Facilitating Real-Time Graph Mining

Zhuhua Cai
Rice University
Houston, TX, USA
caizhua@gmail.com

Dionysios Logothetis
Telefonica Research
Barcelona, Spain
dl@tid.es

Georgos Siganos
Telefonica Research
Barcelona, Spain
georgos@tid.es

ABSTRACT

Real-time data processing is increasingly gaining momentum as the preferred method for analytical applications. Many of these applications are built on top of large graphs with hundreds of millions of vertices and edges. A fundamental requirement for real-time processing is the ability to do incremental processing. However, graph algorithms are inherently difficult to compute incrementally due to data dependencies. At the same time, devising incremental graph algorithms is a challenging programming task.

This paper introduces GraphInc, a system that builds on top of the Pregel model and provides efficient incremental processing of graphs. Importantly, GraphInc supports incremental computations automatically, hiding the complexity from the programmers. Programmers write graph analytics in the Pregel model without worrying about the continuous nature of the data. GraphInc integrates new data in real-time in a transparent manner, by automatically identifying opportunities for incremental processing. We discuss the basic mechanisms of GraphInc and report on the initial evaluation of our approach.

Categories and Subject Descriptors

H.3.4 [**Systems and Software**]: Distributed systems

General Terms

Algorithms, Design, Performance, Experimentation

Keywords

Graph mining, incremental processing, memoization

1. INTRODUCTION

The ability to mine large graphs has become critical for many real-world applications. For instance, recommendation engines analyze the interactions of friends on the social graph to suggest articles or target advertisements [29]. Telcos mine Call Detail Record graphs to detect frauds based on suspicious calling patterns [13, 28, 5, 4], while transportation decision systems analyze road networks for route planning purposes [8].

The importance of large-scale graph mining has given rise to a new class of systems, like Pregel [17] that is based on the Bulk Synchronous Parallel (BSP) abstraction, and Graphlab [15] an asynchronous parallel framework, both designed for these type of applications. These systems provide intuitive and flexible programming models tweaked for graph algorithms and have been shown to be more efficient than systems based on MapReduce like Hadoop for comparable workloads.

At the same time, there is a growing demand for real-time analytics from a variety of organizations. For instance, telcos must be able to detect frauds as soon as they happen, and news article recommendations in social networks must be delivered within minutes. While current graph processing systems can process batch workloads efficiently, mining continuously changing graph data in real-time raises certain challenges.

A fundamental requirement for processing continuous data in real time is the ability to do incremental processing. As the data size and frequency of the updates increase, recomputing from scratch may result in poor resource utilization and high processing latency. This need has been observed in many real-world, large-scale mining scenarios [21, 11, 12, 20, 14]. However, devising incremental graph algorithms for large graphs can be a difficult and sometimes impossible programming task. Often programmers may resort to approximate algorithms that make it hard to reason about their accuracy [7, 2]. Besides that, designing custom incremental algorithms for every possible graph metric requires significant programming effort.

To address these challenges, we introduce GraphInc, a system for real-time graph mining. GraphInc allows users to specify programs in the familiar Pregel programming model and transparently converts batch graph algorithms into incremental ones that can run in real-time mode, without requiring any effort from the developer. Our approach is based on memoization, the ability to save and reuse computation state [3, 22, 11]. GraphInc leverages the structure of computations in the Pregel model to automatically identify opportunities for computation reuse. It saves the state of Pregel subcomputations and re-executes only those that are necessary when the graph changes, using the memoized computations wherever possible. This enables efficient incremental-like performance for generic graph mining programs, without the hassle of designing custom incremental algorithms.

Even though GraphInc can identify opportunities for computation reuse, graph algorithms are inherently difficult to process in an incremental manner due to data dependencies. To verify the potential of our approach, we study a set of common graph algorithms. We analyze the degree to which these algorithms are amenable to incremental algorithms under different workload characteristics. Through this study we identify common computation and communication patterns observed in these algorithms and discuss how they affect the efficiency of our approach.

We have built the GraphInc prototype by extending Giraph [1], an open-source implementation of the Pregel framework, to support memoization. We report our findings based on our prototype and furthermore discuss the most important design issues that allow efficient incremental processing.

The rest of the paper is structured as follows. In Section 2 we provide background knowledge on the Pregel model that GraphInc builds upon. Section 3 describes GraphInc's incremental computation mechanism, while in Section 4 we do a quantitative analysis on the performance gains for various applications. Section 5 discusses the most important design aspects of GraphInc. In Section 6, we present related work and in Section 7 we conclude.

2. BACKGROUND

GraphInc builds upon Pregel [17], a programming model designed specifically for distributed graph algorithms, and its open-source implementation called Giraph [1]. Apart from being flexible and intuitive for graph mining, the Pregel model structures computations in a way that allows GraphInc to identify opportunities for computation re-use when the input graph changes. In the rest of this section, we describe the most important aspects of the Pregel model and illustrate its use through an example.

