symmetry

MDPI

Article

A Local Approximation Approach for Processing Time-Evolving Graphs

Shuo Ji † and Yinliang Zhao *,†

Department of Computer Science and Technology, Xi'an Jiaotong University, No 28, Xianning West Road, Xi'an 710049, China; jishuo@stu.xjtu.edu.cn

* Correspondence: zhaoy@xjtu.edu.cn; Tel.: +86-130-7297-2287

† These authors contributed equally to this work.

Received: 21 April 2018; Accepted: 12 June 2018; Published: 1 July 2018

Abstract: To efficiently process time-evolving graphs where new vertices and edges are inserted over time, an incremental computing model, which processes the newly-constructed graph based on the results of the computation on the outdated graph, is widely adopted in distributed time-evolving graph computing systems. In this paper, we first experimentally study how the results of the graph computation on the local graph structure can approximate the results of the graph computation on the complete graph structure in distributed environments. Then, we develop an optimization approach to reduce the response time in bulk synchronous parallel (BSP)-based incremental computing systems by processing time-evolving graphs on the local graph structure instead of on the complete graph structure. We have evaluated our optimization approach using the graph algorithms single-source shortest path (SSSP) and PageRankon the Amazon Elastic Compute Cloud(EC2), a central part of Amazon.com's cloud-computing platform, with different scales of graph datasets. The experimental results demonstrate that the local approximation approach can reduce the response time for the SSSP algorithm by 22% and reduce the response time for the PageRank algorithm by 7% on average compared to the existing incremental computing framework of GraphTau.

Keywords: distributed computing; time-evolving graph computing; incremental computing

1. Introduction

With the rapid increase in the scales of social networks, web graphs and other biological networks, the necessity of the distribution to process large-scale graphs intensifies [1–5]. What makes graph computing even more difficult is that graphs are time-evolving like the real-world systems where new users or web pages represented by the vertices and new interactions represented by the edges continuously arrive and evolve over time [6–9]. This poses a significant challenge to cope with these changes in graphs while preserving near real-time responses [9,10].

The majority of the distributed systems are developed to process a time-evolving graph by employing a sequence of static snapshots of the time-evolving graph. A snapshot is periodically constructed when a new graph stream arrives and comprises all the vertices and edges seen so far [11–14]. With the arrival of new vertices and edges, the results are updated by re-running the underlying algorithm on the newly-constructed snapshot. To accelerate the computation on the newly-constructed snapshot, an incremental computing model, which updates the vertices of the newly-constructed snapshot on the basis of the results of the computation on the previous snapshot, is widely adopted. Kineograph [11] is a distributed time-evolving graph computing system that takes a stream of incoming graph data to construct a sequence of consistent snapshots and supports the incremental computing model on successive snapshots. Specifically, when a new graph stream arrives, Kineograph proceeds with the computation on the newly-constructed snapshot t based on the results

of the computation on the previous snapshot $t-1$ until finishing the computation on the previous snapshot $t-1$. To eliminate the cost of waiting for the outcome of the computation on the previous snapshot when a new graph stream has arrived, GraphTau [12] has been developed as an improved incremental computing model that allows graph computation to shift from a previous snapshot $t-1$ to a new snapshot t even between successive iterations of the underlying algorithm on the previous snapshot $t-1$.

Pregel [1], a computing framework used to process each snapshot of a time-evolving graph in several distributed time-evolving graph computing systems [11,12], presents a vertex-centric iterative computing pattern based on the bulk synchronous parallel (BSP) programming paradigm [15]. Based on the BSP paradigm, an iteration in a graph algorithm is executed as a superstep on every participating computing node, and all the computing nodes are synchronized at the end of each superstep. In each superstep, a user-defined function is invoked on each vertex to compute the states of vertices based on the messages received from their in-degree neighbors. Then, the newly-updated values of these vertices are propagated directly to their out-degree neighbors by sending messages, each of which consists of a message value and the name of the destination vertex. This computing procedure is carried out iteratively until there is no status change for any vertex. The incremental computing model where the Pregel framework is exploited to process each snapshot shows a better performance in accelerating the time-evolving graph computing.

To further improve the performance of the incremental computing model when a tolerable loss of the quality of the results rather than knowing the final exact answers for a time-evolving graph is allowed, we present a local approximation approach, LocalAppro, to reduce the communication overhead in BSP-based time-evolving graph computing systems and thereby reduce the response time of processing time-evolving graphs in distributed environments. First of all, the local computing model and the global computing model are defined as follows:

Definition 1. *Local computing model: With a local computing model, graph computing is performed on the graph structure present in the same computing node in the distributed graph computing systems.*

Definition 2. *Global computing model: With a global computing model, graph computing is performed on the complete graph structure in the distributed graph computing systems.*

To the best of our knowledge, the existing distributed systems for processing time-evolving graphs are all based on the global computing model. LocalApproaims to reduce the response time of the time-evolving graph computing by allowing users to switch to a local computing model from a global computing model when a new graph stream arrives at the expense of the accuracy. To improve the effectiveness of the local approximation approach, LocalAppro predicts the messages from other computing nodes by using the previous messages that were received under the global computing model. Both the predicted messages and the local messages via reading from the shared memory are used to update the vertices of the newly-constructed graph snapshot iteratively, thereby saving the necessary communication overhead.

We have implemented our optimization approach by extending Giraph [16], an open-source implementation of the Pregel framework built on top of Apache Hadoop [17], to support the incremental computing model with the local approximation, and we have evaluated its effectiveness on several time-evolving graphs. The results demonstrate that LocalAppro can significantly reduce the amount of communications for the SSSP algorithm by 55% and the PageRank algorithm by 22% on average, which have accordingly resulted in a significant reduction of the response time for the SSSP algorithm by 22% and a slight reduction of the response time for the PageRank algorithm by 7% compared to the existing incremental computing framework of GraphTau [12].

