then applied to each vertex for a loosely coupled execution. Iterative graph algorithms perform the same operations on the graph vertices for several iterations. Each vertex $j \in V$ maintains a vertex state v_j that is updated iteratively. The key of a vertex-centric graph computation is the update function f performed on each vertex j:

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

where v_j^k represents vertex j's state after the k^{th} iteration, and $v_1, v_2, \dots, v_n$ are the states of vertex j's neighbors. The state values are passed between vertices through the edges. The iterative process continues until the graph vertex state becomes stable, when the iterative algorithm converges.

For example, the well-known PageRank algorithm iteratively updates all pages' ranking scores. According to the vertex-centric graph processing model, in each iteration, we update the ranking score of each page j, R_j, as follows:

$$R_j^k = d \cdot \sum_{\{i|(i \to j) \in E\}} \frac{R_i^{k-1}}{|N(i)|} + (1-d), \tag{2}$$

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 PageRank scores of all pages are updated round by round until convergence.

In distributed computing, vertices are distributed to multiple processors and perform updates in parallel. For simplicity of exposition, assume that there are enough processors and each processor j performs update for vertex j. All vertices perform the update in lock steps. At step k, vertex j first collects v_i^{k-1} from all its neighbor vertices, 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 be completed before any of the update operations in the k^{th} iteration starts. Clearly, this synchronization is required in each step. These synchronizations might degrade performance, especially in heterogeneous distributed environments.

To avoid the synchronization barriers, **asynchronous iteration** was proposed [1]. Performing update operations asynchronously means that vertex j performs the update

$$v_j = f(v_1, v_2, \dots, v_n) \tag{3}$$

at any time based on the most recent values of its neighbor vertices, $\{v_1, v_2, \dots, v_n\}$. The conditions of convergence of asynchronous iterations have been studied in [1], [2], [3].

By asynchronous iteration, as vertex j is activated to perform an update, it "pulls" the values of its neighbor vertices, *i.e.*, $\{v_1, v_2, \dots, v_n\}$, and uses these values to perform an update on v_j. This scheme does not require any synchronization. However, asynchronous iteration intuitively requires more communications and useless computations than synchronous iteration. An activated vertex needs to pull the values from all its neighbor vertices, but not all of them have been updated, or even worse none of them is updated. In that case, asynchronous iteration performs a useless computation and results in significant communication overhead. Accordingly, "pull-based" asynchronous iteration is only applicable in an environment where the communication overhead is negligible, such as shared memory systems. In a distributed environment or in a cloud, "pull-based" asynchronous model cannot be efficiently utilized.

As an alternative, after vertex i updates v_i, it "pushes" v_i to all its neighbors j, and v_i is buffered as $B_{i,j}$ on each vertex j, which will be updated as new v_i arrives. Vertex j only performs update when there are new values in the buffers and uses these buffered values $B_{i,j}$, to update v_j. In this way, the redundant communications can be avoided. However, the "push-based" asynchronous iteration results in higher space complexity. Each vertex j has to buffer $|N(j)|$ values, where $|N(j)|$ is the number of vertex j's neighbors.

The large number of buffers also leads to considerable maintenance overhead.

To sum up, in a distributed environment, the synchronous iteration results in low performance due to the multiple global barriers, while the asynchronous iteration cannot be efficiently utilized due to various implementation overheads. Also note that, for some iterative algorithms, the asynchronous iteration cannot guarantee to converge to the same fixpoint as the synchronous iteration does, which leads to uncertainty.

2 Proofs of DAIC

In this section, we provide the proofs of Theorem 1, Theorem 2, and Theorem 3 in the TPDS manuscript.

2.1 Proof of Theorem 1

In this section, we will show four lemmas to support our proof of Theorem 1 in the TPDS manuscript. The first two lemmas show the formal representations of a vertex state by synchronous DAIC and by asynchronous DAIC, respectively. The third lemma shows that there is a time instance when the result by asynchronous DAIC is smaller than or equal to the result by synchronous DAIC. Correspondingly, we show another lemma that there is a time instance when the result by asynchronous DAIC is larger than or equal to the result by synchronous DAIC. Once these lemmas are proved, it is sufficient to establish Theorem 1.

Lemma 1: By synchronous DAIC, v_j after k iterations is:

$$v_j^k = v_j^0 \oplus \Delta v_j^1 \oplus \sum_{l=1}^{k} \oplus \Big(\prod_{\{i_0,\ldots,i_{l-1},j\} \in P(j,l)} \oplus g_{\{i,j\}}(\Delta v_i^1) \Big), \tag{4}$$

where

$$\prod_{\{i_0,\ldots,i_{l-1},j\}} \oplus g_{\{i,j\}}(\Delta v_i^1) = g_{\{i_{l-1},j\}}(\ldots g_{\{i_1,i_2\}}(g_{\{i_0,i_1\}}(\Delta v_{i_0}^1)))$$

and $P(j,l)$ is a set of l-hop paths to reach node j.