2.1 The Pregel model

The Pregel model advocates a vertex-centric way of programming graph algorithms that is based on the Bulk Synchronous Parallel (BSP) model [27]. A program is structured as a sequence of *supersteps*. During a superstep every vertex v executes the same user-defined compute function C in parallel. Every vertex has associated *state* and inside function C, the user can access and update this state based on arbitrary application logic. At superstep i, the input to the function C of v consists of messages sent to v by other vertices in superstep $i-1$. Similarly, from within function C, vertex v can send messages to other vertices in the graph.

The termination of an algorithm execution depends on vertices *voting to halt*. From within the function C, a vertex can vote-to-halt, deactivating itself. A deactivated vertex does not execute function C, unless it receives messages. Execution stops when all vertices have voted to halt and there are no messages.

Note that the state of a vertex includes the graph structure, that is, the edges to neighbors and any associated properties such as weights. A user-defined function C has the ability to modify the graph structure by adding or deleting vertices and edges.

2.2 Example: single-source shortest paths

To illustrate the Pregel model we use as an example the computation of single-source shortest paths, a simple yet very common application. Here, every vertex iteratively updates its state, that is, its current shortest distance to the source.

Algorithm 1 Computation function C for the single-source shortest paths algorithm.

```
 1: function $C(v, S, I)$
 2:     $mindist$ = is_source(v) ? 0 : $+inf$;
 3:     for all $msg$ in $I$ do
 4:         $mindist = \min(mindist, msg.value)$
 5:     end for
 6:     if $mindist < S.value$ then
 7:         $S.value = mindist$
 8:         for all $edge$ in $v.get\_edges()$ do
 9:             send_message($edge.dst$, $mindist + edge.value$)
10:         end for
11:         vote_to_halt();
12:     end if
13: end function
```

Algorithm 1 shows the user-defined function C for this algorithm. Starting from an infinite distance (line 2), if there is a decrease in the current distance (line 6), a vertex propagates such change to all its neighbors through messages (line 9). Neighbors receive this change in the next superstep and potentially update their own distance (lines 3-5). At the end of every superstep, every vertex votes to halt (line 9) and is deactivated until another vertex sends a message to it. This process continues until there are no updates to the distance of any vertex.

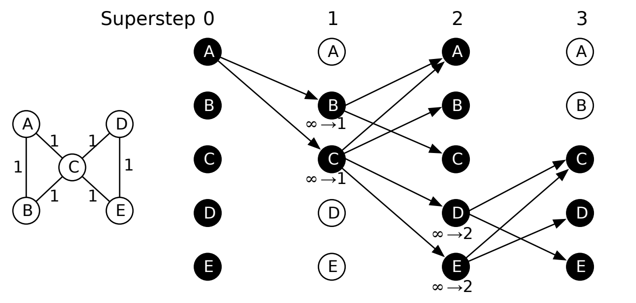
Figure 1(a) shows the execution of the single-source shortest path algorithm on a small graph. At superstep 0, only the source updates its current state (line 2) and propagates this change to its neighbors, vertices B and C. At superstep 1, vertices B and C update their state based on the new minimum distance and, in turn, propagate this change to their neighbors.

3. INCREMENTAL COMPUTATION

The goal of GraphInc is to automatically minimize the computation and communication necessary to recompute the analysis when the graph changes. In this section, we describe the mechanisms that enable GraphInc to incrementally update a graph computation.

Our approach leverages the structure of Pregel programs to unveil opportunities for computation re-use without the need for custom incremental programs. A graph computation consists of a set of per-vertex computations that depend on the vertex state and messages from other vertices. If we modify the graph and re-execute the algorithm, a fraction of these computations repeat: for a specific superstep, a vertex has the same state and receives the same messages as in the original execution. The example of Figure 1 shows the execution of the shortest-paths algorithm before and after a graph update. In Figure 1(b), the grey nodes correspond to vertices that perform exactly the same computation for a specific superstep as in the execution on the original graph.

The role of GraphInc is to identify repeated computations after a graph update and eliminate them, while guaranteeing correctness. GraphInc ensures that the result of the execution is identical as if the same user-defined program


