Distributed Incremental Pattern Matching on Streaming Graphs

Jyun-Sheng Kao
National Tsing Hua University
Computer Science Dept.
Hsinchu, Taiwan
joshkao@lsalab.cs.nthu.edu.tw

Jerry Chou
National Tsing Hua University
Computer Science Dept.
Hsinchu, Taiwan
jchou@lsalab.cs.nthu.edu.tw

ABSTRACT

Big data has shifted the computing paradigm of data analysis. While some of the data can be treated as simple texts or independent data records, many other applications have data with structural patterns which are modeled as a graph, such as social media, road network traffic and smart grid, etc. However, there is still limited amount of work has been done to address the velocity problem of graph processing. In this work, we aim to develop a distributed processing system for solving pattern matching queries on streaming graphs where graphs evolve over time upon the arrives of streaming graph update events. To achieve the goal, we proposed an incremental pattern matching algorithm and implemented it on GPS, a vertex centric distributed graph computing framework. We also extended the GPS framework to support streaming graph, and adapted a subgraph-centric data model to further reduce communication overhead and system performance. Our evaluation using real wiki trace shows that our approach achieves a 3x $\sim$ 10x speedup over the batch algorithm, and significantly reduces network and memory usage.

Keywords

Streaming data, Graph pattern matching, Incremental algorithm, Distributed computing

1. INTRODUCTION

Big data has shifted the computing paradigm of data analysis, and allows people to explore the values from dark data by addressing the challenges of volume, variety, velocity, and other ones, such as veracity. While some of the data can be treated as simple texts or independent data records, many other applications have data with structural patterns which are modeled as graph, such as social media from Twitter and Facebook, Web graph from Wiki, road network traffic and smart grid from civil engineering, and bioinformatics from science domain. Therefore, several graph processing frameworks based on batch processing model have been developed to deal with the volume and variety of graph data. To name a few, these include GraphLab [13], Giraph [1], GPS [19], GraphX [25] and Mizan [12]. However, driven by the recent emerging applications of IoT (Internet of Things) and the advancement of data collection technologies, there is an increasing need to process and analyze streaming data in real-time fashion. Unfortunately, until recently, there is still limited amount of work has been done to address the velocity problem of graph processing.

In this work, we aim to develop a distributed processing system for solving pattern matching queries on streaming graphs where graphs evolve over time upon the arrives of streaming graph update events. Graph pattern matching is a problem seeking to find the subgraphs of a data graph that are similar to a given query graph, and it has been widely used to search and analyze data with graph structure. Variants of the problem have been extensively studied, but they are shown to be costly to compute [9, 6, 11]. But under the big data paradigm, we are facing the challenges of increasing data graph size, and more frequent graph topology updates. Hence, the traditional batch algorithms that recompute results from scratch upon each graph update event has become prohibitively expensive.

While real-life graph has shown the dynamic nature with frequent data updates, the changes are typically small. As reported by [17], only 5% to 10% of nodes from a Web graph are updated weekly. We have also observed a similar behavior from the Wikipedia website where only 0.32% to 0.35% of edges are updated weekly. Therefore, in order to solve aforementioned problem and deliver query results in timing manner, we built our solution based on an incremental pattern matching algorithm, and implemented it on GPS, a vertex centric distributed graph computing framework. In our system, users submit a set of graph pattern matching queries, and they are notified in real-time when matching results are changed by the graph updates. As opposed to batch algorithm recomputes results from scratch, incremental algorithms only trigger computations on the vertices whose matching status is changed by the graph update events. Our implementation on GPS further optimizes the performance by adapting subgraph-centric data model to minimize communication traffic among vertices. As shown by our evaluation results based on a real-world Wikipedia dataset, we achieved a 3x $\sim$ 10x speedup over the batch algorithm, and consumes much less network and computer resources.

As discussed in Section 6, our work is motivated by several recent studies. These include the distributed graph pattern

DOI: 10.1145/1235

matching algorithms proposed in [8, 7, 14], the sequential algorithm for incremental graph pattern matching algorithms described in [5], and the distributed computing frameworks for streaming graph processing discussed in [4, 24]. However, to our best knowledge, incremental graph pattern matching algorithms have not been proposed or implemented in any distributed computing framework. Thus we are the first work that solves and evaluates the incremental graph pattern matching problem on distributed computing framework, and we provide extensive study on the problem in various aspects including algorithm design, implementation technique and performance evaluation.

The rest of this paper is structured as follows. Section 2 introduces the pattern matching problem. Our proposed incremental algorithm and implementation are described in Section 3 and Section 4, respectively. Our experimental results are presented in Section 5. Finally, related work is discussed in Section 6, and the paper is concluded in Section 7.

2. PROBLEM DEFINITION

In our problem description, a graph is denoted as $G = (V, E, f)$. V is the set of vertices in G. $E \subseteq V \times V$ is the set of edges where (u, v) denotes an edge from vertex u to v. $f(\cdot)$ is a function that associates each vertex $v \in V$, so that $f(v)$ returns a tuple of attributes $(A_1 = a_1, \cdots, A_n = a_n)$, where A_i is an attribute name, and a_i is an attribute value. The attributes are used to describe the contents carried on a vertex, such as labels, keywords, text, etc.

