CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. (2014)
Published online in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/cpe.3393

SPECIAL ISSUE PAPER

An efficient graph data processing system for large-scale social network service applications‡

Wei Zhou[1,2], Jizhong Han[1,*,†], Yun Gao[1,2] and Zhiyong Xu[3]

[1]*Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China*
[2]*University of the Chinese Academy of Science, Beijing,100049, China*
[3]*Department of Math and Computer Science Suffolk University, Boston, MA 02114*

SUMMARY

Trust in social network draws more and more attentions from both the academia and industry fields. Public opinion analysis is a direct way to increase the trust in social network. Because the public opinion analysis can be expressed naturally by the graph algorithm and graph data are the default data organization mechanism used in large-scale social network service applications, more and more research works apply the graph processing system to deal with the public opinion analysis. As the data volume is growing rapidly, the distributed graph systems are introduced to process the large-scale public opinion analysis. Most of graph algorithms introduce a large number of data iterations, so the synchronization requirements between successive iterations can severely jeopardize the effectiveness of parallel operations, which makes the data aggregation and analysis operations become slower. In this paper, we propose a large-scale graph data processing system to address these issues, which includes a graph data processing model, *Arbor*. *Arbor* develops a new graph data organization format to represent the social relationship, and the format can not only save storage space but also accelerate graph data processing operations. Furthermore, *Arbor* substitutes time-constrained synchronization operations with non-time-constrained control message transmissions to increase the degree of parallelism. Based on the system, we put forward two most frequently used graph applications on Arbor: shortest path and PageRank. In order to evaluate the system, we compare *Arbor* with the other graph processing systems using large-scale experimental graph data, and the results show that it outperforms the state-of-the-art systems. Copyright © 2014 John Wiley & Sons, Ltd.

Received 19 February 2014; Revised 31 July 2014; Accepted 25 August 2014

KEY WORDS: public opinion analysis; graph processing; graph analysis

1. INTRODUCTION

In recent years, with the growing popularity of Facebook [1], Twitter [2], Renren [3], and so on, social network service (SNS) applications achieve great successes. These applications draw more and more attentions from academia and industry communities. Ordinary users are getting used to

*Correspondence to: Jizhong Han, Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China.
†E-mail: zhouwei@iie.ac.cn

‡The original version of this paper is included in the proceeding of 15th IEEE International Conference on High Performance Computing and Communications (HPCC 2013). This paper is the extended version. Compared with the original version, there are four main differences. First, the introduction of this paper starts with the trust in social network and public opinion analysis, and then the graph processing system is suitable for them, while the original version emphasizes the graph computing model in the system. Second, in the background section, this paper discusses more about the graph representation model, including simple graph and hyper graph. Third, this paper introduces two important public opinion analysis algorithms based on the programming model in the original version, which play important role in the public opinion analysis. Lastly, this paper shows the experimental results for the two algorithms in the experimental, shortest path and PageRank, while the original version only introduces the evaluation results of shortest path.

share their stories and feelings with SNS, and it quickly becomes a commodity service in our daily life. However, it has serious security issues. Many criminals aim to use the information provided in SNS applications to create inflammatory remarks. Such activities greatly endanger the public safety and social security [4]. It jeopardizes the mutual trusts among users and impedes the further development of SNS applications.

Various mechanisms have been proposed to enhance the trust in social networks. Among them, public opinion analysis plays an important role. It aims to excavate the useful information hidden inside large-scale SNS data. This information includes but is not limited to finding the opinion leader, community detection, and friend recommendation [5]. Because of the rapid increasing number of users and user interactions/activities in SNS applications, not only the total volume of data but also the quantity and complexity of hidden information boost dramatically.

In order to discover hidden information, we need to describe the SNS data in a systematical way and then design an appropriate data processing strategy on it. For SNS applications, the graph data model is the de facto data organization format [6–9]. Research and industry communities have developed various data analysis software to discover useful information disseminated in the graph. While data analysis operations on large-scale graph data have been well studied in graph data processing systems such as Pregel [10], Twister [11], Trinity [12], and Hama [13], the larger data sizes and more complex relationships in today's graph data make these operations becoming more and more time consuming with these existing techniques. It becomes extremely difficult to obtain useful information with timely responses. Developing an efficient graph processing model is in great need.

The state-of-the-art data analysis systems can be categorized into two types: graph database and graph data processing system. In the first approach, graph data are stored in database. The graph database can provide efficient query models on graph data with the mature techniques in databases such as transaction, indexing, query language, and traverses on graph. However, these systems can only carry query and aggregation functionalities. They do not provide high performance parallel computing strategies in the graph. In the second approach, the graph data processing system applies *master–slave* framework. Slaves are the worker nodes, which execute the tasks assigned by the master. The master node is in charge of splitting the graph jobs into multiple tasks; thus, the graph jobs are executed by the slaves in parallel. Clearly, this approach can fit the needs of graph data processing tasks for scalable processing requirements. As a result, the researches on graph processing systems draw more and more attentions, and many systems are developed and widely used nowadays, including Twister [11], HaLoop [14], Pregel [10], and so on. However, this approach also has severe problems. For example, the data synchronization requirement between two consecutive processing steps impairs the benefits of parallel processing. The master node has to send synchronization messages to the slave nodes and waits for their replies. This process adds extra overheads and slows down the data operations significantly.

Based on the previous discussion, we conclude the key issues in the large-scale graph data processing system as follows.

- I/O accesses on graph data are frequent, especially for the large-scale data analysis tasks. Apparently, keeping as much data as possible in the physical memory storage space can enhance the access efficiency. However, the amount of physical memory is limited, and it cannot hold all the processing data in most scenarios.
- In the emerging systems, the data processing tasks can only progress to the next super step after all the tasks in the current super step are finished. Clearly, the system overall performance is restrained by the slowest work node. Furthermore, the master node has to conduct data synchronization operations between two consecutive super steps, which is the bottleneck. It increases the workload on the master node and reduces the effectiveness of the parallel processing significantly.

In this paper, we propose an efficient large-scale graph data processing framework. In our model, first, we propose a novel graph data organization strategy to reduce storage consumption.

DOI: 10.1002/cpe

It integrates the advantages of both hyper graph (HG) and simple graph (SG) by introducing extended simple graphs (ESGs). Second, for graph data analysis operations, our system replaces data synchronization coordinations with a control message mechanism. Third, we introduce two optimization methods to further improve the efficiency in messages transmission operations. Overall, the paper has the following contributions:

- Designing a novel data storage mechanism. The main idea is to organize graph data with ESGs. ESG can replace complete connected subgraphs in the original data with hyper edges. It can significantly reduce the storage consumption overhead.
- Replacing synchronization operation between data iterations with a novel message mechanism. A slave node can advance to the next iteration once the control message received from the other slaves matches its own record. It does not have to communicate with the master node for synchronization any more. It greatly reduces the overhead on the master node.
- Implementing an efficient real-time distributed graph data processing model, named *Arbor*, based on the previous ideas. We evaluate *Arbor* with extensive simulations, and the results show that, compared with the widely used open-source systems *Hama* and *HyperGraphDB*, it can achieve significantly higher performance.

The rest of this paper is organized as follows. Section 2 introduces the bulk synchronous parallel (BSP) model, the different representation of graph data, and the standard *graph algorithms*. Section 3 describes the system design and presents the technical details. Section 4 discusses two basic public opinion analysis applications based on the proposed processing system *Arbor*. Section 5 depicts simulation configurations and analyzes experimental results compared with other systems. Section 6 gives the related works. Finally, Section 7 concludes the paper.