(a) Execution on the original graph

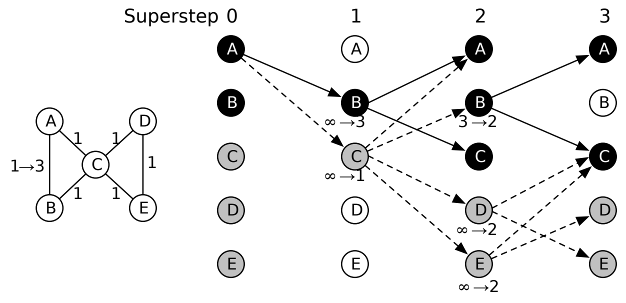

(b) Execution on the modified graph

Figure 1: Execution steps for the single-source shortest-paths application. Here, we compute the shortest distance from all vertices to vertex A. (a) Black nodes represent vertices that execute the user-defined function C. Below a vertex that executes, we indicate the change in the current shortest distance after the execution. White nodes represent deactivated vertices that do not execute. (b) Execution after the weight of edge AB changes from 1 to 3. Dashed arrows and gray nodes represent messages and computations that are exactly the same as in the original execution. Here, 50% of the computations are the same as in the original execution.

had been executed on the modified graph. As we describe in more detail in the following sections, GraphInc achieves this by *memoizing* the state of all computations performed.

3.1 Problem formulation

Let us now describe more formally how GraphInc automatically transforms a graph computation to an incremental one.

Given a graph $G = \{V, E\}$, a program $P(G)$ on the graph consists of a set of per-vertex computations C performed in supersteps as defined by the Pregel model. We denote the computation of a vertex v at superstep i as $C(v, S_{v,i}, I_{v,i}) \rightarrow \{S_{v,i+1}, O_{v,i}\}$, where $I_{v,i}$ is the set of incoming messages, $S_{v,i}$ is the state of v before it processes $I_{v,i}$, $S_{v,i+1}$ is the updated state and $O_{v,i}$ is the set of outgoing messages.

GraphInc constructs a program P' such that when applied to an updated graph $G' = \{V', E'\}$, then $P'(G', P(G)) = P(G')$. Here, the program P' takes as input not only the modified graph G', but also the computations performed on the original graph G. GraphInc memoizes the computations executed during P. Specifically, for every computation $C(v, S_{v,i}, I_{v,i})$, GraphInc memoizes the state $S_{v,i}$ and the set of messages $I_{v,i}$.

The goal of GraphInc is to execute P' with as few new computations as possible, using the memoized computations of P.

3.2 Basic algorithm

We now show how GraphInc constructs P'. Recall that we wish to construct an incremental program in a transparent manner. Therefore, as building blocks for P, we use the original computations C specified by the user unmodified.

However, we make the following assumptions about the user-defined computations C that ensure we can safely reuse them:

1. Computations are side-effect free. A vertex computation depends only on input messages and state. It does not depend on interactions with external services, such as a file system.

2. Computations are deterministic.

The execution of program P', which is described in Algorithm 2, proceeds in two phases. In the first phase, GraphInc detects and labels all vertices that are *affected* by an update to the graph. A vertex labeled as affected is a candidate for reexecution of the computation function C since the output of C may depend on some modified property of the graph (e.g. a new outgoing edge added). In the example of Figure 1, only vertices A and B are affected.

Table 1 summarizes all possible ways that vertices get affected by a change. Notice that edge changes in directed graphs affect only the source vertex since the computation of a vertex does not depend on incoming edges. Notice also that a vertex deletion implicitly affects all the vertices connected to it either with outgoing or with incoming edges.

The second phase of P' starts with superstep 0 by executing computations only for the affected vertices. These are the only vertices for which the computation during superstep 0 in P' may be different than in P. After that, at every superstep $i > 0$, GraphInc executes a computation for a vertex only if it satisfies one of the following conditions:

1. It receives messages and at least one of the messages is different than in P. In this case, the computation of the vertex depends on another vertex that has changed in the previous superstep. In the example of Figure 1(b), at superstep 1 vertex B receives a message from A that is different than in the execution of P and must, therefore, execute.

2. Its messages are the same, but its state is different than in P. In this case, the state of the vertex has changed in a previous superstep, and even if it receives the same messages, it has to execute. In Figure 1(b), at superstep 2, vertex B receives the same messages as in P, but its state is different.

3. Its messages are the same as in P, but it is an affected vertex. As with the previous condition, even if a vertex receives exactly the same messages, since it is affected the result of the computation may be different.

Conditions 2 and 3 imply that either a vertex receives messages in P' and they are exactly the same as in P, or it does

Type of change	Affected vertices
Vertex property change	Only the specific vertex
Vertex addition	The specific vertex and any vertices it points to
Vertex deletion	All neighbors of the vertex, connected with either incoming or outgoing edges
Edge addition/deletion	**Directed**: only the source vertex **Undirected**: both ends of the edge
Edge property change	**Directed**: only the source vertex **Undirected**: both ends of the edge

Table 1: Different types of graph updates and how they affect neighboring vertices.

