Asynchronous Distributed Incremental Computation on Evolving Graphs

Jiangtao Yin(✉) and Lixin Gao

University of Massachusetts Amherst, USA
{jyin, lgao}@ecs.umass.edu

Abstract. Graph algorithms have become an essential component in many real-world applications. An essential property of graphs is that they are often dynamic. Many applications must update the computation result periodically on the new graph so as to keep it up-to-date. Incremental computation is a promising technique for this purpose. Traditionally, incremental computation is typically performed synchronously, since it is easy to implement. In this paper, we illustrate that incremental computation can be performed asynchronously as well. Asynchronous incremental computation can bypass synchronization barriers and always utilize the most recent values, and thus it is more efficient than its synchronous counterpart. Furthermore, we develop a distributed framework, GraphIn, to facilitate implementations of incremental computation on massive evolving graphs. We evaluate our asynchronous incremental computation approach via extensive experiments on a local cluster as well as the Amazon EC2 cloud. The evaluation results show that it can accelerate the convergence speed by as much as 14x when compared to recomputation from scratch.

1 Introduction

A large class of data routinely produced and collected by large corporations can be modeled as graphs, such as web pages crawled by Google (e.g., the web graph) and tweets collected by Twitter (e.g., the mention graph for users). Since graphs can capture complex dependencies and interactions, graph algorithms have become an essential component in many real-world applications [2, 8, 24], including business intelligence, social sciences, and data mining.

An essential property of graphs is that they are often dynamic. As new data and/or updates are being collected (or produced), the graph will evolve. For example, search engines will periodically crawl the web, and the web graph is evolving as web pages and hyper-links are created and/or deleted. Many applications must utilize the up-to-date graph in order to produce results that can reflect the current state. However, rerunning the computation over the entire graph is not efficient (considering the huge size of the graph), since it discards the work done in earlier runs no matter how little changes have been made.

The dynamic nature of graphs implies that performing incremental computation can improve efficiency dramatically. Incremental computation exploits

the fact that only a small portion of the graph has changed. It reuses the result of the prior computation and performs computation only on the part of the graph that is affected by the change. Although a number of distributed frameworks have been proposed to support incremental computation on massive graphs [3, 6, 15–17, 23], most of them apply synchronous updates, which require expensive synchronization barriers. In order to avoid the high synchronization cost, asynchronous updates have been proposed. In the asynchronous update model, a vertex performs the update using the most recent values instead of the values from the previous iteration (and there is no waiting time). Intuitively, we can expect asynchronous updates outperform synchronous updates since more up-to-date values are used and the synchronization barriers are bypassed. However, asynchronous updates might require more communications and perform useless computations (e.g., when no new value available to a vertex), and thus result in limited performance gain over synchronous updates.

In this paper, we provide an approach to efficiently apply asynchronous updates to incremental computation. We first describe a broad class of graph algorithms targeted by this paper. We then present our incremental computation approach through illustrating how to apply asynchronous updates to incremental computation. In order to address the challenge that asynchronous updates might require more communication and computation, we present a scheduling scheme to coordinate updates. Furthermore, we develop a distributed system to support our proposed asynchronous incremental computation approach. We evaluate our approach on a local cluster of machines as well as the Amazon EC2 cloud. More specifically, our main contributions are as follows:

- We propose an approach to efficiently apply asynchronous updates to incremental computation on evolving graphs for a broad class of graph algorithms. In order to improve efficiency, a scheduling scheme is presented to coordinate asynchronous updates. The convergence of our proposed asynchronous incremental computation approach is proved.
- We develop an asynchronous distributed framework, GraphIn, to support incremental computation. GraphIn eases the process of implementing graph algorithms with incremental computation in a distributed environment and does not require users to have the distributed programming experience.
- We extensively evaluate our asynchronous incremental computation approach with several real-world graphs. The evaluation results show that our approach can accelerate the convergence speed by as much as 14x when compared to recomputation from scratch. Moreover, a scalability test on a 50-machine cluster demonstrates our approach works with massive graphs having tens of millions of vertices and a billion of edges.

2 Problem Setting

In this section, we first define the problem of performing algorithms on evolving graphs. We then describe a broad class of graph algorithms which we target.

2.1 Problem Formulation

Many graph algorithms leverage iterative updates to compute states (e.g., scores of importance, closenesses to a specified vertex) of the vertices until convergence points are reached. For example, PageRank iteratively refines the rank scores of the vertices (e.g., web pages) of a graph. Such a graph algorithm typically starts with some initial state and then iteratively refines it until convergence. We refer to this kind of graph algorithms as *iterative graph algorithms*.

We are interested in how to efficiently perform iterative graph algorithms on evolving graphs. More formally, if we use G to denote the original graph and G' to represent the new graph, the question we ask is: for an iterative graph algorithm, given G' and the convergence point on G, how to efficiently reach the convergence point on G'.

2.2 Iterative Graph Algorithms

We here describe the iterative graph algorithms targeted by this paper. Typically, the update function of an iterative graph algorithm has the following form:

$$x^{(k)} = f(x^{(k-1)}), \tag{1}$$

where the n-dimensional vector $x^{(k)}$ presents the state of the graph at iteration k, each of its elements is the state for one vertex (e.g., $x^{(k)}[i]$ for vertex i), and $x^{(0)}$ is the initial state. A convergence point is a fixed point of the update function. That is, if $x^{(*)}$ is a convergence point, we have $x^{(*)} = f(x^{(*)})$.

The update function usually can be decomposed into a series of individual functions. In other words, we can update a vertex's state (e.g., x_j) as follows:

$$x_j^{(k)} = c_j \star \sum_{i=1}^{n} \star f_{\{i,j\}}(x_i^{(k-1)}), \tag{2}$$

where '$\star$' is an abstract operator ($\sum_{i=1}^{n} \star$ represents an operation sequence of length n by '$\star$'), c_j is a constant, and $f_{\{i,j\}}(x_i^{k-1})$ is an individual function denoting the impact from vertex i to vertex j in the k^{th} iteration. The operator '$\star$' typically has three candidates, '+', 'min', and 'max'. In this paper, we target the iterative graph algorithm that can compute the state in the form of Eq. (2).