This paper makes the following contributions:

- We developed a new optimization approach to reduce the response time in the distributed time-evolving graph systems by providing a novel local computing model instead of the global computing model.
- We designed a prediction method for the PageRank algorithm to guess the messages from remote computing nodes and thereby combined the predicted messages and the local messages to update the vertices of the newly-constructed snapshot. This ensures a smaller relative error between the results of the computation by using the local computing model and the results of the computation by using the global computing model.
- We were able to reduce the response time by 22% for the SSSP algorithm and 7% for the PageRank algorithm on average compared to the incremental computing framework of GraphTau [12].

The remainder of the paper is organized as follows. Section 2 introduces the incremental computing model in more detail and shows the potential advantages of the local approximation. Section 3 presents the system design of our proposed optimization approach for the distributed time-evolving graph computing. A more detailed description of the local approximation approach is presented in Section 4. Section 5 provides a performance evaluation of this work. Section 6 describes the related works, and Section 7 concludes this work.

2. Background

In this section, we first describe the incremental computing model built on Giraph [16], an open-source implementation of the Pregel framework. Then, we study the potential advantages of executing graph algorithms on the local structure of a time-evolving graph.

2.1. Incremental Computing Model

Instead of submitting a new task to re-execute the underlying algorithm on the newly-constructed snapshot, the incremental computing model is developed to process time-evolving graphs by proceeding with the computation on the newly-constructed snapshot on the basis of the results of the computation on the previous snapshot. This can guarantee the correctness of the computation on account of the observation that small changes to the previous snapshot often require only small updates to the results. The following aspects need to be taken into consideration when implementing the incremental computing model. Firstly, the results of the computation on the previous snapshot should be consistent. That is, the results should be the final results of the computation on the previous snapshot like Kineograph [11] or should be the intermediate results saved after the same superstep for each vertex like GraphTau [12]. Secondly, the values of all the vertices of the previous snapshot should be broadcast to their out-degree neighbors in case the newly-added vertices could not receive the complete messages from the vertices of the previous snapshot to update their values.

Due to the synchronization phase of each superstep, which ensures the consistent results of each iteration on the previous snapshot in BSP-based graph systems, the GraphTau [12] framework is implemented by extending Giraph [1], a graph processing system appearing as an open-source implementation of Pregel, to load graph streams in a timely manner and support the incremental computing model on time-evolving graphs. Figure 1 below illustrates the framework of this incremental computing system for time-evolving graphs. It follows the master-worker pattern and supports the BSP-based programming paradigm to better facilitate parallel processingfor various graph algorithms in distributed environments, such as Giraph. It extends Giraph to support the incremental computing model by implementing a specific class of *DynamicMasterCompute*, which inherits the abstract class of *MasterCompute*. The abstract class of *MasterCompute* is provided by Giraph for users to specify the responsibility of the master. The *DynamicMasterCompute* class implements the function of *compute*, which will be invoked to have the master observing the changes on the input graph after each superstep. If new vertices and edges arrive, the computing nodes will be informed at the beginning of the next superstep to load the new graph stream to construct a new snapshot and to broadcast the current results of the computation on the previous snapshot. Each

participating computing node subsequently proceeds with executing the underlying algorithm on the newly-constructed snapshot on the basis of the current results of the previous snapshot.

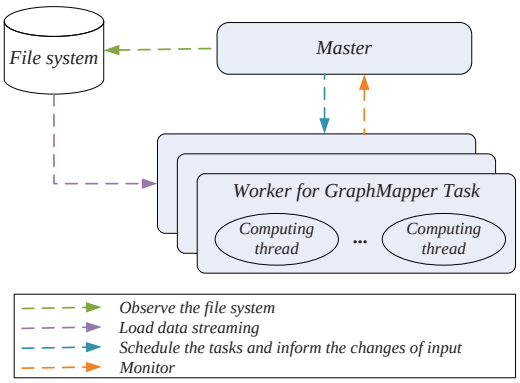

Figure 1. The framework of the incremental computing system for time-evolving graphs.

2.2. Potential Advantages of Local Approximation

In this subsection, we conduct experiments to study the possibility that the results of executing the time-evolving graph computing with a local model could effectively approximate the results of executing the time-evolving graph computing with a global model in distributed time-evolving graph computing systems.

We first take the SSSP algorithm on a sequence of snapshots extracted from YouTube [18] as an example. The first snapshot during the first six months in the dataset is taken as the input, and the subsequent snapshot during the next month in the dataset is loaded at a pre-specified superstep of the computation on the first snapshot. We first run the algorithm with the global computing model and then run with the local computing model. We plot Figure 2 below by counting the number of vertices still to be updated in each superstep following the specific superstep at which the new graph stream has just been inserted. Figure 2a shows that it takes three iterations such that the remaining 35,000 vertices reach the final results with the global computing model; whereas Figure 2b shows that nearly only 100 vertices, a very small percentage of the remaining vertices, have not reached the final results after Superstep 13, which follows a local computing model triggered in Superstep 12.

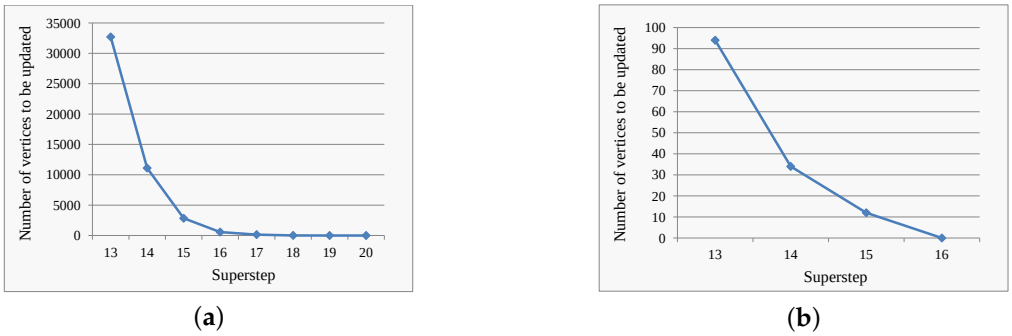

Figure 2. The number of vertices to be updated in single-source shortest path (SSSP). (**a**) With the global computing model. (**b**) With the local computing model.