2. BACKGROUND

2.1. BSP model

BSP model is a parallel computation model proposed by an English scientist Viliant [15]. The key component in BSP is *super step*, which is the minimum computation unit. A super step consists of multiple slave nodes with associated workloads. Each round, only when the assigned tasks on all the slave nodes are finished, the system can proceed to the next super step. The process of checking whether the tasks are finished or not is called synchronization, which is the most time-consuming operation in BSP. If the tasks are not well distributed on the slave nodes, the overall processing speed is restrained by the slowest slave node. As a result, the synchronization requirements seriously jeopardize the efficiency of parallel processing paradigm. To illustrate this issue, Figure 1 shows an example with two consecutive super steps. Clearly, no slave node can advance to the second super step before all tasks in the first step finish.

There are many distributed processing systems applying BSP as the data analysis model, such as Apache Hama [13], Google Pregel [10], and Microsoft Trinity [12]. Because of the nature of graph data analysis, in each iteration, the system can only obtain the local optimal results instead of global ones. In order to refine the results, multiple graph data iterations are unavoidable. For example, in the single-source shortest path (SSSP) problem, initially, each vertex receive a minimum distance information from its neighbor vertexes. However, it is based on the most recent information from its neighbor vertexes, and they are not likely the global optimal minimum distance yet. Each vertex compares this information with the current values and makes updates (if smaller than the current distance). It then sends messages containing updated distance information to its own neighbor vertexes in the next iteration. The neighbor vertexes can start comparisons and send the updated results to their own neighbors. In the consecutive rounds, each vertex keeps receiving messages containing distance information and updates their own accordingly. The whole process continues until no new results are generated. During this process, the delivering messages are not likely to be received by the neighbor vertexes simultaneously. In order to ensure the correctness, a synchronization operation is necessary between iterations. The system cannot move to the next step until the vertexes receive all the messages from their neighbors.

DOI: 10.1002/cpe

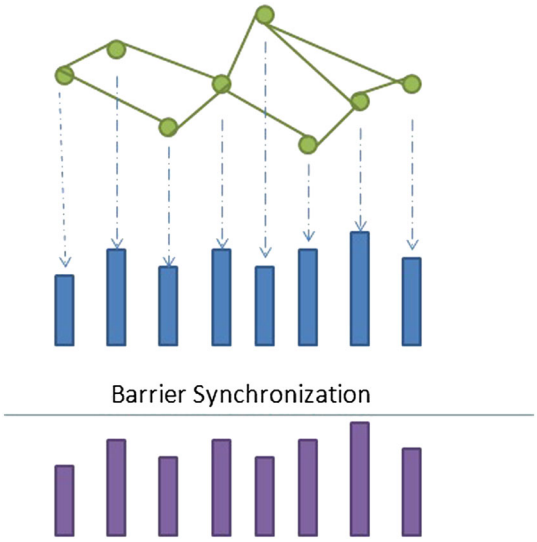

Figure 1. Bulk synchronous parallel model.

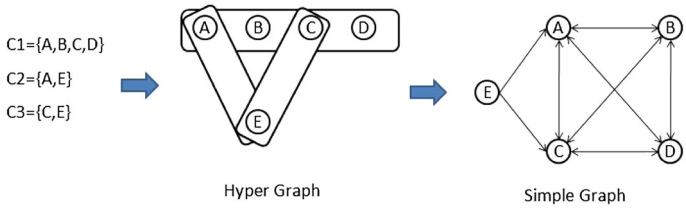

Figure 2. Simple graph (SG) and hyper graph (HG).

Clearly, BSP model is suitable for off-line graph data processing systems because the super step cannot go on until all the vertexes in this super step are finished, which is time consuming. For on-line processing, because of synchronization requirements, the system performance is determined by the slowest node. It is a great challenge to apply BSP model on real-time on-line graph data processing system and achieve satisfactory performance, especially for large-scale SNS applications.

2.2. *Simple graph and hyper graph*

We can choose two data organization mechanisms to store the graph data: SG (Simple Graph) and HG (Hyper Graph). They are applied on different applications. For example, HG is mainly used in image processing [16], human action recognition [17] applications, and so on. SG can be applied on network organization [18], human brain analysis [19] applications, and so on. However, both of them have pros and cons.

As Figure 2 displays, an SG consists of vertexes and simple edge only. A simple edge is an edge that can only connect two vertexes, for example, *AB*. According to the directional information, simple edges can be further divided into two types: directed edges and undirected edges. Based on that, two classes of SGs can be constructed: directed graph and undirected graph.

The description of HG is also shown in Figure 2. Same as SG, an HG consists of vertexes and edges. However, edges in an HG are hyper edges. A hyper edge is an edge that can connect more than two vertexes. For example, the hyper edge C1 connects four vertexes, *A, B, C,* and *D*. Here, all hyper edges are undirected edges.

DOI: 10.1002/cpe

3. GRAPH DATA PROCESSING SYSTEM DESIGN

In this paper, we propose an efficient graph data processing model called *Arbor* to address the aforementioned problems in data analysis tasks for SNS applications. *Arbor* introduces a new data organization format to reduce storage consumption and offer fast I/O accesses. In order to simplify the programming burden for data analysis system programmers, *Arbor* also provides an easy-to-use programming model (API). Thus, users only need to implement the specific interfaces following the predefined rules, and they can leave other jobs to *Arbor* framework. Users submit the graph jobs to the graph data analysis engine in *Arbor* for processing. When a job finishes and the final result is generated, the engine returns it to the corresponding user. *Arbor* applies master–slave framework to handle and execute the graph jobs. The master is responsible for analyzing and dividing the job to several tasks and assigning them to different slave nodes. Then, the slave nodes execute the tasks and return the results back to the master node.

In our system, we use ESGs, which replace simple edges with hyper edges in the storage layer. It can greatly reduce the number of edges to be stored and save space. In order to alleviate the effects of the slowest slave nodes and mitigate the scheduling and maintenance overhead on the master node, *Arbor* eliminates synchronization operations by using a novel control message mechanism. It also introduces several optimization strategies to improve message transmission performance.

3.1. Programming model

We design the programming model for *Arbor* as depicted in Figure 3. There are only two interfaces needed to be implemented by the users. One is the split interface, and the other is the computation interface. If the user chooses not to implement its own split method, then a default implementation can be chosen. It splits the vertexes based on their IDs with a hash function. The computation method corresponds to the data analysis application specifications as shown in Algorithm 1. It has to be provided by the users. No default one is offered.

Algorithm 1 Arbor Programming Model

Require:
$input$: msgs
1: value = compute(msgs)
2: edge = getOutEdgeIterator()
3: **for** each $i \in edge$ **do**
4: sendMessage(edge[i],message)
5: **end for**

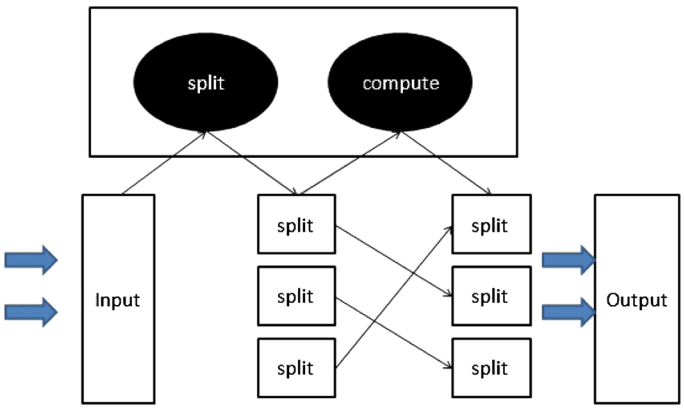

Figure 3. The programming model in *Arbor*.