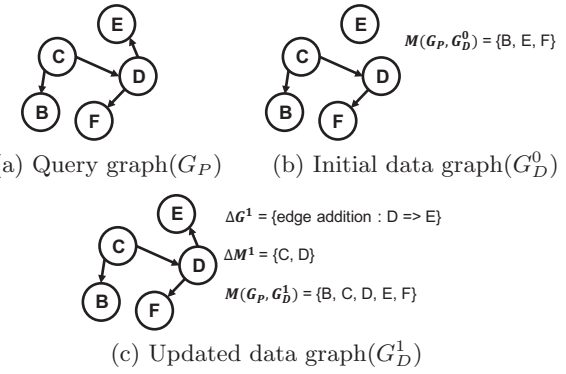
Given a data graph and a pattern graph, variants of the graph pattern matching models have been proposed that form a spectrum with respect to the stringency of the matching conditions. *Subgraph isomorphism* is the most restrictive model, but it has been proven to be intractable [6]. Hence, in this work, we discuss our graph pattern matching problem defined by the *graph simulation* model [11], which is the least restrictive model and it has been commonly used in social community detection [3], wireless and mobile network analyses [10], etc.

First we revisit the definition of graph simulation model [11] as below. A data graph $G_D = (V_D, E_D, f_D)$ matches a pattern graph $G_P = (V_P, E_P, f_P)$, if there exists a binary relation $R \subseteq V_P \times V_D$ which satisfies the following constraints:

Constraints of graph simulation model for pattern matching:

1. *Vertex constraint*: If $(v, v') \in R$, then $f_P(v) \subseteq f_D(v')$;
2. *Edge constraint*: If $(v, v') \in R$, then $\exists (u, u') \in R$, such that $(u, v) \in E_P$, and $(u', v') \in E_D$.
3. *Graph constraint*: $\forall v \in V_P$, $\exists v' \in V_D$ such that $(v, v') \in R$;

When G_D matches G_P, there exists a unique maximum match, such that R contains the most number of vertices. The pattern matching problem is to compute the maximum match from the data graph denoted as a matchset, $M(G_P, G_D)$, such that $M(G_P, G_D) = \{v \mid (v, v') \in R, v \in V_P, v' \in V_D\}$. Figure 1 illustrates the above definition when giving a pattern graph shown in Figure 1(a) and a data graph shown in Figure 1(b). In the example, all the vertices in the data graph satisfy the *vertex constraint* because all of them can find another vertex in pattern graph with the same attribute value. However, vertex D doesn't satisfy the



(a) Query graph(G_P) (b) Initial data graph(G_D^0)

(c) Updated data graph(G_D^1)

Figure 1: Illustration about the member of match set in differnet query and data graphs based on constraints of graph simulation.

edge constraint because it has an outgoing to E in the pattern graph, but the edge does not exist in the data graph. In consequence, vertex C violates the *edge constraint* as well because D is one of its children in the pattern graph, but it is not in the matchset. On the other hand, vertex B, E and F satisfy the *edge constraint* because all their children are in the matchset. Therefore, the matchset only contains vertex B, E and F. Since the members in the matchset does not cover all the vertices from pattern graph, the query result is false.

Now we extend the pattern matching problem to a streaming graph where a data graph G_D are changed over time by an unbounded sequence of updates $S = \{\delta_1, \cdots, \delta_n, \cdots\}$, where an update δ_i can be any of the following events: vertex addition/removal, edge addition/removal, and attribute addition/removal. Due to the high velocity of graph streams and the real time processing constraint, matching operations are considered to be performed periodically. Thus, G_D^t is used to denote the graph at time interval t after all the update events until time t are applied to the initial data graph G_D. $\triangle G^t$ is used to denote the graph changes in time interval t from its previous time interval; that is $\triangle G^t = G_D^t - G_D^{t-1}$. Accordingly, at every time interval t, the traditional batch algorithm aims to find the matching result $M(G_P, G_D^t)$ from scratch. In contrast, an incremental algorithm leverages the previous matching result $M(G_P, G_D^{t-1})$, and aims to find the matching change $\triangle M^t$, such that $M(G_P, G_D^{t-1} \oplus \triangle G^t) = M(G_P, G_D^{t-1}) \oplus \triangle M^t$. Figure 1(c) illustrates the definition of incremental pattern matching by assuming an edge from D to E is added at the next time interval. In the updated data graph, vertex C and D now can both satisfy the *edge constraint* because all their outgoing edges in pattern graph can be found in data graph, and all their children are in matchset as well. Therefore, the goal of incremental algorithm is to find the matchset difference $\triangle M^t = \{C, D\}$.