Proof: According to the update functions shown in Equation (2) in the TPDS manuscript, after k iterations, we have

$$\begin{aligned} v_j^k =& v_j^0 \oplus \Delta v_j^1 \oplus \Big(\sum_{i_1=1}^{n} \oplus g_{\{i_1,j\}}(\Delta v_{i_1}^1) \Big) \oplus \\ & \Big(\sum_{i_1=1}^{n} \oplus g_{\{i_1,j\}}\big(\sum_{i_2=1}^{n} \oplus g_{\{i_2,i_1\}}(\Delta v_{i_2}^1) \big) \Big) \oplus \ldots \oplus \\ & \Big(\sum_{i_1=1}^{n} \oplus g_{\{i_1,j\}}\big(\sum_{i_2=1}^{n} \oplus g_{\{i_2,i_1\}}\big(\ldots \sum_{i_k=1}^{n} \oplus g_{\{i_k,i_{k-1}\}}(\Delta v_{i_k}^1) \big)\big) \Big). \end{aligned}$$

The l^{th} term of the right side this equation corresponds to the received values from the $(l+1)$-hop away neighbors. Therefore, we have the claimed equation. □

In order to describe asynchronous DAIC, we define a continuous time instance sequence $\{t_1, t_2, \ldots, t_k\}$. Correspondingly, we define $S = \{S_1, S_2, \ldots, S_k\}$ as the series of subsets of vertices, where S_k is a subset of vertices, and the propagated values of all vertices in S_k have been received by their direct neighbors during the interval between time t_{k-1} and time t_k . As a special case, synchronous updates result from a sequence $\{V, V, \ldots, V\}$, where V is the set of all vertices.

Lemma 2: By asynchronous DAIC, following an activation sequence S, $\check{v}_j$ at time t_k is:

$$\check{v}_j^k = v_j^0 \oplus \Delta v_j^1 \oplus \sum_{l=1}^{k} \oplus \Big(\prod_{\{i_0,\ldots,i_{l-1},j\} \in P'(j,l)} \oplus g_{\{i,j\}}(\Delta v_i^1) \Big) \tag{5}$$

where $P'(j,l)$ is a set of l-hop paths that satisfy the following conditions. First, $i_0 \in S_l$. Second, if $l > 0$, $i_1, \ldots, i_{l-1}$ respectively belongs to the sequence S. That is, there is $0 < m_1 < m_2 < \ldots < m_{l-1} < k$ such that $i_h \in S_{m_{l-h}}$.

Proof: We can derive $\check{v}_j^k$ from Equation (6) in the TPDS manuscript. □

Lemma 3: For any sequence S that each vertex performs the receive and update operations an infinite number of times, given any iteration number k, we can find a subset index k' in S such that $|v_j^* - \check{v}_j^{k'}| \geq |v_j^* - v_j^k|$ for any vertex j.

Proof: Based on Lemma 1, we can see that, after k iterations, each node receives the values from its direct/indirect neighbors as far as k hops away, and it receives the values originated from each direct/indirect neighbor once for each path. In other words, each node j propagates its own initial value Δv_j^1 (first to itself) and receives the values from its direct/indirect neighbors through a path once.

Based on Lemma 2, we can see that, after time t_k, each node receives values from its direct/indirect neighbors as far as k hops away, and it receives values originated from each direct/indirect neighbor through a path at most once. At time period $[t_{k-1}, t_k]$, a value is received from a neighbor only if the neighbor is in S_k. If the neighbor is not in S_k, the value is stored at the neighbor or is on the way to other nodes. The node will eventually receive the value as long as every node performs receive and update an infinite number of times.

As a result, $\check{v}_j^k$ receives values through a subset of the paths from j's direct/indirect incoming neighbors within k hops. In contrast, v_j^k receives values through all paths from j's direct/indirect incoming neighbors within k hops. $\check{v}_j^k$ receives less values than v_j^k. Correspondingly, $\check{v}_j^k$ is further to the converged point v_j^* than v_j^k. Therefore, we can set $k' = k$ and have the claim. □

Lemma 4: For any sequence S that each vertex performs the receive and update operations an infinite number of times, given any iteration number k, we can find a subset index k'' in S such that $|v_j^* - \check{v}_j^{k''}| \leq |v_j^* - v_j^k|$ for any vertex j.

Proof: From the proof of Lemma 3, we know that v_j^k receives values from all paths from direct/indirect neighbors of j within k hops away. In order to let $\check{v}_j^{k''}$ receives all those values, we have to make sure that all paths from direct/indirect neighbors of j within k hops away are activated and their values are received. Since in sequence S each vertex performs the update an infinite number of times, we can always find k'' such that $\{S_1, S_2, \dots, S_{k''}\}$ contains all paths from direct and indirect neighbors of j within k hops away. Correspondingly, $\check{v}_j^{k''}$ can be nearer to the converged point v_j^* than v_j^k, or at least equal to. Therefore, we have the claim. □

Based on Lemma 3 and Lemma 4, we have Theorem 1.

Theorem 1: If v_j in (1) converges, $\check{v}_j$ in (3) converges. Further, they converge to the same value, *i.e.*, $v_j^\infty = \check{v}_j^\infty = \check{v}_j^*$.

2.2 Proof of Theorem 2

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.*, $|v_j^* - \check{v}_j| \leq |v_j^* - v_j|$.

Proof: In a single machine, the update sequence for asynchronous DAIC is a special S, where only one vertex in S_k for any k and any vertex is appeared once and only once in $\{S_{(k-1)n+1}, S_{(k-1)n+2}, \dots, S_{(k-1)n+n}\}$ for any k, where n is the total number of vertices. Based on Lemma 2, we have

$$\check{v}_j^{kn} = v_j^0 \oplus \Delta v_j^1 \oplus \sum_{l=1}^{kn} \oplus \Big(\prod_{\{i_0,\dots,i_{l-1},j\} \in P'(j,l)} \oplus g_{\{i,j\}}(\Delta v_i^1) \Big), \tag{6}$$

The values sent from any k-hop-away neighbors of j will be received during time period $[t_{(k-1)n}, t_{kn}]$, *i.e.*, the sent values from $\{S_{(k-1)n+1}, S_{(k-1)n+2}, \dots, S_{(k-1)n+n}\}$ are received. Further, $\check{v}_j^{kn}$ receives more values from further hops away, as far as kn-hop-away neighbors. Therefore, $\check{v}_j^{kn}$ is nearer to the converged point v_j^* than v_j^k, *i.e.*, $|v_j^* - \check{v}_j^{kn}| \leq |v_j^* - v_j^k|$. □

2.3 Proof of Theorem 3

We first pose the following lemma.