DOI: 10.1002/cpe

In the algorithm, the message transmission model is embedded inside the *Arbor* system. As a result, the programming model is based on 'think as a vertex', and only the $compute(msgs)$ needs to be implemented by users. The users can only think about what computations are needed, which greatly simplifies the distributed computation complexity in data analysis tasks.

3.2. Graph data storage

In SNS applications, subgraphs of completely connected vertexes are very common, especially for triangle, a special type of completely connected subgraph. For example, as Table I shows, we take three public data sets for analysis, and we find out that the number of those subgraphs is reasonably large.

In SNS data analysis tasks, the users in the same groups or communities can be considered as a completely connected subgraph because they can contact each other directly. Usually, users in the same group have common interests or have similar service requirements, such as 'slave trade transaction group' and 'electronic books reading club'. Other examples include 'students from the same class', 'office workers in the same department' [8, 9], and so on.

Clearly, for graph applications, the complex relationships among users, groups, and communities have to be reflected in graph data organizations. It must support many-to-many and directional relationships simultaneously. Both SG and HG are not able to meet the requirements at the same time because SG can only represent one-to-one relationship, while the standard HG has no directional edges. To address this issue, in *Arbor*, we introduce a new graph data organization format, named ESG. In ESG, *Arbor* groups vertexes that are intensely connected with super vertexes. Fundamentally, ESG means that the graph is SG. However, the simple edge in SG is replaced by hyper edge in ESG. Therefore, the graph can represent many-to-many relationships with hyper edges, and at the same time, it keeps the characteristics of directions in simple edges.

As shown in Figure 4, the left side is the original graph data. If we use *SG* representation, four vertexes and eight directed edges are needed. Here, three vertexes *A, B, and C* are completely connected with six directed edges. To save space and simplify the representation, we can put them together and create a hyper edge to replace all the three nodes and six edges. As shown in the right portion of Figure 4, using *ESG*, one directed edge is defined to connect from *D* to the combination node of (A, B, and C) with just one hyper edge. Only one vertex, one directed edge, and one hyper edge are needed. Clearly, by combining hyper edges with SGs, the number of edges needed in *ESG* is reduced significantly. In *Arbor*, hyper edges are represented as key-value pairs. The *ESG* ID is used as the key. An example is shown in Figure 5.

Table I. The number of completely connected subgraphs of three data sets.

Graph	Number of triangles
Collaboration	173361
Email	727044
Flickr	3985776

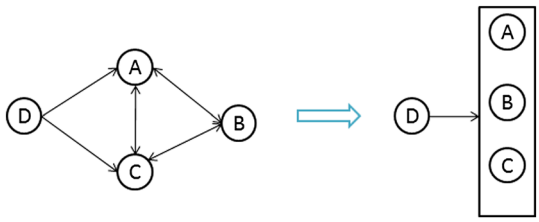

Figure 4. Extended simple graph in *Arbor*.

DOI: 10.1002/cpe

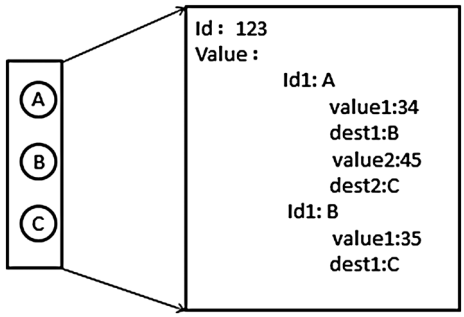

Figure 5. Key-value representation of super vertex in extended simple graph.

3.3. Graph data analysis

In order to dig out the useful hidden information, the system has to conduct graph data analysis operations, which involves complex computations and multiple iterations.

Arbor adopts BSP model for graph data analysis tasks. However, to achieve higher performance, we make significant changes in the original model and develop an efficient message mechanism to eliminate the time-consuming synchronization operations. In addition, we introduce several optimization approaches including Check Before Sending, and Avoid Unnecessary Messages to reduce the overhead even more.

3.3.1. Graph processing framework. In the traditional BSP model, the synchronization time is determined by the operations on the slowest slave. During this period, all the other slave nodes are idle. In our experiments, we found out that in most data analysis tasks, for each iteration, the slowest slave node that finishes the tasks is not always the same. As a result, the task on a certain slave node can process if all of its dependent tasks are finished already, and it does not have to wait until the slowest task to finish. Using this approach, the operations on the slowest slave nodes in multiple iterations (super steps) overlap, and the effects of those slowest tasks reduce a lot. Another advantage of this approach is that the master node does not have to coordinate the synchronizations any more, which greatly reduces the scheduling overhead.

In *Arbor*, we use a novel control message mechanism for this purpose. Arbor splits the original graph data and records the corresponding information such as which partitions containing related tasks. During data processing period, this information is used to determine if the computation tasks on a slave node should communicate with other tasks or not. For example, suppose the number of messages that a vertex should received is 2, and if it receives a message from one of its neighbors, the number is modified to 1, representing that there is only one more message needed to be received to continue this task. When this number reduces to 0, the corresponding slave node can advance to the next super step; no instruction from the master node is needed.

(a) Task Distribution

Before the graph data processing starts, *Arbor* preprocesses the original graph data and checks the legality. The relationships between vertexes and edges are recorded. For example, if vertexes A and B are connected with each other by an edge, this information is recorded as a *control message*, and it will be used later.

Next, to achieve good load balancing, *Arbor* task scheduling module carefully examines the requirements of the coming tasks and distributes them onto the slave nodes according to the available resources on those nodes. The algorithm used in task distribution is shown in Equation 1. *Arbor* takes two factors into accounts, the load balance and the independence, when distributing the tasks. Independence means that the distribution should result in fewer relationships between groups of tasks. After distribution, groups of tasks are assigned on different slave nodes.

DOI: 10.1002/cpe

The communications between the master and slave nodes are implemented by heartbeat messages. The slave nodes record the load information including the CPU, memory usages, and the number of tasks assigned and report to the master node periodically. When there are too many tasks from one job assigned to a slave node, the master node can migrate some of the tasks to other light-loaded slave nodes. The description is shown in Equation 1.

$$Q = P_d * V_d + P_l * V_l \tag{1}$$

Q identifies how suitable for a node to execute a certain task; the higher the better. P_d is the data weight factor. V_d represents whether a piece of data is requested, and it is either 1 or 0. P_l is the load balancing weight factor, and V_l is the value of load balance. P_d and P_l are both within [0, 1], and their sum is 1. V_l is greater than 0, and how to determine it is discussed in Equation 2. In most situations, the values of P_d and P_l are chosen upon experiences. If a job is computationally intensive, P_l should be higher. If it is I/O intensive, P_d should be larger.

$$V_l = 1/(((M_u/M_t) * (C_u/C_t))/((M_u/M_t) + (C_u/C_t))) \tag{2}$$

where M_u and M_t are the used memory and total memory. C_u and C_t are the used CPU and total CPU, respectively.