2.3 Example Graph Algorithms

We next illustrate a series of well-known iterative graph algorithms, the update functions of which can be converted into the form of Eq. (2).

PageRank and Variants: PageRank is a well-known algorithm, which ranks vertices in a graph based on the stationary distribution of a random walk on the graph. Each element (e.g., r_j) of the score vector r can be computed iteratively as follows: $r_j^{(k)} = \sum_{\{i|\{i\to j\}\in E\}} \frac{dr_i^{(k-1)}}{|N(i)|} + (1-d)e_j$, where d $(0 < d < 1)$ is

the damping factor, $\{i \to j\}$ represents the edge from vertex i to vertex j, E is the set of edges, $|N(i)|$ is the number of outgoing edges of vertex i, and e is a size-n vector with each entry being $\frac{1}{n}$. We can convert the update function of PageRank into the form of Eq. (2). If $\{i \to j\} \in E$, $f_{\{i,j\}}(x_i^{(k-1)}) = dx_i^{(k-1)}/|N(i)|$, otherwise $f_{\{i,j\}}(x_i^{(k-1)}) = 0$, $c_j = (1-d)e_j$, and '$\star$' is '+'.

The update function of Personalized PageRank [9] differs from that of PageRank only at vector e. Vector e of Personalized PageRank assigns non-zero values only to the entries indicating the personally preferred pages. Rooted PageRank [19] is a special case of Personalized PageRank. It captures the probability for two vertices to run into each other and uses this probability as the similarity score of those two vertices.

Shortest Paths: The shortest paths algorithm is a simple yet common graph algorithm which computes the shortest distances from a source vertex to all other vertices. Given a weighted graph, $G = (V, E, W)$, where V is the set of vertices, E is the set of edges, and W is the weight matrix of the graph (if there is no edge between i and j, $W[i,j] = \infty$). Then the shortest distance (i.e., d_j) from the source vertex s to a vertex j can be calculated by performing the iterative updates: $d_j^{(k)} = \min\{d_j^{(0)}, \min_i(d_i^{(k-1)} + W[i,j])\}$. For the initial state, we usually set $d_s^{(0)} = 0$ and $d_j^{(0)} = \infty$ for any vertex j other than s. We can map the update function of the shortest paths algorithm into the form of Eq. (2). If there is an edge from vertex i to vertex j, $f_{\{i,j\}}(x_i^{(k-1)}) = x_i^{(k-1)} + W[i,j]$, otherwise $f_{\{i,j\}}(x_i^{(k-1)}) = \infty$, $c_j = d_j^{(0)}$, and '$\star$' is 'min'.

Connected Components: The connected components algorithm is an important algorithm for understanding graphs. It aims to find the connected components in a graph. The main idea of the algorithm is to label each vertex with the maximum vertex id across all vertices in the component which it belongs to. Initially, a vertex j sets its component id $p_j^{(0)}$ as its own vertex id, i.e., $p_j^{(0)} = j$. Then the component id of vertex j can be iteratively updated by $p_j^{(k)} = \max\{p_j^{(0)}, \max_{i \in N(j)}(p_i^{(k-1)})\}$, where $N(j)$ denotes vertex j's neighbors. When no vertex in the graph changes its component id, the algorithm converges. As a result, the vertices having the same component id belong to the same component. We can map the update function of the connected components algorithm into the form of Eq. (2). If there is an edge from vertex i to vertex j, $f_{\{i,j\}}(x_i^{(k-1)}) = x_i^{(k-1)}$, otherwise $f_{\{i,j\}}(x_i^{(k-1)}) = -\infty$, $c_j = j$, and '$\star$' is 'max'.

Other Algorithms: There are many more iterative graph algorithms, update functions of which can be mapped into the form of Eq. (2). We name several ones here. Hitting time is a measure based on a random walk on the graph. Penalized hitting probability [8] and discounted hitting time [18] are variants of hitting time. The adsorption algorithm [2] is a graph-based label propagation algorithm proposed for personalized recommendation. HITS [10] utilizes a two-phase iterative update approach to rank web pages of a web graph. SALSA [13] is another link-based ranking algorithm for web graphs. Effective Importance [4] is a proximity measure to capture the local community structure of a vertex.

3 Asynchronous Incremental Computation

As the underlying graph evolves, the states of the vertices also change. Obviously, rerunning the computation from scratch over the new graph is not efficient, since it discards the work done in earlier runs. Intuitively, performing computations incrementally can improve efficiency. In this section, we present our asynchronous incremental computation approach. The convergence of our approach is proved.

3.1 Asynchronous Updates

In order to describe our asynchronous incremental computation approach, we define a time sequence $\{t_0, t_1, \ldots, t_\infty\}$. Let $\hat{x}^{(k)}$ denote the state vector at time t_k. Also, we introduce the delta state vector $\Delta\hat{x}^{(k)}$ to represent the difference between $\hat{x}^{(k+1)}$ and $\hat{x}^{(k)}$ in the operator '$\star$' manner, i.e., $\hat{x}^{(k+1)} = \hat{x}^{(k)} \star \Delta\hat{x}^{(k)}$. The goal of introducing $\Delta\hat{x}^{(k)}$ is to perform accumulative computations. When the operator '$\star$' has the commutative property and the associative property and the function $f_{\{i,j\}}(x_i)$ has the distributive property over '$\star$', the computation can be performed accumulatively. All the graph algorithms discussed in Section 2.3 satisfy these properties. It is straightforward to verify that accumulative computations are equivalent to normal computations. The benefit of performing accumulative computations is that only changes of the states (i.e., delta states) are used to compute new changes. If there is no change for the state of a vertex, no communication or computation is necessary. The general idea of separating fixed parts from changes and leveraging changes to compute new changes also shows efficiency in many other algorithms, such as Nonnegative Matrix Factorization [21] and Expectation-Maximization [22].

