Accelerate Large-Scale Iterative Computation through Asynchronous Accumulative Updates

Yanfeng Zhang†‡, Qixin Gao†, Lixin Gao‡, Cuirong Wang†

†Northeastern University, China
‡University of Massachusetts Amherst
{yanfengzhang, lgao}@ecs.umass.edu; {gaoqx, wangcr}@neuq.edu.cn

ABSTRACT

Myriad of data mining algorithms in scientific computing require parsing data sets iteratively. These iterative algorithms have to be implemented in a distributed environment to scale to massive data sets. To accelerate iterative computations in a large-scale distributed environment, we identify a broad class of iterative computations that can accumulate iterative update results. Specifically, different from traditional iterative computations, which iteratively update the result based on the result from the previous iteration, accumulative iterative update accumulates the intermediate iterative update results. We prove that an accumulative update will yield the same result as its corresponding traditional iterative update. Furthermore, accumulative iterative computation can be performed asynchronously and converges much faster. We present a general computation model to describe asynchronous accumulative iterative computation. Based on the computation model, we design and implement a distributed framework, Maiter. We evaluate Maiter on Amazon EC2 Cloud with 100 EC2 instances. Our results show that Maiter achieves as much as 60x speedup over Hadoop for implementing iterative algorithms.

Categories and Subject Descriptors

C.2.4 [**Distributed Systems**]: Distributed applications

General Terms

Algorithms, Design, Theory, Performance

Keywords

asynchronous accumulative update, Maiter, iterative computation

1. INTRODUCTION

The advances in sensing, storage, and networking technology have created huge collections of high-volume, high-dimensional data. For example, with the success of Web 2.0 and the popularity of online social networks, huge amounts of data, such as Facebook activities, Flickr photos, Web pages, eBay sales records, and cell phone records, are being collected. Making sense of these data is critical for companies and organizations to make better business decisions, and even bring convenience to our daily life. Recent advances in scientific computing, such as computational biology, have led to a flurry of data analytic techniques that typically require an iterative refinement process [6, 24, 20, 9]. 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, distributed computing frameworks such as MapReduce and Dryad [12, 15, 2] have been proposed to perform large-scale data processing in a cluster of machines or in a cloud environment. Furthermore, a series of distributed frameworks [25, 22, 23, 10, 26] have been proposed for accelerating large-scale iterative computations. Common to these proposed distributed frameworks, iterative updates are performed iteration by iteration. Specifically, an iterative update in iteration k is performed after all updates in iteration $k-1$ have completed. Within the same iteration, the results from an earlier update cannot be used for a later update. While this is the traditional model for iterative computations, such a model does slow down the progress of the computation. First, the concurrent update model does not utilize the already retrieved partial results from the same iteration. This can prolong iteration process and lead to slow convergence. Second, a synchronization step is required for every iteration. This can be time consuming in a heterogeneous distributed environment.

In this paper, we propose to implement a class of iterative algorithms through accumulative updates. Instead of iteratively updating a new result with the old result, accumulative updates aggregate the intermediate results from both the previous iterations and the current iteration. Unlike traditional iterative computations, which require synchronizations between iterations, accumulative iterative computations allow to accumulate the intermediate iteration results asynchronously. While the asynchrony has been demonstrated to accelerate the convergence of iterative computations [14], the asynchronous accumulative updates can further accelerate an iterative process. We provide the sufficient conditions for the iterative algorithms that can be performed by accumulative iterative updates, and show that a large number of well-known algorithms satisfy these conditions. In addition, we present a computation model with a

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ScienceCloud'12, June 18, 2012, Delft, The Netherlands.
Copyright 2012 ACM 978-1-4503-1340-7/12/06 ...$10.00.

few abstract operations for describing an asynchronous accumulative iterative computation.

Based on the accumulative computation model, we design and implement a distributed framework, Maiter. Maiter relies on Message Passing Interface (MPI) for communication and provides intuitive API for users to specify their accumulative iterative computations. We systematically evaluate Maiter on Amazon EC2 Cloud [1]. Our results are presented in the context of four popular applications. The results show that Maiter can accelerate the convergence of 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 accumulative iterative updates, followed by a description of several example algorithms in Section 3. In Section 4, we describe Maiter design. The experimental results are shown in Section 5. We outline the related work in Section 6 and conclude the paper in Section 7.

2. ACCUMULATIVE ITERATIVE UPDATES

We first introduce the background of iterative computation and then describe the accumulative iterative updates.

2.1 Iteration Computation

Iterative algorithms typically perform the same operations on a data set for several iterations. The key of an iterative algorithm is the update function F:

$$v^k = F(v^{k-1}),$$

which is performed on an n-dimensional vector, $v^k = \{v_1^k, v_2^k, \ldots, v_n^k\}$, where v^{k-1} represents the $(k-1)^{\text{th}}$ iteration result. (Note that the iterative update function $F(v)$ might be a series of functions that are performed in tandem.) We can further represent $F(v^k)$ by a set of functions of the form $f_j(v_1, v_2, \ldots, v_n)$, each of which performs an update on an element j of vector v. That is,

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

In distributed computing, multiple processers perform the updates in parallel. For simplicity of exposition, assume that there are n processors and processor j performs an update for data element j (we will generalize this model in Section 2.3). All processors perform the update in lock steps. At step k, processor j first collects v_i^{k-1} from all processors, followed by performing the update function f_j based on v_i^{k-1}, $i = 1, 2, \ldots, n$. The main drawback of implementing synchronous iteration in a distributed fashion is that all the update operations in the $(k-1)^{\text{th}}$ iteration have to be completed before any of the update operations in the kth iteration starts. Clearly, 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 [11]. Performing update operations asynchronously means that processor j performs the update

$$v_j = f_j(v_1, v_2, \ldots, v_n)$$

at any time based on the most recent values of all data elements, $\{v_1, v_2, \ldots, v_n\}$. The conditions of convergence of asynchronous iterations have been studied in [11, 7, 8].