Arbor is a distributed framework that can handle multiple *SNS* jobs at the same time. In *Arbor* system, we define a two-layer architecture to trace load balancing status. When the jobs submitted by users are received, the master node tries to assign these jobs to different nodes based on their requirements and currently available resources in the whole cluster. The purpose is to achieve certain level of load balancing. Next, *Arbor* checks tasks in each given SNS job. If the tasks from the same job become unbalanced, the dynamic scheduling module is triggered. The system migrates the tasks on the heavy-loaded slave nodes to some other light-loaded nodes for execution. With such a strategy, *Arbor* achieves both static and dynamic load balancing with minimal overhead.

The task distribution algorithm is shown in Algorithm 2.

Algorithm 2 The Task Distribution Algorithm

Require:

$weight$: data intensive or computing intensive job

$slaves$: the load on all the slave nodes

Ensure:

Q_p: the slave nodes selected to execute a task

1: **for** each $i \in slaves$ **do**

2: $\quad Q_i = P_{di} * V_{di} + P_{li} * V_{li}$

3: **end for**

4: $Q_s = Sort(Q)$

5: $Q_p = Pick(TaskNum, Q_s)$

As Figure 6 displays, there are two vertexes in task1 that have directed edges with the vertexes in task2. Thus, we denote that the *control message* from task1 to task2 has a message number *2*. Similarly, the number of messages from task2 to task1 is also *2*. Thus, we can initialize that the *control message number* from task2 to task1 is *2* as well.

(b) Control message

The control messages generated earlier are associated with the corresponding tasks. Each task has to update the control messages as the data processing moves from one super step to the next one. The format of a control message is [source task, destination task, iteration number, message number], where the iteration number is the super step ID that the message should be transferred. For example, when task1 is first launched, the master node sends a

DOI: 10.1002/cpe

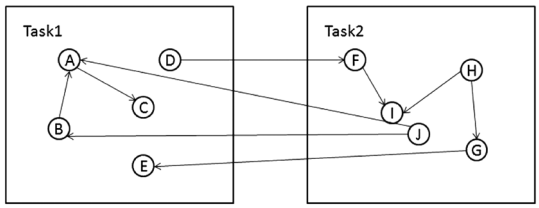

Figure 6. Example of graph data split.

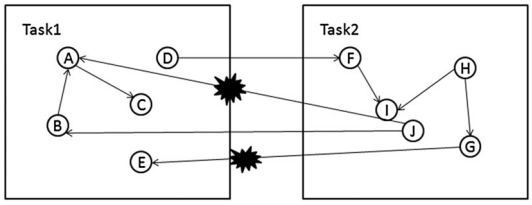

Figure 7. Control messages in *Arbor*.

message *[1,2,1,2]* to task2. It means that, in the super step 1, task1 sends a control message to task2 and the message number is *2*.

When a slave node receives a message, it checks whether the destination ID in the message matches its task ID or not as shown in Figure 7. If they are the same, the control message reaches the destination. The slave node checks the message number information, which represents how many data messages (explained next) that the destination task should receive from the source task.

(c) Data Message

Besides the control messages, *Arbor* uses data messages to transfer the actual data and information from one task to other tasks between successive super steps. The format of the data message is [source task, destination task, iteration number, value], which is similar to the control message. The value field maintains the actual data that the source task should send to the destination task. It will be used in the following iteration (super step). For example, assume task1 sends a message *[1,2,1,45]* to task2. When task2 receives this message, it sets the parameter (the number of the received messages) to *1*. It is not equal to 2 in the control message. The destination node, which is executing task1, keeps waiting for the next data message. After a while, tasks1 sends another data message *[1,2,1,56]* to task2. Now, the number of the received messages is updated to *2*, and it is equal to the number in the control message. Thus, the destination node knows that it receives both messages, and it can proceed to the next super step now.

(d) Iteration Message

During the graph data processing, some vertexes may obtain the final results earlier than other vertexes and do not have to be involved in later iterations. During the iterations, to reduce the computation overhead, the master node updates the computation information by removing the edges connected to the vertexes that obtained the final results from the computations. Thus, unnecessary messages are eliminated, and the total number of messages to be transmitted in the later super steps can be reduced significantly. Thus, it can further speed up the overall performance.

Arbor implements this strategy using the following approach. Before starting a new data iteration super step, on each vertex, the system checks the information of neighboring vertexes and marks all the vertexes that already obtain the final results. Those vertexes should be excluded from future data message transmissions. It then modifies the control messages accordingly. For example, as Figure 7 shows, if one of the two connected vertexes AJ or EG already obtained the final result, this edge is removed from later iterations. Thus, the corresponding control messages are updated to *[1,2,2,1]* and *[2,1,2,1]*, respectively.

DOI: 10.1002/cpe

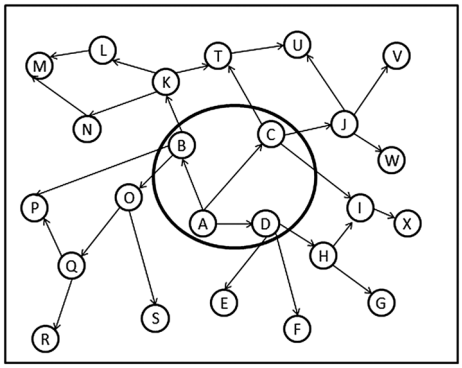

Figure 8. Example of Avoid Unnecessary Messages in *Arbor*.

3.3.2. Optimization strategies. In SNS applications, because of the complex relationships/interactions among users, a large number of data messages have to be transferred between slave nodes. It has a great impact on the overall system performance. *Arbor* introduces two optimization strategies to reduce the overhead.

(a) Check Before Sending

During the data processing, a slave node sends the final result of a vertex when obtaining it. However, sometimes, the messages do not have to be sent under certain conditions. In *Arbor*, when a slave node receives a data message, if the value in the data message is invalid, it drops the message immediately. For example, in SSSP algorithm, a smaller value means that there is a closer path from the source vertex to the current vertex. If a slave node receives a message containing a larger value than the current value, the message is useless and should be dropped. For example, when a slave node receives a data message that contains value *5*, while the stored value is only *4*, the message is dropped.

(b) Avoid Unnecessary Messages

In the graph data processing, a data iteration super step may only involve a subset of the vertexes. Other vertexes have to wait until certain messages are received before sending out their own messages. *Arbor* takes this factor into consideration and avoids generating unnecessary messages. Taking SSSP as an example, assume the source vertex is *A* and we are calculating the shortest distances from A to all other vertexes. As Figure 8 shows, in the first iteration, only nodes *B*, *C*, and *D* should be involved because they are the direct neighbor nodes of *A*. No message needs to be sent for other vertexes. In the second iteration, only the direct neighbors of *B*, *C*, and *D* should be involved. All the other vertexes are not touched yet.

4. APPLICATIONS

In this section, we describe how to use *Arbor* framework to develop the algorithms for real-world problems. The two sample applications are PageRank and shortest path. These two applications are the basic for public opinion analysis tasks. For example, PageRank can be used to calculate the influence importance of each user in the SNS. Based on this operation, the system can discover the opinion leader information.

4.1. PageRank

PageRank is a widely used graph algorithm. It can be applied on the applications such as web analysis [20], human protein interactome [21], and influence maximization in the social network [22]. An *Arbor*-based implementation of PageRank algorithm is shown in Algorithm 3.