Algorithm 2 Construction of incremental program P'. $I'_{v,i}$ denotes messages sent to v in P' and $I_{v,i}$ denotes messages memoized from P.

```
1: function $P'(G', P(G))$
2:     $A \leftarrow \{$affected vertices$\}$                ▷ Phase 1
3:     for all $v$ in $A$ do                    ▷ Phase 2, superstep 0
4:         compute $C(v, S_{v,0}, I_{v,0})$
5:     end for
6:     for $(i = 1; i <= S_{max}; i++)$ do
7:         for all $v$ in $V$ do
8:             if $(I'_{v,i} \neq \emptyset$ AND $I'_{v,i} \neq I_{v,i})$ then
9:                 compute $C'(i, v, I'_{v,i})$
10:            else if $(I_{v,i} \neq \emptyset$ AND $S_{v,i}$ is modified$)$ then
11:                compute $C'(i, v, I'_{v,i})$
12:            else if $(I_{v,i} \neq \emptyset$ AND $v \in A)$ then
13:                compute $C'(i, v, I'_{v,i})$
14:            else
15:                NO-OP
16:            end if
17:        end for
18:    end for
19: end function
```

Algorithm 3 Modified computation function C'

```
1: function $C'(i, v, I'_{v,i})$
2:     read saved state $S_{v,i}$
3:     read saved messages $I_{v,i}$
4:     overwrite $I_{v,i}$ with messages in $I'_{v,i}$
5:     save new $I_{v,i}$
6:     compute $C(v, S_{v,i}, I_{v,i}) \rightarrow \{S_{v,i+1}, O_{v,i}\}$
7:     save new $S_{v,i+1}$
8:     send $O_{v,i}$
9: end function
```

not, but it has messages memoized from P. This is reflected in lines 10 and 12 of Algorithm 2. In both these cases, the vertex must execute since its state is different than in P. If none of these three conditions hold, then a vertex does not need to execute.

Note that in P' GraphInc does not execute computations C, rather a modified computation C' that uses the memoized results of P. Algorithm 3 describes the computation C' at every vertex. Here I'_i is the set of messages received while executing P'. Some of these may be different than the ones received in P. For example, in Figure 1(b), the messages that node A receives from B at superstep 2 are different than in P. In this case, GraphInc overwrites old memoized messages with the new ones as shown in step 4 of the algorithm.

The first assumption mentioned above ensures that the computation of a vertex depends only on data controlled by the underlying processing system, such as the messages exchanged and the state computation. This allows GraphInc to reliably detect when the result of a vertex computation may change and whether the conditions are met. However, in certain algorithms vertex computations depend on additional state such as global counters. These applications are supported simply by enforcing access to such state through well defined APIs controlled by the underlying system.

4. WORKLOAD ANALYSIS

Even though our approach allows us to reuse subcomputations of a graph algorithm, graph analytics are inherently difficult to compute incrementally due to the data dependencies. A change in a vertex may affect a fraction of the graph, ranging from a small neighborhood around the affected vertex to potentially the entire graph. Whether an analysis is amenable to incremental computation depends on data-specific parameters, such as graph connectivity, as well as algorithm-specific factors, such as computation and message-propagation patterns. In this section, we analyze a set of common graph algorithms on typical graphs to verify the potential of our approach. In Section 4.3, we provide insights about the dependence of performance on graph and application properties.

We study a set of algorithms with different computation and communication patterns. While this is not by any means an exhaustive list of graph algorithms they exhibit patterns found in other several graph algorithms, such as path traversal, community detection, and message propagation. Additionally, the algorithms under study are the base in many applications.

Note that we characterize graph algorithms specifically in the context of the Pregel compute model. While there are different compute models and specialized dynamic graph mining algorithms that may yield different characteristics with respect to incremental computation, such a study is beyond the scope of this paper.

4.1 Measuring incremental performance

Characterizing the degree to which a graph change affects the execution of graph algorithm is different than in computations like aggregates. To analyze the benefit of incremental computation for the various workloads, we first need an appropriate metric that measures computation and communication savings.

Our metric takes into consideration the structure of a

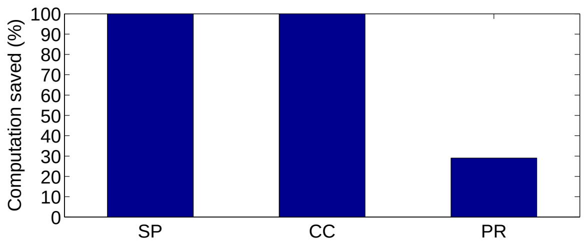


Figure 2: Average computation saved for a single edge addition.

Pregel program. If we consider the execution of an algorithm on a graph G and on a modified version G' of the original graph, then a fraction of all the per-vertex computations on G' are exactly the same. These are the computations that our approach prunes. We define as the *computation saved* the fraction of the computation on G' that GraphInc prunes over all the computations on G'. By pruning vertex computations, GraphInc also prunes the messages sent as a result of a vertex computation. We define as *communication saved* the fraction of messages saved over all messages during the execution of the algorithm on G'.