In our asynchronous incremental computation approach, each vertex i updates its $\Delta\hat{x}_i^{(k)}$ and $\hat{x}_i^{(k)}$ independently and asynchronously, starting from $\Delta\hat{x}_i^{(0)}$ and $\hat{x}_i^{(0)}$ (we will illustrate how to construct them soon). In other words, there are two separate operations for vertex j:

- *Accumulate* operation: whenever receiving a value (e.g., $f_{\{i,j\}}(\Delta\hat{x}_i)$) from a neighbor (e.g., i), perform $\Delta\hat{x}_j = \Delta\hat{x}_j \star f_{\{i,j\}}(\Delta\hat{x}_i)$;
- *Update* operation: perform $\hat{x}_j = \hat{x}_j \star \Delta\hat{x}_j$; for any neighbor l, if $f_{\{j,l\}}(\Delta x_j^{(1)}) \neq o$, send $f_{\{j,l\}}(\Delta\hat{x}_j)$ to l; and then reset $\Delta\hat{x}_j$ to o;

where o is the identity value of the operator '$\star$'. That is, for $\forall z \in R$, $z = z \star o$ (if '$\star$' is '+', $o = 0$; if '$\star$' is 'min', $o = \infty$; if '$\star$' is 'max', $o = -\infty$). Basically, the *accumulate* operation accumulates received values between two consecutive updates on $\hat{x}_j$. The *update* operation adjusts $\hat{x}_j$ by absorbing $\Delta\hat{x}_j$, sends useful values to other vertices, and resets $\Delta\hat{x}_j$.

We now illustrate how to construct $\hat{x}_i^{(0)}$ and $\Delta\hat{x}_i^{(0)}$ by leveraging the computation result on the previous graph, G. We need to make sure that the constructed $\hat{x}_i^{(0)}$ and $\Delta\hat{x}_i^{(0)}$ can guarantee the correctness of the result on the new graph. Let $\bar{x}^{(*)}$ denote the convergence point on G. We next show how to construct $\hat{x}_i^{(0)}$

and $\Delta\hat{x}_i^{(0)}$ when the operator '$\star$' is '+' (for all the graph algorithms discussed in Section 2.3 except shortest paths and connected components) and when '$\star$' is 'min/max' (shortest paths and connected components), respectively.

For an iterative graph algorithm with the operator '$\star$' as '+', we first leverage $\bar{x}^{(*)}$ to construct $\hat{x}^{(0)}$ in the following way: for a kept vertex (e.g., i), we set $\hat{x}_i^{(0)} = \bar{x}_i^{(*)}$; for a newly added vertex (e.g., j), we set $\hat{x}_j^{(0)} = 0$. In contrast, *recomputation from scratch* typically utilizes $\mathbf{0}$ as $\hat{x}^{(0)}$ (where $\mathbf{0}$ is a vector with all its elements being zero). In order to construct $\Delta\hat{x}^{(0)}$, we compute $\hat{x}^{(1)}$ using $\hat{x}^{(1)} = f(\hat{x}^{(0)})$ and then construct $\Delta\hat{x}^{(0)}$ by making sure $\Delta\hat{x}^{(0)}$ satisfying $\hat{x}^{(1)} = \hat{x}^{(0)} \star \Delta\hat{x}^{(0)}$. Since '$\star$' is '+', we can calculate $\Delta\hat{x}^{(0)}$ by $\Delta\hat{x}^{(0)} = \hat{x}^{(1)} - \hat{x}^{(0)}$. It is important to note that here the deleted vertices and/or edges do not affect the way we construct $\hat{x}_i^{(0)}$ and $\Delta\hat{x}_i^{(0)}$. In other words, no matter whether there are deleted vertices and/or edges, the way we construct $\hat{x}_i^{(0)}$ and $\Delta\hat{x}_i^{(0)}$ can guarantee the correctness of the result on the new graph.

For an iterative graph algorithm with the operator '$\star$' as 'min/max', we construct $\hat{x}_i^{(0)}$ and $\Delta\hat{x}_i^{(0)}$ as follows. When the operator '$\star$' is 'min' (e.g., shortest paths), if any vertex's initial state is not smaller than its final converged state, the algorithm will converge. This is because of the following reason. When the algorithm has not converged, in each iteration there must be at least one vertex whose state is becoming smaller, and thus the overall state vector is becoming closer to the final converged state vector. When there is no vertex changing its state, the algorithm converges. Generally, it is hard to know the final converged state vector. Therefore, for the shortest paths algorithm, recomputation from scratch usually sets the initial state of a vertex (other than the source vertex) as ∞ to guarantee that it is not smaller than the final converged state. Fortunately, when the graph grows (vertices and/or edges are added and no vertices or edges are deleted), the previous converged state of a kept vertex must be not smaller than its converged state on the new graph. Therefore, for the graph growing scenario, we construct $\hat{x}_i^{(0)}$ in the following way: for a kept vertex (e.g., i), we set $\hat{x}_i^{(0)} = \bar{x}_i^{(*)}$; for a newly added vertex (e.g., j), we set $\hat{x}_j^{(0)} = \infty$. Similarly, for the connected component algorithm, whose operator '$\star$' is 'max', we can construct $\hat{x}_i^{(0)}$ (for the graph growing scenario) as follows: for a kept vertex (e.g., i), we set $\hat{x}_i^{(0)} = \bar{x}_i^{(*)}$; for a newly added vertex (e.g., j), we set $\hat{x}_j^{(0)} = j$. To construct $\Delta\hat{x}^{(0)}$, we also compute $\hat{x}^{(1)}$ using $\hat{x}^{(1)} = f(\hat{x}^{(0)})$ and then simply set $\Delta\hat{x}_j^{(0)} = \hat{x}_j^{(1)}$. It can satisfy $\hat{x}^{(1)} = \hat{x}^{(0)} \star \Delta\hat{x}^{(0)}$, no matter '$\star$' is 'min' or 'max'.