By asynchronous iteration, as processor j is activated to perform an update, it "pulls" the values of data elements from the other processors, *i.e.*, $\{v_1, v_2, \ldots, v_n\}$, 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 processor needs to pull the values from all the other processors, but not all of them have been updated, or even worse none of them is updated. In that case, asynchronous iteration performs a meaningless 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 processor i updates v_i, it "pushes" v_i to every other processor j, and v_i is buffered as $B_{i,j}$ on processor j. When processor j is activated, it uses the buffered values $B_{i,j}$, $i = 1, 2, \ldots, n$, to update v_j. In this way, the redundant communications can be avoided. However, the "push-based" asynchronous iteration results in considerable memory overhead. Each processor has to buffer n values, and the system totally needs $O(n^2)$ space. We will introduce accumulative updates, which can be executed asynchronously while reducing memory consumption.

2.2 Accumulative Updates

Before giving the definition of accumulative updates, we first pose the condition of performing accumulative updates. The condition is that the iterative update function $v_j^k = f_j(v_1^{k-1}, v_2^{k-1}, \ldots, v_n^{k-1})$ can be represented by:

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

where $k = 1, 2, \ldots$, c_j is a constant, '$\oplus$' is an abstract operator, and $g_{\{i,j\}}(x)$ is a function denoting the impact from element i to element j. In other words, processor i pushes value $g_{\{i,j\}}(v_i)$ (instead of v_i) to processor j, and on processor j, these $g_{\{i,j\}}(v_i)$ from any processor i and c_j can be aggregated (by '$\oplus$' operation) to update v_j.

For example, the well-known PageRank algorithm iteratively updates an n-dimensional PageRank score vector R. 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 (1), where $c_j = 1 - d$, '$\oplus$' is '+', and if there is a link from page i to page j, $g_{\{i,j\}}(v_i^{k-1}) = d \cdot \frac{v_i^{k-1}}{|N(i)|}$, otherwise $g_{\{i,j\}}(v_i^{k-1}) = 0$.

Next, we derive the accumulative updates. Let Δv_j^k denote the "change" (in the '$\oplus$' operation manner) from v_j^{k-1} to v_j^k. That is,

$$v_j^k = v_j^{k-1} \oplus \Delta v_j^k. \quad (2)$$

In order to derive Δv_j^k we assume that

- function $g_{\{i,j\}}(x)$ has 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 (1) with $v_i^{k-2} \oplus \Delta v_i^{k-1}$, we have

$$\begin{aligned} v_j^k =& g_{\{1,j\}}(v_1^{k-2}) \oplus g_{\{1,j\}}(\Delta v_1^{k-1}) \oplus \ldots \oplus \\ & g_{\{n,j\}}(v_n^{k-2}) \oplus g_{\{n,j\}}(\Delta v_n^{k-1}) \oplus c_j. \end{aligned} \tag{3}$$

Further, let us assume that

- operator '$\oplus$' has the *commutative property*, *i.e.*, $x \oplus y = y \oplus x$;
- operator '$\oplus$' has the *associative property*, *i.e.*, $(x \oplus y) \oplus z = x \oplus (y \oplus z)$;
- operator '$\oplus$' has the *identity property* of abstract value $\mathbf{0}$, *i.e.*, $x \oplus \mathbf{0} = x$.

Then we can combine these $g_{\{i,j\}}(v_i^{k-2})$, $i = 1, 2, \ldots, n$, and c_j in Equation (3) to obtain v_j^{k-1}. Considering Equation (2), the combination of the remaining $g_{\{i,j\}}(\Delta v_i^{k-1})$, $i = 1, 2, \ldots, n$, results in Δv_i^k. Therefore, We have the following two-step *accumulative updates*:

$$\begin{cases} v_j^k = v_j^{k-1} \oplus \Delta v_j^k, \\ \Delta v_j^{k+1} = \sum_{i=1}^{n} \oplus g_{\{i,j\}}(\Delta v_i^k), \end{cases} \tag{4}$$

where $k = 1, 2, \ldots$, and $\sum_{i=1}^{n} \oplus x_i = x_1 \oplus x_2 \oplus \ldots \oplus x_n$. v_j^0 can be initialized to be any value, and v_j^1 can be computed based on Equation (1). Then, we can obtain $\Delta v_j^1 = v_j^1 - v_j^0$ for the accumulative updates in (4). As a simple initialization, suppose $v_j^0 = \mathbf{0}$, then $v_j^1 = c_j$ and $\Delta v_j^1 = v_j^1 - v_j^0 = c_j$.

The accumulative updates can be described as follows. Processor j first updates v_j^k by accumulating Δv_j^k (by '$\oplus$' operation) and then updates Δv_j^{k+1} with $\sum_{i=1}^{n} \oplus g_{\{i,j\}}(\Delta v_i^k)$. Δv_j^{k+1} will be used for the $(k+1)^{\text{th}}$ update. $\sum_{i=1}^{n} \oplus g_{\{i,j\}}(\Delta v_i^k)$ is the accumulation of the received values from all processors since the kth update. Apparently, this still requires all processors to start the update synchronously when all the processors have received these values. That is, Δv_j^{k+1} has accumulated all the values $g_{\{i,j\}}(\Delta v_i^k)$, $i = 1, 2, \ldots, n$, at which time it is ready to be used in the $(k+1)^{\text{th}}$ iteration. Therefore, we refer to the updates in (4) as *synchronous accumulative updates*.