DOI: 10.1002/cpe

Algorithm 3 PageRank Algorithm with *Arbor*

```
Require:
    input : msgs
 1: sum: 0
 2: superstep: 0
 3: if superstep ⩾ 0 then
 4:    superstep = superstep + 1
 5:    while !msgs.isEmpty() do
 6:       sum = sum + (Integer)msgs.getNextValue()
 7:       value = 1/VertexNum
 8:       value = 0.15 * value + 0.85 * sum
 9:    end while
10: end if
11: if superstep < 20 then
12:    for each i ∈ outEdges do
13:       OutedgeNum = i.getDest().size()
14:       SendMessage(i.getDest(),value/OutedgeNum)
15:    end for
16: end if
```

At the beginning, the super step ID is set to 0 when the algorithm starts. The initial value on each vertex is set to $1/VertexNum$. Then, in each of the first 20 super steps (20 is a threshold to stop the computation, and it can be any positive number according to the application specifications), each vertex sends the data messages containing this value to its neighbor vertexes through the outgoing edges. When a vertex receives messages from all its neighbor vertexes, it adds up the values and stores them in sum. It updates its current value using the formula $0.15 * VertexNum + 0.85 * sum$ and then moves to the next super step. Here, 0.15 and 0.85 are weights defined by the users according to the specific application requirements; they can be changed accordingly. When the number of super steps reaches 20, the graph data processing stops, and the current final result represents the ranks for all the web pages.

The algorithm is related to the number of edges because the messages are transferred by the edges. As a result, the time complexity is $O(m)$, where m is the number of edges. The space complexity corresponds to the message number because the messages consume the memory space and the message number is related to the number of edges.

4.2. Single-source shortest path

The shortest path problem is one of the well-known problems in graph theory [23, 24]. The applications based on it are very common in our daily life, for example, the geographic navigation systems. There are many variants, among which SSSP is the most popular one. It aims to find the shortest path from the source s to the destination t. An *Arbor*-based SSSP algorithm is shown in Algorithm 4.

As shown in Algorithm 4, the value associated with each vertex is initialized to **INF**, which is larger than any possible distance from the source to any destination. When the computation starts, each vertex sends its value to the neighbors. Meanwhile, it waits for the new values sent from other vertexes. When a vertex receives a new value from a neighbor, it compares with its own and updates with the minimum one. Clearly, in the first super step, only the source vertex needs to update its value, changing it from **INF** to zero and sending this value to its neighbors. In the second super step, when one of its neighbors receives the value, the neighbor vertex updates its own value and forwards to its neighbors in the following super step. The algorithm continues until there are no more updates occurring. After the process finishes, the value in each vertex is the shortest distance to the source vertex.

DOI: 10.1002/cpe

Algorithm 4 Shortest Path Algorithm with *Arbor*

Require:

$input$: msgs

```
1: initialNum: 1000
2: super: 0
3: mindist: Integer.valueOf(value)
4: tempDist: mindist
5: if superstep==0 then
6:    mindist = isSource(Integer.valueOf(vertexId))? 0 :initialNum
7:    lastValue = String.valueOf(initialNum)
8: end if
9: while !msgs.isEmpty() do
10:   tempValues = (String)msgs.getNextValue()
11:   arr = tempValues.split(",")
12:   valueNum = arr.length
13:   for each i ∈ [1valueNum] do
14:      mindist = min(mindist, arr[i])
15:   end for
16: end while
17: for each i ∈ outEdges do
18:   SendMessage(i.getDest(),mindist+value)
19: end for
20: value = mindist
21: lastValue = value
```

5. EXPERIMENTAL EVALUATION

In this section, we present *Arbor* performance analysis results. In our experiments, we use a pseudo graph data set, which is generated with the similar characteristics as the real SNS graph data [25]. We compare *Arbor* with *Hama*, one of the most popular graph data processing systems, and *HyperGraphDB*, a typical graph database implementation.

5.1. Experiment configurations

The pseudo test graph data used in our simulations contain of up to 10M vertexes. In the data set, the vertexes are represented by their IDs.

All the experiments are conducted on a cluster consisting of six Sugon X8DTL servers. These machines are connected with gigabyte networks. One of the servers is chosen as the master node, and the others are used as slave nodes. We installed the distributed memory storage system Redis [26] on all the machines, and the master node is also the master of Redis. The storage layer used in *Arbor* is Hadoop Distributed File System. The detailed environmental settings are listed in Table II.

Table II. The experimental environmental settings.

Type	Model or performance metrics
CPU number	Intel(R) Xeon(R) CPU E5620@2.4 GHz
Memory size	12 G
Disk size	267 G
Internet connection	Gigabyte bandwidth connections between nodes
Operation system	CentOS 505, Linux 2.6.18
Software	HyperGraphDB 1.0.0, Hama 0.4.0, Redis 2.0.4

DOI: 10.1002/cpe

5.2. *Graph data organization performance*

The storage layer keeps the graph data and provides the data access interface. It has a direct impact on the system overall performance. In the first experiment, we examine the average response time and storage space consumption metrics in those systems.

5.2.1. Average response time. The average response time of graph data analysis tasks is a key metric to measure the storage layer performance. It determines how fast the final answers can be generated and returned to the users. First, we check the performance of read operations because the majority of data accesses in data analysis tasks are reads. We compare *Arbor* with *HyperGraphDB*, and the results are shown in Figure 9. Here, we do not include *Hama* for comparison because it integrates the storage and computation operations together; no direct result can be obtained on its storage layer only.

From the results, we can observe that, in both *Arbor* and *HyperGraphDB*, the average response time becomes larger as the number of vertexes in the graph data increases. However, in all scenarios, *Arbor* has significantly smaller response time than *HyperGraphDB*. For example, when there are 100 vertexes, the average response time in *Arbor* is 33 ms while it is 3100 ms in *HyperGraphDB*. The average response time in *Arbor* is only 1.06% of that in *HyperGraphDB*. When the number of vertexes becomes 10 million, the average response time is 108 ms in *Arbor*, while it is 6300 ms in *HyperGraphDB*. The performance in *Arbor* is still much better than that of *HyperGraphDB*. We believe that the smaller storage requirement in *Arbor* is the major contribution factor for the superior performance. *Arbor* can easily place all or most of the data to be processed in the cache while it is not possible for *HyperGraphDB*.

5.2.2. Extended simple graph performance. *Arbor* uses the extended SG as the de facto data organization format in its storage layer, in which a group of vertexes can be replaced by a single hyper edge. Because of this, *Arbor* can significantly reduce the number of edges to be stored and thus reduce the storage space requirements. Additionally, with less edges, the graph partition operations can be executed in a smaller scale, which further speeds up the process. In this experiment, we measure the benefits of using hyper edges.

As shown in Figure 10, with the usage of hyper edges, the number of edges needed to be generated in *Arbor* is much smaller than the original graph data with SG representation. The number of edges in *Arbor* is only 9% of the original graph representation when the total number of vertexes is 100. While with 10 million vertexes, it achieves even better performance. The number of edges in *Arbor* is almost 99% less compared with the original data organization. A direct impact of the reduction on the number of edges is that *Arbor* uses much less storage space to keep the data, the graph data representations are much simpler, and data analysis operations can be executed much faster.