3.2 Selective Execution

One potential problem of basic asynchronous updates is that they might require more computation and communication when compared to their synchronous counterparts. This is because vertices are updated in a round-robin manner no matter how many new values available to a vertex. To solve this problem, instead of updating vertices in a round-robin manner, we update vertices selectively by identifying their importance. The motivation behind it is that not all vertices

contributes the same to the convergence. We refer to this scheduling scheme as *selective execution*. The vertices are selected according to their importance (in terms of contribution to the convergence).

Our selective execution scheduling scheme selects a block of m vertices (instead of one) to update each round. The reason is that if one vertex is chosen to update at a time, the scheduling overhead (e.g., maintaining a priority queue to always choose the vertex with the highest importance) is high. Once the block of the selected vertices are updated, it selects another block to update. Every time our scheme selects the top-m vertices in terms of the importance value. The size of the block (i.e., m) balances the tradeoff between the gain from selective execution and the cost of selecting vertices. Setting m too small may incur considerable overhead, while setting m too large may degrade the effect of selective execution, e.g., if setting m as the number of total vertices, it degrades to the round-robin scheduling. We will discuss how to determine m in Section 4.1.

We then illustrate how to quantify a vertex's importance when '$\star$' is 'min/max' and when the operator '$\star$' is '+', respectively. Ideally, the vertex whose update decreases the distance to the fixed point (i.e., $||x^{(*)} - \hat{x}^{(k)}||_1$) most should have the highest importance. For an iterative graph algorithm with the operator '$\star$' as 'min/max', the iterative updates either monotonically decrease (e.g., shortest paths) or monotonically increase (e.g., connected components) any element of $\hat{x}^{(k)}$. For ease of exposition, we assume the monotonically decreasing case. In this case, $x_j^{(*)} \leq \hat{x}_j^{(k)}$ for any j, and thus we have $||x^{(*)} - \hat{x}^{(k)}||_1 = ||\hat{x}^{(k)}||_1 - ||x^{(*)}||_1$. An update on vertex j decrease $||\hat{x}^{(k)}||_1$ by $|\hat{x}_j^{(k)} \star \Delta\hat{x}_j^{(k)} - \hat{x}_j^{(k)}|$. Therefore, we use $|\hat{x}_j^{(k)} \star \Delta\hat{x}_j^{(k)} - \hat{x}_j^{(k)}|$ to represent the importance of the vertex j (denoted as η_j), i.e. $\eta_j = |\hat{x}_j^{(k)} \star \Delta\hat{x}_j^{(k)} - \hat{x}_j^{(k)}|$.

For an iterative graph algorithm with the operator '$\star$' as '+', it is difficult to directly measure how the distance to the fixed point decreases. Update one single vertex may even increase the distance to the fixed point. Fortunately, for such an algorithm, its update function ($f()$) typically can be seen as a $||\cdot||$-*contraction mapping*. That is, there exists an α ($0 \leq \alpha < 1$), such that $||f(x) - f(y)|| \leq \alpha||x-y||, \forall x, y \in R^n$. Therefore, we can provide an upper bound on the distance, as stated in Theorem 1. The proof is omitted due to the space limitation. We then analyze how the upper bound decreases.

Theorem 1. $||x^{(*)} - \hat{x}^{(k+1)}||_1 \leq \frac{||\Delta\hat{x}^{(k+1)}||_1}{1-\alpha}$.

Without loss of generality, assume that current time is t_k and that during interval $[t_k, t_{k+1}]$ we only update vertex j. When updating vertex j, we accumulate $\Delta\hat{x}_j^{(k)}$ to $\hat{x}_j$, send $f_{(j,l)}(\Delta\hat{x}_j^{(k)})$ to a vertex l (and the total sending out value is no larger than $\alpha|\Delta\hat{x}_j^{(k)}|$), and reset $\Delta\hat{x}_j^{(k)}$ to 0. Therefore, we have the following theorem.

Theorem 2. $||\Delta\hat{x}^{(k+1)}||_1 \leq ||\Delta\hat{x}^{(k)}||_1 - (1-\alpha)|\Delta\hat{x}_j^{(k)}|$.

Theorem 2 implies that the upper bound monotonically decreases. When updating vertex j, we have $\frac{||\Delta\hat{x}^{(k+1)}||_1}{1-\alpha} \leq \frac{||\Delta\hat{x}^{(k)}||_1}{1-\alpha} - |\Delta\hat{x}_j^{(k)}|$. It shows that the reduc-

tion in the upper bound is at least $|\Delta\hat{x}_j^{(k)}|$. Given a graph, α is a constant. Hence, we define the importance of the vertex j to be $|\Delta\hat{x}_j^{(k)}|$, i.e., $\eta_j = \arg\max_j |\Delta\hat{x}_j^{(k)}|$.

3.3 Convergence

Our asynchronous incremental computation approach yields the same result as recomputation from scratch. To prove it, we first show that if synchronous updates (i.e., $x^{(k)} = f(x^{(k-1)})$) converge (and synchronous updates converge for all the graph algorithms discussed in Section 2.3), any asynchronous update scheme that can guarantee every vertex is updated infinitely often (until its state is fixed) will yield the same result as synchronous updates, as stated in Lemma 1.

Lemma 1. *If updates $x^{(k)} = f(x^{(k-1)})$ converge to $x^{(*)}$, any asynchronous update scheme that guarantees every vertex is updated infinitely often will converge to $x^{(*)}$ as well, i.e., $\hat{x}^{(\infty)} = x^{(*)}$.*

We then show that our asynchronous incremental computation approach fulfills this requirement, as stated in Lemma 2. The proofs of both Lemma 1 and Lemma 2 are omitted.