Lemma 5: 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.*, $\check{v}'^\infty_j = v_j^\infty = v_j^*$.

Proof: There are two cases to guide priority scheduling. We only prove the case that schedules vertex j that results in the largest $(\check{v}_j \oplus \Delta\check{v}_j - \check{v}_j)$. The proof of the other case is similar.

We prove the lemma by contradiction. Assume there is a set of vertices, S_*, which is scheduled to perform update only before time t_*. Then the accumulated values on the vertices of S_*, $\check{v}_{S_*}$, will not change since then. While they might receive values from other vertices, *i.e.*, $||\check{v}_{S_*} \oplus \Delta\check{v}_{S_*} - \check{v}_{S_*}||_1$ might become larger. On the other hand, the other vertices $(V - S_*)$ continue to perform the update operation, the received values on them, $\Delta\check{v}_{V-S_*}$, are accumulated to $\check{v}_{V-S_*}$ and propagated to other vertices again. As long as the iteration converges, the difference between the results of two consecutive updates, $||\check{v}_{V-S_*} \oplus \Delta\check{v}_{V-S_*} - v_{V-S_*}||_1$ should decrease "steadily" to 0. Therefore, eventually at some point,

$$\frac{||\check{v}_{S_*} \oplus \Delta\check{v}_{S_*} - \check{v}_{S_*}||_1}{|S_*|} > ||\check{v}_{V-S_*} \oplus \Delta\check{v}_{V-S_*} - \check{v}_{V-S_*}||_1. \tag{7}$$

That is,

$$\max_{j \in S_*}(\check{v}_j \oplus \Delta\check{v}_j - \check{v}_j) > \max_{j \in V-S_*}(\check{v}_j \oplus \Delta\check{v}_j - \check{v}_j). \tag{8}$$

Since the vertex that has the largest $(\check{v}_j \oplus \Delta\check{v}_j - \check{v}_j)$ should be scheduled under priority scheduling, a vertex in S_* should be scheduled at this point, which contradicts with the assumption that any vertex in S_* is not scheduled after time t_*. □

Then, with the support of Lemma 5 and Theorem 1, we have Theorem 3.

Theorem 3: By priority scheduling, $\check{v}'_j$ in (3) converges to the same fixed point v_j^* as v_j in (**??**) converges to, *i.e.*, $\check{v}'^\infty_j = v_j^\infty = v_j^*$.

3 DAIC ALGORITHMS

In this section, we show a broad class of DAIC algorithm examples.

3.1 Single Source Shortest Path

The *single source shortest path* algorithm (SSSP) has been widely used in online social networks and web mapping. Given a source node s, the algorithm derives the shortest distance from s to all the other nodes on a directed weighted graph. Initially, each node j's distance d_j^0 is initialized to be ∞ except that the source s's distance d_s^0 is initialized to be 0. In each iteration, the shortest distance from s to j, d_j, is updated with the following update function:

$$d_j^k = \min\{d_1^{k-1} + A(1,j), d_2^{k-1} + A(2,j), \dots, \\ d_n^{k-1} + w(n,j), d_j^0\},$$

where $A(i,j)$ is the weight of an edge from node i to node j, and $A(i,j) = \infty$ if there is no edge between i and j. The update process is performed iteratively until convergence, where the distance values of all nodes no longer change.

Following the guidelines, we identify that operator '$\oplus$' is 'min', function $g_{\{i,j\}}(d_i) = d_i + A(i,j)$. Apparently, the function $g_{\{i,j\}}(x)$ has the distributive property, and the operator 'min' has the commutative and associative properties. The initialization can be $d_j^0 = \infty$

and $\Delta d_j^1 = 0$ if $j = s$, or else $\Delta d_j = \infty$. Therefore, SSSP can be performed by DAIC. Further, suppose Δd_j is used to accumulate the received distance values by 'min' operation, the scheduling priority of node j would be $d_j - \min\{d_j, \Delta d_j\}$.

3.2 Linear Equation Solvers

Generally, DAIC can be used to solve systems of linear equations of the form

$$A \cdot \chi = b,$$

where A is a sparse $n \times n$ matrix with each entry A_{ij}, and χ, b are size-n vectors with each entry χ_j, b_j respectively.

One of the linear equation solvers, *Jacobi method*, iterates each entry of χ as follows:

$$\chi_j^k = -\frac{1}{A_{jj}} \cdot \sum_{i \neq j} A_{ji} \cdot \chi_i^{k-1} + \frac{b_j}{A_{jj}}.$$

The method is guaranteed to converge if the spectral radius of the iteration matrix is less than 1. That is, for any matrix norm $|| \cdot ||$, $\lim_{k\to\infty} ||B^k||^{\frac{1}{k}} < 1$, where B is the matrix with $B_{ij} = -\frac{A_{ij}}{A_{ii}}$ for $i \neq j$ and $B_{ij} = 0$ for $i = j$.

Following the guidelines, we identify that operator '$\oplus$' is '+', function $g_{\{i,j\}}(\chi_i) = -\frac{A_{ji}}{A_{jj}} \cdot \chi_i$. Apparently, the function $g_{\{i,j\}}(x)$ has the distributive property, and the operator '+' has the commutative and associative properties. The initialization can be $\chi_j^0 = 0$ and $\Delta\chi_j^1 = \frac{b_j}{A_{jj}}$. Therefore, the Jacobi method can be performed by DAIC. Further, suppose $\Delta\chi_j$ is used to accumulate the received delta message, the scheduling priority of node j would be $\Delta\chi_j$.