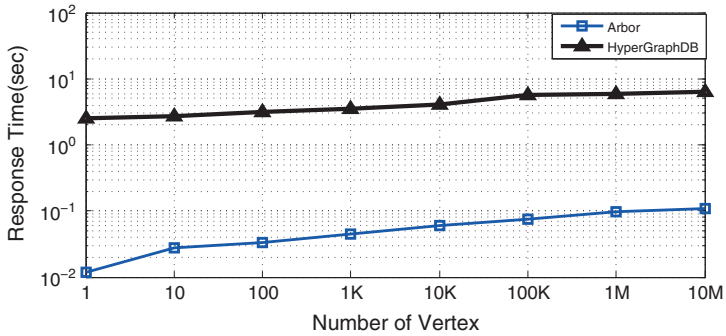

Figure 9. Response time comparison of *HyperGraphDB* and *Arbor*. The x-axis is the number of vertexes. The y-axis is the response time (in second).

DOI: 10.1002/cpe

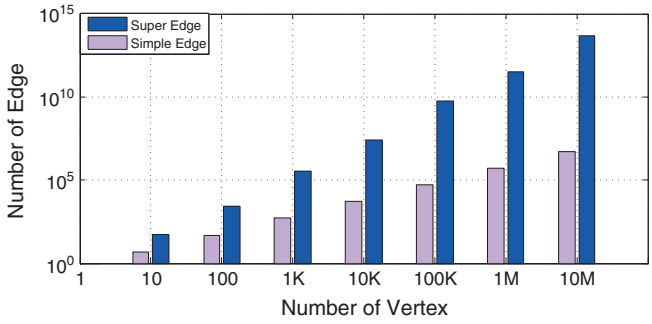

Figure 10. Storage consumption comparison with super vertexes. The x-axis is the number of super vertexes. The y-axis is the number of edges.

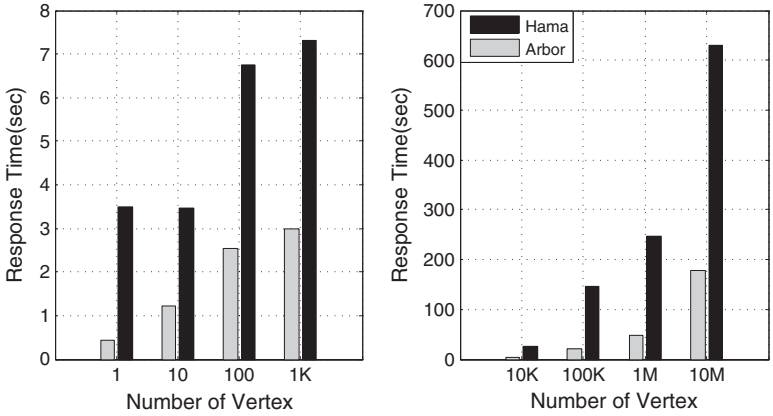

Figure 11. Response time comparison of shortest path application.

5.3. Graph data analysis

Finally, we evaluate the graph data analysis performance in *Arbor* and compare it with *Hama*. Again, we use the average response time as the major metric for comparison. We also compare the total number of super steps needed to fulfill a data analysis operation. The test case we choose is *finding the closest person to Rose* in the pseudo SNS data set. It can be solved as the SSSP problem. In this experiment, *HyperGraphDB* is not used because it is a graph database implementation. It only provides data storage, and no distributed graph processing functionality is offered.

As Figure 11 shows, the average performance in *Arbor* is much higher than that of *Hama*. In all scenarios, the average response time in *Arbor* is at least 59% lower than *Hama*. As the number of vertexes increases, the performance difference becomes larger. With 100K number of vertexes, *Arbor* achieves the biggest performance gain, and it only consumes 14.25% of the average response time in *Hama*. Apparently, *Arbor* has better scalability and is able to generate the final results much faster than *Hama*.

Figure 12 compares the average number of super steps needed to execute a graph data analysis operation. In this figure, the example is PageRank. Again, we can observe the similar trend. In all situations, *Arbor* outperforms *Hama* significantly. It takes much less number of iterations to generate the final result than *Hama*. For example, when the vertex number reaches one million, the iteration number of *Arbor* is 2134, while *Hama* needs 3324. The iteration number is only 64.2% of *Hama*.

In addition, we compare the two systems using PageRank information, which can reflect the contributions of each vertex in the graph.

As shown in Figure 13, the average performance is much higher in *Arbor* than in *Hama*. In all scenarios, the average response time in Arbor is at least 17% lower than *Hama*. As the number

DOI: 10.1002/cpe

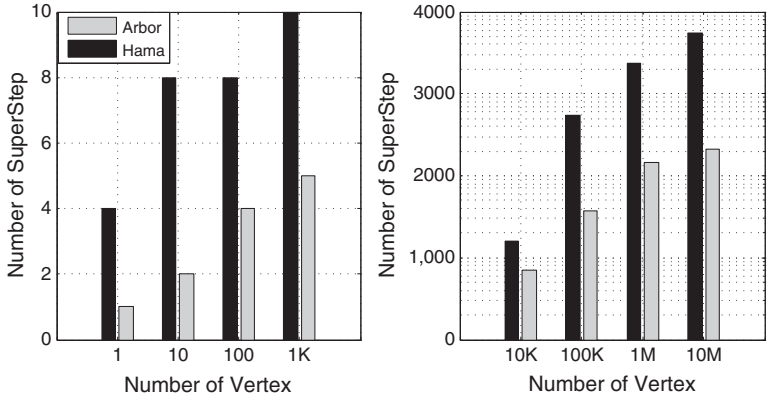

Figure 12. The number of super steps comparison of data analysis.

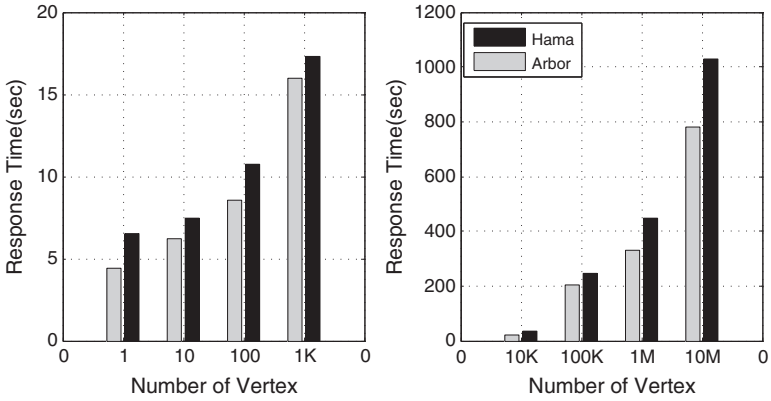

Figure 13. Response time comparison of PageRank.

Table III. The number of messages comparison using Check Before Sending optimization.

Vertex number	1	10	100	1K
Check before sending	0	15	2398	226548
No check before sending	0	21	2767	281254

of vertexes increases, the performance difference becomes larger. With 10M number of vertexes, *Arbor* achieves the biggest performance gain, and it only consumes 75.34% of the average response time in *Hama*. Similar to Figure 11, *Arbor* offers higher scalability. We also compare the number of super steps (iterations) to generate the final results. The similar trend shown in Figure 12 can be obtained.

From the previous experiments, we can draw the conclusion that *Arbor* reduces the maintenance overhead for synchronization operations significantly with the introduced control message strategy. As mentioned, *Arbor* also introduces several improvements to further increase the performance. In the following experiments, we evaluate their effectiveness.

5.3.1. Check Before Sending. In Arbor, this technique allows the system to check the validity of the received messages and avoids unnecessary message transmissions.

As Tables III and IV show, this method works very well. In all scenarios, the system reduces the number of messages to be transmitted significantly. Furthermore, this method does not result in the

DOI: 10.1002/cpe

Table IV. The number of messages comparison using Check Before Sending optimization.