We also take the PageRank algorithm on a sequence of snapshots extracted from Wikipedia [18] as an example. The first snapshot during the first five years in the dataset is taken as the input, and the subsequent snapshot during the the next two weeks in the dataset is loaded at a pre-specified superstep of the computation on the first snapshot. As the example of the SSSP algorithm above, we first run the algorithm with the global computing model and then run with the local computing model. We then

plot Figure 3 below by computing the relative error between the results with the local computing model and that with the global computing model. Figure 3 illustrates that along with the number of iterations increasing when the new graph stream is inserted, the results with the local computing model get closer to the results with the global computing model. Besides, more iterations executed on local graph structure will lead to a larger relative error between the results with the local computing model and the results with the global computing model as shown in Figure 3.

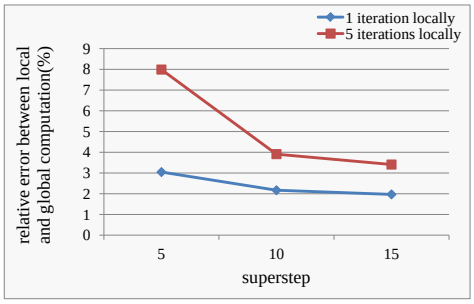

Figure 3. The relative error between the results of executing PageRank on the new snapshot with the local computing model and the global computing model.

From the analysis above, we conclude that when a new graph stream arrives at a pre-specific superstep, the results of processing a time-evolving graph with the local computing model can effectively approximate that of the global computing model in distributed time-evolving graph computing systems. This provides a simple way to reduce the communication overhead between computing nodes when relaxing the accuracy of the obtained results.

3. System Design

In this section, we describe the system design of our proposed optimization approach and present the message passing model adopted in this optimization approach, respectively. First of all, the local neighbors and remote neighbors are defined respectively as follows. The notations are listed in Table 1 below.

Definition 3. *Local neighbors: Local neighbors of a vertex v are defined as a set of vertices that link to the vertex v and are located in the same computing node as the vertex v.*

Definition 4. *Remote neighbors: Remote neighbors of a vertex v are defined as a set of vertices that link to the vertex v and are located in different computing nodes from where the vertex v is located.*

Table 1. Notations.

Notation	Description
$G = (V, E)$	Represents a graph where V is the set of the vertices and E is the set of the edges
$G_l = (V_l, E_l)$	Represents a subgraph present in the same computing node where a specified vertex v is located, where V_l is the set of the vertices and E_l is the set of the edges in this subgraph.
$G_r = (V_r, E_r)$	Represents a subgraph present in the different computing node where a specified vertex v is located, where V_r is the set of the vertices and E_r is the set of the edges in this subgraph.

3.1. LocalAppro System

LocalAppro is developed for time-evolving graph computing in distributed environments. Figure 4 depicts the high-level design of LocalAppro implemented by extending the incremental computing system (illustrated in Figure 1) with two modules, a local computing module to execute the

underlying algorithm on the graph structure present in each local computing node and a predictor module to predict the potential messages from remote computing nodes (the potential messages would not be sent by the remote neighbors when executing computation with the local computing model) for each vertex under the local computing model. Specifically, when a new graph stream arrives during a user-specific period of the computation on the previous snapshot, the local computing model is triggered to approximate the accurate final results. The local computing model proceeds to update the values of vertices with the constantly updated local messages via reading from shared memory and the predicted messages, which would been sent from the remote neighbors when executing graph computing with the global computing model.

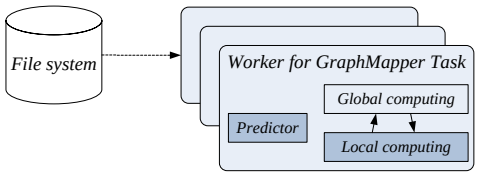

Figure 4. The added components in LocalAppro.

The workflow of LocalAppro on the master is the same as that of the incremental computing system [12] built on Giraph. Specifically, after initializing the job that users have submitted, the input graph data as the first snapshot are split for each computing node to load, and then, the mapper tasks are created for each computing node to execute (Giraph is built on Hadoop and executes each superstep as a mapper task). Then, the master periodically observes the file system to check whether a new graph stream arrives. With the arrival of a new stream, the master notifies each computing node to load splits for the new graph stream to construct a new snapshot accordingly. This procedure on the master is terminated until the job has completed.

The workflow of LocalAppro on computing nodes works as follows: after initialization, the input splits are loaded by each participating computing node accordingly, then a user-defined function for updating the state of each vertex is iteratively executed. At the very beginning of each superstep, each participating computing node examines whether a new stream has arrived. If there exists new splits for the computing nodes, each computing node will load the new stream with hash partitioning provided by Giraph and switch to a local computing model instead of a global computing model to approximate the final results. This procedure on the computing nodes is terminated until no status changes on the vertices or there are no more graph streams for the input graph data.

3.2. Message Passing Model

The push model is widely adopted in the BSP-based distributed graph computing platforms. In the push model, a message flows from a vertex v executing the user-defined function out to the neighbors of the vertex v. Particularly with the Pregel framework, a user-defined function is firstly invoked to update the value of each vertex by processing the received messages, then the newly-updated value is broadcast to the out-degree neighbors of the vertex through message passing. Different from the push model, the message flows from the neighbors of the vertex v into the vertex v in the pull mode. In the pull model, a vertex v needs to read the values of its in-degree neighbors from shared memory before invoking the user-defined function to update its value. LocalAppro exploits the push model in the global computing model as Giraph and exploits the pull model in the local computing model where the iteratively updated values of the local neighbors of a vertex v are required to update the value of the vertex v.