Lemma 2. *Our asynchronous incremental computation approach can guarantee that every vertex is updated infinitely often (until its state is fixed).*

We can also prove that recomputation from scratch converges to $x^{(*)}$ (no matter what type of updates it uses). As a result, we have the following theorem.

Theorem 3. *Our asynchronous incremental computation approach converges and yields the same result as recomputation from scratch.*

4 Distributed Framework

Oftentimes, iterative graph algorithms in real-world applications need to process massive graphs. Hence, it is desirable to leverage the parallelism of a cluster of machines to run these algorithms. Furthermore, it is troublesome to implement asynchronous incremental computation for each individual algorithm. Therefore, we propose GraphIn, an in-memory asynchronous distributed framework, for supporting iterative graph algorithms with incremental computation. GraphIn provides several high-level APIs to users for implementing asynchronous incremental computation and meanwhile hides the complexity of distributed computation. It leverages the proposed selective execution to accelerate convergence.

GraphIn consists of a number of workers and one master. Workers perform vertex updates, and the master controls the flow of computation. The new graph and the previous computed result are taken as the input of GraphIn. The input graph is split into partitions and each worker is responsible for one partition. Each worker leverages an in-memory table to store the vertices assigned to it. A worker has two main operations for its stored vertices: the accumulate operation and the update operation, as illustrated in Section 3.1. The accumulate

operation utilizes a user-defined function to aggregate incoming messages for a vertex and also triggers another user-defined function to calculate the vertex's importance. The update operation uses a user-defined function to update the states of scheduled vertices and compute outgoing messages.

The prototype of GraphIn is built upon Maiter [26]. Maiter is designed for processing static graphs, and thus has inherent impediments to the execution of graph algorithms with incremental computation. First, it relies on the specific initial state to guarantee the convergence of a graph algorithm. However, incremental computation leverages the previous result as the initial state, which can be arbitrary. Second, although Maiter supports prioritized updates, its scheduling scheme assumes that Δx_i is always positive for any vertex i, which can be not true under incremental computation. Last, the termination check mechanism of Maiter assumes that $||x||_1$ varies monotonically, which can be not true as well under incremental computation. GraphIn removes all these impediments to efficiently support incremental computation.

4.1 Distributed Selective Execution

GraphIn leverages the proposed selective execution scheduling as its default scheduling scheme. Since a centralized approach of finding the top-m elements is inefficient in a distributed environment, GraphIn allows each worker to build its own selective execution scheduling. Round by round (except the first round in which all vertices are selected to derive $\hat{x}^{(0)}$ and $\Delta\hat{x}^{(0)}$), each worker selects its local top-m vertices in terms of the importance. The number m is crucial to the effect of selective execution.

For the iterative graph algorithm with the operator '$\star$' as '+', GraphIn learns m online. We use $\mu \cdot n$ to quantify the overhead of selecting such m vertices (where μ represents the amortized overhead), which is proportional to the total number (n) of vertices with an efficient selection algorithm (e.g., quick-select). Also, we assume that the average cost of updating one vertex is ν, and then the cost of updating those m vertices is $\nu \cdot m$. Let $c(m)$ be the total cost of updating those m vertices (including both selection and update), then $c(m) = \mu \cdot n + \nu \cdot m$. Let $g(m) = \sum_{j \in S} |\Delta\hat{x}_j|$ (recall that $|\Delta\hat{x}_j|$ represents the importance of vertex i), where S denotes the set of the m selected vertices. For each round, we aim to find the m that can achieve the largest efficiency, i.e., $m = \arg\max_m \frac{g(m)}{c(m)}$. It is computationally impossible to try every value (from 1 to n) to figure out the best m. Therefore, our practical approach chooses several values ($0.05n$, $0.1n$, $0.25n$, $0.5n$, n), which cover the entire range of possible m, as the candidates. For each candidate m, we leverage quick-select to find the m-th $|\Delta\hat{x}_j|$, which is used as a threshold, and all $|\Delta\hat{x}_i|$ no less than the threshold are counted into $g(m)$. By testing each candidate (we set ν/μ as 4 by default), we can figure out the best m and the set S. The practical approach leverages quick-select to avoid the time-consuming sorting, and thus takes $O(n)$ time on extracting the top-m vertices instead of $O(n \log n)$ time. For the iterative graph algorithm with the operator '$\star$' as 'min/max', the importance of a vertex might be close to ∞. If we

still use the above idea, $g(m)$ might easily be overflown. Therefore, in this case, we simply set m as $0.1n$, which shows good performance in experiments. Note that if there are only m' ($m' < m$) vertices with the importance being larger than 0, we only select these m' vertices to update.

4.2 Distributed Termination Check

We design termination check mechanisms for the iterative graph algorithm with the operator '$\star$' as 'min/max' and for that with the operator '$\star$' as '+', respectively. When '$\star$' is 'min/max', $||\hat{x}^{(k)}||_1$ monotonically decreases or increases. Therefore, we can utilize $||\hat{x}^{(k)}||_1$ to perform the termination check. If $||\hat{x}^{(k)}||_1 - ||\hat{x}^{(k-1)}||_1 = 0$, the algorithm has converged, and thus the computation can be terminated. When '$\star$' is '+', $||x^{(*)} - \hat{x}^{(k)}||_1$ is the choice for measuring convergence. However, it is difficult to directly quantify $||x^{(*)} - \hat{x}^{(k)}||_1$, since the fixed point $x^{(*)}$ is always unknown during the computation. Fortunately, we know $||x^{(*)} - \hat{x}^{(k)}||_1 \leq ||\Delta\hat{x}^{(k)}||_1/(1-\alpha)$ from Theorem 1, and thus can leverage $||\Delta\hat{x}^{(k)}||_1$ to measure convergence. We use the convergence criterion, $||\Delta\hat{x}^{(k)}||_1 \leq \epsilon$, where the convergence tolerance ϵ is a pre-defined constant.