Here, we do not model the absolute cost of an algorithm execution in terms of running time, CPU operations or bytes transferred. While we could use these metrics to measure computation and communication at a finer granularity, their interpretation varies across different execution environments. Instead, we model the relative difference in the number of per-vertex operations and messages sent. These two metrics combined provide a more intuitive view of the benefit.

4.2 Applications

We now describe the different algorithms we have implemented and discuss the observed performance. We first give an overview of our experiments, and then discuss the results for every application. To get a better understanding of the results, we discuss the results with respect to the differences in computation and communication patterns among the different algorithms.

As a point of reference, we first examine how much a single change in the graph can affect the computation (Figure 2). In this case, we modify the graph by adding an edge, and measure the computation saved. We repeat this for 50 randomly added edges and report the average.

Subsequently, we study the effect of the size of the graph updates on the computation and communication saved (Figure 4.2). In many scenarios an application may apply several graph updates before updating the analysis. Here, we modify the percentage of edges added from 0.01% to 0.1% and measure how the computation and communication savings change.

Finally, we measure how the computation saved changes as a graph evolves over time (Figure 4). This gives us an idea of how the fact that graphs naturally become more connected over time affects the efficiency of our approach. In this case, we continuously modify the same graph by adding a certain percentage of new edges (1%) and measure the computation saved, each time with respect to the previous instance of the graph.

For all the experiments, we use a synthetic power-law graph with 1 million vertices and 20 million edges.

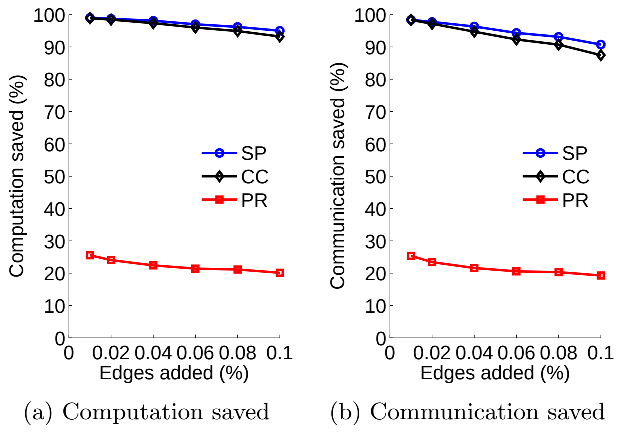


Figure 3: The percentage of computation and communication saved as a function of the percentage of edges added to the graph.

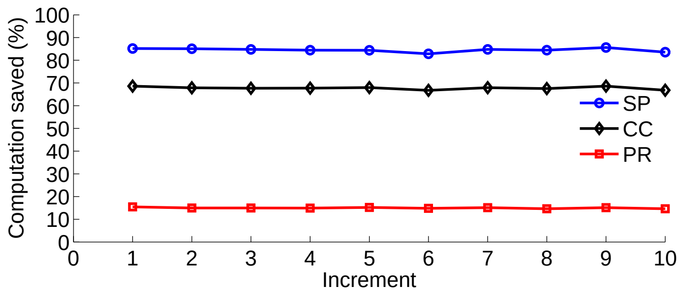


Figure 4: The percentage of computation saved as the graph evolves. Every increment on the x-axis corresponds to adding 1% of new edges.

4.2.1 *Shortest paths*

While simple, the shortest paths (SP) algorithm is useful in many real applications, including transportation systems [8], and social network analysis [6]. Here, we implement the SP algorithm as described in Section 2.

In Figure 2, we see that for a single edge addition, the computation saved is 99.99%. This implies that GraphInc can incorporate single updates to the graph efficiently for this algorithm. Note that although it is not shown here, edge deletions are handled in the same manner and have the same effect on computation savings. Note that while typically edge deletions require special incremental and often approximate algorithms [2], GraphInc can handle deletions efficiently in a transparent manner using memoization.

Furthermore, Figure 3(a) shows that even for larger update sizes, the computation saved remains high, ranging from 96.1% to 99.2%. In this case, even 0.01% of new edges do not significantly affect the shortest distances of the vertices. In Figure 3(b), we see that our approach prunes a large percentage of the messages as well with the communication saved ranging from 90.8% to 98.3%. As expected, since the vast majority of the vertices do not execute the associated messages are not sent.

Figure 4 shows that as the graph evolves over time, the computation saved remains near constant. Notice that because the percentage of added edges is now higher (1%), the computation saved is lower than in the previous experiment. However, in this experiment the number of new edges is not able to change the connectivity of the graph significantly across time. We expect, though, that as more updates are applied and the graph becomes more connected, the computation saved drops.

4.2.2 *Connected components*

The connected components (CC) algorithm is important in itself in understanding graphs but is also a requirement in several other applications. For instance, it may be used as a first step in graph pattern matching [16] or graph partitioning algorithms.