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

$$\begin{aligned} &\texttt{receive:} \begin{cases} \texttt{Whenever a value } m_j \texttt{ is received,} \\ \quad \Delta \check{v}_j \leftarrow \Delta \check{v}_j \oplus m_j. \end{cases} \\ &\texttt{update:} \begin{cases} \check{v}_j \leftarrow \check{v}_j \oplus \Delta \check{v}_j; \\ \texttt{For any } h\texttt{, if } g_{\{j,h\}}(\Delta \check{v}_j) \neq \mathbf{0}, \\ \quad \texttt{send value } g_{\{j,h\}}(\Delta \check{v}_j) \texttt{ to processor } h; \\ \Delta \check{v}_j \leftarrow \mathbf{0}, \end{cases} \end{aligned} \tag{5}$$

where $\check{v}_j$ is initialized to be $\mathbf{0}$, $\Delta \check{v}_j$ is initialized to be c_j, and m_j is the received value $g_{\{i,j\}}(\Delta \check{v}_i)$ sent from any processor i. The *receive* operation accumulates the received value m_j to $\Delta \check{v}_j$, which stores the accumulated received value between consecutive update operations. The *update* operation updates $\check{v}_j$ by accumulating $\Delta \check{v}_j$, sends value $g_{\{j,h\}}(\Delta \check{v}_j)$ to processor h, and resets $\Delta \check{v}_j$ to $\mathbf{0}$. To avoid useless communication, it is also necessary to check that $g_{\{j,h\}}(\Delta \check{v}_j) \neq \mathbf{0}$ before sending. For example, in PageRank, each page j has a buffer ΔR_j to accumulate the received PageRank scores. When page j performs an update, R_j is updated by accumulating ΔR_j. Then, $d \frac{\Delta R_j}{|N(j)|}$ is sent to page j's linked pages, and ΔR_j is reset to 0.

By asynchronous accumulative updates, the two operations, receive and update, on a processor are completely independent from those on other processors. Any processor is allowed to perform the operations at any time. There is no lock step to synchronize any operation between processors. Moreover, implementing asynchronous accumulative iteration only needs two buffers for storing $\check{v}_j$ and $\Delta \check{v}_j$. The space complexity of the system is $O(n)$. Therefore, accumulative iteration provides a memory-efficient solution for asynchronous iteration.

To study the convergence property, we first give the following definition of convergence of asynchronous accumulative iterative computation.

DEFINITION 1. *Asynchronous accumulative iterative computation as shown in (5) converges as long as that after each element has performed the receive and update operations an infinite number of times, $\check{v}_j$ converges to a fixed value $\check{v}_j^\infty$.*

Then, we have the following theorem to guarantee that by asynchronous accumulative updates the iterative computation converges to the correct result.

THEOREM 1. *If v_j in (1) converges, $\check{v}_j$ in (5) converges. Further, they converge to the same value, i.e., $v_j^\infty = \check{v}_j^\infty$.*

We can explain the intuition behind Theorem 1 as follows. Consider the accumulative iteration process as information propagation in a graph with each processor as a node. Node i with initial value c_i propagates value $g_{\{i,j\}}(c_i)$ to its "neighboring" node j, where value $g_{\{i,j\}}(c_i)$ is accumulated to v_j and value $g_{\{j,h\}}(g_{\{i,j\}}(c_i))$ is produced and propagated to its "neighboring" node h. By synchronous accumulative iteration, the values propagated from all nodes should be received by all their neighboring nodes before starting the next round propagation. That is, the values originated from a node are propagated strictly hop by hop. In contrast, by asynchronous accumulative iteration, whenever some values arrive, a node accumulates them to $\check{v}_j$ and propagates the newly produced values to its neighbors. No matter synchronously or asynchronously, the spread values are never lost, and the values originated from each node will be eventually spread along all paths. For a destination node, it will eventually collect the values originated from all nodes along various propagation paths. Therefore, synchronous iteration and asynchronous iteration will converge to the same result. The formal proof of Theorem 1 is provided in the extended version of this paper [28].

2.3 Optimal Scheduling

In reality, a subset of data elements are assigned to a processor, and multiple processors run in parallel. In each processor, computation resources can be assigned to each element in equal portions and in circler order, which is referred

to as *round-robin scheduling.* Moreover, it is possible to schedule the updates of these local elements dynamically by identifying their importance, which is referred to as *priority scheduling.* In our previous work [27], we have proposed that selectively processing a subset of the data elements has the potential of accelerating the convergence of iterative computation. Some of the data elements can play an important decisive role in determining the final converged outcome. Giving an update execution priority to some of the data elements can accelerate the convergence. For example, the well-known shortest path algorithm, Dijkstra's algorithm, greedily expands the node with the shortest distance first, which allows it to derive the shortest distance of all nodes fast.

In order to show the iteration 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 accumulative iterative computation either monotonically increases or monotonically decreases $||\check{v}||_1$ to a fixed point $||v^*||_1$. According to (5), an update of element 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, data element $j = arg\max_j |\check{v}_j \oplus \Delta\check{v}_j - \check{v}_j|$ is scheduled first. In other words, The bigger $|\check{v}_j \oplus \Delta\check{v}_j - \check{v}_j|$ is, the higher update priority data element j has. For example, in PageRank, we set each page j's scheduling priority value based on $|R_j + \Delta R_j - R_j| = \Delta R_j$. Then, we will schedule page j with the largest ΔR_j first.

Furthermore, the following lemma combining with Theorem 1 guarantees the convergence under the priority scheduling. The proof of Lemma 1 can be found in the extended version of this paper [28].

Lemma 1. By priority scheduling, each element will be scheduled to perform the update an infinite number of times.