4. Local Approximation

In this section, we first illustrate the local computing model and next present two typical graph algorithms, SSSP and PageRank, optimized with this proposed model.

4.1. Local Computing Model

The global computing model is depicted in Figure 5a below, with which message passing could be initiated between both intra- and inter-computing nodes. Typically in the Giraph platform, a vertex v is updated by processing the messages that are received in superstep $t-1$ from its neighbors both located in the same computing node as the vertex v and in other remote computing nodes. Then, the newly-updated value of the vertex is broadcast to its neighbors both located in the same computing nodes and in other remote computing nodes in the current superstep t. Different from the global computing model in distributed environments, message passing could only be initiated in the same computing node with the local computing model. As depicted in Figure 5b below, with the local computing model, a vertex v is updated by processing the iteratively updated values of its local neighbors and the predicted values, which approximate the values of its remote neighbors in the current superstep t. The iteratively updated values of its local neighbors can be read from shared memory, and the predicted values are guessed based on the previous values of its remote neighbors, which had been sent under the global computing model. Specifically, before a new stream arrives, graph computing is executed with the global computing model. When a new stream arrives and graph computing is switched to the local computing model, the approximations will be made actually based on the messages passed under the global computing model. That is, the approximations will be made based on the messages passed before the new stream arrives. Then, under the local computing model, we use the old messages (passed before the new stream arrives) to predict (approximate) the messages that would be transferred when the new snapshot is constructed and executed under the global computing model. For the next superstep $t+1$, the local neighbors of the vertex v will pull the value of the vertex v, whereas the remote neighbors of the vertex v will predict the value of the vertex v based on the old value of the vertex v that was sent under the global computing model.

Whether the local computing model, as described above, should be triggered depends on the time-point of a new stream arriving and the scale of a new stream. Considering that each vertex tends to reach a stable state along with iteratively executing the underlying algorithms in a consistent snapshot and a tiny change of the graph structure causes a smaller change of the results, the local computing model can be switched to when a new stream arrive at a later period of the computation on the old snapshot and when the scale of a new graph stream is smaller compared to the input graph data to ensure a reasonable and low error of the whole application.

Figure 5. The computing models. The solid circles with the same color represent that these vertices are located in the same computing node. The solid lines represent that the vertices are connected and communicate with each other. The dotted lines represent that the vertices are connected and do not communicate with each other. (**a**) The global computing model. (**b**) The local computing model.

4.2. Algorithms with the Local Computing Model

4.2.1. SSSP Algorithm

The SSSP algorithm aims to find the shortest paths from a single vertex to all the other vertices in a graph. In distributed environments, the shortest value of a vertex v is iteratively updated by taking

the minimum value from the incoming messages, which represent the distances from the designated source vertex to v through different paths. The updated function can be described as follows:

$$d\left(v^{t+1}\right) = min_{u \in S}\left(d\left(u^{t}\right) + w\left(u,v\right)\right) \quad S = \{u | u \in V, (u,v) \in E\} \tag{1}$$

where $d\left(v^{t+1}\right)$ represents the shortest value of the vertex v at iteration $t+1$, $d\left(u^{t}\right)$ is the shortest value of the vertex u at iteration t and $w\left(u,v\right)$ is the weight of the edge between the vertex u and vertex v. u is a vertex that links to the vertex v in the graph. We can further formalize the function considering whether the neighbors of the vertex v are located in the same computing node as v as follows:

$$\begin{gathered} d\left(v^{t+1}\right) = min\{min_{l \in S_l}\left(d\left(l^{t}\right) + w\left(l,v\right)\right), \\ min_{r \in S_r}\left(d\left(r^{t}\right) + w\left(r,v\right)\right)\} \\ S_l = \{l | l \in V_l, (l,v) \in E_l\} \quad S_r = \{r | r \in V_r, (r,v) \in E_r\} \end{gathered} \tag{2}$$

where l is a vertex in the subgraph of G_l and r is a vertex in the subgraph of G_r. G_l and G_r for the vertex v are defined in Table 1. From the formula above, to reduce the communication overhead, which degrades the response time, the formula for SSSP algorithms to update a vertex with local neighbors can be described as:

$$d\left(v^{t+1}\right) = min_{l \in S_l}\left(d\left(l^{t}\right) + w\left(l,v\right)\right) \quad S_l = \{l | l \in V_l, (l,v) \in E_l\} \tag{3}$$

Specifically, Algorithm 1 below illustrates how to approximate the final exact results with the local computing model for the SSSP algorithms. To approximate the results when a new graph stream has arrived, the *SSSPComputation* function is invoked to get the local neighbors of the vertex v firstly (as Line 2) and then traverse the values of the local neighbors of the vertex v (as Lines 3–5) to take a minimum value to update the shortest value of the vertex v (as Lines 6–8).

Algorithm 1 SSSP algorithm with local computing model
Input: Vertex v
Output: *minDist*

```
SSSPComputation (Vertex v) {
1.  minDist ← initialization(v);
2.  indegreeVertexList ← getInEdges(v);
3.  for each vertex u ∈ indegreeVertexList
4.      minDist ← findMinimum(minDist,getValue(u)+getWeight(u,v));
5.  end for
6.  if minDist < getValue(v)
7.      setValue(v,minDist);
8.  end if
9.  return minDist; }
```