GraphIn adopts a passively monitoring model to perform the termination check, which works by periodically (and the period is configurable) measuring $||\hat{x}^{(k)}||_1$ if the operator '$\star$' is 'min/max' (or $||\Delta\hat{x}^{(k)}||_1$ if '$\star$' is '+'). To complete the measure, each worker computes the sum of $|\hat{x}_j^{(k)}|$ (or $|\Delta\hat{x}_j^{(k)}|$) of its local vertices and sends the local sum to the master. The master aggregates the local sums into a global sum. The challenge of performing such a distributed termination is to make sure that the local sum at each worker are calculated from the snapshot of the values at the same time (especially for $|\Delta\hat{x}_j^{(k)}|$). To address the challenge, GraphIn asks all the workers to pause vertex updates before starting to calculate the local sums. The procedure of the distributed termination check is as follows.

1. When it is the time to perform the termination check, the master broadcasts a chk_{pre} message to all the workers.
2. Upon receiving the chk_{pre} message, every worker pauses vertex updates and then replies a chk_{ready} message to the master.
3. The master gathers those chk_{ready} messages from all the workers, and then broadcasts a chk_{begin} message to them.
4. Upon receiving the chk_{begin} message, every worker calculates the local sum, $\sum_j |\hat{x}_j^{(k)}|$ (or $\sum_j |\Delta\hat{x}_j^{(k)}|$), and reports it to the master.
5. The master aggregates the local sums to the global sum $||\hat{x}^{(k)}||_1$ (or $||\Delta\hat{x}^{(k)}||_1$). If $||\hat{x}^{(k)}||_1 - ||\hat{x}^{(k-1)}||_1 \neq 0$ (or $||\Delta\hat{x}^{(k)}||_1 > \epsilon$), the master broadcasts a chk_{fin} message to all the workers. Otherwise, it broadcasts a *term* message.
6. When a worker receives the chk_{fin} message, it resumes vertex updates. When a worker receives the *term* message, it dumps the result to a local disk and then terminates the computation.

It is important to note that since calculating the local sums is inexpensive and it is done periodically, the overhead of the termination check is ignorable.

5 Evaluation

In this section, we evaluate the performance of our asynchronous incremental computation approach. We compare it with re-computation from scratch. Both approaches are supported by GraphIn. To show the performance of the selective execution scheduling, we compare it with the round-robin scheduling. The performance of other distributed frameworks that can support synchronous incremental computation are also evaluated.

5.1 Experiment Setup

The experiments are performed on both a local cluster and a large-scale cluster on Amazon EC2. The local cluster consists of 4 machines. The large-scale cluster consists of 50 EC2 medium instances.

Table 1. Graph Dataset Summary

Dataset	Vertices	Edges
Amazon co-purchasing graph (Amz) [14]	403K	3.4M
Web graph from Google (Gog) [14]	876K	5.1M
LiveJournal social network (LJ) [14]	4.8M	69M
Web graph from UK (UK) [5]	39M	936M
Web graph from IT (IT) [5]	41M	1.2B

Two graph algorithms are implemented on GraphIn, PageRank and the shortest paths algorithm. For PageRank, the damping factor is set to 0.8, and if not stated otherwise, the convergence tolerance ϵ (which is discussed in Section 4.2) is set to $10^{-2}/n$ (n is the number of vertices of the corresponding graph). The shortest paths algorithm stops running only when the convergence point is reached (i.e., all the vertices reach their shortest paths to the source vertex). The measurement of each experiment is averaged over 10 runs. Real-world graphs of various sizes are used in the experiments and are summarized in Table 1.

5.2 Overall Performance

We first show the convergence time of PageRank on the local cluster. The convergence time is measured as the wall-clock time that PageRank uses to reach the convergence criterion (i.e., $||\Delta\hat{x}^{(k)}||_1 \leq \epsilon$). We consider both the edge change case and the vertex change case. Under the edge change case, we randomly pick a number of vertices to change their edges. In the graph evolving process, there are usually more added edges than deleted edges. Therefore, for 80% of the picked vertices, we add one outgoing edge to it with a randomly picked neighbor. For the rest 20% vertices, we remove one randomly picked edge from it. Under the vertex change case, we pick a number (e.g., p, some percentage of the total number of vertices) for each experiment. We add $0.8p$ new vertices to the graph and

delete $0.2p$ vertices. For each added vertex, we put two edges (one incoming edge and one outgoing edge) with randomly picked neighbors. For each deleted vertex, we also delete all its edges.

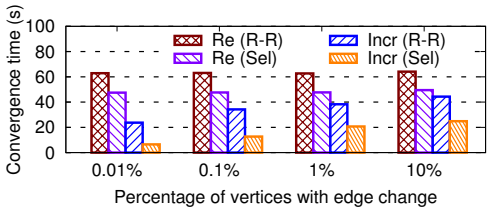


Fig. 1. PageRank on Amz graph (edge change).

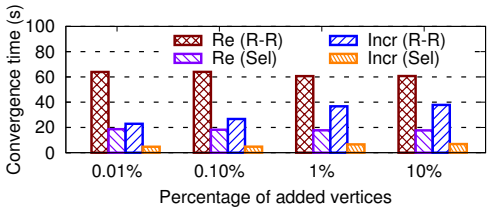


Fig. 2. Shortest paths on weighted Amz graph.