At a high level, the distributed connected components algorithm labels every vertex with the minimum vertex ID across all vertices in the component it belongs. To do this, every vertex iteratively updates the minimum vertex ID it has seen. Starting with every vertex propagating its own ID at superstep 0, whenever a vertex sees a new minimum ID, it updates its state and propagates this change to its own neighbors.

In Figure 2, we observe that the addition of a single edge to the graph has a similar effect on the computation as in the SP algorithm. However, in Figure 4.2, we see that as we add more edges to the graph, the computation saved in the CC algorithm is in the range 94.9%-98.9%, while the communication savings range from 90.7% to 98.3%.

Although still high, the computation and communication saved in CC is lower than in SP due to the different computation and communication patterns. In SP, a computation starts with the source node propagating messages in a breadth-first manner. Therefore, a change in the graph that is far from the source node may affect only supersteps toward the end of the computation. In contrast, in the CC algorithm every node starts propagating messages from the very first superstep. This way, multiple changes in various parts of the graph will cause changes in the computation from the very first supersteps, affecting a larger fraction of the computation.

4.2.3 *PageRank*

PageRank (PR) and the basic operation underlying the algorithm, which is power iterations, can be viewed as graph problems and are the base for several important algorithms, including machine learning algorithms and optimization problems. Here, we implement the PageRank algorithm as described in [17].

In Figures 2 and 4.2, we see that the PageRank algorithm is more sensitive to graph changes with respect to the computation saved. A single edge addition may cause the computation savings to drop to 29%, while for large update sizes, the computation and communication savings may both drop to less than 22%.

To better understand the difference, let us contrast PageRank with the previous two algorithms. Notice that in the previous algorithms whether a vertex propagates a message depends on an algorithm-specific condition. For instance, in SP if a message does not change the minimum distance the vertex does not propagate it. Similarly in CC, if a vertex does not change the minimum ID seen, it does not propagate the change. This condition effectively puts a bound on how much a change may affect the computation.

In PageRank, however, a vertex propagates messages to its neighbors at every superstep. At the same time, even if a graph update causes a negligible change in the state of a vertex, its current PageRank, such a change is caught and propagated by GraphInc. This is a natural limitation of attempting to automatically transform an algorithm to an incremental one.

Note, though, that other common variations of the PageRank algorithm may propagate messages only if the change in the PageRank of a vertex with respect to the previous iteration exceeds a threshold. This again introduces a condition for propagating messages that can potentially pose a bound on how much a change in the graph can affect the total computations.

4.3 Discussion

In the above, we examined applications with different computation and communication patterns. A common property of all these algorithms is that a single change has the potential to propagate across the entire graph. In general, the effect of a change may depend on data- and application-specific factors. For instance, the more connected the graph the more a change affects the computation. In the case of the SP algorithm, how much a change affects may also depend on the relative value of the weights in the graph.

Additionally, as Figure 4.2 shows the gain from incremental degrades as the update size increases. Even though in this study we want to analyze the limitations of our approach by looking at a wide range of workloads with respect to the update size, real graphs like the web graph typically change slowly [19]. Our own measurements with the graph of a large social network in Spain shows also that a typical hourly change in the graph is less than 0.01%. Therefore, at smaller time granularity, for instance minutes, we expect applications on top of real graphs to benefit more from our approach.

Further, Figure 4.2 also exhibits a non-linear relation between computation saved, which is more evident in the PageRank algorithm. This is due to the data dependencies that cause a single change to propagate to multiple computations. Although not studied in this paper, there are graph algorithms that exhibit more locality in the computation and communication. For example, a common operation used in many applications, such as fraud detection in telco Call Detail Record graphs [28], is the calculation of the friends-of-friends relation for every vertex. In such an algorithm, a single update to a vertex may affect only the vertices that are up to two hops away, potentially increasing the computation and communication savings.

Finally, we performed this study using a power-law graph, a type of graph that represents the worst case scenario for the applications we evaluate. Such graphs are well connected and have a small diameter, causing changes to propagate faster. However, many real large graphs like Facebook's social graph diverge from this type, for instance, because of limitations on the number of friends imposed by the social network or even user cognitive limitations [9]. Even Call Detail Record graphs exhibit much lower connectivity than social graphs [13]. Therefore, such graphs have better characteristics with respect to incremental processing.

5. DESIGN ISSUES

Our analysis shows that GraphInc's memoization approach to automatically converting algorithms to incremental ones is promising. However, building a system for real-time mining on large graphs raises certain design challenges. In this section, we briefly discuss two of the most important design aspects of GraphInc.

Memoization mechanism. GraphInc depends heavily on the ability to memoize and reuse messages and computed state. Saving computations must impose little overhead during the operation of the system. At the same time, to enable real-time operation GraphInc must be be able to access memoized results with low latency.