4.2.2. PageRank Algorithm

The PageRank algorithm aims to rank the importance of a web page v based on the number of links to it from other pages, by iteratively having v receiving messages about the ranks of other web pages that link to v to compute a new rank for v. The updated function can be described as follows:

$$PR\left(v^{t+1}\right) = d * \sum_{u \in M(v)} \frac{PR\left(u^{t}\right)}{D\left(u\right)} + \frac{1-d}{N} \tag{4}$$

where $PR(v^{t+1})$ represents the PageRank value of the vertex v updated at the superstep $t+1$, d is a damping factor, $M(v)$ is the set of vertices that link to the vertex v, $PR(u^{t})$ is the PageRank value of

the vertex u at superstep t, $D(u)$ is the number of outbound links on the vertex u and N is the total number of vertices. For a vertex-centric distributed graph system, the PageRank value for the vertex v is updated by processing the messages with the above formula. The messages the vertex v received can be divided into two parts, local messages from the in-degree neighbors of the vertex v located in the same computing node as the vertex v and the remote messages from the in-degree neighbors of the vertex v located in the remote computing nodes. We can further formalize the function considering where the neighbors of vertex v are located as follows:

$$PR\left(v^{t+1}\right) = d * \sum_{l \in M_l(v)} \frac{PR\left(l^t\right)}{D\left(l\right)} + d * \sum_{r \in M_r(v)} \frac{PR\left(r^t\right)}{D\left(r\right)} + \frac{1-d}{N} \tag{5}$$

where l is a vertex in the subgraph of G_l and r is a vertex in the subgraph of G_r. G_l and G_r for the vertex v are defined in Table 1. $M_l(v)$ and $M_r(v)$ respectively represent the in-degree neighbors of the vertex v in the G_l and G_r. Therefore, to reduce the communication overhead, which degrades the response time, the formula for the PageRank algorithm to update a vertex with local neighbors can be described as follows:

$$PR\left(v^{t+1}\right) = d * \sum_{l \in M_l(v)} \frac{PR\left(l^t\right)}{D\left(l\right)} + d * sum' + \frac{1-d}{N} \tag{6}$$

where sum' is an approximate value of the sum of messages from the in-degree neighbors of the vertex v located in the remote computing nodes. To ensure that the final results updated with the local computing model for PageRank are acceptable, we predict the sum of messages from the remote neighbors of the vertex v by using the following rules:

1. For the vertices that have been in the old snapshot, based on Formula (5) above, the sum of the messages from the remote neighbors of the vertex v multiplied by d at superstep t can be described as Formula (7). We make the assumption that the messages from the remote neighbors of the vertex v gradually increase with the pattern that the increasing rate gradually decreases. Thereby, we predict the sum of the remote messages by using Formula (8) with the local messages. We set α to be 0.05 in our implementation.

$$d * \sum_{r \in M_r(v)} \frac{PR\left(r^t\right)}{D\left(r\right)} = PR\left(v^{t+1}\right) - d * \sum_{l \in M_l(v)} \frac{PR\left(l^t\right)}{D\left(l\right)} - \frac{1-d}{N} \tag{7}$$

$$sum' = (1 + \alpha^t) * \sum_{r \in M_r(v)} \frac{PR\left(r^t\right)}{D\left(r\right)} \quad \alpha < 1 \tag{8}$$

2. For the newly-inserted vertices, we make the assumption that the neighbors of the new vertices are uniformly distributed in each computing node; thereby, we predict the sum of the remote messages as the sum of the local messages.

Specifically, the PageRank algorithm with the local computing model is presented as Algorithm 2 below. The *getInEdges*() function is invoked to get the local neighbors of the vertex v. The iteratively-updated values of the local neighbors of the vertex v are then read from the shared memory and processed as Lines 3–6. The *predict*() function is invoked to predict the sum of the messages from the remote neighbors of vertex v (Line 8). The value of the vertex v is locally updated based on the sum of the iteratively-updated values of the local neighbors and the predicted sum of the messages from the remote neighbors of the vertex v (as Lines 9–10).

Algorithm 2 PageRank algorithm with local computing model
Input: Vertex v
Output: $value$

PageRankComputation (Vertex v) {
1. $sum \leftarrow 0$;
2. $indegreeVertexList \leftarrow$ getInEdges(v);
3. for each vertex $u \in indegreeVertexList$
4. $edges \leftarrow$ getNumEdges(u);
5. $sum \leftarrow sum$+getValue(u)/$edges$;
6. end for
7. $localValue \leftarrow d$*sum;
8. $remoteValue \leftarrow d$*predict(v);
9. $value \leftarrow localValue$+$remoteValue$+(1-$d$)/$N$
10. setValue(v,$value$);
11. return $value$;}

5. Evaluation

We have implemented our proposed local approximation approach by extending the open-source platform Giraph [16]. No change is made to the underlying communication and synchronization mechanisms in Giraph. We implemented the graph algorithms single-source shortest path (SSSP) and PageRank, both using the global computing model and the local computing model, to study its effectiveness, particularly its implications on the communications behavior, the response time performance and the accuracy of the final results of the algorithms.

5.1. Experimental Setup

The classical graph algorithms, SSSP and PageRank, are implemented to study the performance. We conducted all experimental studies on the Amazon EC2 Cloud with five small instances; each small instance is with one CPU and 2 G memory. Each node of the cluster is configured with Ubuntu Server 14.04 (64 bit), Apache Hadoop 0.20.203 and Java 1.7. Each graph algorithm is evaluated based on the global computing model and the local computing model with the cluster configurations.

Typical graph streams, YouTube and Wikipedia, listed in the following Table 2, from [18] are used to study the SSSP and PageRank algorithms, respectively. Specifically, the graph stream $s1$ in YouTube consists of the data stream during the first six months in the dataset; the graph stream $s2$ consists of the data stream during the next two weeks following the first snapshot; and the graph stream $s3$ consists of the data stream during the next month following the first snapshot. The graph stream $s1$ in Wikipedia-small consists of the data stream during the first five years in the dataset; the graph stream $s2$ consists of the data stream during the next two weeks following the first snapshot; and the graph stream $s3$ consists of the data stream during the next month following the first snapshot. The Wikipedia data include all the graph stream provided by [18] and are divided into different sizes of graph streams, as shown in Table 2.

Table 2. Graph data in our experiments.