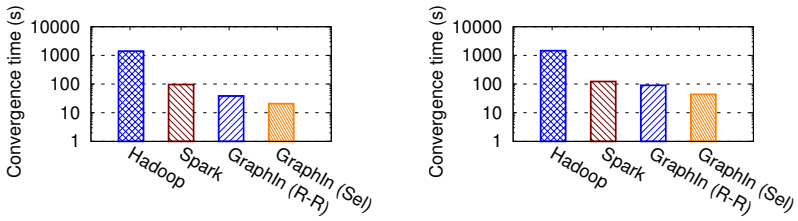
Figure 1 shows the performance on the Amz graph under the edge change case. We can see that incremental computation (denoted as "Incr") is much faster than re-computation from scratch (denoted as "Re") for different percentages of vertices with edge change. The selective execution scheduling (denoted as "Sel") is faster than the round-robin scheduling (denoted as "R-R") with either approach. The efficiency of incremental computation is more prominent when the change is smaller. For example, when the percentage of vertices with edge change is 0.01%, incremental computation with the selective execution scheduling is about 10x faster than recomputation from scratch with the round-robin scheduling and 7x faster than recomputation from scratch with the selective execution scheduling. Not surprisingly, incremental computation takes longer time as the percentage of vertices with edge change becomes larger, and the convergence time of the re-computation is almost the same since the change to the graph is relatively small. Similar trends are observed for the vertex change case.

We then present the result of the shortest paths algorithm, which runs on weighted graphs. Here the convergence time is measured as the wall-clock time that the shortest paths algorithm uses to reach the convergence point. All the graphs summarized in Table 1 are unweighed. We generate a weighted graph by assigning weights to the Amz graph. The weight of each edge is an integer, which

is randomly drawn from the rang $[1, 100]$. Figure 2 plots the performance comparison under the vertex adding case. The percentage means the ratio between the number of added vertices to the number of original vertices. For each added vertex, we put two weighted edges (one incoming edge and one outgoing edge) with randomly picked neighbors. From the figure, we can see that incremental computation with the selective execution scheduling is about 14x faster than recomputation from scratch with the round-robin scheduling when the percentage of added vertices is 0.01% and still 9x faster even when the percentage is 10%. Similar results are observed for the edge adding case as well.

5.3 Comparison with Synchronous Incremental Computation

It is also possible to build a framework to support incremental computation upon other systems, such as Hadoop and Spark. To demonstrate the efficiency of GraphIn, we compare it with both Hadoop and Spark for the 1% of vertices with edge change scenario. We restrict our performance comparison to PageRank, since it is a representative graph algorithm. For fair comparison, we instruct both systems to use the prior result as the starting point. For Hadoop, if there is no change in the input of some Map/Reduce tasks, we proportionally discount the running time. In this way, we can simulate task-level reusing, which is the key of MapReduce-based incremental processing frameworks. For Spark, we choose its Graphx [7] component to implement PageRank.



(a) Amz (b) Gog

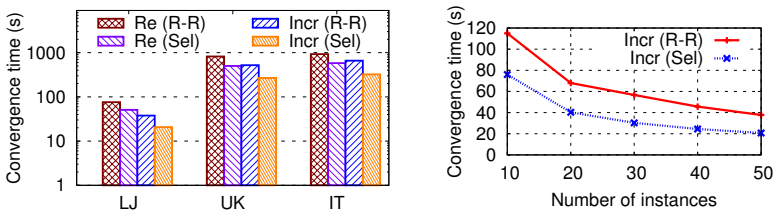
Fig. 3. PageRank on different frameworks.

Figure 3 shows that GraphIn (especially with selective execution) is much faster than Hadoop and Spark. Hadoop is a disk-based system and uses synchronous updates. Even though Spark is a memory-based system, it also utilizes synchronous updates. Therefore, it is still slower than GraphIn.

5.4 Scaling Performance

We further evaluate incremental computation on the large-scale Amazon cluster to test its scalability. We consider the 1% of vertices with edge change scenario, and concentrate on PageRank (and set the convergence tolerance ϵ to 10^{-4}). We

first use the three large real-world graphs, LJ, UK, and IT (both UK and IT have tens of millions of vertices and a billion of edges), as input graphs when all the 50 instances are used. As shown in Figure 4a (note that the y-axis is in log scale), on the large-scale cluster incremental computation is still much faster than re-computation from scratch, and both approaches can benefit from the selective execution scheduling.



(a) 50 instances (b) Varying instance number

Fig. 4. Performance on Amazon cluster.

We then show the performance of incremental computation when different numbers of instances are used. Figure 4b shows the convergence time on LJ as we increase the number of instances from 10 to 50. It can be seen that by increasing the number of instances, the convergence time is reduced, and that the selective execution scheduling is always faster than the round-robin scheduling.

6 Related Work

Due to the dynamic nature of graphs in real-world applications, incremental computation has been studied extensively. In terms of iterative graph algorithms, most of the studies [1, 11, 12] focus on PageRank. The basic idea behind approaches in [11,12] is that when a change happens in the graph, the effect of the change on the PageRank scores is mostly local. These approaches start with the exact PageRank scores of the original graph but provide approximate scores for the graph after the change, and the estimations may drift away from the exact scores. On the contrary, our approach can provide exact scores. The work in [1] utilizes the Monte Carlo method to approximate PageRank scores on evolving graphs. It precomputes a number of random walk segments for each vertex and stores them in distributed shared memory. Besides of the approximate result, it also incurs high memory overhead.

In recent years, the growing scale and importance of graph data have driven the development of a number of distributed graph systems. Graphx [7] is a graph system built on top of Spark. It stores graphs as tabular data and implements graph operations using distributed joins. PrIter [25], Maiter [26], and Prom [20], introduce prioritized updates to accelerate convergence. PrIter is a MapReduce-based framework, which requires synchronous iterations. Maiter and Prom utilize

asynchronous iterative computation. All these graph systems aim at supporting graph computation on static graph structures.

There are several systems for supporting incremental parallel processing on massive datasets. Incoop [3] extends the MapReduce programming model to support incremental processing. It saves and reuses states at the granularity of individual Map or Reduce tasks. Continuous bulk processing (CBP) [15] provides a groupwise processing operator to reuse prior state for incremental analysis. Similarly, other systems like DryadInc [17] support incremental processing by allowing their applications to reuse prior computation results. However, most of the studies focus on one-pass applications rather than iterative applications. Several recent studies address the need of incremental processing for iterative applications. Kineograph [6] constructs incremental snapshots of the evolving graph and supports reusing prior states in processing later snapshots. Naiad [16] presents a timely dataflow computational mode, which allows stateful computation and nested iterations. Spark Streaming [23] extends the cyclic batch dataflow of original Spark to allow dynamic modification of the dataflow and thus supports iteration and incremental processing. However, most of these systems apply synchronous updates to incremental computation. Our work illustrates how to efficiently apply asynchronous updates to incremental computation.