3. ALGORITHM

To enable incremental pattern matching on a distributed computing framework, this section presents our algorithm under BSP(Bulk Synchronous Parallel)-based vertex-centric programming model. It is the programming model used by many distributed graph processing systems, including Pregel [15], GPS [19], and others [1, 25, 13]. In this model,

computation is a series of supersteps. Each superstep consists of three ordered stages: (1) concurrent computation, (2) communication, and (3) barrier synchronization. Each vertex of the data graph is a computing unit which can be conceptually mapped to a process in the BSP model. Each vertex only records its own local information, and the same predefined function is executed by all the vertices in the stage of computation. Then, vertices can exchange messages in the stage of communication to learn the status of its neighbors. Finally, a vertex votes to halt and goes to inactive mode when it believe it has accomplished its tasks and reach a local termination condition. It remains inactive until it is triggered externally by a message from another vertex. The computation terminates when all vertices become inactive.

In our approach, a vertex stores its local data graph information including the attributes and outgoing/incoming edges. The information is initialized when the data graph is loaded into our system, and then modified according to the update events during runtime. Besides, a buffer is allocated on each vertex to keep the arrival update events between update intervals. It is noted that an update event of an edge is only buffered at its source vertex to reduce duplicate information in the system. On the other hand, the pattern graph of users' matching query is duplicated and stored on every vertex. But this only causes limited overhead because in practice the size of pattern graph is relatively small compared to the size of data graph, and the number of queries subscribed in the system is limited compared to the number of update events that stream into system with high velocity. Finally, a match flag and a matchset variable are kept on each vertex to indicate its current matching status. A match flag is set to False, if its vertex is not satisfied the *vertex* or *edge constraint*. Otherwise the match flag is set to True. The matchset records its potential match vertices from the pattern graph. It is noted that because our system uses incremental algorithm to deal with streaming graph updates, the status of match flag and matchset from the previous update event ($M(G_P, G_D^{t-1})$) will be kept to detect the matching change of the next update event ($\triangle M^t$). In contrary, a batch matching algorithm will reset the match flag on all vertices and re-compute them from scratch for each arrival update. In the rest of section, we explain our vertex-centric incremental algorithm based on the above information.

To simplify the description of our algorithm, our discussion below is based on a basic update event with the following assumptions (a) only one pattern graph query exists in the system, (b) only one "update" event arrives in between graph update intervals, and (c) the update is either adding or removing an edge from the data graph. However, the algorithm can be generalized to remove all these assumptions to handle multiple queries, multiple updates and any type of graph update events as we briefly explain at the end of the section. The generalized algorithm has been implemented to handle a real Wiki trace with multiple quires and updates in our experimental evaluations.

Figure 1 shows the pesudecode of our algorithm, which can be divided into three phases in order to handle an arrival edge update event, and re-evaluate the matching results according the three constraints defined in Section 2. Due to the BSP programming model, each phase may be implemented in one or multiple supersteps. For the ease of our explanation, we abbreviate the source vertex of the update event as "the source", and the destination vertex of the update event as "the destination". We also use the term children (parents) to refer the set of vertices connected by the outgoing (incoming) edges of a vertex. We describe the supersteps of each phase as follows.

The "*update*" phase lets both the source and destination to update their data graph information according to the update event. Since the update event is only sent to the source, the source updates its data graph first in superstep 1, and then notifies the destination. After receiving an update message from the source, the destination updates its data graph in superstep 2. It is noted that the destination only sends its matchset to the source and trigger the next phase if its match flag was True. This is because is the destination was not matched to any vertex in the pattern graph, then the update definitely will not change the match result of the source. Thus, the matching processing can terminate directly.

After the data graph is updated, the "*self-checking*" phase lets the source using on its local information to quickly filter the update events that won't change match results. Specifically, under incremental algorithm, it is easy to see that an update definitely won't change the match flag if one of the following condition occurs: (1) the source is not in the pattern graph; (2) the source is already in the matchset, and the update event is adding another outgoing edge; (3) the source is not in the matchset, and the update event is removing another outgoing edge. Therefore, only when all of the conditions above don't exist, the source will then send a re-evaluate message to itself to trigger the next propagation phase. As observed from our evaluation results, more than 75% of update events could be filtered by the previous two checking phases in our experiments, and it gives a significant advantage to the incremental algorithm over the traditional batch algorithm.

Finally, the "*propagation*" phase lets a vertex to re-evaluation its matching status. Different from the *self-checking* phase which only checks the *vertex constraint*, this phase checks the *edge constraint*. Therefore, it needs multiple supersteps for collecting the match status from all its children before re-evaluating its match result. Specifically, in superstep 4, requests are sent to all children to collect their matching status. The requests are replied by the children in superstep 5. Then the matching status is re-evaluated according to the *edge constraint* in superstep 6. Based on the matching result, if the matching status altered, then the vertex must inform all its parents to re-evaluate their matching status as well. The propagation phase will repeat until there is no more propagation messages generated from any vertex. Then the master process of the graph processing system will collect the match results from all vertices and evaluate the last *graph constraint*.