2.4 Summary

To sum up, as long as an iterative update function can be written as a series of distributive functions $g_{\{i,j\}}(x)$ combined by an operator '$\oplus$' that has the commutative property, the associative property, and the identity property of **0** as shown in Equation (1), the iterative computation can be performed by accumulative iterative updates. The accumulative iterative computation can be performed asynchronously as shown in (5). By asynchronous accumulative updates, the sent values of an update are applied immediately, and the following updates will be performed more efficiently based on the most up-to-date accumulated received values. Further, we have an optimal scheduling policy that selects data element $j = arg\max_j |\check{v}_j \oplus \Delta\check{v}_j - \check{v}_j|$ to perform the update first, which accelerates convergence.

We have designed a distributed framework to support the implementations of accumulative updates. But first, we will present a broad class of iterative algorithms that can be performed by accumulative updates.

3. EXAMPLE ALGORITHMS

In this section, we show how to perform iterative algorithms by accumulative iterative updates through a few 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:

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

where $w(i,j)$ is the weight of an edge from node i to node j, and $w(i,j) = \infty$ 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.

The update function of SSSP is in the form of Equation (1). Operator '$\oplus$' is 'min', function $g_{\{i,j\}}(x) = x + w(i,j)$, and $c_j = d_j^0$. Apparently, the function $g_{\{i,j\}}(x) = x + w(i,j)$ has the distributive property, and the operator 'min' has the commutative and associative properties and the identity property of ∞. Therefore, SSSP can be performed by accumulative updates. Further, if Δd_j is used to accumulate the received distance values by 'min' operation, the scheduling priority of node j is set as $d_j - \min\{d_j, \Delta d_j\}$.

3.2 Linear Equation Solvers

Generally, accumulative updates 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

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

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 $i = j$.

The update function of Jacobi method is in the form of Equation (1). Operator '$\oplus$' is '+', $g_{\{i,j\}}(x) = -\frac{a_{ji}}{a_{jj}} \cdot x$, and $c_j = \frac{b_j}{a_{jj}}$. Apparently, the function $g_{\{i,j\}}(x) = -\frac{a_{ji}}{a_{jj}} \cdot x$ has the distributive property, and the operator '+' has the commutative and associative properties and the identity property of 0. Therefore, the Jacobi method can be performed by accumulative updates. Further, if $\Delta\chi_j$ is used to accumulate the received values, the scheduling priority of node j is set as $\Delta\chi_j$.

3.3 Other Algorithms

Many other iterative algorithms can be performed by accumulative updates. Table 1 shows a list of such algorithms, and each of their update functions is represented with a tuple $(\{c_1, c_2, \ldots, c_n\}, g_{\{i,j\}}(x), \oplus, \mathbf{0})$. The Connected Components algorithm [17] finds connected components in a graph by letting each node propagate its component id to its neighbors. The component id of each node is initialized to be its node id. Each node updates its component id with the largest received id and propagates its component id, so that the algorithm converges when all nodes belonging to the same connected component have the same component

Table 1: A list of accumulative iterative algorithms

algorithm	c_j	$g_{\{i,j\}}(x)$	$\oplus$	$\mathbf{0}$
SSSP	$0\ (j = s)$ or $\infty\ (j \neq s)$	$x + w(i, j)$	min	$+\infty$
Connected Components	j	x	max	$-\infty$
PageRank	$1 - d$	$d \cdot \frac{x}{\lvert N(j)\rvert}$	+	0
Adsorption	$p_j^{inj} \cdot I_j$	$p_i^{cont} \cdot W(i, j) \cdot x$	+	0
HITS (*authority*)	1	$d \cdot A^T A(i, j) \cdot x$	+	0
Katz metric	$1\ (j = s)$ or $0\ (j \neq s)$	$\beta \cdot x$	+	0
Jacobi method	$\frac{b_j}{a_{jj}}$	$-\frac{a_{ji}}{a_{jj}} \cdot x$	+	0
Rooted PageRank	$1\ (j = s)$ or $0\ (j \neq s)$	$P(j, i) \cdot x$	+	0

id. *Adsorption* [6] is a graph-based label propagation algorithm that provides personalized recommendation for contents. Each 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. Each node 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. *Hyperlink-Induced Topic Search* (HITS) [19] ranks web pages in a web linkage graph A by a two-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 $A^T A(i, j)$, while the hub update scales a node's output values by $AA^T(i, j)$. The *Katz metric* [18] is a proximity measure between two nodes in a graph. 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* [24] captures the probability for any node j running into node s, based on the node-to-node proximity, $P(j, i)$, indicating the probability of jumping from node j to node i.

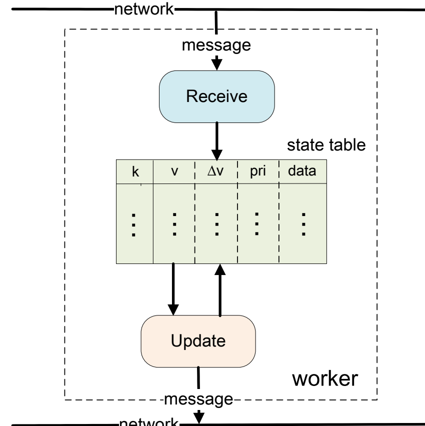


Figure 1: Worker overview.

4. MAITER PROTOTYPE

This section presents the design and implementation of Maiter prototype. Maiter is a message-passing framework that supports accumulative iterative updates [3], which is implemented by modifying Piccolo [23]. Maiter framework contains a master and multiple workers. The master coordinates the workers and monitors the status of workers. These workers run in parallel and communicate with each other through MPI. Each worker performs the updates for a subset of data elements. We first describe how to partition the data and how to load a data partition in each worker (Section 4.1), and present how to implement the receive and update operations (Section 4.2) and the scheduling policies (Section 4.3). Then, we show how the communication between workers is implemented with efficient message passing (Section 4.4). We will describe iteration termination in Section 4.5 and introduce Maiter API in Section 4.6.