To achieve this, GraphInc partitions the graph into blocks with each block holding the memoized state of the corresponding vertices. Similarly to the Pregel and Giraph architectures, GraphInc assigns vertex partitions to worker machines and a worker is responsible for executing the compute function for all vertices it holds. In GraphInc, a worker is additionally responsible for accessing the memoized state blocks. A worker reads a block whenever a vertex contained in it must re-execute its compute function. Adapting the granularity of a block allows GraphInc to control the amount of data it reads off the disk and minimize I/O operations.

GraphInc implements a simple caching scheme that fetches and keeps blocks in memory. Subsequent vertex computations that require memoized state may be served from the cache or the disk. Modified cached blocks are written periodically back to disk or before they must be evicted from the cache. To further speed up state access, we have designed GraphInc to prefetch and cache memoized data using the graph structure as a hint for what state will be read next.

Graph partitioning. Performance depends on graph partitioning as it affects how computation is distributed across the compute machines and how much network communication occurs. In the case of GraphInc specifically, partitioning also affects how many blocks of memoized state must be read off the disk when changes propagate between vertices resident in different blocks. Currently, GraphInc partitions data to blocks randomly, an approach that achieves good load balancing. Although small updates do not result in reading many blocks off the disk, we are exploring how GraphInc can improve performance for a wide range of workloads with partitioning techniques that take locality into consideration [25, 23].

Handling update streams. Real-time operation requires the ability to handle potentially high rates of changes to the graph. To allow for low latency and consistency under streaming updates, GraphInc separates the storage of the graph structure from the storage required for Pregel computations and for memoization. This allows for incoming updates to the graph while computation is taking place, a technique also used in [6].

6. RELATED WORK

Various systems, like Percolator [21] and CBP [14], have been designed for incremental analytics. Such systems are targeted toward general analytics and require users to program custom incremental program. Instead, the goal of GraphInc is to automate this process and target graph mining analytics in specific. Kineograph [6], on the other hand, is built for real-time graph mining, but unlike GraphInc it requires users to specify custom incremental graph algorithms.

Systems like Incoop [3], Nectar [11], Comet [12] and DryadInc [22] are similar in that they advocate a transparent approach to incremental processing based on memoization. However, they are designed having in mind the MapReduce and DryadLINQ programming models and not for graph mining applications.

There is a large body of work on designing dynamic and streaming graph algorithms [18, 10, 26, 24, 25]. Such approaches aim to efficiently update the analysis with minimum work, but focus on calculating specific graph metrics. While these approaches are efficient, devising incremental graph mining algorithms remains a difficult task. Instead, GraphInc aims to create a generic mechanism for incremental graph mining on top of which many algorithms can be implemented.

7. CONCLUSION

Incremental processing is critical for real-time graph mining, however designing incremental graph algorithms is challenging. GraphInc allows users to write programs as if on batch workloads and automatically converts the algorithm to an incremental one by memoizing and reusing subcomputations. While graph algorithms are inherently difficult to compute incrementally, through a sample of real applications, our experiments illustrate the viability and limits of this approach. We showed that for certain algorithms, updates can be incorporated with minimal recomputation, while others trigger larger changes due to their computation and communication patterns. At the same time, this limitation opens interesting questions, such as the ability to automatically profile whether an application is amenable to incremental computation and tune the system accordingly, making the system more usable.

8. REFERENCES

[1] Apache Giraph Project. http://giraph.apache.org/.

[2] A. Bernstein and L. Roditty. Improved dynamic algorithms for maintaining approximate shortest paths under deletions. In *ACM-SIAM Symposium on Discrete Algorithms*, pages 1355–1365, Jan. 2011.

[3] P. Bhatotia, A. Wieder, R. Rodrigues, U. A. Acar, and R. Pasquini. Incoop: MapReduce for Incremental Computations. In *ACM Symposium on Cloud Computing*, Cascais, Portugal, Oct. 2011.

[4] Q. Cao, M. Sirivianos, X. Yang, and T. Pregueiro. Aiding the Detection of Fake Accounts in Large Scale Social Online Services. In *USENIX Symposium on Networked Systems Design and Implementation*, 2012.

[5] P. Chau. Catching bad guys with graph mining. *XRDS*, 17(3):16–18, 2011.

[6] R. Cheng, F. Yang, L. Zhou, and E. Chen. Kineograph: Taking the Pulse of a Fast-Changing and Connected World. In *EuroSys European Conference on Computer Systems*, Bern, Switzerland, Apr. 2012.

[7] S. Chien, C. Dwork, R. Kumar, D. R. Simon, and D. Sivakumar. Link Evolution: Analysis and Algorithms. *Internet Mathematics*, 1:277–304, 2004.