Vertex number	10K	100K	1M	10M
Check before sending	1974623	436315798	404325654	3267907535
No check before sending	2976769	504587439	556734543	4333286385

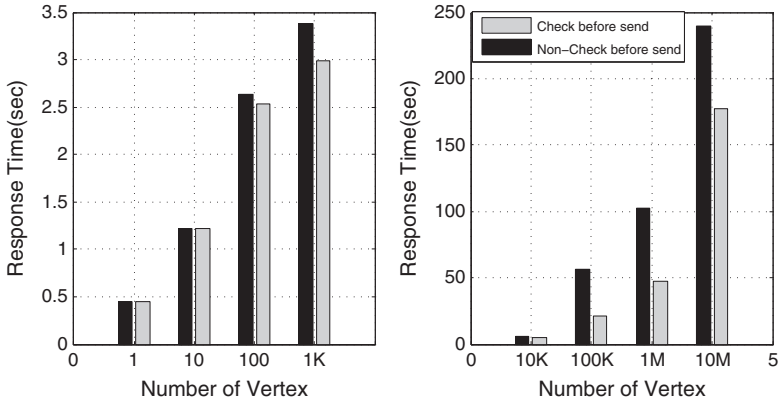

Figure 14. Check Before Sending optimization performance.

Table V. The number of messages comparison using Avoid Unnecessary Messages optimization.

Vertex number	1	10	100	1K
Send for the first time	0	15	2770	260454
Non-sending for the first time	0	9	2574	273467

Table VI. The number of messages comparison using Avoid Unnecessary Messages optimization.

Vertex number	10K	100K	1M	10M
Send for the first time	3148769	504587439	556734532	4333286385
Non-sending for the first time	2964347	487567849	531467389	4276643376

increases on the number of super steps. As Figure 14 shows, using this technique, *Arbor* always has a lower average response time under different numbers of vertexes. The larger the number of vertexes, the more the average response time reduction we can acquire.

5.3.2. Avoid Unnecessary Messages. As we discussed in previous sections, some messages are meaningless and should not be sent to neighbors. In *Arbor*, each node carefully examines the received messages and abandons those unnecessary messages.

Tables V and VI compare the number of total messages sent with different methods. Avoid Unnecessary Messages method always has a less number than the original approach. However, this reduction does not always result in better performance. As shown in Figure 15, when the number of vertexes is small, this strategy helps. However, when the number of vertexes is larger than 1000, it has a negative effect and actually increases the response time. The main reason of this phenomenon is that although it can reduce the number of messages to be sent, it turns out to add more data iterations and consequently increases the average response time. Thus, as the number of vertexes increases, the benefits it achieves cannot compensate the associated overhead any more. The system performance becomes worse.

DOI: 10.1002/cpe

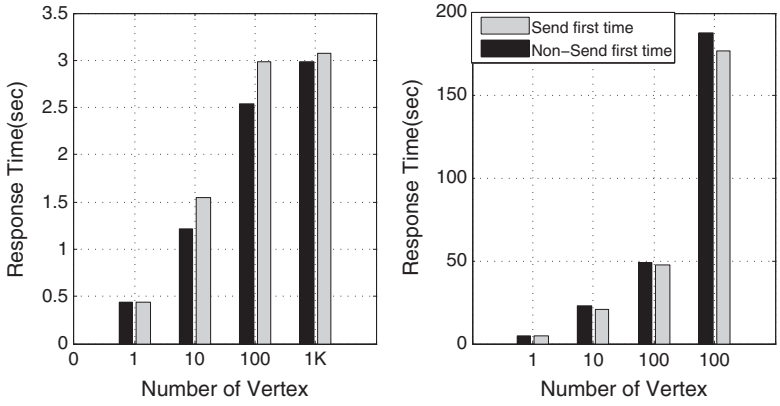

Figure 15. Avoid Unnecessary Messages optimization performance.

6. RELATED WORKS

With the popularity of SNS applications, many large-scale graph processing systems are proposed, such as *MapReduce* [27, 28]. Although *MapReduce* is a very good fit for a wide range of computing problems and sometimes is used for large graphs, it has suboptimal performance and usability issues. If a user wants to use *MapReduce* for the graph data processing, it has to implement the iterative *MapReduce* by itself. In order to reduce the complexity of the iterative implementation, there are emerging open-source implementations proposed, such as HaLoop [14] and Twister [11]. Twister [11] is a system of iterative *MapReduce* [29], which is suitable for the graph processing because the graph analysis involves multiple iterations. However, Twister is limited by the *MapReduce* [29] architecture, which is designed for off-line analysis purpose. HaLoop [14] is another variant of *MapReduce*. In *HaLoop*, the master node has to handle the synchronization for each iteration, and it is also suitable for off-line analysis only.

Novel large-scale graph data processing systems have been introduced to relieve the issues. These systems can be divided into two types: the graph database and the graph data processing system. The graph database provides direct database type organization for the graph data storage and simple data access interface. It includes HyperGraphDB [30], Neo4J [31], InfiniteGraph [32], and so on. Graph database is important in the development of graph data processing. With the growing volume of the graph data in SNS applications, this approach gains more attentions. Among all the graph database, *HyperGraphDB* is the most popular one. *HyperGraphDB* is a generally used, scalable, portable storage mechanism. It is designed for artificial intelligence and semantics web applications. It uses Berkeley DB as the storage layer. *HyperGraphDB* can support graph data storage and SG data queries and analysis. However, *HyperGraphDB* is a single-node database, which cannot support the large-scale graph data aggregation and complex data analysis tasks.

The graph data processing systems, such as Google Pregel [10], Apache Hama [13], and Microsoft Trinity [12], manage the system main memory with the underlying storage layer together. Such an integration mechanism can speed up data processing operations. Pregel [10] is a large-scale off-line graph data processing framework. It adopts BSP model for synchronization. However, Pregel only supports graph data analysis instead of graph query and aggregation. It is designed for Google internal applications and may not suitable for other applications. Trinity is a query and analysis system. It provides two main functions: graph data query and graph data analysis. However, it is used for off-line batch processing tasks and does not satisfy the on-line requirement. Furthermore, Trinity does not support the graph data aggregation. Hama [13] is a distributed system designed for graph data processing. However, in *Hama*, the synchronization of two super steps is performed by the Zookeeper, which is time consuming. Furthermore, it is an off-line batch processing system and does not fit the on-line graph data processing requirements.

DOI: 10.1002/cpe

7. CONCLUSION

In this paper, we first summarize the key problems in the existing large-scale graph data processing systems. Then, we propose a novel graph data processing system called *Arbor* to address these problems. *Arbor* introduces a new data organization model called ESG, which uses hyper edges to reduce the data representation complexity. It cuts the storage consumption greatly and speeds up graph data processing operations. *Arbor* also proposes a novel control message mechanism to replace expensive synchronization operations during data iterations. It is proven to be very effective, especially for large-scale graph data processing tasks. Furthermore, *Arbor* designs two optimization strategies including Check Before Sending and Avoid Unnecessary Messages to further improve the efficiency. The evaluation results show clearly that *Arbor* is superior than the state-of-the-art systems.

ACKNOWLEDGEMENTS

This research is supported in part by the National High Technology Research and Development Program of China (863 Program) under grant 2012AA01A401, and the Strategic Priority Research Program of the Chinese Academy of Sciences under grant XDA06030200.