As mentioned, all the predefined assumptions of our algorithm can be removed in brief as follows. To handle multiple queries, the match flag can be defined as an array to record the match result of each query. To handle multiple update events in a time interval, all those update events can be processed simultaneously through the procedure described in the algorithm. Finally, all other types of update events can be converted into a set of edge addition/removal events. For instance, adding edge attribute can be considered as an edge addition with new attribute values, and adding a vertex can be considered as a set of edge addition for each of

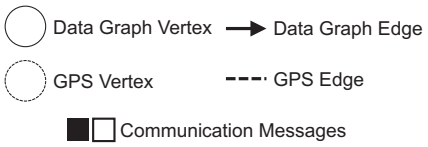
Algorithm 1: Vertex-centric incremental graph pattern matching algorithm

update phase
▷*superstep 1*: on the source vertex
if *receives an update from buffer* **then**
  • update its outgoing edge and attributes
  • send the update event to the destination
else
  • vote to halt
end
▷*superstep 2*: on the destination vertex
if *receives an update message* **then**
  • update its incoming edge
  if *it is not in the pattern graph* **then**
    • vote to halt
  else
    • send its matchset to the source
  end
else
  • vote to halt
end

self-checking phase
▷*superstep 3*: on the source vertex v
if *receives a message from the destination* **and** *the match flag could be changed due to update* **then**
  • send a re-evaluation message to itself
else
  • vote to halt
end

propagation phase (repeat until termination)
▷*superstep 4*: on any vertex
if *receives a re-evaluation message* **then**
  • ask children about their matching status
else
  • vote to halt
end
▷*superstep 5*: on any vertex
if *receives a message* **and** *match flag is True* **then**
  • reply its matchset
else
  • vote to halt
end
▷*superstep 6* on any vertex
if *receives matchset from its children* **then**
  • re-evaluate its own match flag and matchset
  if *the match flag is altered* **then**
    • send a re-evaluation message to all its parents
  else
    • vote to halt
  end
else
  •vote to halt
end

(a) Viewpoint of data subgraph. (b) Viewpoint of GPS.



(c) Legend of 2(a) and 2(b)

Figure 2: An illustration of a data graph on GPS under the subgraph-centric data model. Data graph is partitioned into subgraphs, and each subgraph is mapped to a single vertex in GPS. The messages from a subgraph are merged and encapsulated before transmission to reduce network traffic.

the outgoing edges from the new vertex.

As can be seen from our pesudecode, in contrast to a batch algorithm that always has to re-compute all matching results from scratch, our incremental algorithm only trigger re-computations on the vertices affected by the data graph update event. Moreover, the propagation phases can be prevented by the checking from the first two phases. Therefore, better performance can be achieved by having fewer number of supersteps, fewer number of active vertices, and less communications.

4. IMPLEMENTATION

This section discusses the details of our implementation. In particularly, we aim to address two key challenges for supporting incremental graph pattern matching on streaming graph. One is to support streaming graph on a traditional graph processing system such as GPS. The other one is to minimize the network overhead of our incremental algorithm. To achieve these goals, we adapt a subgraph-centric data model in our implementation as described below.

Support streaming graph is a challenge problem as discussed by several recent works [4, 24, 21, 16]. A few prototype systems have been proposed [4, 24], but they are not available to public yet. On the other hand, traditional distributed graph processing systems such like GPS don't support data graph mutation in runtime. Thus, in order to support streaming graph update in our system, we adapt a subgraph-centric data model to overcome the limitation and implement our solution on GPS. Specifically, we partition data graph into several disjoint subgraphs. Then, the information of a subgraph is encapsulated into a data object stored in a single vertex in GPS as shown in Figure 2. Accordingly, in the rest of section, we use *logical* graph to refer the original data graph, and we use *physical* graph to refer the graph seen by GPS.

Once introducing the subgraph-centric data model into our system, a graph update can be performed by editing the vertex data in GPS. Furthermore, our implementation uses a hash function to place a logical vertex from the data graph onto a physical vertex in GPS according to the vertex id. Hence, a vertex id in GPS can also be referred as a subgraph

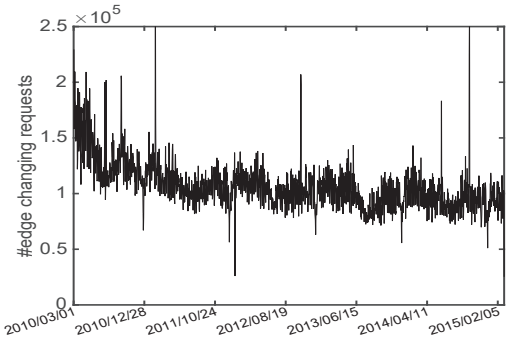


Figure 3: The number of edge updates in Wiki trace from 2010/3 to 2015/2.

id in our logical graph. With the help of using hashing function, we ensure the physical vertex (i.e., subgraph) id of a logical vertex can be found in constant time without additional computation or communication overhead. It also minimizes the effort to implement our incremental algorithm in the subgraph-centric data model.