4.1 Loading Data

Each worker loads a subset of data elements in memory for processing. Each data element is indexed by a global unique *key*. The assignment of a data element to a worker depends solely on the key. A data element with key j is assigned to worker $h(j)$, where $h()$ is a hash function applied on the key. For example, for distributing workload evenly, we can adopt a MOD operation on the continuous integer keys as the hash function.

The data elements in a worker are maintained in a local in-memory key-value store, *state table*. Each state table entry corresponds to a data element indexed by its key. As depicted in Figure 1, each table entry contains five fields. The first field stores the key j of a data element; the second field stores v_j; the third field stores Δv_j; the forth field stores the priority value of element j for priority scheduling; the fifth field stores the input data associated with element j, such as the adjacency information for node j.

Before iterative computation starts, the input data stored on HDFS are partitioned into multiple shards, and each of them is assigned to a worker. Several workers parse the input data in parallel, and the input data of element j is sent to worker $h(j)$. Then worker $h(j)$ fills the data field of entry j (*i.e.*, the entry with key field being j) with the assigned input data of element j. Users are responsible for initializing the v fields (*i.e.*, $v_j = \mathbf{0}$) and the Δv fields (*i.e.*, $\Delta v_j = c_j$) through the provided API (will be described in Section 4.6). The priority fields are automatically initialized based on the values of the v field and Δv field.

4.2 Receive Thread and Update Thread

As described in (5), asynchronous accumulative iteration 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 elements. Each worker receives messages from other workers and updates the Δv fields by accumulating the received messages. The update thread performs the update operation for all local elements. When operating on a data element, it updates the corresponding entry's v field and Δv field, and sends messages to other elements. The update of the priority field will be discussed in Section 4.3. The data field stores the input data, which is never changed during the iterative computation.

The receive thread writes on the Δv field, while the update thread both reads and writes on the Δv field. In order to avoid the read-write and write-write conflict risks on a table entry's Δv field, the update operation on a table entry has to be atomic, where the read and write on the Δv field are implemented in critical section. The update thread selects the table entries to perform the update according a scheduling policy. We will describe the scheduling policies and their implementations in the next subsection.

4.3 Scheduling within Update Thread

The simplest scheduling policy is to schedule the local data elements 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 2.3, 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 keys that have larger priority values. The update thread dequeues the key 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 data elements in the priority queue have been processed, the update thread extracts a new subset of high-priority keys for next round update. The extraction of keys is in terms of the priority field of each table entry. 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 message values are applied on this entry).

The number of extracted keys 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. We have provided an optimal queue size analysis in our previous work [27], which assumes a synchronous execution environment. 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 in the state table for processing. In addition, we also use the sampling technique proposed in [27] for efficient queue extraction, which only needs $O(N)$ time, where N is the local state table size.

4.4 Message Passing

Maiter uses OpenMPI [4] to implement message passing between workers. A message contains a key indicating the message's destination element and a message value. Suppose that a message's destination element key is k. The message will be sent to worker $h(k)$, where $h()$ is the partition function for data partition (Section 4.1), so the message will be received by the worker where the destination element resides.

A naive implementation of message passing is to send the output messages as soon as they are produced. However, initializing message passing leads to system overhead. To reduce this overhead, Maiter buffers the output messages and flushes them to remote workers after a short timeout. If a message's destination worker is the host worker, the output message is directly applied to the local state table.

The output messages are buffered in multiple *msg table*s, each of which corresponds to a remote destination worker. The reason why Maiter exploits this table buffer design is that we can leverage early aggregation to reduce network communications. Each msg table entry consists of a destination key field and a value field. As mentioned in Section 2.2, 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 key can be aggregated at the sender side or at the receiver side. Therefore, Maiter worker combines (by '$\oplus$' operation) the output messages with the same key in a msg table entry before sending them.

4.5 Iteration Termination

To terminate iteration, Maiter exploits *progress estimator* in each worker and a global *terminator* in the master. The master periodically broadcasts a *termination check 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 key and v fields of the local table entries to HDFS, which are the converged results. Note that, even though we exploit a synchronous termination check periodically, it will not impact the asynchronous computation. The workers proceed the iterative computation after producing the local progress reports without waiting for the master's feedback.

A commonly used termination check approach compares the two consecutive global iteration progresses. If the difference between them is minor enough, the iteration is terminated. For example, to terminate the SSSP computation, the progress estimator in each worker calculates the sum of the v field values (the sum of the shortest distance values of all the local nodes) and sends a report with the summed value to the master. Based on these local sums, the terminator in the master calculates a global sum, which indicates the iteration progress. If there is no change between the two global sums collected during a termination check period (*i.e.*, no node's distance is changed during that period), the SSSP computation is considered converged and is terminated.

4.6 Maiter API

Users can implement a Maiter program using the provided API, which is written in C++ style. The accumulative iterative computation in Maiter are specified by implementing three functionality components, including `Partitioner`, `IterateKernel`, and `TermChecker` as shown in Figure 2.

`K`, `V`, and `D` are the template types of element keys, element values, and element data respectively. Particularly, for an 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 a partition of input file line by line. The `parse_line` function extracts element key and element data by parsing the given