REFERENCES

1. FaceBook. Available from: http://www.facebook.com/ [Accessed on 10 June 2013].
2. Twitter. Available from: http://www.twitter.com/ [Accessed on 10 June 2013].
3. Renren. Available from: http://www.renren.com/ [Accessed on 19 May 2013].
4. Huohuo Q. Available from: http://zh.wikipedia.org/wiki/Qinhuohuo [Accessed on 13 August 2013].
5. Chen W, Cheng S, He X, Jiang F. InfluenceRank: an efficient social influence measurement for millions of users in microblog. *2012 Second International Conference on Cloud and Green Computing (CGC)*, Xiangtan, Hunan, China, 2012; 563–570.
6. Hido S, Kashima H. Graph similarity calculation system, method, and program 2012. EP Patent 2,442,239.
7. Fire M, Katz G, Elovici Y, Shapira B, Rokach L. Predicting student exam's scores by analyzing social network data. *Active Media Technology* 2012:584–595.
8. Yuehua Y, Junping D, Yingmin J, Zengqi S. Study on SNS graph generation and prediction. *2010 International Conference on Control Automation and Systems (ICCAS)*, KINTEX, Gyeonggi-do, Korea, 2010; 1188–1191.
9. Wakita K, Tsurumi T. Finding community structure in mega-scale social networks. *arXiv preprint cs/0702048*, 2007.
10. Malewicz G, Austern MH, Bik AJC, Dehnert JC, Horn I, Leiser N, Czajkowski G. Pregel: a system for large-scale graph processing. *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, 2010; 135–146.
11. Ekanayake J, Li H, Zhang B, Gunarathne T, Bae SH, Qiu J, Fox G. Twister: a runtime for iterative MapReduce. *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, Chicago, Illinois, 2010; 810–818.
12. The graph processing system form Microsoft, called Trinity. Available from: http://research.microsoft.com/en-us/projects/trinity/ [Accessed on 12 September 2013].
13. Graph computation system from Apache *hama*.
14. Bu Y, Howe B, Balazinska M, Ernst MD. HaLoop: efficient iterative data processing on large clusters. *Proceedings of the VLDB Endowment* 2010; **3**(1–2):285–296.
15. The introduction of BSP. Available from: http://baike.baidu.com/view/757269.htm [Accessed on 15 April 2013].
16. Yu J, Tao D, Wang M. Adaptive hypergraph learning and its application in image classification. *IEEE Transactions on Image Processing* 2012; **21**(7):3262–3272.
17. Celiktutan O, Wolf C, Sankur B, Lombardi E. Real-time exact graph matching with application in human action recognition. *Human Behavior Understanding* 2012:17–28.
18. Bossard A, Kato T, Masuda K. Supporting reconstruction of the blood vessel network using graph theory: an abstraction method. *2012 Annual International Conference of the IEEE Engineering in Medicine and Biology Society (EMBC)*, San Diego, California, USA, 2012; 5470–5473.
19. Vertes PE, Alexander-Bloch AF, Gogtay N, Giedd JN, Rapoport JL, Bullmore ET. Simple models of human brain functional networks. *Proceedings of the National Academy of Sciences* 2012; **109**(15):5868–5873.
20. Polyak BT, Timonina A. PageRank: new regularizations and simulation models. *World Congress* 2011; **18**(1): 11202–11207.
21. Du D, Lee CF, Li XQ. Systematic differences in signal emitting and receiving revealed by pagerank analysis of a human protein interactome. *PLOS One* 2012; **7**(9):e44872. Public Library of Science.
22. Luo ZL, Cai WD, Li YJ, Peng DA. PageRank-based heuristic algorithm for influence maximization in the social network. In *Recent Progress in Data Engineering and Internet Technology*. Springer, 2012; 485–490.

DOI: 10.1002/cpe

23. Cherkassky BV, Goldberg AV, Radzik T. Shortest paths algorithms: theory and experimental evaluation. *Mathematical Programming* 1996; **73**:129–174.
24. Gross JL, Yellen J. *Graph Theory and Its Applications* (2nd edn). Chapman and Hall/CRC, CRC press, 2005.
25. Low Y, Gonzalez J, Kyrola A, Bickson D, Guestrin C, Hellerstein JM. Graphlab: a new framework for parallel machine learning. *arXiv preprint arXiv 1006. 4990*, 2010.
26. *The Homepage of Redis*. Available from: http://redis.io/ [Accessed on 7 January 2014.
27. Cohen J. Graph Twiddling in a MapReduce World. *Comparative in Science and Engineering* 2009; **11**(4):29–41.
28. Kung U, Tsourakakis CE, Faloutsos C. Pegasus: a peta-scale graph mining system – implementation and observations. *Proceedings of International Conference in Data Mining*, Miami, Florida, USA, 2009; 229–238.
29. Dean J, Ghemawat S. MapReduce: simplified data processing on large clusters. *Communications of the ACM* 2008; **51**(1):2008.
30. Graph Database HyperGraphDB. Available from: http://www.open-open.com/open316576.htm [Accessed on 19 November 2013].
31. Graph Database Neo4J. Neo4J. http://neo4j.org/ [Accessed on 19 May 2013].
32. Graph Database Infinitegraph. Available from: http://www.infinitegraph.com/ [Accessed on 7 December 2013].
33. Sherchan W, Nepal S, Paris C. A survey of trust in social networks. *ACM Computing Surveys (CSUR)* 2013; **47**(4):1–13.
34. Social Network Service. Available from: http://newsroom.fb.com/Key-Facts [Accessed on 2 December 2013].
35. Ghemawat S, Gobioff H, Leung ST. The Google file system. *ACM SIGOPS Operating Systems Review* 2003; **37**(5):29–43.
36. Chang F, Dean J, Ghemawat S, Hsieh WC, Wallach DA, Burrows M, Chandra T, Fikes A, Gruber RE. Bigtable: a distributed storage system for structured data. *ACM Transactions on Computer Systems(TOCS)* 2008; **26**(2):4.
37. Yuehua Y, Junping D, Yingmin J, Zengqi S. Study on SNS graph generation and prediction. *2010 International Conference on Control Automation and Systems (ICCAS)*, KINTEX, Gyeonggi-do, Korea, 2010.
38. Khetrapal A, Ganesh V. *Hbase and hypertable for large scale distributed storage systems*. Department of Computer Science, Purdue University, 2006. http://www.uavindia.com/ankur/downloads/HypertableHBaseEval2.pdf.
39. The Home Page of Sina Weibo, it is a widely used SNS platform in China. Available from: http://weibo.com [Accessed on 12 April 2013].
40. iResearch report about the SNS in China, The speedup of chinese SNS and weibo platform, 2011.
41. Borthakur D. The Hadoop Distributed File System: Architecture and Design. *Hadoop Project Website* 2007; **11**:21.
42. Frischbier S, Margara A, Freudenreich T, Eugster P, Eyers D, Pietzuch P. ASIA: application-specific integrated aggregation for publish/subscribe middleware. *Proceedings of the Posters and Demo Track*, ACM New York, NY, USA, 2012.
43. Ausiello G, Franciosa P, Italiano G, Ribichini A. Computing graph spanners in small memory: fault-tolerance and streaming. *Discrete Mathematics, Algorithms and Applications* 2010; **2**(04):591–605. World Scientific.
44. Lou YS, Zhang WY, Xu F, Wang Y, Chen S. Parallel implementation of single-source shortest path algorithm based on haloop. *Applied Mechanics and Materials* 2012; **220**:2428–2432.

DOI: 10.1002/cpe