Previous work from [22, 20] has shown subgraph-centric data model can be used to support asynchronous computation by allowing the vertex within a subgraph to read the data from each other directly without restricting by the message passing and superstep synchronous communication model. However, because pattern matching problem has dependency between each pair of edges, it is a class of problem that cannot benefit from asynchronous communication. Therefore, different from the previous study, our work focus on minimizing the network communication overhead by packaging the message from a subgraph before transmission. It is known that graph algorithms are mostly bounded by communication. In particularly, by using an incremental algorithm, lots of small messages are generated by vertices which often causes longer latency delay and lower network bandwidth utilization. Therefore, packaging the message from all the vertices in a subgraph together before transmitting can significantly network utilization. Moreover, in our pattern matching, we can greatly reduce the amount of network traffic by removing the redundant message among vertices when the vertices in the same subgraph ask data from their common children in another subgraph. It is a common scenario in graph pattern matching algorithm, especially when the ratio between the number of edges and the number of vertices gets higher. Our evaluation also shows this implementation technique can effectively reduce the network and memory consumptions during computations.

5. EXPERIMENTAL EVALUATION

We implemented our incremental graph patten matching algorithm on GPS. This section presents our experimental evaluation results using real-life trace from Wikipedia. Our goal is to compare the performance between incremental and batch algorithm, analyze the resource usage and performance bottleneck of our implementation, and investigate the performance impact of using the subgraph-centric data model.

5.1 Environment Setup & Wikipedia Dataset

We implemented our approach on the GPS platform [19], which is a vertex centric graph processing system similar to Google's proprietary Pregel system [15]. The experiments were conducted in the Amazon EC2 environment by launching by computing clusters with a master node, and 16 compute nodes. Each node is a r3.xlarge instances with 4 virtual cores and 30.5 GBytes of RAM. Besides evaluate the system performance based on the GPS logging information, we also installed a monitor tool (NMON) to collect the fine-grained (in seconds) resource usage information of compute nodes, including CPU utilization, memory usage, disk I/O and network traffic, etc.

To evaluate our pattern matching problem on streaming graphs, we use the real-life data collected from Wikipedia, which is a large-scaled Web graph where each web page can be seen as a vertex, the hyperlinks connect between these pages can be seen as an edge, and the content of each web page can be seen as the attributes of a vertex. The graph constantly changes as people editing the web page content and adding the web page or inner links. For every Wikipedia web page, it records all its previous version after each content update from a user since 2001. Therefore, we are able to regenerate the graph update events from parsing the dataset and identify the content changes over time. Until 2015/03, there are about 9.5 million pages and about 199 million inner page links. We report the number of total vertices, and updated vertices from 2010/3 to 2015/2 in Figure 3, and we found that only 0.32% to 0.35% of edges are updated weekly. Therefore, incremental algorithm is obviously needed to process the dataset more efficiently.

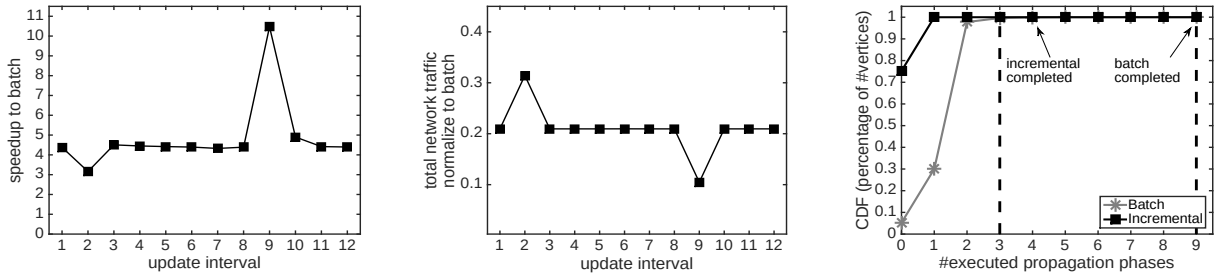
In the experiments, we generate pattern graphs with different structures that are widely used in real-world applications, including star, line, ring and tree. For instance, line structure pattern is used by biologists to find the occurrence of a specified DNA sequence; star structure pattern can be used to find the links of a webpage or the citation of a paper. The diameter of these pattern graphs varies from 3 to 8, and the maximum degree of vertices is 4. We choose the dataset snapshot at 0 o'clock on March 1st 2015 as the initial data graph. Each update interval contains 600 edge update events. At each update interval, the graph is updated and query result is re-computed. We present the results over 12 intervals to show the consistency of our improvement.

5.2 Incremental vs. Batch Algorithm

Here we compare the performance between incremental and batch algorithms. Both algorithms were implemented by the same code as shown in Figure 1, but the match algorithm resets its match status on vertices and re-compute the results from scratch at every time interval. The comparison results are summarized in Figure 4.