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

template <class K, class V, class D>
struct IterateKernel {
  virtual void init_c(const K& k, V* delta) = 0;
  virtual const V& default_v() const = 0;
  virtual void accumulate(V* a, const V& b) = 0;
  virtual void g_func(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;
};
```

Figure 2: Maiter API summary.

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 shards/workers). The framework will assign each element data to a host worker based on this function. It is also used for determining a message's destination worker. In the `IterateKernel` component, users specify a tuple ($\{c_1, c_2, \ldots, c_n\}$, $g_{\{i,j\}}(x)$, $\oplus$, $\mathbf{0}$) for describing an accumulative iterative computation. Specifically, we initialize each data element j's constant value c_j (*i.e.*, the initial value of Δv field) by implementing the `init_c` interface; specify the '$\oplus$' operation by implementing the `accumulate` interface; specify the identity element $\mathbf{0}$ by implementing the `default_v` interface; and specify the function $g_{\{i,j\}}(x)$ by implementing the `g_func` interface with the given Δv_i and element i's adjacency information, which generates the output pairs $\langle j, g_{\{i,j\}}(\Delta v_i)\rangle$ to any element j that is in element i's adjacency list. 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. The application implementation examples can be found at [3].

5. EVALUATION

We evaluate Maiter on Amazon EC2 Cloud with 100 medium instances, each with 1.7GB memory and 5 EC2 compute units [1]. We compare Maiter with Hadoop, iMapReduce, PrIter and Piccolo in the context of four applications, SSSP, PageRank, Adsorption, and Katz metric.

5.1 Preparation

We first review a few related frameworks used for comparison and generate the synthetic data sets for the applications.

5.1.1 Related Frameworks

Hadoop [2] is an open-source MapReduce implementation. It relies on HDFS to 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.

iMapReduce [26] 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. We also consider an in-memory version of iMapReduce (**iMR-mem**), which loads data into memory for efficient data access.

PrIter [27] enables prioritized iteration. 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 optimal scheduling in each iteration in a synchronous manner.

Piccolo [23] allows to operate distributed tables, where iterative algorithm can be implemented by updating the distributed tables iteratively. Even though synchronous iteration is required, these workers can communicate asynchronously. That is, the intermediate data are shuffled between workers continuously as long as some amount of the intermediate data are produced, 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.

To evaluate Maiter with different scheduling policies, we consider the round robin scheduling (**Maiter-RR**) as well as the optimal priority scheduling (**Maiter-Pri**). For Maiter-Pri, we set the priority queue size as 1% of the state table size (Section 4.3). In addition, we manually add a synchronization barrier controlled by the master to let these workers perform iteration synchronously, and we also let these workers only shuffle their msg tables at the end of an iteration, which maximizes the advantage of early aggregation but loses the benefit of asynchronous communication. We call this version of Maiter as **Maiter-Sync**.

Table 2 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), early aggregation (early agg.), asynchronous communication (asyn. com.), asynchronous iteration (asyn. iter.), and optimal scheduling (opt. sched.). Note that, Hadoop provides early aggregation option. The map output can be compressed by Combiner before it is sent to reduce, but we turn it off in order to clearly see the effects of these factors.

5.1.2 Synthetic Data Generation

Four algorithms described in Section 3 are implemented, including SSSP, PageRank, Adsorption, and Katz metric. 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 follow-

Table 2: A list of frameworks

name	sep. data	in mem	early agg.	asyn. com.	asyn. iter.	opt. sched.
Hadoop	×	×	×	×	×	×
iMR	✓	×	×	×	×	×
iMR-mem	✓	✓	×	×	×	×
PrIter	✓	✓	×	×	×	✓
Piccolo	✓	✓	×	✓	×	×
Maiter-Sync	✓	✓	✓	×	×	×
Maiter-RR	✓	✓	✓	✓	✓	×
Maiter-Pri	✓	✓	✓	✓	✓	✓

ing 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 [5].

5.2 Running Time to Convergence

We first show the running time to convergence of these applications. For Hadoop, iMR, iMR-mem, Piccolo, and Maiter-Sync, we check the convergence (termination condition) after every iteration. For PrIter, Maiter-RR, and Maiter-Pri, we check the convergence every termination check interval (10 seconds for SSSP, and 20 seconds for PageRank, Adsorption and Katz metric). For SSSP, it is terminated when there is no change between two consecutive iteration results (Hadoop, iMR, iMR-mem, Piccolo, and Maiter-Sync) or when there is no change of the iteration progress during a 10-second termination check interval (PrIter, Maiter-RR, and Maiter-Pri). For PageRank, Adsorption, and Katz metric, we start these iterative computations with the same initial values. We first run these applications off-line for 200 iterations to obtain a resulted vector, which is assumed to be the converged vector v^*. We terminate these applications when the difference between the checked resulted vector v (after every iteration or after a time interval) and the converged vector v^* is less than $0.0001 \cdot ||v^*||$, *i.e.*, $\sum_j v_j^* - \sum_j v_j < 0.0001 \cdot \sum_j v_j^*$. Note that, for fairness, the time for summing the values of all keys (through another job) in Hadoop and Piccolo, has been excluded from the total running time (other frameworks provide termination check functionality). The data loading time for the memory-based systems (other than Hadoop) is included in the total running time. The same is true for all experiments described in this section.