Dataset	Stream	$\|V\|$	$\|E\|$	Size (MB)	Description
YouTube	s1	2,438,091	6,294,386	106	The social network of YouTube users and their friendship connections.
	s2	361,085	840,673	17	
	s3	714,061	1,888,816	36	
Wikipedia-small	s1	1,092,282	16,810,630	181	The hyperlink network of the English Wikipedia with edge arrival times.
	s2	338,588	1,259,596	19	
	s3	529,574	2,565,247	35	
Wikipedia	s1	1,631,890	37,675,911	545	The hyperlink network of the English Wikipedia with edge arrival times.
	s2	367,747	1,233,340	22	
	s3	336,098	2,261,332	39	

5.2. Performance Study

In this subsection, we analyze the performance in the following aspects. First, we compare the communication overhead and the response time of using our optimization with that of using the global computing model in the incremental computing as described in GraphTau [12]. Then, we analyze the overall relative error between the results of using the local computing model and that of using the global computing model in the incremental computing for time-evolving graphs.

5.2.1. Communication Overhead and Response Time

We first validate that the local approximation can efficiently improve the overall performance in time-evolving graph computing compared with using the global computing model in the incremental computing. We read the graph stream $s1$ as the first snapshot and the corresponding new graph streams $s2$ and $s3$ to construct new snapshots during a pre-defined period of the computation on the first snapshot. The response time is counted from loading the first snapshot to finishing the computation on the newly-updated snapshot. We first perform the incremental computing with the original global computing model. Then, we perform the local approximation approach with which the computing model is switched to the local computing model from the global computing model when a new graph stream arrives.

Figure 6 compares the amount of communications of these algorithms implemented using incremental computing with the global computing model and the local computing model, respectively. The bars where their horizontal axes are labeled as $s2$ and $s3$ present the amount of communications transferred from starting the computation on the first snapshot $s1$ of the input data to finishing the computation on the newly-constructed snapshots with $s2$ and $s3$, respectively. Figure 7 shows the respective response time of these algorithms. The bars where their horizontal axes are labeled as $s2$ and $s3$ present the response time from loading the first snapshot $s1$ of the input data to finishing the computation on the newly-constructed snapshots with $s2$ and $s3$, respectively. Here, the local approximation approach has significantly reduced the amount of communications for the SSSP algorithm by 55% and for the PageRank algorithm by 22% on average. Benefiting from the reduction of the communication overhead, the local approximation approach has accordingly reduced the response time for the SSSP algorithm by 22% and slightly reduced the response time for the PageRank algorithm by 7% on average, as described in Figure 7.

Figure 8 presents the execution time (normalized to the execution time of the implementation under the global computing model) when using the local computing model to conduct the SSSP algorithm and the PageRank algorithm on varying sizes of the Wikipedia graph [18]. We artificially set the sizes of the graph to be 20%, 40%, 60%, 80% and 100% of the Wikipedia graph. Figure 8 addresses that the runtime performances of the implementations on different scales of the input graphs benefits from using the local computing model.

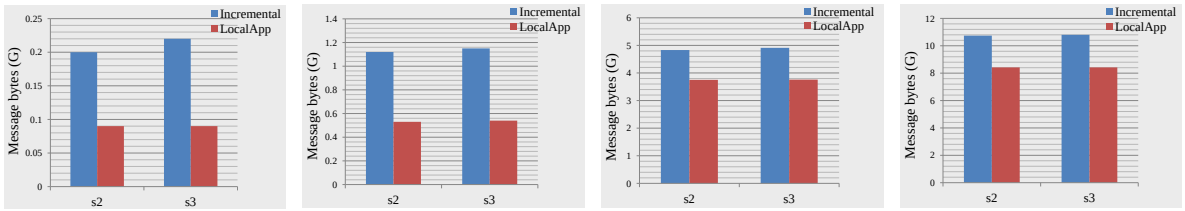

Figure 6. Comparison of the communication overhead between the global computing model and the local computing model. (**a**) SSSP on YouTube. (**b**) SSSP on Wikipedia. (**c**) PageRank on Wikipedia-small. (**d**) PageRank on Wikipedia.

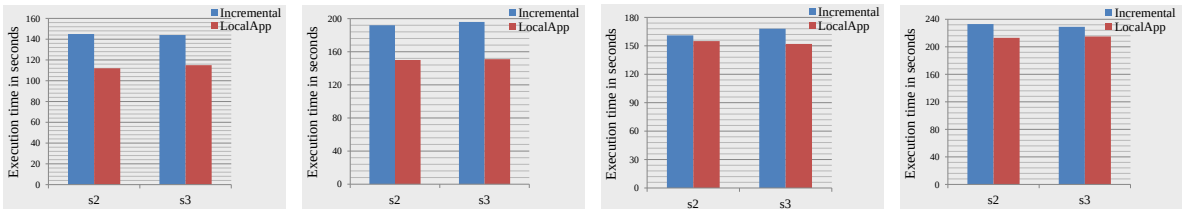

Figure 7. Comparison of the response time between the global computing model and the local computing model. (**a**) SSSP on YouTube. (**b**) SSSP on Wikipedia. (**c**) PageRank on Wikipedia-small. (**d**) PageRank on Wikipedia.

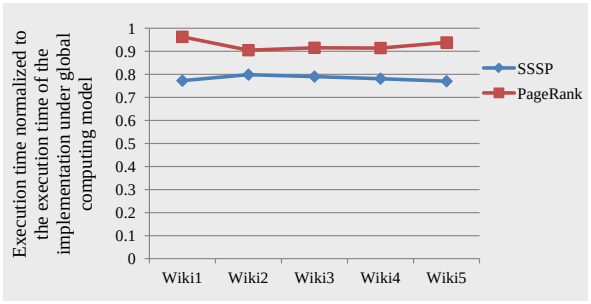

Figure 8. The performance of the execution time of executing the graph algorithms under the local computing model as the size of the graph Wikipedia grows.