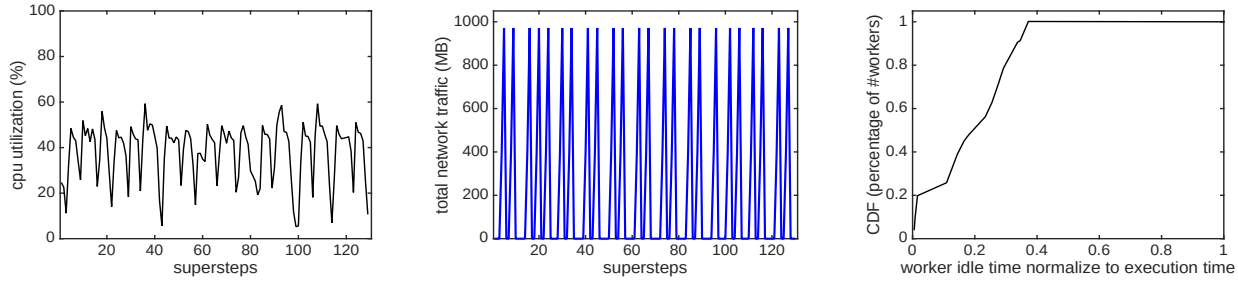
We compare the overall performance by showing the speedup of increment algorithm over batch algorithm in term of execution time in Figure 4(a). At each update interval, we collect the total execution time of the two algorithms, and plot the speedup factor in the figure. During these 12 update intervals, the incremental algorithm shows a consistent performance improvement with 3x $\sim$ 10x speedup to the batch algorithm.

Since the graph algorithm is known to be a network bound problem, we further investigate the performance by plotting the total network traffic at the each time interval in Figure 4(b). As shown by the results, the network traffic is significantly reduced by more than 60% when using



(a) Performance improvement (b) Network traffic amount (c) Degree of convergence

Figure 4: The comparison results between incremental and batch algorithm in terms of computation time, network traffic and degree of convergence.



(a) Average cpu utilization of machines (b) Total network traffic (c) CDF idle time on workers.

Figure 5: Resource usage and idle time analysis from the first update interval.

the incremental algorithm, and the amount of traffic reduction shows strong correlation to the speedup factor. Hence, higher speedup and more traffic reduction is observed at update interval 9, while lower speedup and the less traffic reduction is observed at update interval 2.

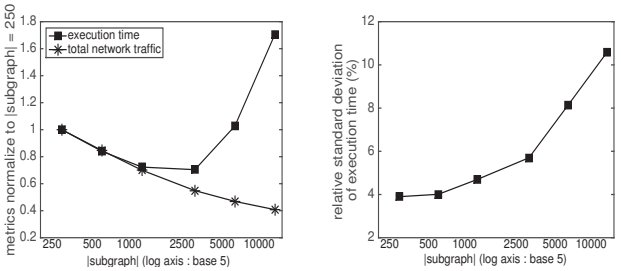
Finally, we can explain the reason of traffic reduction by comparing how the algorithms were executed on vertices. As described in Section 3, both algorithms must repeat the propagation phase until the results become stable. Hence, we draw the CDF of vertices according to the number of executed propagation phases in Figure 4(c).

As observed, only 30% of vertices execute the propagation phases once or less in batch. In contrast, almost all the vertices execute the propagation phases once or less in incremental. We even observed 75% of vertices didn't execute the propagation phase at all because the update events were filtered by the first two checking phases in our incremental algorithm. Therefore, the amount of network traffic can be reduced significantly. Moreover, the incremental algorithm can also reduce the maximum number of propagation phases on vertices. For instance, the maximum number of propagations is reduced from 9 to 3 in our experiments. Since graph processing system is synchronous computation, the higher number of propagation phases implies more superstep and longer execution. Due to the above reason, the network traffic and computation time can be greatly reduced by the incremental algorithm.

5.3 Resource Usage Analysis

Next, we present the resource usage and workload distribution among compute nodes for our incremental algorithm. Figure 5 shows the change of CPU utilization and network traffic over time. Since graph processing is consisted of supersteps, we report the resource usage from each supersteps during computation. Clearly, the resource usage varied widely between supersteps because each superstep may perform tasks with a different amount of computation and communication workload. However, it is clear that CPU utilization is relatively low with peak loading at only 60%. This is not a surprise result because graph processing is known to be network bound. On the other hand, the network traffic varied much more drastically between 970 MBytes to 0 MBytes. The 0 MBytes is due to the superstep that only performs computation. This behavior is commonly seen in incremental algorithm, because most of the vertices may vote to halt immediately after one superstep. Therefore, the traffic loading is much less stable over time for incremental algorithm, and it is a problem that could be addressed in the future to achieve higher network utilization.

From our resource monitor trace, we also found the CPU loading are not balanced between compute nodes. Therefore, we investigate the problem by recording the total idle time from each GPS worker process during the execution of a single update interval. The CDF of workers with respect to their normalized idle time is plotted in Figure 5(c). A 50% normalized idle time means a worker spends half of its execution in idle state waiting for communication message. Thus, as shown by Figure 5(c), only 20% of the GPS workers have almost no idle time. The rest of 80% GPS workers all experienced a lengthty idle time upto 40% of the execution time. This is caused by the unbalanced workload among GPS worker under synchronous computation. Therefore, all the results above indicate the need of techniques like workload migration.



(a) Performance improvement and network traffic reduction

(b) Degree of workload imbalance

Figure 6: Performance comparison under different subgraph sizes

5.4 Subgraph Size Analysis

As described in Section 4, we adapt a subgraph-centric data model to reduce network traffic. Hence, we evaluate the performance impact of the subgraph size and summarize the results in Figure 6. First of all, Figure 6(a) compares the execution time and network traffic under subgraph size varied from 250 vertices to 10,000 vertices. As expected, the network traffic is reduced as the subgraph size increase. This is because more messages can be encapsulated together to reduce redundant information when using larger subgraph size.