Figure 3 shows the running time to convergence of SSSP on a 100-million-node graph under these frameworks. By separating the iteration-variant state data from the static data, iMR reduces the running time of Hadoop by around 60%. iMR-mem further reduces it by providing faster memory access. PrIter identifies the more important elements to perform the update and ignores the useless updates, by which the running time is further reduced by half. Piccolo, with asynchronous communication without optimal scheduling, can also reduce the running time of iMR-mem a lot. The synchronous accumulative iteration framework, Maiter-Sync, with early aggregation but without asynchronous communication consumes similar running time with Piccolo. While the asynchronous accumulative iteration frameworks, Maiter-RR and Maiter-Pri, only need about 50 seconds to converge, which is more than 30x faster than Hadoop.

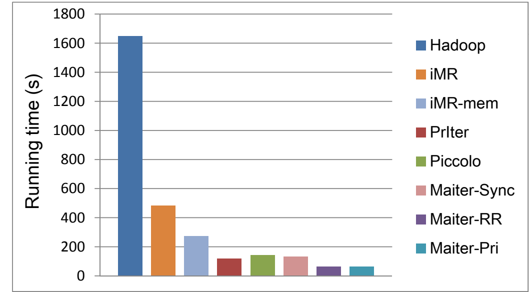


Figure 3: Running time of SSSP on different frameworks.

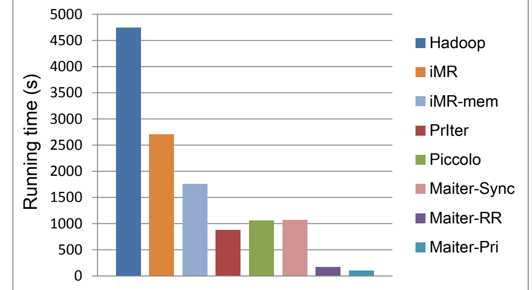
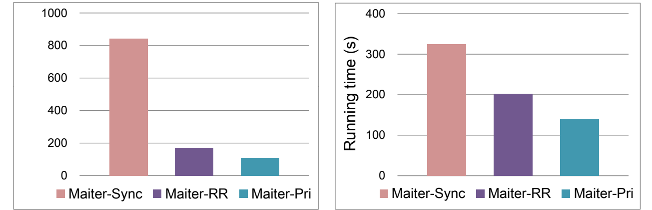


Figure 4: Running time of PageRank on different frameworks.



(a) Adsorption (b) Katz metric

Figure 5: Running time of Adsorption and Katz metric.

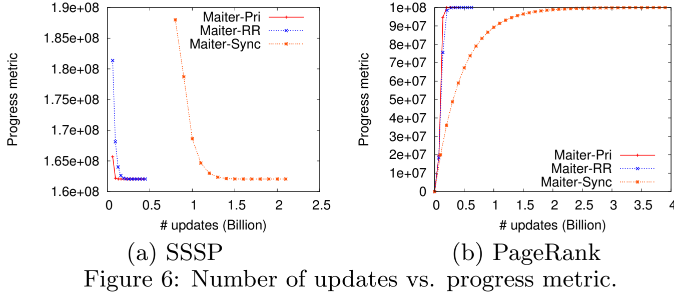
Figure 4 shows the running time to convergence of PageRank on a 100-million-node graph under these frameworks. We can see the similar results with SSSP. By asynchronous accumulative iteration, Maiter-RR is 30x faster than Hadoop and 7x faster than the synchronous accumulative iteration framework Maiter-Sync. Further, by optimal scheduling, Maiter-Pri further reduces the running time of Maiter-RR by around half.

Figure 5a and Figure 5b show the running time to convergence of Adsorption and Katz metric on 100-million-node graphs respectively under Maiter-Sync, Maiter-RR, and Maiter-Pri. We can see that Maiter-RR and Maiter-Pri significantly reduce the running time of Maiter-Sync, and Maiter-Pri further reduces the running time of Maiter-RR by around half.

5.3 Efficiency of Accumulative Updates

With the same number of updates, asynchronous accumulative iteration results in more progress than synchronous accumulative iteration. In this experiment, we measure the number of updates that SSSP and PageRank 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 $\sum_j d_j$ for SSSP and $\sum_j R_j$ for PageRank.

Then, the *efficiency* of accumulative updates can be seen as the ratio of the progress metric to the number of updates. For Maiter-Sync, the number of updates and the progress metric are measured after every iteration, while for Maiter-RR and Maiter-Pri, the number of updates and the progress metric are measured every termination check interval (10 seconds for SSSP and 20 seconds for PageRank).



(a) SSSP (b) PageRank

Figure 6: Number of updates vs. progress metric.

We run SSSP for a 500-million-node graph and PageRank for a 100-million-node graph. Figure 6a 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 $c_j = \infty$ for any node $j \neq s$, which cannot be drawn in the figure, we start plotting when any $d_j < \infty$. Figure 6b shows the progress metric against the number of updates for PageRank. In PageRank, the progress metric $\sum_j R_j$ should be increasing. Each R_j is initialized to be $c_j = 1 - d = 0.2$ (the damping factor $d = 0.8$). The progress metric $\sum_j R_j$ is increasing from $0.2 \cdot |v|$ to $|v|$, where $|v| = 10^8$ (number of pages). From Figure 6a and Figure 6b, we can see that by asynchronous accumulative updates, Maiter-RR and Maiter-Pri require much less updates to converge than Maiter-Sync. That is, the update in asynchronous accumulative updates is more effective than that in synchronous accumulative updates. Further, Maiter-Pri selects more effective updates to perform the update, so the update in Maiter-Pri is even more effective.

5.4 Communication Cost

Distributed applications need high-volume communication between workers, and the communications between workers become the bottleneck of distributed computations. Saving the communication cost correspondingly helps improve system performance. By asynchronous accumulative updates, the iteration converges with much less number of updates, and as a result needs less communication.

We run PageRank on a 100-million-node graph to measure the communication cost. 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 7 depicts the total volume of data transferred in Hadoop, Piccolo, Maiter-Sync, Maiter-RR, and Maiter-Pri. By Hadoop, we mix the iteration-variant state data with the static data and shuffle them in each iteration. While by Piccolo we can separate the state data from the static data and only communicate the state data. Accordingly, 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 accumulative updates, we need less number of updates and as a result need less amount of data to transfer. 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.

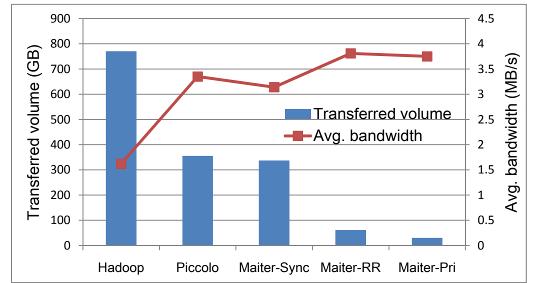


Figure 7: Communication cost.

Asynchronous accumulative iteration consumes more bandwidth to shuffle data. In Figure 7, 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. Despite Maiter-RR and Maiter-Pri run significantly faster, the amount of shuffled data is much less, and the average consumed bandwidth is not significantly higher.

6. RELATED WORK

The original idea of asynchronous iteration, chaotic iteration, was introduced by Chazan and Miranker in 1969 [11]. Motivated by that, Baudet proposed an asynchronous iterative scheme for multicore systems [7], and Bertsekas presented a distributed asynchronous iteration model [8]. These early stage studies laid the foundation of asynchronous iteration and have proved its effectiveness and convergence. Asynchronous methods are being increasingly studied since then, particularly so in connection with the use of heterogeneous workstation clusters.