3.3 PageRank

The *PageRank* algorithm [4] is a popular algorithm proposed for ranking web pages. Initially, the PageRank scores are evenly distributed among all pages. In each iteration, the ranking score of page j, R_j, is updated as follows:

$$R_j = d \cdot \sum_{\{i|(i\to j)\in E\}} \frac{R_i}{|N(i)|} + (1-d), \tag{9}$$

where d is damping factor, $|N(i)|$ is the number of outbound links of page i, and E is the set of link edges. The iterative process terminates when the sum of changes of two consecutive iterations is sufficiently small. The initial guess of R_i can be any value. In fact, the final converged ranking score is independent from the initial value.

Following the guidelines, we identify that operator '$\oplus$' is '+', function $g_{\{i,j\}}(R_i) = d \cdot A_{i,j} \frac{R_i}{N(i)}$, where A represents the adjacency matrix and $A_{i,j} = 1$ if there is a link from i to j or else $A_{i,j} = 0$. Apparently, the function $g_{\{i,j\}}(x)$ function has distributive property and the operator '+' has the commutative and associative properties. The initialization can be $R_j^0 = 0$ and $\Delta R_j^1 = 1 - d$. Therefore, PageRank can be performed by DAIC. Further, suppose ΔR_j is used to accumulate the received PageRank values, the scheduling priority of node j would be ΔR_j.

3.4 Adsorption

Adsorption [5] is a graph-based label propagation algorithm that provides personalized recommendation for contents (e.g., video, music, document, product). The concept of *label* indicates a certain common feature of the entities. Given a weighted graph $G = (V, E)$, where V is the set of nodes, E is the set of edges. A is a column normalized matrix (i.e., $\sum_i A(i,j) = 1$) indicating that the sum of a node's inbound links' weight is equal to 1. Node j carries a probability distribution L_j on label set L, and each node j is initially assigned with an *initial distribution* I_j. The algorithm proceeds as follows. For each node j, it iteratively computes the weighted average of the label distributions from its neighboring nodes, and then uses the random walk probabilities to estimate a new label distribution as follows:

$$L_j^k = p_j^{cont} \cdot \sum_{\{i|(i\to j)\in E\}} A(i,j) \cdot L_i^{k-1} + p_j^{inj} \cdot I_j,$$

where p_j^{cont} and p_j^{inj} are constants associated with node j. If Adsorption converges, it will converge to a unique set of label distributions.

Following the guidelines, we identify that operator '$\oplus$' is '+', $g_{\{i,j\}}(L_i) = p_j^{cont} \cdot A(i,j) \cdot L_i$. Apparently, the function $g_{\{i,j\}}(x)$ has the distributive property, and the operator '+' has the commutative and associative properties. The initialization can be $L_j^0 = 0$ and $\Delta L_j^1 = p_j^{inj} \cdot I_j$. Therefore, Adsorption can be performed by accumulative updates. Further, suppose ΔL_j is used to accumulate the received distance values, the scheduling priority of node j would be ΔL_j.

3.5 SimRank

SimRank [6] was proposed to measure the similarity between two nodes in the network. It has been successfully used for many applications in social networks, information retrieval, and link prediction. In SimRank, the similarity between two nodes (or objects) a and b is defined as the average similarity between nodes linked with a and those with b. Mathematically, we iteratively update $s(a,b)$ as the similarity value between node a and b:

$$s^k(a,b) = \frac{C}{|I(a)||I(b)|} \sum_{c\in I(a), d\in I(b)} s^{k-1}(c,d),$$

where $s^1(a,b) = 1$ if $a = b$, or else $s^1(a,b) = 0$, $I(a) = b \in V|(b,a) \in E$ denoting all the nodes that have a link to a, and C is a decay factor satisfying $0 < C < 1$.

However, this update function is applied on node-pairs. It is not a vertex-centric update function. We should rewrite the update function. Cao *et. al.* has proposed *Delta-SimRank* [7]. They first construct a node-pair graph $G^2 = \{V^2, E^2\}$. Each node denotes one pair of nodes of the original graph. One node ab in G^2 corresponds to a pair of nodes a and b in G. There is one edge $(ab, cd) \in E^2$ if $(a, c) \in E$ and $(b, d) \in E$. If the graph size $|G| = n$, the node-pair graph size $|G^2| = n^2$. Let vertex j represent ab and vertex i represent cd. Then, the update function of a vertex $j \in G^2$ is:

$$s^k(j) = \frac{C}{|I(a)||I(b)|} \sum_{i \in I(j)} s^{k-1}(i),$$

where $I(a)$ and $I(b)$ denote the neighboring nodes of a and b in G respectively, and $I(j)$ denotes the neighboring nodes of j in G^2.

The new form of SimRank update function in the node-pair graph G^2 is vertex-centric. Following the DAIC guidelines, we identify that operator '$\oplus$' is '$+$', and function $g_{\{i,j\}}(s(i)) = \frac{C \cdot A(i,j)}{|I(a)||I(b)|} \cdot s(i)$, where $A_{i,j} = 1$ if $i \in I(j)$ (*i.e.*, $cd \in I(ab)$) or else $A_{i,j} = 0$. Apparently, the function $g_{\{i,j\}}(x)$ has the distributive property, and the operator '$+$' has the commutative and associative properties. The initialization of $s^0(j)$ can be $s^0(j) = s^0(ab) = 1$ if $a = b$, or else $s^0(j) = s^0(ab) = \sum_{c \in I(a) \& c \in I(b)} 1 = |I(a) \cap I(b)|$. The initialization of $\Delta s^1(j)$ can be $\Delta s^1(j) = \Delta s^1(ab) = 0$ if $a = b$, or else $\Delta s^1(j) = \Delta s^1(ab) = \frac{|I(a)||I(b)|}{C}$. Therefore, SimRank can be performed by DAIC. Further, suppose $\Delta s(j)$ is used to accumulate the received delta messages, the scheduling priority of node j would be $\Delta s(j)$.