Less network loading also helps to reduce the execution time as we observed when the subgraph size is less than 2,500. However, as the subgraph size continues to increase over 2,500, surprisingly the execution time starts to grow. After further investigation, the growing execution time is caused by the unbalanced workload under larger subgraph size. As known, all the vertices in a subgraph are mapped into a single GPS vertex, and then executed sequentially by a GPS worker. For the incremental pattern matching problem, the active vertices are located in a near region because they have to be triggered by their neighbors. As a result, the chance of active vertices located in the same subgraph becomes higher when the subgraph size gets larger, and therefore causes less balanced workload. We verify this reason by plotting the relative standard deviation(RSD) of worker execution time in Figure 6(b). A larger value of RSD implies higher variance between the worker execution time. Clearly, there is a strong correlation between the RSD value and the execution time when the subgraph size is too large. Therefore, the result implies the subgraph size should be chosen more carefully, and an optimal subgraph setting exists to achieve the best performance.

6. RELATED WORK

The graph pattern matching problem has been studied extensively by the previous work. Much of the work has been in graph theory to analyze the complexity of graph pattern matching problem. Several variants of the problem have been proposed with respect to the stringency of the matching. [9] shows subgraph isomorphism is NP-complete. [6] shows bounded simulation is cubic-time, and [11] shows graph simulation is quadratic-time. Others discussed the design of incremental pattern matching algorithms. [5] shows the incremental algorithm for pattern matching in different definitions and proves the complexity of these algorithms is bounded or unbounded depending on the complexity is related to the size of *changes* or not. But only a few works have designed the algorithm and implement the algorithm in distributed computing environment. One of these attempts is [8], which proposed the distributed vertex-centric algorithm for several different graph simulation models, and also evaluated their performance on GPS. Differ to prior works, we propose a distributed incremental graph pattern matching for graph simulation. Besides, we view subgraph as computation unit to reduce crossing edges and eliminate redundant messages.

[7] proves the response time of distributed graph simulation is related to the number of edges across different fragments. Thus, partitioning graph evenly across machines can have the impact on performance. [19] has shown that the dynamic migration mechanism doesn't speedup performance unless it runs long enough. In the case of dealing a streaming graph, the data graph is persistently stored in the system for processing arrival jobs. Therefore, our problem can benefit from dynamic migration. As shown by [23], the heuristic dynamic strategy they proposed can save over half of execution time than hash-partitioning on large-scale dynamic graph. Under our subgraph-centric data mode, vertex can be more easily migrated between subgraphs, so we plan to explore dynamic graph partition in the future.

While several batch graph processing systems [13, 1, 19, 25] have been developed, many open questions still remain to support streaming graphs. Kineograph [4] is a distributed system supporting graph topology mutations with steaming incoming graph updates. [24] purposes a general model to support query analytic on streaming graphs, and it lists several research directions and challenges for implementing one such system. Different from these works, we focus on the performance evaluation and optimization of a specific application called graph simulation pattern matching problem, and we adapt a subgraph-centric data model to reduce network traffic. Finally, X-stream [18] and Stinger [2] are streaming graph systems on a single shared memory machine, and they aim to support fast update and access to the graph topology on disk or in memory. Different from these works, we aim to address the network bottleneck problem of a distributed graph processing system rather than the graph update problem on a single machine.

7. CONCLUSIONS

This work investigates the velocity problem of graph processing. We tackle this problem by developing a system to support the pattern matching queries on streaming graphs. Our main contributions include the following: (1) It is the first work to propose a distributed incremental graph pattern matching algorithm and implement it on a vertex-centric computing framework. (2) We adapt a subgraph-centric data model to extend GPS for handling graph mutation and optimizing performance. (3) Our implementation is evaluated using a real-life trace from Wikipedia to analyze the performance bottleneck and the advantage of incremental pattern matching algorithm.

As opposed to the previous works that either focus on studying the pattern matching algorithms [11, 9, 6, 5] on single machine or develop a general data analysis system [4, 24, 21, 16] for stream graphs, our work provides a more thorough study of the incremental pattern matching pattern

problem on a real implementation, and provide the following observations. First, incremental algorithms are essential to solve data analytic problem on streaming graphs. In our approach, a 3x $\sim$ 10x speedup over the batch algorithm were observed. The improvement is expected to grow even further as the data size increases. Second, the performance of graph processing is mainly bounded by communication overhead and unbalanced workload. It is a difficult problem to solve because of the nature of graph algorithm. But we did find that using subgraph-centric data model can effectively reduce the amount of network traffic for our pattern matching algorithm and likely for other algorithms having duplicate message among neighbors. Therefore, more performance optimization techniques and resource management at the subgraph level should be considered in the future. Last but not least, due to the dynamic nature of streaming graph and query driven analysis, a more adaptive technique is required to manage the data graph. As shown by our results, the adjustment of subgraph size is critical to the performance, but it is still an open question on how to find the proper setting and how to change subgraph size efficiently at runtime. Therefore, we plan to further investigate these open questions in the future.