5.2.2. Local Approximation Error Analysis

To determine how the local computing model affects the overall accuracy of these graph algorithms, we compute an overall error rate for each algorithm (SSSP and PageRank). For the SSSP algorithm, we calculate the percentage of the number of vertices that do not obtain the shortest distances accurately compared to the results with the global computing model. For the PageRank algorithm, we compute an overall error rate as the absolute sum of the differences in the final results (with the local computing model) normalized against the absolute sum of the original final results (with the global computing model). The final results of the algorithms are represented by the collection of eventual values computed for each vertex of the input graphs. Thereby, an overall error rate is defined as the following formula:

$$R\left(u,v\right) = \frac{\sum_{i=1}^{n} |u_i - v_i|}{\sum_{i=1}^{n} |v_i|} \tag{9}$$

where u and v are the vectors representing the eventual values computed for all vertices of the input graph, by using the local computing model and the global computing model, respectively.

We then study the approximation error rate to validate the effectiveness of our local approximation approach by comparing the relative error between the results with using the local computing model and those with using the global computing model in the incremental computing. We simulate a series of experiments on different scales of data streams inserted. Table 3 shows the resulting relative error rates, which almost range between 2%–9%, from applying our local approximation optimization. However, the relative error of the case of executing the SSSP algorithms on a snapshot of YouTube constructed with the graph stream s3 is 16.59%. This is due to the scale of the graph stream s3 being nearly 34% to the scale of the input data s1 of YouTube. Table 3 addresses that the local approximation approach is able to lead to an acceptable relative error when the scale of a new graph stream is smaller compared to the input graph data. This also addresses that a tiny change of the graph structure causes a smaller change of the results on which the incremental computing is based.

Table 3. The resulting relative error rates.

Algorithm	Dataset	Snapshot	Relative Error (%)
SSSP	YouTube	s2	8.67
		s3	16.59
	Wikipedia	s2	2.39
		s3	4.17
PageRank	Wikipedia-small	s2	5.61
		s3	8.06
	Wikipedia	s2	5.26
		s3	6.14

Figure 9 illustrates the relative error between the results with using the local approximation and the final results with using the global computing model in the incremental computing for the SSSP algorithm and the PageRank algorithm when the newly-incoming graph streams are inserted at different supersteps, respectively. Figure 9 presents that the relative errors between the results with the local computing model and the results with the global computing model tend to become smaller along with the new data being inserted at a later superstep for both the graph algorithms. This is because each vertex tends to reach a stable state along with iteratively executing the underlying algorithms on a consistent snapshot.

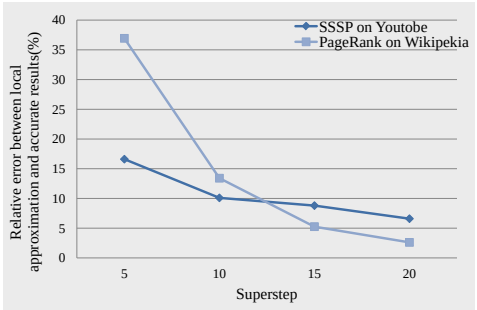

Figure 9. Relative errors between the results with the local computing model and the final results with the global computing model.

6. Related Work

6.1. Distributed Time-Evolving Graph Computing Systems

Many distributed systems have been recently proposed to handle time-evolving graphs efficiently by exploiting incremental computing. Kineograph [11] is a distributed system that takes a stream of

incoming data to construct a series of consistent snapshots. Even Kineograph supports an incremental graph computation, but newly-arrived updates must wait for the completion of the current snapshot rather than being processed immediately. GraphInc [13] aims to reduce the amount of computation by automatically identifying opportunities for computation reuse to save the computations and re-executing only those that are necessary when the graph changes. GraphTau [12] is a distributed computing system for time-evolving graphs that enables efficient computations by allowing graph computations to "shift" from a previous snapshot to a new snapshot even between iterations of executing the underlying algorithm on the previous snapshot. Different from the time-evolving graph systems above that load new graph streams when the new data accumulatively reach a pre-defined size, Das [19] explores the effect of the size of batches on the performance of time-evolving graph processing. They address that the response time of the systems can have complicated relationships with batch sizes, data ingestion rates, variations in available resources and workload characteristics and then propose a simple yet robust control algorithm that automatically adapts batch sizes as the situation necessitates optimizing the performance of time-evolving graph systems.

6.2. Asynchronous Graph Systems

In the asynchronous iteration model, no explicit synchronization points, i.e., barriers, are provided, so any vertex is eligible for computation whenever processor and network resources are available [2]. GraphLab [20] presents an asynchronous graph system by providing a set of data consistency models that enable the user to specify the minimal consistency requirements of his/her application without having to build his/her own complex locking protocols. Maiter [21] presented a delta-based accumulative iterative computation framework that selectively processes a subset of the vertices, which has the potential of accelerating the iterative computation. This paper similarly relied on local computing to approximate the final results, but without the need to change the underlying communication framework. Asynchronous processing on the other hand changes the underlying communication framework.

7. Conclusions and Future Work

This paper presents a local approximation approach, LocalAppro, for distributed time-evolving graph computing systems to reduce the response time. It provides a local computing model in the incremental computing when a new graph stream arrives to approximate the final outcomes at the expense of the accuracy. Different from the global computing model in the incremental computing with which a vertex v is updated by taking in the messages from its neighbors both located in the same computing node as the vertex v and the other computing nodes, with the local computing model, a vertex v is updated by taking in the message from its neighbors located in the same computing nodes as the vertex v and the messages that are predicted on the basis of the previous messages from the neighbors located in the other computing nodes. This local approximation approach has effectively reduced the response time of the SSSP algorithm by 22% and slightly reduced the response time of the PageRank algorithm by 7% on average.

However, the local computing model proposed in this paper will lead to a certain degree of error due to the approximation of the messages in the other remote computing nodes. In the future work, we will study the predictability of the error, thereby providing a strategy of switching between the global computing model and the local computing model. On the other hand, we will further explore a more reasonable approach to approximate the required remote messages, which would be sent by the remote computing nodes when using the global computing model, to improve the accuracy with the local approximation approach.