3.6 Other Algorithms

We have shown several typical DAIC algorithms. Following the guidelines, we can rewrite them in DAIC form. In addition, there are many other DAIC algorithms. Table 1 of the main manuscript 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).

The *Connected Components* algorithm [8] finds connected components in a graph (the graph adjacency information is represented in matrix A, $A_{i,j} = 1$ if there is a link from i to j or else $A_{i,j} = 0$). Each node updates its component id with the largest received id and propagates its component id to its neighbors, so that the algorithm converges when all the nodes belonging to the same connected component have the same component id.

Hyperlink-Induced Topic Search (HITS) [9] ranks web pages in a web linkage graph W by a 2-phase iterative update, the *authority update* and the *hub update*. Similar to Adsorption, the authority update requires each node i to generate the output values damped by d and scaled by $A(i, j)$, where matrix $A = W^T W$, while the hub update scales a node's output values by $A'(i, j)$, where matrix $A' = W W^T$.

The *Katz metric* [10] is a proximity measure between two nodes in a graph (the graph adjacency information is represented in matrix A, $A_{i,j} = 1$ if there is a link from i to j, or else $A_{i,j} = 0$). It is computed as the sum over the collection of paths between two nodes, exponentially damped by the path length with a damping factor β.

Rooted PageRank [11] captures the probability for any node j running into node s, based on the node-to-node proximity, $A(j, i)$, indicating the probability of jumping from node j to node i.

```
template <class K, class D>
struct Partitioner {
  virtual void parse_line(string& line, K* vid, D* data) = 0;
  virtual int partition(const K& vid, int shards) = 0;
};

template <class K, class V, class D>
struct IterateKernel {
  virtual void init(const K& vid, V* c) = 0;
  virtual void accumulate(V* a, const V& b) = 0;
  virtual void send(const V& delta, const D& data,
                        list<pair<K, V> >* output) = 0;
};

template <class K, class V>
struct TermChecker {
  virtual double estimate_prog(LocalTableIterator<K, V>*
                                    table_itr) = 0;
  virtual bool terminate(list<double> local_reports) = 0;
};
```

Fig. 1. Maiter API summary.

4 MAITER API

Users implement a Maiter program using the provided API, which is written in C++ style. A DAIC algorithm is specified by implementing three functionality components, `Partitioner`, `IterateKernel`, and `TermChecker` as shown in Fig. 1.

`K`, `V`, and `D` are the template types of data element keys, data element values (v and Δv), and data element-associate data respectively. Particularly, for each entry in the state table, `K` is the type of the key field, `V` is the type of the v field/Δv field/priority field, and `D` is the type of the data field. The `Partitioner` reads an input partition line by line. The `parse_line` function extracts data element id and the associate data by parsing the given line string. Then the `partition` function applied on the key (e.g., a MOD operation on integer key) determines the host worker of the data element (considering the number of workers/shards). Based on this function, the framework will assign each data element to a host worker and determines a message's destination worker. In the `IterateKernel` component, users describe a DAIC algorithm by specifying a tuple ($g_{\{i,j\}}(x)$, $\oplus$, v_j^0, Δv_j^1). We initialize v_j^0 and Δv_j^1 by implementing the `init` interface; specify the '$\oplus$' operation by implementing the `accumulate` interface; and specify the function $g_{\{i,j\}}(x)$ by implementing the `send` interface

```
class PRPartitioner : public Partitioner<int,vector<int> >{

   void parse_line(string& line, int* key, vector<int>* data) {
      node  = get_source(line);
      adjlist = get_adjlist(line);

      *key = node;
      *data = adjlist;
   }

   int partition(const int& key, int shards) {
      return key % shards;
   }
}
```

Fig. 2. PageRankPartitioner implementation.

with the given Δv_i and data element i's associate data, which generates the output pairs $\langle j, g_{\{i,j\}}(\Delta v_i)\rangle$ to data element i's out-neighbors. To stop an iterative computation, users specify the `TermChecker` component. The local iteration progress is estimated by specifying the `estimate_prog` interface given the local state table iterator. The global terminator collects these local progress reports. In terms of these local progress reports, users specify the `terminate` interface to decide whether to terminate.

For better understanding, we walk through how the PageRank algorithm is implemented in Maiter [1]. Suppose the input graph file of PageRank is line by line. Each line includes a node id and its adjacency list. The input graph file is split into multiple slices. Each slice is assigned to a Maiter worker. In order to implement PageRank application in Maiter, users should implement three functionality components, `PRPartitioner`, `PRIterateKernel`, and `PRTermChecker`.

In `PRPartitioner`, users specify the `parse_line` interface and the `partition` interface. The implementation code is shown in Fig. 2. In `parse_line`, users parse an input line to extract the node id as well as its adjacency list and use them to initialize the state table's key field (`key`) and data field (`data`). In `partition`, users specify the partition function by a simple *mod* operation applied on the key field (`key`) and the total number of workers (`shards`).

In `PRIterateKernel`, users specify the asynchronous DAIC process by implementing the `init` interface, the `accumulate` interface, and the `send` interface. The implementation code is shown in Fig. 3. In `init`, users initialize node k's v field (`value`) as 0 and Δv field (`delta`) as 0.2. Users specify the `accumulate` interface by implementing the '$\oplus$' operator as '$+$' (*i.e.*, $a = a + b$). The `send` operation is invoked after each update of a node. In `send`, users generate the output messages (contained in `output`) based on the node's Δv value (`delta`) and data value (`data`).