8. REFERENCES

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

[2] D. A. Bader, J. Berry, A. Amos-Binks, D. Chavarría-Miranda, C. Hastings, K. Madduri, and S. C. Poulos. Stinger: Spatio-temporal interaction networks and graphs (sting) extensible representation. *Georgia Institute of Technology, Tech. Rep*, 2009.

[3] J. Brynielsson, J. Hogberg, L. Kaati, C. Martenson, and P. Svenson. Detecting social positions using simulation. In *Advances in Social Networks Analysis and Mining, International Conference on*, pages 48–55, Aug 2010.

[4] R. Cheng, J. Hong, A. Kyrola, Y. Miao, X. Weng, M. Wu, F. Yang, L. Zhou, F. Zhao, and E. Chen. Kineograph: Taking the pulse of a fast-changing and connected world. In *ACM EuroSys*, pages 85–98, 2012.

[5] W. Fan, J. Li, J. Luo, Z. Tan, X. Wang, and Y. Wu. Incremental graph pattern matching. In *ACM SIGCOM*.

[6] W. Fan, J. Li, S. Ma, N. Tang, Y. Wu, and Y. Wu. Graph pattern matching: From intractable to polynomial time. *Proc. VLDB Endow.*, 3(1-2):264–275, Sept. 2010.

[7] W. Fan, X. Wang, Y. Wu, and D. Deng. Distributed graph simulation: Impossibility and possibility. *Proc. VLDB Endow.*, 7(12):1083–1094, Aug. 2014.

[8] A. Fard, M. Nisar, L. Ramaswamy, J. Miller, and M. Saltz. A distributed vertex-centric approach for pattern matching in massive graphs. In *Big Data, 2013 IEEE International Conference on*, pages 403–411, Oct 2013.

[9] M. R. Garey and D. S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1990.

[10] J. C. Godskesen and S. Nanz. Mobility models and behavioural equivalence for wireless networks. In *Coordination Models and Languages*, pages 106–122, 2009.

[11] M. R. Henzinger, T. A. Henzinger, and P. W. Kopke. Computing simulations on finite and infinite graphs. In *Proceedings of the 36th Annual Symposium on Foundations of Computer Science*, FOCS '95, pages 453–, 1995.

[12] Z. Khayyat, K. Awara, A. Alonazi, H. Jamjoom, D. Williams, and P. Kalnis. Mizan: a system for dynamic load balancing in large-scale graph processing. In *ACM EuroSys*, pages 169–182, 2013.

[13] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein. Distributed graphlab: A framework for machine learning and data mining in the cloud. *Proc. VLDB Endow.*, 5(8):716–727, Apr. 2012.

[14] S. Ma, Y. Cao, J. Huai, and T. Wo. Distributed graph pattern matching. In *Proceedings of the 21st International Conference on World Wide Web*, pages 949–958, 2012.

[15] 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 *ACM SIGCOM*, pages 135–146, 2010.

[16] A. McGregor. Graph stream algorithms: A survey. *SIGMOD Rec.*, 43(1):9–20, May 2014.

[17] A. Ntoulas, J. Cho, and C. Olston. What's new on the web?: The evolution of the web from a search engine perspective. In *Proceedings of the 13th International Conference on World Wide Web*, pages 1–12, 2004.

[18] A. Roy, I. Mihailovic, and W. Zwaenepoel. X-stream: Edge-centric graph processing using streaming partitions. In *Proceedings of ACM Symposium on Operating Systems Principles*, pages 472–488, 2013.

[19] S. Salihoglu and J. Widom. Gps: A graph processing system. In *SSDBM*.

[20] Y. Simmhan, A. Kumbhare, C. Wickramaarachchi, S. Nagarkar, S. Ravi, C. Raghavendra, and V. Prasanna. Goffish: A sub-graph centric framework for large-scale graph analytics. In *Euro-Par*, pages 451–462. 2014.

[21] I. Stanton and G. Kliot. Streaming graph partitioning for large distributed graphs. In *Proceedings of the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1222–1230, 2012.

[22] Y. Tian, A. Balmin, S. A. Corsten, S. Tatikonda, and J. McPherson. From think like a vertex to think like a graph. *Proceedings of the VLDB Endowment*, 7(3):193–204, 2013.

[23] L. Vaquero, F. Cuadrado, D. Logothetis, and C. Martella. Adaptive partitioning for large-scale dynamic graphs. In *Proceedings of Symposium on Cloud Computing*, pages 35:1–35:2, 2013.

[24] C. Wickramaarachchi, M. Frincu, and V. Prasanna. Enabling real-time pro-active analytics on streaming graphs. *algorithms*, 15:18, 2010.

[25] R. S. Xin, J. E. Gonzalez, M. J. Franklin, and I. Stoica. Graphx: A resilient distributed graph system on spark. In *First International Workshop on Graph Data Management Experiences and Systems*, pages 2:1–2:6, 2013.