On the other hand, to support iterative computation, a series of distributed frameworks have emerged. In addition to the frameworks described in Section 5.1.1, many other synchronous frameworks are proposed recently. HaLoop [10], a modified version of Hadoop, improves the efficiency of iterative computations by making the task scheduler loop-aware and employing caching mechanisms. Pregel [22] aims at supporting graph-based iterative algorithms by proposing a graph-centric programming model. Spark [25] uses a collection of read-only objects, which maintains several copies of data across memory of multiple machines to support iterative algorithm recovery from failures. Twister [13] employs a lightweight iterative MapReduce runtime system by logically constructing a reduce-to-map loop.

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 [16] investigates the notion of partial synchronizations in iterative MapReduce applications to overcome global synchronization overheads. It performs more frequent local synchronizations but with less frequent global synchronizations. GraphLab [21] supports asynchronous iterative computation with sparse computational dependencies while ensuring data consistency and achieving a high degree of parallel performance. Our work is the first that proposes to perform ac-

cumulative updates for iterative computations and identify a broad class of iterative computations that can perform asynchronous accumulative updates.

Our previous work [27] focused on prioritized execution of iterative computation. In order to illustrate the advantage of prioritized execution, we proved that asynchronous accumulative iteration derives the same result as synchronous iteration under only an example application, PageRank, and developed a framework, PrIter, which is Java implementation with synchronous iteration, synchronous communication and file-based transfer. In this paper, we ask the fundamental question: what kind of iterative computations can be performed with asynchronous accumulative updates? By answering this question, we extract an abstract model to identify a broad class of applications that can be performed with asynchronous accumulative updates. We further develop a framework, Maiter, which is C++ implementation with asynchronous iteration and MPI-based asynchronous communication.

7. CONCLUSIONS

We propose accumulative iterative computation and identifies a broad class of algorithms that can be performed by accumulative updates. Accumulative iterative computation can be performed asynchronously and converges with much less workload. To support accumulative computation, we design and implement Maiter, which relies on message passing to communicate between hundreds of distributed machines. We deploy Maiter on Amazon EC2 and evaluate its performance in the context of four iterative algorithms. The results show that by asynchronous accumulative updates the iterative computation performance is significantly improved.

Acknowledgment

This work is partially supported by U.S. NSF grants CCF-1018114. Yanfeng Zhang was a visiting student at UMass Amherst, supported by China Scholarship Council.

8. REFERENCES

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

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

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

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

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

[6] 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 *WWW '08*, pages 895–904, 2008.

[7] G. M. Baudet. Asynchronous iterative methods for multiprocessors. *J. ACM*, 25:226–244, April 1978.

[8] D. P. Bertsekas. Distributed asynchronous computation of fixed points. *Math. Programming*, 27:107–120, 1983.

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

[10] Y. Bu, B. Howe, M. Balazinska, and D. M. Ernst. Haloop: Efficient iterative data processing on large clusters. In *VLDB '10*, 2010.

[11] D. Chazan and W. Miranker. Chaotic relaxation. *Linear Algebra and its Applications*, 2(2):199 – 222, 1969.

[12] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. In *OSDI'04*, pages 10–10, 2004.

[13] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S.-H. Bae, J. Qiu, and G. Fox. Twister: a runtime for iterative mapreduce. In *MAPREDUCE '10*, pages 810–818, 2010.

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

[15] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *EuroSys '07*, pages 59–72, 2007.

[16] K. Kambatla, N. Rapolu, S. Jagannathan, and A. Grama. Asynchronous algorithms in mapreduce. In *CLUSTER' 10*, pages 245 –254, 2010.

[17] U. Kang, C. Tsourakakis, and C. Faloutsos. Pegasus: A peta-scale graph mining system implementation and observations. In *ICDM '09*, pages 229 –238, 2009.

[18] L. Katz. A new status index derived from sociometric analysis. *Psychometrika*, 18:39–43, 1953.

[19] J. M. Kleinberg. Authoritative sources in a hyperlinked environment. *J. ACM*, 46:604–632, September 1999.

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

[21] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. Graphlab: A new framework for parallel machine learning. In *UAI '10*, 2010.

[22] 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 *SIGMOD '10*, pages 135–146, 2010.

[23] R. Power and J. Li. Piccolo: Building fast, distributed programs with partitioned tables. In *OSDI'10*, pages 1–14, 2010.

[24] H. H. Song, T. W. Cho, V. Dave, Y. Zhang, and L. Qiu. Scalable proximity estimation and link prediction in online social networks. In *IMC '09*, pages 322–335, 2009.

[25] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: cluster computing with working sets. In *HotCloud'10*, 2010.

[26] Y. Zhang, Q. Gao, L. Gao, and C. Wang. imapreduce: A distributed computing framework for iterative computation. In *DataCloud '11*, pages 1112–1121, 2011.

[27] Y. Zhang, Q. Gao, L. Gao, and C. Wang. Priter: A distributed framework for prioritized iterative computations. In *SOCC '11*, 2011.

[28] Y. Zhang, Q. Gao, L. Gao, and C. Wang. Maiter: A message-passing distributed framework for accumulative iterative computation, http://rio.ecs.umass.edu/∼yzhang/maiter-full.pdf. Technical report, 2012.