In `PRTermChecker`, users specify the *estimate_prog* interface and the `terminate` interface. The implementation code is shown in Fig. 4. In `estimate_prog`, users compute the summation of v value in local state table. The *estimate_prog* function is invoked after each period of time. The resulted local sums from various workers are sent to the global termination checker, and then the `terminate` operation in the global termination checker is invoked. In `terminate`, based on these received local sums, users compute a global sum, which is considered as the iteration progress. It is compared with the previous iteration's progress to calculate a progress difference. The asynchronous DAIC is terminated when the progress difference is smaller than a pre-defined threshold.

1. More implementation example codes are provided at Maiter's Google Code website https://code.google.com/p/maiter/.

```
class PRIterateKernel : public IterateKernel<int, float, vector<int> > {

   void initialize(const int& k, float* value, float* delta){
      *value = 0;
      *delta = 0.2;
   }

   void accumulate(float* a, const float& b){
      *a = *a + b;
   }

   void send(const float& delta, const vector<int>& data,
                                  vector<pair<int, float> >* output){
      int size = (int) data.size();
      float outdelta = delta * 0.8 / size;
      for(vector<int>::const_iterator it=data.begin(); it!=data.end(); it++){
         int target = *it;
         output->push_back(make_pair(target, outdelta));
      }
   }
}
```

Fig. 3. PRIterateKernel implementation.

```
class PRTermChecker : public TermChecker<int, float> {

   double prev_prog = 0.0;
   double curr_prog = 0.0;

   double estimate_prog(LocalTableIterator<int, float>* statetable){
      double partial_curr = get_sum_v(statetable);
      return partial_curr;
   }

   bool terminate(list<double> local_sums){
      curr_prog += get_sum_v(local_sums);

      if(abs(curr_prog - prev_prog) < term_threshold){
         return true;
      }else{
         prev_prog = curr_prog;
         return false;
      }
   }
}
```

Fig. 4. PRTermChecker implementation

5 Supplementary Materials to Evaluation Section

This section provides supplementary materials to the evaluation section of the TPDS manuscript.

5.1 Frameworks For Comparison

Hadoop [12] is an open-source MapReduce implementation. It relies on HDFS for storage. Multiple map tasks process the distributed input files concurrently in the map phase, followed by that multiple reduce tasks process the map output in the reduce phase. Users are required to submit a series of jobs to process the data iteratively. The next job operates on the previous job's output. Therefore, two synchronization barriers exist in each iteration, between map phase and reduce phase and between Hadoop jobs. In our experiments, we use Hadoop 1.0.2.

iMapReduce [13] is built on top of Hadoop and provides iterative processing support. In iMapReduce (**iMR**), reduce output is directly passed to map rather than dumped to HDFS. More importantly, the iteration variant state data are separated from the static data. Only the state data are processed iteratively, where the costly and unnecessary static data shuffling is eliminated. The original iMapReduce stores data relying on HDFS. iMapReduce can load all data into memory for efficient data access and can store the intermediate data in files for better scalability. We refer to the memory-based iMapReduce as *iMR-mem* and the file-based iMapReduce as *iMR-file*.

Spark [14] was developed to optimize large-scale iterative and interactive computation. It uses caching techniques and operates in-memory read-only objects to improve the performance for repeated operations. The main abstraction in Spark is resilient distributed dataset (RDD), which maintains several copies of data across memory of multiple machines to support iterative algorithm recovery from failures. The read and write of RDDs is coarse-grained (*i.e.*, read or write a whole block of RDD), so the update of RDDs in iterative computation is coarse-grained. Besides, in Spark, the iteration variant state data can also be separated from the static data by specifying `partitionBy` and `join` interfaces. The applications in Spark can be written with Java or Scala. Spark is open-source and can be freely downloaded. In our experiments, we use Spark 0.6.2.

PrIter [15] enables prioritized iteration by modifying iMapReduce. It exploits the dominant property of some portion of the data and schedules them first for computation, rather than blindly performs computations on all data. The computation workload is dramatically reduced, and as a result the iteration converges faster. However, it performs the priority scheduling in each iteration in a synchronous manner. PrIter provides in-memory version (PrIter 0.1) as well as in-file version (PrIter 0.2). We refer to the memory-based PrIter as *PrIter-mem* and the file-based PrIter as *PrIter-file*.

Piccolo [16] is implemented with C++ and MPI, which allows to operate distributed tables. The iterative algorithm can be implemented by updating the distributed tables iteratively. The intermediate data are shuffled between workers continuously as long as some amount of the intermediate data are produced (fine-grained write), instead of waiting for the end of iteration and sending them together. The current iteration's data and the next iteration's data are stored in two global tables separately, so that the current iteration's data will not be overwritten. Piccolo can maintain the global table both in memory and in file. We only consider the in-memory version.

TABLE 1
Comparison of Distributed Frameworks

name	sep data	in mem	fine-g update	async iter	pri sched
Hadoop	×	×	×	×	×
iMR-file	✓	×	×	×	×
iMR-mem	✓	✓	×	×	×
Spark	✓	✓	×	×	×
PrIter-file	✓	×	×	×	✓
PrIter-mem	✓	✓	×	×	✓
Piccolo	✓	✓	✓	×	×
GraphLab-Sync	✓	✓	✓	×	×
GraphLab-AS-fifo	✓	✓	✓	✓	×
GraphLab-AS-pri	✓	✓	✓	✓	✓
Maiter-Sync	✓	✓	✓	×	×
Maiter-RR	✓	✓	✓	✓	×
Maiter-Pri	✓	✓	✓	✓	✓

GraphLab [17] supports both synchronous and asynchronous iterative computation with sparse computational dependencies while ensuring data consistency and achieving a high degree of parallel performance. It is also implemented with C++ and MPI. It first only supports the computation under multi-core environment exploiting shared memory (GraphLab 1.0). But later, GraphLab supports large-scale distributed computation under cloud environment (GraphLab 2.0). The static data and dynamic data in GraphLab can be decoupled and the update of vertex/edge state in GraphLab is fine-grained. Under asynchronous execution, several scheduling policies including FIFO scheduling and priority scheduling are supported in Graphlab. GraphLab performs a fine-grained termination check. It terminates a vertex's computation when the change of the vertex state is smaller than a pre-defined threshold parameter.