7 Conclusion

In this paper, we propose an approach to efficiently apply asynchronous updates to incremental computation on evolving graphs. Our approach works for a family of iterative graph algorithms. We also present a scheduling scheme, selective execution, to coordinate asynchronous updates so as to accelerate convergence. Furthermore, to facilitate the implementation of iterative graph algorithms with incremental computation in a distributed environment, we design and implement an asynchronous distributed framework, GraphIn. Our evaluation results show that our asynchronous incremental computation approach can significantly boost the performance.

Acknowledgments

We would like to thank anonymous reviewers for their insightful comments. This work is partially supported by NSF grants CNS-1217284 and CCF-1018114.

References

1. Bahmani, B., Chowdhury, A., Goel, A.: Fast incremental and personalized pagerank. Proc. VLDB Endow. 4(3), 173–184 (Dec 2010)
2. Baluja, S., Seth, R., Sivakumar, D., Jing, Y., Yagnik, J., Kumar, S., Ravichandran, D., Aly, M.: Video suggestion and discovery for youtube: taking random walks through the view graph. In: WWW '08. pp. 895–904 (2008)

3. Bhatotia, P., Wieder, A., Rodrigues, R., Acar, U.A., Pasquin, R.: Incoop: Mapreduce for incremental computations. In: SoCC '11. pp. 7:1–7:14 (2011)
4. Bogdanov, P., Singh, A.: Accurate and scalable nearest neighbors in large networks based on effective importance. In: CIKM '13. pp. 1009–1018 (2013)
5. Boldi, P., Vigna, S.: The WebGraph framework I: Compression techniques. In: WWW '04. pp. 595–601 (2004)
6. Cheng, R., Hong, J., Kyrola, A., Miao, Y., Weng, X., Wu, M., Yang, F., Zhou, L., Zhao, F., Chen, E.: Kineograph: Taking the pulse of a fast-changing and connected world. In: EuroSys '12. pp. 85–98 (2012)
7. Gonzalez, J.E., Xin, R.S., Dave, A., Crankshaw, D., Franklin, M.J., Stoica, I.: Graphx: Graph processing in a distributed dataflow framework. In: OSDI'14. pp. 599–613 (2014)
8. Guan, Z., Wu, J., Zhang, Q., Singh, A., Yan, X.: Assessing and ranking structural correlations in graphs. In: SIGMOD '11. pp. 937–948 (2011)
9. Jeh, G., Widom, J.: Scaling personalized web search. In: WWW '03. pp. 271–279 (2003)
10. Kleinberg, J.M.: Authoritative sources in a hyperlinked environment. J. ACM 46, 604–632 (Sep 1999)
11. Langville, A.N., Meyer, C.D.: Updating PageRank with iterative aggregation. In: WWW '04. pp. 392–393 (2004)
12. Langville, A.N., Meyer, C.D.: Updating markov chains with an eye on Google's PageRank. SIAM J. Matrix Anal. Appl. 27(4), 968–987 (2006)
13. Lempel, R., Moran, S.: Salsa: The stochastic approach for link-structure analysis. ACM Trans. Inf. Syst. 19(2), 131–160 (Apr 2001)
14. Leskovec, J., Krevl, A.: SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data` (Jun 2014)
15. Logothetis, D., Olston, C., Reed, B., Webb, K.C., Yocum, K.: Stateful bulk processing for incremental analytics. In: SoCC '10. pp. 51–62 (2010)
16. Murray, D.G., McSherry, F., Isaacs, R., Isard, M., Barham, P., Abadi, M.: Naiad: A timely dataflow system. In: SOSP '13. pp. 439–455 (2013)
17. Popa, L., Budiu, M., Yu, Y., Isard, M.: Dryadinc: Reusing work in large-scale computations. In: HotCloud'09 (2009)
18. Sarkar, P., Moore, A.W.: Fast nearest-neighbor search in disk-resident graphs. In: KDD '10. pp. 513–522 (2010)
19. Song, H.H., Cho, T.W., Dave, V., Zhang, Y., Qiu, L.: Scalable proximity estimation and link prediction in online social networks. In: IMC '09. pp. 322–335 (2009)
20. Yin, J., Gao, L.: Scalable distributed belief propagation with prioritized block updates. In: CIKM '14. pp. 1209–1218 (2014)
21. Yin, J., Gao, L., Zhang, Z.M.: Scalable nonnegative matrix factorization with block-wise updates. In: ECML/PKDD '14. pp. 337–352 (2014)
22. Yin, J., Zhang, Y., Gao, L.: Accelerating expectation-maximization algorithms with frequent updates. In: CLUSTER '12. pp. 275–283 (2012)
23. Zaharia, M., Das, T., Li, H., Hunter, T., Shenker, S., Stoica, I.: Discretized streams: Fault-tolerant streaming computation at scale. In: SOSP '13. pp. 423–438 (2013)
24. Zhang, C., Jiang, S., Chen, Y., Sun, Y., Han, J.: Fast inbound top-k query for random walk with restart. In: ECML/PKDD' 15. pp. 608–624 (2015)
25. Zhang, Y., Gao, Q., Gao, L., Wang, C.: PrIter: A distributed framework for prioritized iterative computations. In: SoCC '11. pp. 13:1–13:14 (2011)
26. Zhang, Y., Gao, Q., Gao, L., Wang, C.: Maiter: An asynchronous graph processing framework for delta-based accumulative iterative computation. IEEE Transactions on Parallel and Distributed Systems 25(8), 2091–2100 (2014)