[8] U. Demiryurek, F. Banaei-kashani, and C. Shahabi. TransDec: A Data-Driven Framework for Decision-Making in Transportation Systems. In *Transportation Research Forum*, pages 1–15, Portland, OR, Mar. 2009.

[9] R. Dunbar. Neocortex size as a constraint on group size in primates. *Journal of Human Evolution*, 22(6):469–493, June 1992.

[10] W. Fan. Graph Pattern Matching Revised for Social Network Analysis. In *International Conference on Database Theory*, Berlin, Germany, Mar. 2012.

[11] P. K. Gunda, L. Ravindranath, C. A. Thekkath, Y. Yu, and L. Zhuang. Nectar: Automatic Management of Data and Computation in Datacenters. In *USENIX Symposium on Operating Systems Design and Implementation*, Vancouver, BC, Canada, Oct. 2010.

[12] B. He, M. Yang, Z. Guo, R. Chen, B. Su, W. Lin, and L. Zhou. Comet: batched stream processing for data intensive distributed computing. In *ACM Symposium on Cloud Computing*, pages 63–74, Indianapolis, IN, June 2010. ACM.

[13] N. Jiang, Y. Jin, A. Skudlark, W.-L. Hsu, G. Jacobson, S. Prakasam, and Z.-L. Zhang. Isolating and Analyzing Fraud Activities in a Large Cellular Network via Voice Call Graph Analysis. In *International Conference on Mobile Systems, Applications, and Services*, Low Wood Bay, Lake District, UK, June 2012.

[14] D. Logothetis, C. Olston, B. Reed, K. Webb, and K. Yocum. Stateful Bulk Processing for Incremental Analytics. In *ACM Symposium on Cloud Computing*, Indianapolis, IN, June 2010.

[15] Y. Low, J. Gonzalez, D. Bickson, C. Guestrin, and J. M. Hellerstein. Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud. *The VLDB Endowment*, pages 716–727, Aug. 2012.

[16] S. Ma, Y. Cao, J. Huai, and T. Wo. Distributed graph pattern matching. In *International Conference on World Wide Web*, page 949, New York, New York, USA, 2012. ACM Press.

[17] 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. *ACM SIGMOD International Conference on Management of Data*, 2010.

[18] H. Ning, W. Xu, Y. Chi, Y. Gong, and T. Huang. Incremental Spectral Clustering With Application to Monitoring of Evolving Blog Communities. In *SIAM International Conference on Data Mining*, pages 261–272, 2007.

[19] A. Ntoulas, J. Cho, and C. Olston. What's new on the web?: the evolution of the web from a search engine perspective. In *International Conference on World Wide Web*, page 1, New York, New York, USA, May 2004. ACM Press.

[20] C. Olston, G. Chiou, L. Chitnis, F. Liu, Y. Han, M. Larsson, A. Neumann, V. B. N. Rao, V. Sankarasubramanian, S. Seth, C. Tian, T. Zicornell, and X. Wang. Nova: Continuous Pig / Hadoop Workflows. In *International Conference on Management of Data*, Athens, Greece, June 2011.

[21] D. Peng and F. Dabek. Large-scale incremental processing using distributed transactions and notifications. In *USENIX Symposium on Operating Systems Design and Implementation*, pages 1–15, Vancouver, BC, Canada, Oct. 2010.

[22] L. Popa, M. Budiu, Y. Yu, and M. Isard. DryadInc: Reusing work in large-scale computations. In *USENIX Workshop on Hot Topics in Cloud Computing*, page 21. USENIX Association, 2009.

[23] J. M. Pujol, V. Erramilli, G. Siganos, X. Yang, N. Laoutaris, P. Chhabra, and P. Rodriguez. The little engine(s) that could: scaling online social networks. *ACM SIGCOMM Computer Communication Review*, 41(4):375–375–375–386–386–386, Oct. 2011.

[24] A. D. Sarma, S. Gollapudi, and R. Panigrahy. Estimating PageRank on Graph Streams. In *ACM Principles of Database Systems*, Vancouver, BC, Canada, June 2008.

[25] I. Stanton and G. Kliot. Streaming Graph Partitioning for Large Distributed Graphs. In *ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2012.

[26] H. Tong, P. S. Yu, and C. Faloutsos. Proximity Tracking on Time-Evolving Bipartite Graphs. In *SIAM International Conference on Data Mining*, Atlanta, GA, Apr. 2008.

[27] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, Aug. 1990.

[28] S. Weigert, M. Hiltunen, and C. Fetzer. Mining large distributed log data in near real time. In *Workshop on Managing Large-Scale Systems via the Analysis of System Logs and the Application of Machine Learning Techniques*, Cascais, Portugal, Oct. 2011.

[29] S. H. Yang, B. Long, and A. Smola. Like like alike - Joint Friendship and Interest Propagation in Social Networks. In *International Conference on World Wide Web*, Hyderabad, India, Mar. 2011.