Table 1 summarizes these frameworks. These frameworks are featured by various factors that help improve performance, including separating static data from state data (sep data), in-memory operation (in mem), fine-grained update (fine-g update), asynchronous iteration (async iter), and the priority scheduling mechanism under asynchronous iteration engine (pri sched).

5.2 Data Set Description

We generate synthetic massive data sets for these algorithms. The graphs used for SSSP and Adsorption are weighted, and the graphs for PageRank and Katz metric are unweighted. The node ids are continuous integers ranging from 1 to size of the graph. We decide the in-degree of each node following log-normal distribution, where the log-normal parameters are ($\mu = -0.5$, $\sigma = 2.3$). Based on the in-degree of each node, we randomly pick a number of nodes to point to that node. For the weighted graph of SSSP computation, we use the log-normal parameters ($\mu = 0$, $\sigma = 1.0$) to generate the float weight of each edge following log-normal distribution. For the weighted graph of Adsorption computation, we use the log-normal parameters ($\mu = 0.4$, $\sigma = 0.8$) to generate the float weight of each edge following log-normal distribution. These log-normal parameters for these graphs are extracted from a few small real graphs downloaded from [18].

5.3 Termination Condition of the Experiments

To terminate iterative computation in PageRank experiment, we first run PageRank off-line to obtain a resulted rank vector, which is assumed to be the converged vector R^*. Then we run PageRank with different frameworks. We terminate the PageRank computation when the L1-Norm distance between the iterated vector R and the converged vector R^* is less than $0.001 \cdot N$, where N is the total number of nodes, *i.e.*, $\sum_j (|R_j^* - R_j|) < 0.001 \cdot N$. For the synchronous frameworks (*i.e.*, Hadoop, iMR-file, iMR-mem, Spark, PrIter-file, PrIter-mem, Piccolo, and Maiter-Sync), we check the convergence (termination condition) after every iteration. For the asynchronous frameworks (*i.e.*, Maiter-RR, and Maiter-Pri), we check the convergence every termination check interval. For GraphLab variants, we set the parameter of convergence tolerance as 0.001 to terminate the computation. Note that, the time for termination check in Hadoop and Piccolo (computing the L1-Norm distance through another job) has been excluded from the total running time, while the other frameworks provide termination check functionality.

For SSSP, the computation is terminated when there is no update of any vertex. For Adsorption and Katz metric, we use the similar convergence check approach as PageRank.

5.4 Comparison of Asynchronous Frameworks: Maiter vs. GraphLab

In this experiment, we focus on comparing Maiter with another asynchronous framework GraphLab. Even though GraphLab support asynchronous computation, as shown in Fig. 2 of the TPDS manuscript, it shows poor performance under asynchronous execution engine. Especially for priority scheduling, it extremely extends the completion time.

GraphLab relies on chromatic engine (partially asynchronously) and distributed locking engine (fully asynchronous) for scheduling asynchronous computation. Distributed locking engine is costly, even though many optimization techniques are exploited in GraphLab. For generality, the scheduling of asynchronous computation should guarantee the dependencies between computations. Distributed locking engine is proposed for the generality, but it becomes the bottleneck of asynchronous computation. Especially for priority scheduling, the cost of managing the scheduler tends to exceed the cost of the PageRank computation itself, which leads to very slow asynchronous Pagerank computation in GraphLab. Actually, GraphLab's priority scheduling policy is designed for some high-workload applications, such as Loopy Belief Propagation [19], in which case the asynchronous computation advantage is much more substantial.

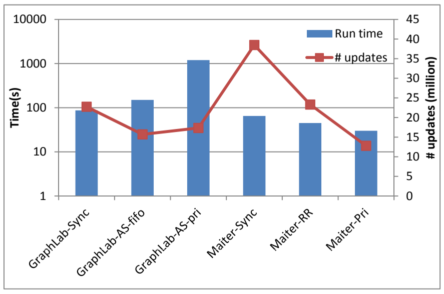

Fig. 5. Running time and number of updates of PageRank computation on GraphLab and Maiter.

To verify our analysis, we run PageRank on Maiter and GraphLab to compare the running time and the number of updates. The experiment is launched in the local cluster, and the graph dataset is the Google Webgraph dataset. Fig. 5 shows the result. In GraphLab, the number of performed updates under asynchronous engine (both fifo scheduling and priority scheduling) is less than that under synchronous engine, but the running time is longer. Under asynchronous engine, the number of updates by priority scheduling is similar to that by fifo scheduling, but the running time is extremely longer. Even though the workload is reduced, the asynchronous scheduling becomes an extraordinarily costly job, which slows down the whole process.

On the contrary, asynchronous DAIC exploits the cumulative operator '$\oplus$', which has commutative property and associative property. This implicates that the delta values can be accumulated in any order and at any time. Therefore, Maiter does not need to guarantee the computation dependency while allows all vertices to update their state totally independently. Round-robin scheduling, which performs computation on the local vertices in a round-robin manner, is the easiest one to implement (*i.e.*, with low overhead). Further, priority scheduling identifies the vertex importance and executes computation in their importance order, which can accelerate convergence. Both of them do not need to guarantee the global consistency and do not result in serious overhead. As shown in Fig. 5, round-robin scheduling and priority scheduling first reduce the workload (less number of updates), and as result shorten the convergence time.