Author Contributions: All authors contributed equally to this paper. The individual contributions of each author are described as follows: S.J. conceived of and designed the experiments and also completed the manuscript preparation; Y.Z. provided constructive guidance and revised the manuscript.

Funding: This research was funded by the National Natural Science Foundation of China through grant number 61640219. Check carefully that the details given are accurate and use the standard spelling of funding agency names at https://search.crossref.org/funding, any errors may affect your future funding.

Acknowledgments: We thank our colleagues for their collaboration and the present work. We also thank all the reviewers for their specific comments and suggestions. This work is supported by the National Natural Science Foundation of China through grant number 61640219.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Malewicz, G.; Austern, M.H.; Bik, A.J.; Dehnert, J.C.; Horn, I.; Leiser, N.; Czajkowski, G. Pregel: A system for large-scale graph processing. In Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data (SIGMOD'10), Indianapolis, IN, USA, 6–10 June 2010; pp. 135–146.
2. McCune, R.R.; Weninger, T.; Madey, G. Thinking like a vertex: A survey of vertex-centric frameworks for large-scale distributed graph processing. *ACM Comput. Surv.* **2015**, *48*, 25. [CrossRef]
3. Gonzalez, J.E.; Low, Y.; Gu, H.; Bickson, D.; Guestrin, C. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs. In Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI'12), Hollywood, CA, USA, 8–10 October 2012; pp. 17–30.
4. Gonzalez, J.E.; Xin, R.S.; Dave, A.; Crankshaw, D.; Franklin, M.J.; Stoica, I. GraphX: Graph processing in a distributed dataflow framework. In Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation, Broomfield, CO, USA, 6–8 October 2014; pp. 599–613.
5. Kang, U.; Tsourakakis, C.E.; Faloutsos, C. PEGASUS: A Peta-scale graph mining system implementation and observations. In Proceedings of the 2009 Ninth IEEE International Conference on Data Mining, Miami, FL, USA, 6–9 December 2009; pp. 229–238.
6. Leskovec, J.; Kleinberg, J.M.; Faloutsos, C. Graphs over time: Densification laws, shrinking diameters and possible explanations. In Proceedings of the eleventh ACM SIGKDD International Conference on Knowledge Discovery in Data Mining (KDD'05), Chicago, IL, USA, 21–24 August 2005; pp. 177–187.
7. Gaito, S.; Zignani, M.; Rossi, G.P.; Sala, A.; Zhao, X.; Zheng, H.; Zhao, B.Y. On the bursty evolution of online social networks. In Proceedings of the First ACM International Workshop on Hot Topics on Interdisciplinary Social Networks Research (HotSocial'12), Beijing, China, 12–16 August 2012; pp. 1–8.
8. Vaquero, L.M.; Cuadrado, F.; Ripeanu, M. Systems for near real-time analysis of large-scale dynamic graphs. *arXiv* **2014**, arxiv: 1410.1903.
9. Murray, D.G.; Mcsherry, F.; Isaacs, R.; Isard, M.; Barham, P.; Abadi, M. Naiad: A timely dataflow system. In Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP'13), Farminton, PA, USA, 3–6 November 2013; pp. 439–455.
10. Morshed, S.J.; Rana, J.; Milrad, M. Real-time Data analytics: An algorithmic perspective. In Proceedings of the IEEE International Conference on Data Mining, Barcelona, Spain, 12–15 December 2016; pp. 311–320.
11. 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 Proceedings of the 7th ACM European Conference on Computer Systems (EuroSys'12), Bern, Switzerland, 10–13 April 2012; pp. 85–98.
12. Iyer, A.P.; Li, L.E.; Das, T.; Stoica, I. Time-evolving graph processing at scale. In Proceedings of the 4th International Workshop on Graph Data Management Experience and Systems (GRADES'16), Redwood Shores, CA, USA, 24–24 June 2016.
13. Cai, Z.; Logothetis, D.; Siganos, G. Facilitating real-time graph mining. In Proceedings of the Fourth International Workshop on Cloud Data Management (CloudDB'12), Maui, HI, USA, 29 October 2012.
14. Shi, X.; Cui, B.; Shao, Y.; Tong, Y. Tornado: A system for real-time iterative analysis over evolving data. In Proceedings of the 2016 International Conference on Management of Data (SIGMOD'16), San Francisco, CA, USA, 26 June–1 July 2016; pp. 417–430.
15. Valiant, L.G. A bridging model for parallel computation. *Commun. ACM* **1990**, *33*, 103–111. [CrossRef]
16. Apache. Apache Giraph. 2012. Available online: http://giraph.apache.org/ (accessed on 7 January 2015).
17. Dean, J.; Ghemawat, S. MapReduce: simplified data processing on large clusters. *Commun. ACM* **2008**, *51*, 107–113. [CrossRef]

18. Konect. Konect Network Dataset. 2017. Available online: http://konect.uni-koblenz.de/ (accessed on 21 May 2017).
19. Das, T.; Zhong, Y.; Stoica, I.; Shenker, S. Adaptive stream processing using dynamic batch sizing. In Proceedings of the ACM Symposium on Cloud Computing (SOCC'14), Seattle, WA, USA, 3–5 November 2014; pp. 1–13.
20. Low, Y.; Gonzalez, J.E.; Kyrola, A.; Bickson, D.; Guestrin, C.E.; Hellerstein, J. Graphlab: a new framework for parallel machine learning. *arXiv* **2014**, arxiv:1408.2041 .
21. Zhang, Y.; Gao, Q.; Gao, L.; Wang, C. Maiter: an asynchronous graph processing framework for delta-based accumulative iterative computation. *IEEE Trans. Parallel Distrib. Syst.* **2014**, *25*, 2091–2100. [CrossRef]