5.5 Communication Cost

Distributed applications need high-volume communication between workers. The communication between workers becomes the performance bottleneck. Saving the communication cost correspondingly helps improve performance. By asynchronous DAIC, the iteration converges with much less number of updates, and as a result needs less communication.

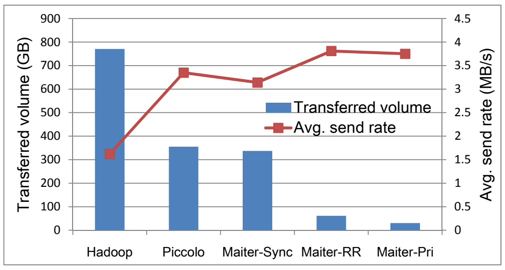

Fig. 6. Communication cost.

To measure the communication cost, we run PageRank on a 100-million-node synthetic graph on the EC2 cluster. We record the amount of data sent by each worker and sum these amounts of all workers to obtain the total volume of data transferred. Figure 6 depicts the total volume of data transferred in Hadoop, Piccolo, Maiter-Sync, Maiter-RR, and Maiter-Pri. We choose Hadoop for comparison for its generality and popularity. Hadoop mixes the iteration-variant state data with the static data and shuffles them in each iteration, which results in high volume communication. Piccolo can separate the state data from the static data and only communicate the state data. Besides, unlike the file-based transfer in Hadoop, Piccolo communicates between workers through MPI. As shown in the figure, Piccolo results in less transferred volume than Hadoop. Maiter-Sync utilizes msg tables for early aggregation to reduce the total transferred volume in a certain degree. By asynchronous DAIC, we need less number of updates and as a result less amount of communication. Consequently, Maiter-RR and Maiter-Pri significantly reduce the transferred data volume. Further, Maiter-Pri transfers even less amount of data than Maiter-RR since Maiter-Pri converges with even less number of updates. Maiter-RR and Maiter-Pri run significantly faster, and at the same time the amount of shuffled data is much less.

In Figure 6, we also show the average bandwidth that each worker has used for sending data. The worker in Maiter-RR and Maiter-Pri consumes about 2 times bandwidth than that in Hadoop and consumes only about 20% more bandwidth than the synchronous frameworks, Piccolo and Maiter-Sync. The average consumed bandwidth in asynchronous DAIC frameworks is a little higher. This means that the bandwidth resource in a cluster is highly utilized.

5.6 Scaling Performance When Increasing Input Size

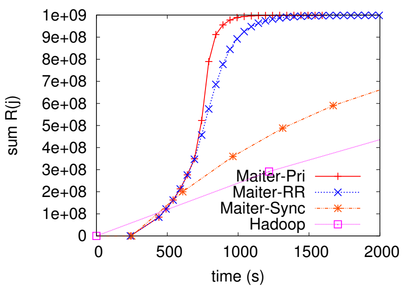

Fig. 7. Running time vs. progress metric of PageRank on a 1-billion-node synthetic graph.

In order to measure how Maiter scales with increasing input size, we perform PageRank for a 1-billion-node graph on the 100-node EC2 cluster. Maiter runs normally without any problem. Figure 7 shows the progress metric against the running time of Hadoop, Maiter-Sync, Maiter-RR, and Maiter-Pri. Since it will take considerable long time for PageRank convergence in Hadoop and Maiter-Sync, we only plot the progress changes in the first 2000 seconds. Maiter-Sync, Maiter-RR, and Maiter-Pri spend around 240 seconds in loading data in memory before starting computation. The PageRank computations in the asynchronous frameworks (Maiter-RR and Maiter-Pri) converge much faster than that in the synchronous frameworks (Hadoop and Maiter-Sync). In addition, to evaluate how large graph Maiter can process at most in the 100-node EC2 cluster, we continue to increase the graph size to contain 2 billion nodes, and it works fine with memory usage up to 84.7% on each EC2 instance.

REFERENCES

[1] D. Chazan and W. Miranker, "Chaotic relaxation," *Linear Algebra and its Applications*, vol. 2, no. 2, pp. 199 – 222, 1969.
[2] G. M. Baudet, "Asynchronous iterative methods for multiprocessors," *J. ACM*, vol. 25, pp. 226–244, April 1978.
[3] D. P. Bertsekas, "Distributed asynchronous computation of fixed points," *Math. Programming*, vol. 27, pp. 107–120, 1983.
[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] 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.
[6] G. Jeh and J. Widom, "Simrank: a measure of structural-context similarity," in *Proc. ACM Int'l Conf Knowledge Discovery and Data Mining (KDD '02)*, 2002, pp. 538–543.
[7] L. Cao, H.-D. Kim, M.-H. Tsai, B. Cho, Z. Li, and I. Gupta, "Delta-simrank computing on mapreduce," in *Proc. Int'l Workshop Big Data Mining (BigMine '12)*, 2012.

[8] U. Kang, C. Tsourakakis, and C. Faloutsos, “Pegasus: A peta-scale graph mining system implementation and observations,” in *Proc. IEEE Int'l Conf. Data Mining (ICDM '09)*, 2009, pp. 229 –238.
[9] J. M. Kleinberg, “Authoritative sources in a hyperlinked environment,” *J. ACM*, vol. 46, pp. 604–632, 1999.
[10] L. Katz, “A new status index derived from sociometric analysis,” *Psychometrika*, 1953.
[11] 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.
[12] Hadoop. http://hadoop.apache.org/.
[13] 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.
[14] 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.
[15] 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.
[16] 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.
[17] 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.
[18] Stanford dataset collection. http://snap.stanford.edu/data/.
[19] A. T. Ihler, J. W. Fischer III, and A. S. Willsky, “Loopy belief propagation: Convergence and effects of message errors,” *J. Mach. Learn. Res.*, vol. 6, pp. 905–936, Dec. 2005.